

Д. Джамп

AutoCAD

ПРОГРАММИРОВАНИЕ

AutoCAD® PROGRAMMING

Dennis N. Jump



TAB Professional and Reference Books

Division of TAB BOOKS Inc.

Blue Ridge Summit, PA

Д. Джамп

AutoCAD

ПРОГРАММИРОВАНИЕ

ПЕРЕВОД С АНГЛИЙСКОГО
С. С. БОГДАНОВА
ПОД РЕДАКЦИЕЙ
А. С. БОГДАНОВА

МОСКВА
«РАДИО И СВЯЗЬ»
1992

ББК 32.973
Д40
УДК 519.682.1

Редакция литературы по вычислительной технике и информатике

Д ж а м п Д.
Д 40 AutoCAD. Программирование: Пер. с англ. — М.: Радио и
связь, 1992. — 336 с.: ил.
ISBN 5-256-00807-2.

В книге американского автора описывается система автоматизированного проектирования AutoCAD, позволяющая синтезировать двух- и трехмерные изображения на экране дисплея, редактировать их, получать копии в виде чертежей. Рассмотрены особенности реализации системы на компьютерах различных типов.

Для разработчиков графических систем, пользователей системы AutoCAD.

Д $\frac{2404020000 - 069}{046(01) - 92}$ 61 - 92

ББК 32.973

ISBN 5-256-00807-2 (рус.)
ISBN 0-8306-9393-9 (англ.)

© 1989 by TAB BOOKS Inc.
© Перевод на русский язык.
Богданов С.С., 1992
© Предисловие к русскому
изданию. Примечания.
Богданов А.С., 1992

Предисловие к русскому изданию

Читателю, открывшему эту книгу, наверняка не стоит представлять систему автоматизированного проектирования AutoCAD. Разработанная фирмой Autodesk Inc. и появившаяся на рынке в конце 1982 г. система AutoCAD получила необычайно широкое распространение. Она успешно применяется в различных областях приложений: машиностроении, архитектуре, электронике — везде, где требуется построение и подготовка чертежей, планов, схем, иллюстраций. Система AutoCAD работает на различных типах персональных компьютеров, поддерживает большое число различных графических периферийных устройств, существует даже ее реализация на рабочих станциях серии VAXstation фирмы DEC.

Популярность системы AutoCAD столь велика, что созданы различные объединения пользователей этой системы, выпускается ряд целиком посвященных системе журналов, таких, как CAD User, CADalyst. Причина популярности — открытость системы, возможность разработки на базе системы AutoCAD своих собственных прикладных пакетов программ, в которых используется инженерная графика. Система AutoCAD представляет собой своего рода графическое ядро, в котором реализованы многие функции, специфичные для инженерной графики. Набор этих функций может быть эффективно использован для различных приложений, но также легко он может быть модифицирован и расширен.

Книга посвящена вопросам создания программных интерфейсов для взаимодействия и обмена графической информацией с системой AutoCAD. В книге рассматриваются структура и формат внешних файлов обмена графической информацией: файлов формата DXF, DXB и PGP, пакетных файлов системы AutoCAD. Формат внутренних файлов системы AutoCAD — файлов чертежей (DWG-файлы) — является конфиденциальной информацией фирмы Autodesk Inc. и к тому же претерпевает изменения с выпуском новых версий системы. Формат внешних файлов остается практически неизменным. Программисты, пишущие прикладные программы для взаимодействия с системой AutoCAD, смогут по достоинству оценить информацию, содержащуюся в данной книге. Словами автора можно сказать: "Настоящие программисты используют DXF-файлы (и только садисты работают с файлами чертежей — DWG-файлами)".

В качестве рабочего прототипа (на момент написания книги) автор выбрал версию 2.9 системы AutoCAD. Но тем не менее в книге упоминаются и некоторые особенности системы AutoCAD версии 10. В процессе подготовки русского издания была анонсирована новая

версия системы AutoCAD — версия 11. Однако материал данной книги не устареет и для последующих версий системы.

Книга окажется полезной всем, кто хочет углубить знания по системе AutoCAD.

А. Богданов

Благодарности

Существуют два способа избежать неудачу:

1. Не начинать.
2. Не останавливаться.

Я признателен многим людям, заслужившим благодарность (или проклятье), всем тем, благодаря кому я начал работу над этой книгой. А также людям, не позволившим мне остановиться, их по праву можно назвать святыми. Я назову всех вместе, так чтобы не было ясно, к какой категории я их отношу: Ларри Хейгер, Рон Пауэрс, моя семья, Ренцо Спанхофф (Фанфары!), сотрудники фирмы The Great SoftWestern Company, (Билл, Диана, Сюзан и Терри).

Гуинн, Шелли и Хилари,
большинству их пациентов

P.S.

Тем из вас, кто все еще удивляется, что я сделал для жизни, это объяснит немного лучше.

ВВЕДЕНИЕ

Существует множество доводов в защиту применения систем автоматизированного проектирования на персональных компьютерах. Большая часть этих доводов объясняет успех системы AutoCAD. Прошу простить, если я пропущу основное, но я могу привести следующее:

- очень широкий набор команд;
- хорошее сочетание простых и сложных функций;
- интуитивное понимание назначения команд;
- широкий выбор периферийных устройств и их модификаций;
- несколько национальных языковых вариантов системы;
- возможность изменять меню команд, как устанавливаемых на планшете, так и появляющихся на экране;
- наличие встроенного языка программирования;
- хорошая документированность системы;
- большое число прикладных программ, разработанных различными фирмами.

Что касается последнего соображения — если не ясно, откуда оно взялось? — то это и есть предмет обсуждения данной книги. Мы собираемся рассмотреть, как можно писать прикладные программы, взаимодействующие с системой AutoCAD. Система AutoCAD может как передавать данные прикладным программам, так и отображать чертежи, созданные с помощью прикладных программ.

Не рассчитывайте на то, что книга позволит написать все необходимые для вас прикладные программы. Это невозможно. Она только покажет, как можно получить данные из системы AutoCAD для использования их в программах анализа. Но программы анализа вы разработаете сами. Книга покажет, как можно подготовить данные для ввода в систему AutoCAD. Но предоставить данные системе — снова ваша задача.

Я разделил материал по разработке прикладных программ на четыре раздела. Каждый раздел состоит из двух глав, относящихся к рассматриваемой теме. Это следующие разделы: методы программирования; настройка параметров системы AutoCAD; система AutoCAD как управляющая система; система AutoCAD как исполнительная система.

AutoCAD является зарегистрированным фирменным знаком фирмы Autodesk Inc.

В разделе "Методы программирования" (гл.1 и 2) вы найдете обзор различных способов и приемов программирования, применяемых в данной книге. Тема весьма общая, но необходимая, чтобы подвести всех к общему старту при рассмотрении вопросов программиро-

вания для систем автоматизированного проектирования. В разделе даны пояснения и приведено множество примеров организации структур данных типа двоичных деревьев, работы со строками, а также описана программная реализация конечных автоматов, широко используемых в данной книге.

Все части раздела "Настройка параметров системы AutoCAD" (гл.3 и 4) довольно легкие. В этом разделе даны определения и основные понятия системы AutoCAD. Пользователи системы AutoCAD могут выполнять свою работу, не вникая во внутреннюю организацию и нюансы системы. В разделе рассматриваются структура команд, интерфейс с операционной системой MS-DOS и назначение файлов системы AutoCAD.

Следующий раздел — "Система AutoCAD как управляющая система" — посвящен вопросам обработки выходных файлов системы, которые можно использовать как источник графической информации для ваших программ. Имеются два типа файлов, представляющие интерес в этом смысле.

В гл.5 рассматривается формат файлов обмена описаниями чертежей (DXF-файлы). Формат этих файлов в настоящее время является стандартным для обмена графической информацией между системами автоматизированного проектирования, реализованными на персональных компьютерах. Если вы используете систему AutoCAD или какую-либо другую систему, то данная глава является для вас важной. В ней, правда, есть немного резкие переходы. Надеюсь, что мне удалось их сгладить.

Глава 6 посвящена системе атрибутов. Существуют два способа извлечения атрибутов из чертежа. Это осуществляется с помощью команды ATTEXT (извлечь атрибуты) и файла типа TXT, а также с помощью DXF-файлов. В главе рассматриваются оба способа. Конечно, чтобы помочь читателю, в главе приводятся многочисленные листинги программ.

Система AutoCAD может управлять работой ваших прикладных программ. И наоборот, ваши прикладные программы могут управлять работой системы AutoCAD. В разделе "Система AutoCAD как исполнительная система" рассматриваются способы передачи графической информации системе AutoCAD.

В гл. 7 обсуждаются некоторые аспекты использования пакетных файлов системы AutoCAD. Пакетные файлы позволяют перенаправить ввод команд с клавиатуры на ввод команд из файла. Сгенерированные с помощью программ и записанные в пакетный файл команды передаются непосредственно командному процессору системы AutoCAD. Вы не должны сразу же "окунуться с головой" в генерацию пакетных файлов. Я позволю вам сперва "смочить ноги". В главе приводятся несколько небольших пакетных файлов, которые удобно использовать для ежедневной работы. Кроме того, я покажу мой способ генерации пакетных файлов системы AutoCAD для различных национальных версий системы AutoCAD.

Глава 8 изрядно насыщена, но не содержит таких резких переходов, как гл. 5. В данной главе рассматривается генерация файлов типа DXF и DXB (двоичный формат файлов обмена описаниями чертежей) с помощью внешних программ. В главе разбираются все составные части DXF-файла и даются рекомендации по формированию обязательно требующихся записей. В качестве примеров приводится несколько вариантов реализации программ. Это наглядно показывает, что существует множество различных способов генерации данных. Большинство примеров — это тексты реальных программ, которые я заимствовал из своих работ (Хорошо, что я имею свою фирму.). Они не все взяты из одной программы, так что их стиль и приемы программирования не всегда совпадают.

Что бы я хотел показать в этих главах, так это сходство подготовки данных для записи их в файлы. Хотя вы можете идти своим путем, несколько отличающимся от моего в зависимости от формата файлов, вам все равно следует иметь возможность изменения подпрограмм вывода для перестройки программ вывода в формате пакетных файлов на вывод в формате DXF- и DXB-файлов. При переходе от одного формата вывода к другому обязательно требуемые данные не изменяются, изменяется только "начинка" между ними.

В приложениях приведены большие таблицы и листинги программ. Они необходимы для разработки серьезных прикладных программ, но весьма громоздки для включения их в текст обычных программ.

Во время подготовки книги к изданию фирма AutoDesk Inc. выпустила новую версию системы — AutoCAD версия 10. В этой версии имеются долгожданные команды для работы с трехмерными объектами, предназначенные для более сложных разработок в области машиностроения и архитектуры. Я имел возможность использовать этот продукт совместно с некоторыми нашими коммерчески распространяемыми пакетами программ, алгоритмы этих программ используются в книге. Я не вдавался глубоко в проблемы совместимости. В алгоритмах, приведенных в данной книге, опущены новейшие элементы, введенные в систему AutoCAD версии 10. Я задумал еще одну книгу, в которой будут освещены детали системы AutoCAD версии 10, язык программирования АвтоЛИСП, а также другие аспекты разработки прикладных программ для системы AutoCAD.

Прошу пользоваться.

Глава 1

СТРУКТУРЫ ДАННЫХ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

Добро пожаловать в мир программирования для систем автоматизированного проектирования. Для вас есть приятные известия, эта область не так уж велика и в ней найдется место для хороших прикладных программ, написанных для систем автоматизированного проектирования (пока вы не являетесь конкурентами: конкурентам не следует покупать эту книгу).

Мы собираемся совершить экскурс в методы программирования, которые можно использовать при написании программ, взаимодействующих с системой AutoCAD. Начнем с подробного разбора нескольких основных соглашений по программированию, которые будут применяться при написании прикладных программ.

Затем достаточно скоро мы перейдем к рассмотрению интерфейсных программ для системы AutoCAD. Будут рассмотрены способы извлечения графической информации из файлов типа DXF и файлов атрибутов системы AutoCAD. Подвигав ящики столов, мы ознакомимся с несколькими форматами графических данных, используемых в системе AutoCAD, и подготовкой данных в этих форматах. Далее узнаем, как можно управлять системой AutoCAD с помощью своих собственных команд. В заключение будет рассмотрено написание прикладных программ для системы AutoCAD на языке АвтоЛИСП - встроенном языке этой системы.

Я ничего не знаю о вас, но я не люблю долгие вступления, впрочем, так же, как и долгие прощания. Давайте начнем с программирования.

Структуры данных

Хотя кажется, что создание хорошего программного обеспечения зависит от разработки хороших алгоритмов, на самом деле, эффективное использование и хранение информации может превратить разработку алгоритмов в весьма заурядную задачу. В следующих двух главах будут рассмотрены некоторые основные методы организации и обработки данных.

Основная цель данной главы - исследовать, объяснить и построить простейшие структуры данных и алгоритмы их обработки. Эти вопросы составляют основу применяемых в дальнейшем методов.

В гл. 2 будут рассмотрены более сложные вопросы, внимание уделится объединению алгоритмов для выполнения более сложной обработки. Как только мы разложим на столе все составные элементы головоломки, мы сможем приступить к ее решению. Поэтому некоторые из рассматриваемых в последующих главах вопросов будут являться продолжением обсуждаемых в данной главе.

Но основным, что вы почерпнете из этих глав, будет накопление полезной информации о файлах данных системы AutoCAD. Это сведение не только об организации и хранении данных; они позволят составить представление о том, как должна работать ваша программа - об управлении работой программы потоками данных, а не процедурами. В данной главе рассматриваются следующие темы: представление линий и точек; двоичные деревья; представление двоичных деревьев; хранение символьных строк; таблицы имен.

Представление линий и точек

Поскольку основным графическим примитивом в системах машинной графики является линия, первая структура данных, с которой мы будем иметь дело, представляет собой средство организации и хранения совокупности линий. Так как информация о длине линий и соединении их между собой может быть важным аспектом в некоторых прикладных программах и программах машинной графики, то в данном разделе внимание будет уделено разработке структур данных, с помощью которых на базе информации о линиях строится и сохраняется соотношение между линиями.

Начнем с простейшего случая задания и сохранения информации о линии. Но для этого сначала необходимо дать определение точки. На рис. 1.1 показана точка в трехмерном пространстве. Хотелось бы, чтобы рисунок был объемным, но это невозможно сделать при двумерном представлении. Точка не имеет размеров, она просто занимает некоторое положение в пространстве. В трехмерном пространстве, местоположение точки может быть определено расстоянием относительно трех взаимноперпендикулярных осей X , Y и Z . Единицы измерения расстояния могут быть произвольными.

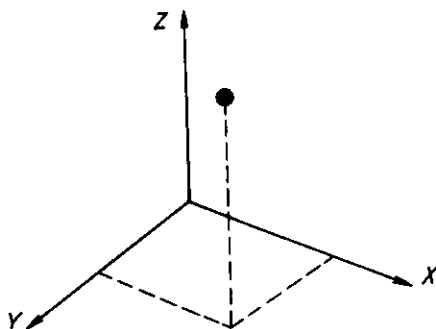


Рис.1.1. Точка в трехмерном пространстве

В нашем случае линия идентифицируется выбором двух точек, принадлежащих этой линии. В элементарной геометрии прямая линия имеет бесконечную длину и проходит через две заданные точки. На рис. 1.2 показана прямая, проходящая через две точки. Однако в системах машинной графики в действительности имеют дело с отрезками прямых. Это обычные прямые, которые ограничены точками начала и конца. В данном случае две точки, определяющие прямую, являются ее концами. На двухмерном чертеже местоположение точек определяется их координатами X и Y . Для трехмерного чертежа необходимо добавить третью координату Z . На рис. 1.3 показан отрезок, заданный двумя точками. Далее мы будем употреблять термин линия, подразумевая отрезок прямой.

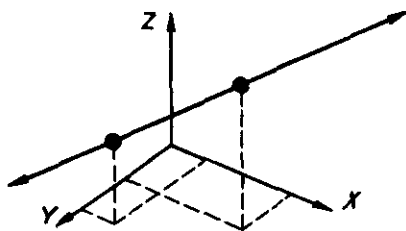


Рис. 1.2. Линия в пространстве

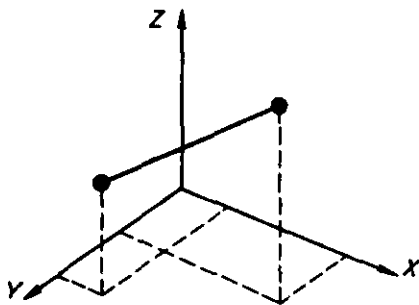


Рис. 1.3. Точки концов отрезка прямой

При программировании для запоминания значений координат точек концов простых линий или в ситуациях, когда желательно быстро получить эту информацию, используйте простые переменные. Взглянув на отрезок, точки концов которого заданы координатами X , Y и Z , можно использовать следующие переменные. На лист. 1.1

приведен пример определения группы переменных для хранения координат отрезка.

```
double      from_x;
double      from_y;
double      from_z;
double      to_x;
double      to_y;
double      to_z;
```

Лист. 1.1. Простые переменные для координат

Данный способ прекрасно подходит для не очень сложных случаев программной реализации. Существует множество обстоятельств, когда данная форма организации удобна, поскольку в силу своей простоты она не требует дополнительных затрат. Обычно такое представление применяется в примитивных подпрограммах. Возьмем, например, задачу нахождения длины отрезка. На лист. 1.2 показан исходный текст функции для определения длины отрезка. В этой программе переменные, содержащие значения координат, являются параметрами функции. С точки зрения функции передаваемые значения - это простые переменные. Когда координаты точек концов отрезка передают функции, производится вычисление расстояния между точками. Единицы измерения расстояния могут быть произвольными.

```
double sqrt(); /* для использования определения */
double distance(from_x,from_y,from_z,to_x,to_y,to_z);
double      from_x;
double      from_y;
double      from_z;
double      to_x;
double      to_y;
double      to_z;
{
double      x;
double      y;
double      z;

x = to_x - from_x;
y = to_y - from_y;
z = to_z - from_z;
return sqrt (x*x + y*y + z*z);
}
```

Лист. 1.2. Функция определения длины отрезка

Дополнительные переменные в этом примере делают исходный текст программы более наглядным. В реальных ситуациях и в реальных программах локальные переменные x , y и z не применяются. Без них можно было бы написать весь текст функции в одной строке.

Повторим, использование простых переменных для запоминания координат точек не является чем-то неверным. Однако наступает момент, когда программа становится настолько перегружена переменными, содержащими координаты, что становится трудно уследить, какая переменная используется для какой координаты. Следующий шаг при организации программы - группировка координат x , y и z вместе.

По логике координаты точек концов отрезков представляют единое целое. Чтобы полностью идентифицировать конкретную точку, должны быть заданы все три координаты (в случае трехмерной системы координат). Итак, следующее усовершенствование - группировка значений координат в структуры данных типа записей. Такие структуры становятся исходными для развития наших структур данных. Вы можете быть уверены, что мы затронем лишь немного, но на этом будет строиться весь последующий материал книги. Начнем с простого определения координат. На лист. 1.3 дано определение типа нашей основной структуры данных для представления координат.

```
typedef
struct
{
    double      x;
    double      y;
    double      z;
}              coord;
```

Лист. 1.3. Основная структура данных для координат

Две точки определяют линию. А местоположение точки в пространстве определяется координатами X , Y и Z . Структура данных для задания линии в данном случае может быть структурой типа записи, в которой отрезок определяется по двум концевым точкам. На лист. 1.4 приведено простое, довольно грубое определение линии.

```
typedef
struct
{
    double      fx;      /* from - начало */
    double      fy;
    double      fz;
    double      tx;      /* to - конец */
```

```
double      ty;
double      tz;
}           line;
```

Лист. 1.4. Первая попытка организации данных для линии

Если задача нахождения независимых отрезков представляет для вас интерес, рассматриваемая структура данных может оказаться весьма полезной. На лист. 1.5 показано объявление массива, элементами которого являются структуры данных указанного типа. Массив может содержать большое число отрезков.

```
line        lines[200];
line        *next_line = lines;
```

Лист. 1.5. Структура данных для хранения значений нескольких отрезков

В процессе распознавания и сохранения в программе информации об отрезках при определении следующего свободного места в массиве, происходит увеличение значения счетчика. В данном примере счетчик представляет собой адресный указатель на очередной элемент массива, предназначенный для хранения отрезка. Значение адресного указателя также может увеличиваться. В качестве указателя используется переменная `next_line`. Начальным значением указателя является адрес начала массива отрезков.

При применении такой весьма грубой организации данных, поиск и нахождение отрезков, составляющих линию (цепочки отрезков, соединенных друг с другом), могут быть весьма продолжительным процессом. Основной причиной является то, что для нахождения смежных отрезков необходимо либо иметь дополнительные данные, либо ввести дополнительные шаги обработки. Требуется осуществлять поиск и находить в построенном списке отрезков соответствующие отрезки, которые имеют общую конечную точку. Зная координаты точки конца одного отрезка (отрезок задается двумя точками - `from` (начало) и `to` (конец)), необходимо искать в списке отрезков совпадающую с ней точку начала другого отрезка. В среднем при поиске надо просмотреть половину элементов списка. Поиск завершится, когда координаты начала отрезка совпадут с известными координатами конца отрезка. При таком совпадении вы будете знать, что отрезки являются смежными.

Однако, когда моделируется множество линий, состоящее из сотен и тысяч отрезков, существуют методы, позволяющие сокращать как требуемый объем памяти, так и время поиска.

Выберем в качестве основного критерия сокращение объема памяти. Задача поиска будет рассматриваться в разделе, посвященном двоичным деревьям. Во время обработки смежных отрезков ломаной

линии, при использовании уже рассмотренной простой структуры организации данных, программа сохраняет значения координат в структуре данных отрезка дважды. Один раз как координаты точки to - конца отрезка, второй раз как следующие координаты точки from - начала отрезка. Исключение составляют только точки начала и конца ломаной линии, координаты которых появляются только один раз. Если интерес представляет сокращение объема памяти, то не надо дублировать координаты; следует учесть, что для каждого значения координаты, представленного как число с плавающей точкой двойной точности, требуется 8 байт памяти. На рис. 1.4 представлена иллюстрация простого метода организации хранения координат.

<i>From (начало)</i>		<i>To (конец)</i>	
1.234	0.0	2.345	1.200
2.345	1.200	2.345	2.431
2.345	2.431	2.500	3.125
2.500	3.125	3.000	3.500
0.000	0.000	1.000	1.000
1.000	1.000	0.000	0.500
0.000	0.500	0.000	2.500

Рис. 1.4. Простой способ задания отрезков

Предположим, что для представления координат мы оставляем прежнюю структуру данных, а структура данных отрезка будет содержать просто ссылки или указатели на пару координат, определяющих отрезок. На рис. 1.5 показано небольшое улучшение структуры данных, позволившее перейти впоследствии к более эффективным методам хранения координат. На лист. 1.6 показано новое определение структуры данных для хранения отрезка.

Впоследствии я добавлю в эту структуру данных некоторую до сих пор неопределенную ссылку на структуру данных, содержащую информацию о слое. Поскольку в системе автоматизированного проектирования слои могут нести существенную информационную нагрузку, я мог бы ввести эту ссылку уже сейчас, чтобы показать, как легко добавляются дополнительные данные к структуре данных отрезка. Вы можете сами добавить ссылку, указывающую на тип линии, чтобы можно было различать сплошные, пунктирные или штриховые линии. Ширина линии могла бы быть еще одним пара-

метром, включенным в эту структуру. Конечно, то, какие именно данные должны быть включены в структуру данных, зависит от конкретного приложения.

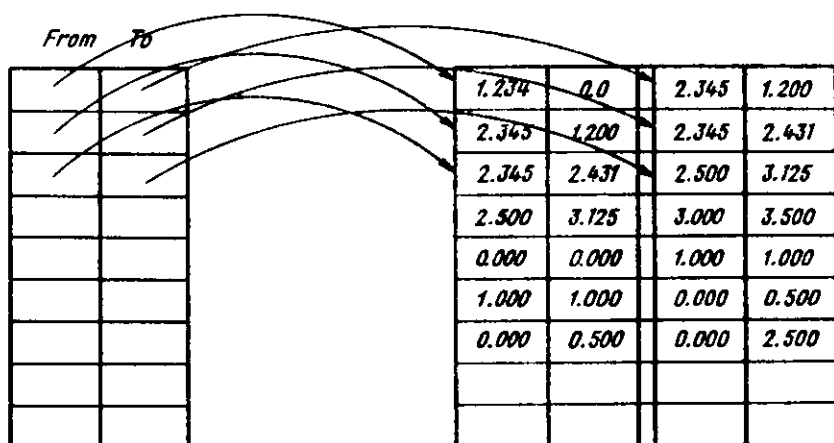


Рис. 1.5. Организация ссылок на точки концов отрезков

```

typedef
    struct          line;
    {
        coord      from;
        coord      to;
    };

line      lines[100];
int      num_lines = 0;
int      i;
double   fx;
double   fy;
double   tx;
double   ty;

for(i=0; i<num_lines; i++)
{
    fx = lines[i].from.x;
    fy = lines[i].from.y;
    tx = lines[i].to.x;
    ty = lines[i].to.y;
}
    
```

Лист. 1.6. Модифицированное определение линии

На рис. 1.6 показана компактная форма соотношения между отрезком и его координатами. Указатели на значение координат точки конца отрезка ссылаются на общую для двух отрезков пару координат. Это значительно уменьшает объем памяти, требуемой для хранения списка координат.

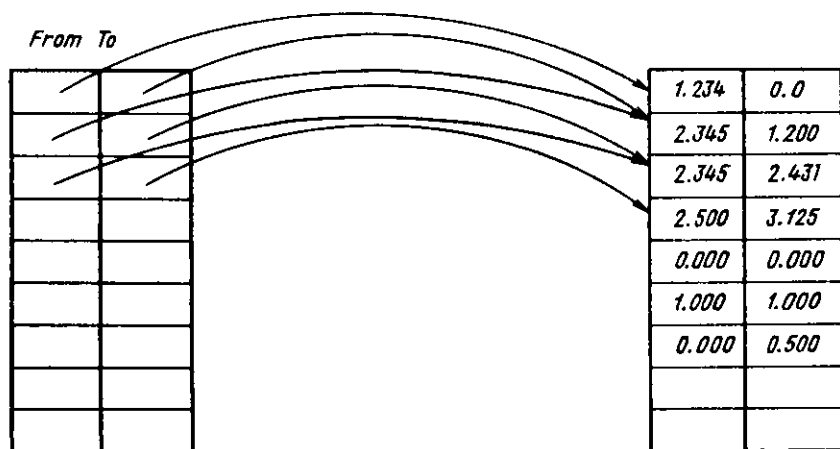


Рис. 1.6. Компактный способ организации ссылок

Можно было бы занять еще меньший объем памяти, если координаты концов отрезков хранились бы в массиве. Вместо использования адресных указателей в структуре данных отрезка, на каждый из которых отводится от 2 до 4 байтов (в зависимости от используемой модели памяти), можно ввести в структуру целое число (2 байта) - индекс координат в массиве значений координат отрезков. Применение индекса делает структуру более компактной. Тем не менее это требует дополнительных вычислений в программе для определения адреса нахождения координат в массиве при каждом обращении к ним, что может увеличить общее время вычислений и потребует большего объема памяти для инструкций преобразования. Вы не должны писать программный код для преобразований сами, он генерируется компилятором для каждого массива и используемого индекса. Применение рассмотренного метода зависит от того, насколько свободно вы себя чувствуете при работе с адресами и указателями. На лист. 1.7 показана реализация целочисленной индексации.

Я предпочитаю использовать указатели. Они дают возможность гибко организовывать ссылки на предопределенные данные, массивы, стеки и область динамической памяти, не скрывая одно от другого. С помощью единственной переменной указателя можно обращаться ко всем этим областям памяти.

```

coord        lines[100];
int          next_line = 0;
int          from_ref[100];
int          to_ref[100];
int          i;

for(i=0;i<num_line;i++)
{
    fx = lines[from_ref[i]].x;
    fy = lines[from_ref[i]].y;
    tx = lines[to_ref[i]].x;
    ty = lines[to_ref[i]].y;
}

```

Лист. 1.7. Представление линии с использованием индексации массива

Теперь давайте вернемся на шаг назад и рассмотрим организацию хранения координат и отрезков, построив набор небольших массивов данных, а также функции работы с отрезками. Это позволит нам понять существующие соотношения между данными. На лист. 1.8 представлена секция определения структур данных для рассматриваемой программы.

```

typedef
struct
{
    double      x;
    double      y;
    double      z;
}              coord;

typedef
struct
{
    coord      *from;
    coord      *to;
}              line;

line          lines[100];
line          *next_line = lines;
coord         points[170];
coord         *next_coord = points;

```

Лист. 1.8. Типичный набор определений данных

Дополнительные переменные `next_line` и `next_coord` предназначены для указания следующей свободной позиции для запоминания

новых отрезков и координат. Когда значение этих переменных достигает некоторого максимального, то это свидетельствует о том, что структура заполнена и в нее не могут быть включены какие-либо новые элементы.

Создание активного списка с использованием массивов отрезков и координат точек концов отрезков осуществляется с помощью функций, подобных `insert_line` (вставить линию). Я обычно не применяю такую функцию, поскольку она слишком проста для эффективной обработки сложных чертежей. Здесь просто иллюстрируется идея, как вообще может быть реализована подобная обработка. На лист. 1.9 показан исходный текст вспомогательной функции поиска координат - `find_coord`:

```
coord *find_coord(x,y,z)
double      x;
double      y;
double      z;
{
  coord      *c;
  /* * * * * * * * * * * * * * * * * * * * * */
  /* Функция находит и возвращает адрес структуры */
  /* данных, содержащей переданные значения */
  /* координат. Если значения координат не найдены, */
  /* то функция добавит эти значения в массив */
  /* точек POINTS. */
  /* * * * * * * * * * * * * * * * * * * * */

  c = points;
  while (c < next_coord) /* для всех имеющихся координат */
  {
    if ((c->x == x) && (c->y == y) && (c->z == z))
      return c; c++;
    /* эти координаты не совпали -- попробуем следующие */
  } /* координаты не найдены в таблице -- добавим их */
  c = next_coord++;
  /* присвоим значения */
  c->x = x;
  c->y = y;
  c->z = z;
  return c; /* возврат адреса добавленных значений */
}
```

Лист. 1.9. Функция поиска координат `find_coord`

Основная функция `insert_line` представлена на лист. 1.10. Я назвал эти функции основной и вспомогательной, поскольку в основ-

ной программе будет использоваться функция `insert_line`, а функция `find_coord` вызывается только из функции `insert_line`.

```

line *insert_line (fx,fy,fz,tx,ty,tz)
double      fx;
double      fy;
double      fz;
double      tx;
double      ty;
double      tz;
{
    line      *l;
    /* * * * * * */
    /* Функция добавляет новый отрезок. параметры */
    /* которого определяются аргументами функции. */
    /* Чтобы избежать дублирования координат, */
    /* применяется функция поиска координат find_coord */
    /* * * * * * */

    l = next_line++; /* получить следующий */
                    /* имеющийся отрезок */
    l->from = find_coord (fx,fy,fz);
    l->to = find_coord (tx,ty,tz);
    return l;
}

```

Лист. 1.10. Функция `insert_line`

Чтобы не загромождать текст программы, функции не содержат соответствующие контрольные проверки на достижение конца массивов. Такой контроль заключается в проверке значений переменных `next_coord` и `next_line` на то, больше ли их значения, чем адрес последнего элемента массива. Если это так, то программа выведет соответствующее сообщение и возвратит вызывающей программе специальное значение указателя `NULL`.

Для вставки отрезка ваша программа должна содержать вызов функции `insert_line`, точки концов отрезка передаются как параметры этой функции. Данная функция не должна сама осуществлять поиск на наличие в списке дублирующего отрезка. Такой поиск выполняется в функции `find_coord`, вызываемой из функции `insert_line`. В процессе поиска просматривается список координат и дублирующие координаты не будут вноситься в список. Вызов функции поиска стоит первым в тексте программы, так как это могло быть и в вашей программе. Таким, образом функции `insert_line` будет известно, что возвращается адрес группы координат, а не предполагаемое по умолчанию в языке Си целое число.

Одним из недостатков данной реализации является линейный поиск в таблице координат. Как было отмечено ранее, это весьма медленный метод обработки. В процессе поиска определяется, имеются ли уже указанные координаты в списке. Если координаты есть в списке, то нет необходимости вносить их в список повторно. Функция поиска возвращает адрес местонахождения заданных координат.

Очевидно, что, поскольку текст программы прост, маловероятно возникновение каких-либо логических ошибок во время работы программы. Это может явиться основным доводом в пользу применения данного метода поиска. Он может использоваться как быстрый способ проверки правильности замыслов разработчика. Тем не менее даже без привлечения математического аппарата для оценки времени поиска можно сказать, что это один из самых медленных из известных методов поиска. Все решения необходимо взвешивать. Если программа и структуры данных простые, то ускорение процесса обработки скорей всего будет незначительным. Более быстрая обработка базируется на применении более сложных структур данных.

Двоичные деревья

Для повышения скорости поиска координат мы будем использовать двоичное дерево. Обычно двоичное дерево изображается (рис. 1.7) как бы перевернутым кроной вниз (в отличие от настоящих натуральных деревьев). Верхний элемент, или узел, называется корнем. Он является основным исходным узлом для всего дерева.

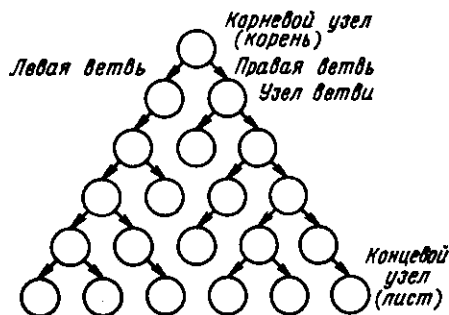


Рис. 1.7. Двоичное дерево

Ветви (или узлы ветвей) берут начало от корня и других ветвей. Ветви, которые берут начало от других ветвей, называются поддеревьями.

На концах ветвей находятся листья. Листьями называются узлы, не имеющие поддеревьев. Листья являются концевыми элементами дерева. Если присоединить к ним листья или ветви, то они также могут стать ветвями.

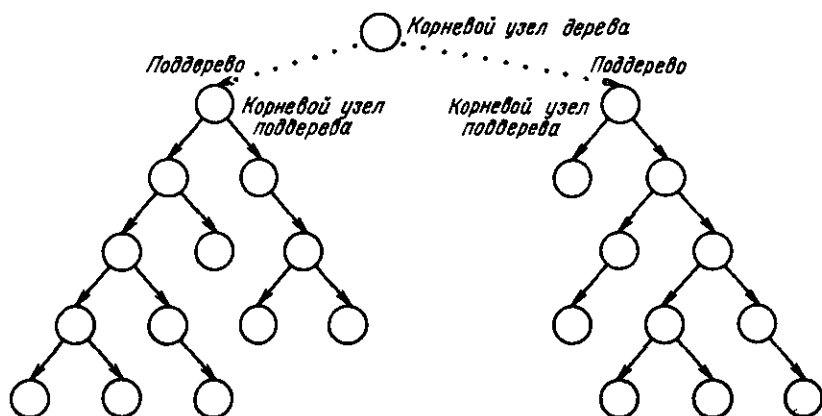


Рис. 1.8. Поддеревья в двоичном дереве

Отметим, что каждый узел может быть исходным только для двух поддеревьев. Отсюда происходит и название двоичные деревья. Обратимся к рис. 1.8: для каждого узла может быть одно поддерево слева (левая ветвь) и одно поддерево справа.

Это подходит для программы, чтобы построить дерево присоединяя узлы с правой и с левой стороны заданного узла. И мы подойдем к этому. Вы можете использовать любую по вашему желанию последовательность присоединения конкретных узлов. Однако обычным является присоединение узлов в зависимости от значений данных в узлах. Узлы с меньшим по величине значением присоединяются слева, а узлы с большим значением - справа.

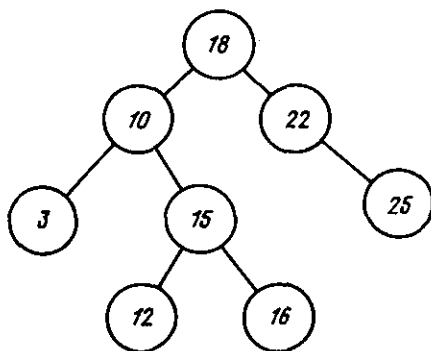


Рис. 1.9. Двоичное дерево для сортировки чисел

Двоичное дерево на рис. 1.9 построено для сортировки группы чисел. Порядок поступления чисел был следующим: 18, 22, 10, 15, 25,

3, 12, 16. Можете ли вы высказать соображение, почему дерево было построено именно так, как показано на рисунке?

Такое расположение чисел делает поиск чрезвычайно простым, здесь не требуется дополнительных комментариев. Проверьте работу следующего алгоритма, используя дерево, изображенное на рис. 1.9, для поиска числа 12. Начните от корневого узла.

При нахождении в узле искомого значения поиск завершается. Если искомое значение меньше, чем значение в данном узле, то поиск следует продолжить по левому поддереву, в противном случае - по правому поддереву. Поскольку каждый новый рассматриваемый узел является корневым для нового поддерева, вы оказываетесь в той же ситуации, что и прежде: имеется ли искомое значение в данном узле или необходимо выполнять поиск слева или справа от узла? Вы можете оформить эту повторяющуюся часть алгоритма либо в виде цикла, либо в виде рекурсивной программы. Но для программы такой поиск может быть не столь прост, как на словах.

Перед тем как перейти к рассмотрению примера программы построения двоичного дерева, включающего координаты отрезков, обновим структуру данных для координат. В структуру данных для координат будут внесены изменения. Мы добавим к ней адресные указатели на левое и правое поддерево, с тем чтобы программа могла находить нужные поддеревья. На лист. 1.11 приведено определение структуры данных, которая будет использоваться в примерах программ для работы с двоичными деревьями.

```
typedef
struct a
{
    double      x;
    double      y;
    double      z;
    struct a     *left;
    struct a     *right;
} coord;
```

Лист. 1.11. Структура данных для координат, адаптированная для двоичного дерева

В следующей группе листингов приведены исходные тексты вспомогательных подпрограмм для функции `find_coord`. На лист. 1.12 представлена функция `comp_coord` (сравнение координат двух точек). По заданным координатам двух точек (x , y и z) эта функция определяет, является ли первое множество значений координат точки меньше, равно или больше, чем второе. Функция возвращает одно из трех следующих значений в зависимости от результата сравнения: -1 , если координаты первой точки меньше, чем второй точки; 0 , если равны, и $+1$, если больше. Эта операция требует большего, чем простое сравнение двух чисел, поскольку имеется 6 значений,

сравниваемых в определенной последовательности, зависящей от предполагаемого упорядочивания координат x, y и z.

```
int comp_coord(x1,y1,z1,x2,y2,z2)
double      x1;
double      y1;
double      z1;
double      x2;
double      y2;
double      z2;
{
/* Функция сравнивает два множества координат      */
/* возвращаемый результат:                          */
/*          0 -- координаты равны                    */
/*          -1 -- первые < вторых                    */
/*          1 -- первые > вторых                    */
/* избавиться от простого -- сперва желаемый вариант */

    if ((x1 == x2) && (y1 == y2) && (z1 == z2))
        return 0;

    /* проконтролировать все возможные варианты. */
    /* что первые координаты меньше, чем вторые */

    if (x1 < x2) return -1;
    if ((x1 == x2) && (y1 < y2)) return -1;
    if ((x1 == x2) && (y1 == y2) && (z1 < z2))
        return -1;

    /* первые координаты должны быть больше, чем вторые */

    return 1;
}
```

Лист. 1.12. Функция сравнения координат comp_coord

Функция new_coord (лист. 1.13) предназначена для выполнения основных операций при создании нового узла двоичного дерева. Эта подпрограмма резервирует память для узла из области динамической памяти, инициализирует некоторые переменные и заносит координаты. Функция возвращает адрес нового узла. Другие функции служат для присоединения узла к дереву.

Если мы обратимся снова к уже рассмотренным функциям find_coord и insert_coord, то мы должны будем изменить только функцию find_coord.

```

coord *new_coord(x,y,z)
double      x;
double      y;
double      z;
{
    coord    c;
    /* Функция получает память для нового узла координат */
    /* из области динамической памяти, и за тем заполняет */
    /* поля структуры данных значениями координат.      */
    c = (coord *)malloc(sizeof(coord));
    c->x = x;
    c->y = y;
    c->z = z;
    c->left = NULL;
    c->right = NULL;
    return c;
}

```

Лист. 1.13. Функция new_coord

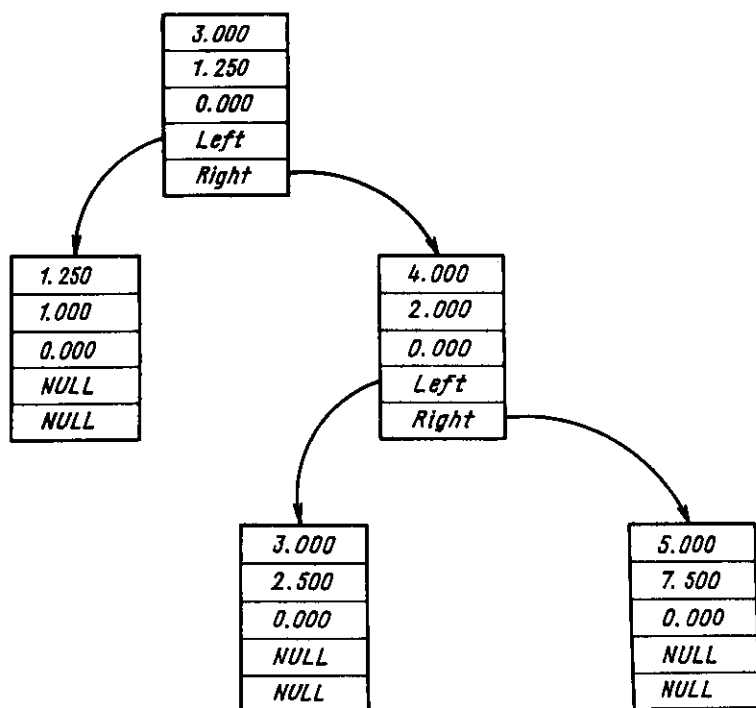


Рис. 1.10. Двоичное дерево, включающее значения координат

Функция `insert_coord` по-прежнему будет получать указатель на группу координат, и для нее безразлично, на что в действительности будет указывать указатель. На лист. 1.14 и 1.15 продемонстрированы два варианта реализации функции `find_coord`. По своему действию они идентичны, но в варианте на лист. 1.15 используются некоторые возможные сокращения текста программы, допускаемые в языке Си. Информации, представленной на лист. 1.11 - 1.13, по-моему вполне достаточно для самостоятельного разбора. В процессе разбора нового алгоритма мы получили новую структуру данных, см. рис. 1.10.

```
coord *find_coord(n,x,y,z)
coord      *n;
double     x;
double     y;
double     z;
{
/* Функция сравнивает координаты со значениями в          */
/* текущем узле дерева (см. примечания к comp_coord)      */
/* Если имеются поддеревья, поиск по дереву продолжается */
/* Поиск заканчивается, если найдено соответствие.        */
/* иначе добавляется узел с новыми координатами          */

switch(comp_coord(x,y,z,n->x,n->y,n->z))
{
case -1 : if (n->left) return find_coord
              (n->left,x,y,z);
          n->left = new_coord(x,y,z);
          return n->left;
case 0 : return n;
case 1 : if (n->right) return find_coord
              (n->right,x,y,z);
          n->right = new_coord(x,y,z);
          return n->right;
}
return NULL;
}
```

Лист. 1.14. Функция `find_coord` для двоичного дерева

```
coord *find_coord(n,x,y,z)
coord      *n;
double     x;
double     y;
double     z;
{
switch(comp_coord(x,y,z,n->x,n->y,n->z))
```

```

{
    case -1 : return ((n->left) ? find_coord(n->left,x,
        y,z) : (n->left = new_coord(x,y,z));
    case 0 : return n;
    case 1 : return ((n->right) ? find_coord(n->right,x,
        y,z) : (n->right = new_coord(x,y,z));
}
return NULL;
}

```

Лист. 1.15. Еще один вариант функции find_coord

Украшение двоичного дерева

Если мы развесим на нашем дереве отрезки (можете представить куда это нас приведет?), то украсим дерево.

До сих пор мы описывали отрезки как средство нахождения координат. Давайте подойдем к этому с другой стороны. Вообразим себе, что не точки привязываются к отрезкам, а отрезки к точкам. До сих пор мы говорили, что отрезок задается двумя точками его концов. Предположим, что мы все перевернули, и скажем, что точка относится к конкретному отрезку. Тогда, зная координаты точек, мы могли бы по ним находить соответствующий отрезок, а не наоборот - зная отрезок находить координаты его концевых точек.

Данный подход состоит в том, чтобы, имея одну точку, идентифицировать отрезки, которым принадлежит эта точка. Бесспорно, и это следует иметь в виду, - точка может быть образована пересечением нескольких линий.

Следующее, что нам нужно, - это изменить способ задания отрезков. Несомненно, отрезок должен идентифицировать его две концевые точки, но переосмысление этого может привести к новому подходу. Поскольку имеются две концевые точки, включенные в построение отрезка, то что нам может помешать для представления отрезка применить две отдельные структуры данных, в которых будут использоваться эти две точки концов отрезка? Теперь не остается других соображений кроме тех, как должны быть связаны между собой эти две структуры данных? Мы подходящим образом упаковали наши прежние определения отрезков и сделали явным отношение между точками концов отрезка. В определении отрезка появились ссылки на обе точки. Теперь я буду говорить о разделении определения отрезка с помощью двух указателей на координаты его точек.

Решение заключается в добавлении еще одного указателя-ссылки. (Хорошенькое дело! Этот парень должно быть любит указатели.) Мы добавим ссылку, которая скажет: "Вон там находится моя другая концевая точка". Рис. 1.11 иллюстрирует предлагаемый подход.

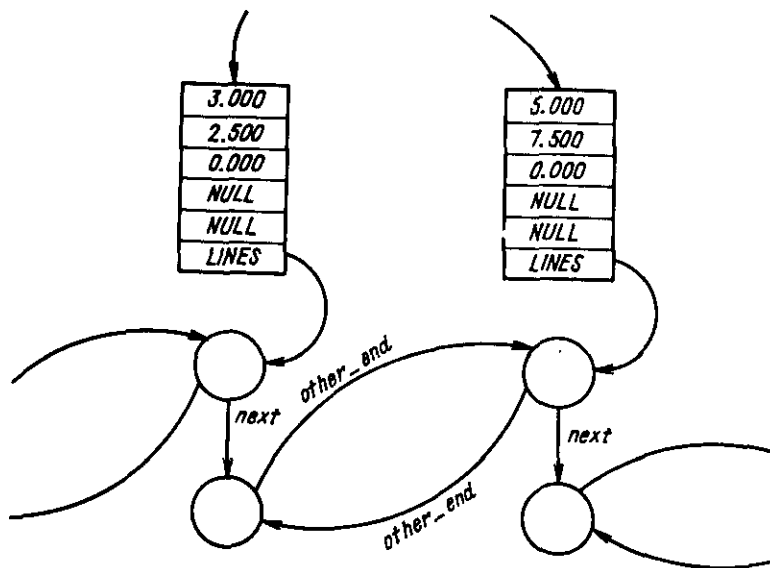


Рис. 1.11. Ссылки на концы отрезков в двоичном дереве

Если мы уж собрались изменить наш подход, чтобы позволить структуре данных для координат "указывать" на отрезки, то в структуре координат должна поддерживаться адресация этих отрезков. Для подключения отрезков к координатам данной точки строятся ссылки. На лист. 1.16 в структуру данных для представления координат добавлена ссылка на отрезки.

```
typedef
struct a
{
    double      x;
    double      y;
    double      z;
    struct a    *left; /* указатели на узлы */
    struct a    *right; /* двоичного дерева */
    line        *lines; /* отрезки с этими координатами */
} coord;
```

Лист. 1.16. Структура данных для координат, содержащая ссылки на отрезки

Как мы уже видели, структура данных для представления отрезков включает в себя большинство из требуемых значений. Тем не менее мы можем взглянуть по-новому на некоторые из них.

При перемещении по сети отрезков и координат, по мере обнаружения соединений и отношений, процесс обработки будет приводить нас от одного узла отрезка к другому. Если это так, то мы должны будем взглянуть на соответствующий координатный узел дерева, чтобы определить, где (*where*) мы находимся. На рис. 1.12 раскрывается данное соотношение.

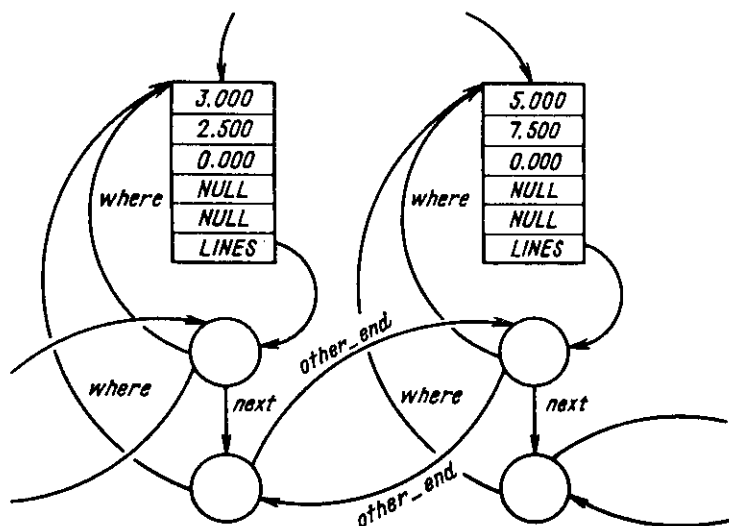


Рис. 1.12. Организация обратных ссылок на отрезки с использованием точек концов отрезков

Итак круг замкнулся. Координаты точек идентифицируют свои отрезки и, в свою очередь, отрезки идентифицируют точки своих концов. Поэтому мы можем добавить к структуре данных отрезка указатель, который скажет: "Это мои координаты". Как и раньше, этот указатель будет указателем на узел координат.

```

typedef
struct b
{
    struct b    *other_end;
    coord      *where;
    layer      *lyr;
    double     width;
    }          line;

```

Лист. 1.17. Разделенное определение отрезка

На лист. 1.17 проиллюстрированы детали этого нового определения.

Что можно сказать о пересечении нескольких отрезков в одной точке? Координатный узел содержит ссылку на отрезки, но такая ссылка только одна. Необходимо ли иметь несколько ссылок на отрезки (массив указателей) в одном координатном узле? Нет, одной ссылки достаточно, если мы связываем определения отрезков друг с другом. Было бы накладно иметь в координатном узле много неиспользованных ссылок на отрезки, потому что вы всегда должны иметь число ссылок на единицу большее, чем максимальное число отрезков, просто для уверенности! Отсутствие достаточного объема памяти, выделяемого для ссылок на отрезки, будет ограничивать возможности программы. Весьма накладно иметь большой объем памяти для ссылок. Но как мы можем знать, сколько памяти потребуется для работы? Мы этого не знаем, так как строим цепочку - связный список узлов отрезков. Это медленнее, чем иметь непосредственно доступные ссылки, организованные в виде массива, но это снимает проблему, связанную с попыткой угадать объем данных. Определение узла отрезка с добавленной ссылкой на цепочку приводится на лист. 1.18.

```
typedef
struct b
{
    struct b *next;          /* следующий в цепочке узел отрезка */
    struct b *other_end;     /* другой конец отрезка */
    coord    *where;         /* координаты, где мы "находимся" */
    layer     *lyr;
    double    width;
}
                    line;
```

Лист. 1.18. Связное определение отрезка

Функция `insert_line` (рис. 1.19) теперь изменяет свой вид. В нее включено совсем немного новых операций, что вызвано появлением новых структур данных. Одно из основных изменений состоит в том, чтобы получить из массива узлы определения отрезков и зарезервировать под них место в области динамической памяти. Это узлы, которые будут развешиваться на двоичном дереве координат.

```
void insert_line(fx,fy,fz,tx,ty,tz)
double      fx;
double      fy;
double      fz;
double      tx;
```

```

double      ty;
double      tz;
{
line        *l1;
line        *l2;

/* выделить память для двух узлов отрезков */
/* см. рис. 1.15а */
l1 = (line *) malloc(sizeof(line));
l2 = (line *) malloc(sizeof(line));
/* очистить поля, которые могут быть заполнены позже*/
/* см. рис. 1.15б */
l1->width = 0.0;
l1->lyr = NULL;
l2->width = 0.0;
l2->lyr = NULL;
/* занести координаты отрезков */
/* см. рис. 1.15в */
if (!root) root = new_coord(fx,fy,fz);
l1->where = find_coord (root,fx,fy,fz);
l2->where = find_coord (root,tx,ty,tz);
/* соединить концы отрезков */
/* см. рис. 1.15г */
l1->other_end = l2;
l2->other_end = l1;
/* присоединить узел к узлу координат */
/* см. рис. 1.15д */
l1->next = l1->where->lines;
l1->where->lines = l1;
l2->next = l2->where->lines;
l2->where->lines = l2;
}

```

Лист. 1.19. Связное определение отрезка

Рисунок украшенного дерева (рис. 1.13) показывает результаты работы рассматриваемого алгоритма на примере нескольких отрезков.

Итак, мы имеем украшенное двоичное дерево. Дерево приобрело ветви и листья с координатами, используемыми накопленными данными. Мы украсили дерево, развесив кусочки информации о внутренней структуре дерева. В конкретном случае мы украсили дерево отрезками, использующими координаты.

Даже Рождество приходит к концу, а дерево должно рухнуть. Это справедливо и для украшенных деревьев. Мы загрузили дерево координатами и отрезками. Теперь настало время извлечь из этого некоторые преимущества. Цель всего нашего упражнения - получить

последовательность соединения отрезков на основе списка отдельных несвязанных между собой координат отрезков.

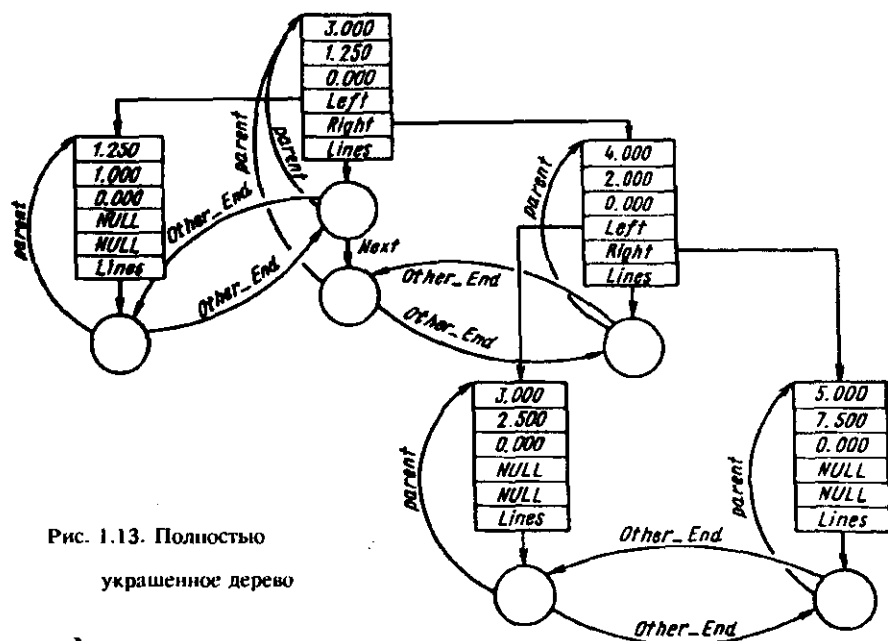


Рис. 1.13. Полностью
украшенное дерево

В последующей главе будут рассмотрены основные детали алгоритма обхода двоичного дерева и нахождения смежных отрезков. набросок алгоритма, используемого для обхода узлов двоичного дерева, приведен на лист. 1.20. Это рекурсивная программа, которая представляет собой реализацию описанного ранее алгоритма для нахождения узла на дереве. Повторим, если имеется поддереву слева от узла (поддереву с наименьшими значениями в узлах), то осуществляется обход всех узлов этого поддерева.

```
void process_coord(n)
coord      *n
{
    if (n->left) process_coord(n->left);
    /* выполните любые действия, которые вы собирались */
    /* осуществить при обработке узла */
    if (n->right) process_coord(n->right);
}
```

Лист. 1.20. набросок функции для обработки координат

После завершения обработки левого поддерева соответствующая обработка выполняется над текущим узлом. При ее завершении происходит переход к обработке правого поддерева. Отметим, что обработка начинается только при наличии поддерева на соответствующей стороне от узла.

Для любого заданного координатного узла двоичного дерева мы можем найти все отрезки, присоединенные к этому узлу. В функции, приведенной на лист. 1.21 мы видим, что "посещение" координатного узла (лист. 1.20) может включать получение информации о концевых точках отрезков, имеющих данные координаты. Этот процесс связан с обходом значений всех отрезков, связанных с данным координатным узлом. При обходе к определению отрезка добавляется признак посещения узла для того, чтобы можно было ответить, рассматривалась или нет какая-либо конкретная точка конца отрезка.

```
void visit_node(n)
coord      *n;
{
    line     *l;
    line     *o;

    l = n->lines;
    while (l)
    {
        if (l->visited == FALSE)
        {
            o = l->other_end;
            /* выполнение операций над отрезком */
            l->visited = TRUE;
        }
        l = l->next;
    }
}
```

Лист. 1.21. Обход отрезков, связанных с заданным координатным узлом

Организация хранения строк

Если в программе необходимо использовать несколько типов данных, то удобно отводить для хранения всех этих данных одинаковый объем памяти. Когда данные занимают одинаковый объем памяти (фиксированное число байт), вы можете лучше планировать размер структур данных. Поначалу это кажется простым.

Для персональных компьютеров (ПК) целые числа занимают 16 бит (32 бита для более сложных компьютеров), числа с плавающей

точкой - 32 бита (64 бита - для представления чисел с плавающей точкой с двойной точностью). Представление адреса зависит от используемой модели памяти, но тем не менее обычно для представления адресов типа NEAR (близкая адресация) используется 16 бит, а для адресов типа FAR (дальняя адресация) - 32 бита. Мы рассматриваем представление данных различных типов применительно к микропроцессорам семейства Intel 8086.

Если для представления значений требуется одинаковый объем памяти, вы можете знать, сколько памяти используется для каждого значения данных. Что же является наилучшим? Наилучшим являются строки. Возникает традиционный вопрос: "какова должна быть длина составной части строки?" Или в другой интерпретации: "сколько символов содержится в строке?" При расшифровке файлов обмена описаниями чертежей (DXF-файлы) системы AutoCAD мы будем обрабатывать имена разного сорта. Имеются имена слоев, имена блоков, значения атрибутов, текстовые строки и маркеры (tag labels).

Например, для полей внутренних имен, имен слоев и блоков система AutoCAD позволяет отводить до 32 символов. Для текста и значений атрибутов ограничений нет. Итак, если возможно подсчитать длину строки, изменяющуюся от нуля до большого числа символов, то как можно оценить объем памяти, резервируемый для символьной строки в наших структурах данных? Несомненно, нас не радует наличие значений, которые могут быть представлены в фиксированном объеме памяти.

Применяемый в системе AutoCAD подход к представлению внутренних имен (длина имен ограничивается 32 символами) заключается в том, что для них в конкретных структурах данных резервируется память для хранения 33 символов. Почему именно 33? Правильно, потому, что длина строки может быть от 0 до 32 символов. Должен существовать способ определения числа символов в строке. Это можно сделать одним из двух способов.

В языке Паскаль первый байт строки является байтом длины строки. Поскольку байт - это 8-битовое поле, то в его формате можно было бы указать длину строки от 0 до 255 символов. В строке, максимальная длина которой может достигать 32 символа, первый байт может содержать любое значение от 0 до 32, показывающее число символов, следующих после байта длины строки. Если для строки резервируется 32 байта, то наличие дополнительного байта длины строки увеличит это число до 33.

Однако система AutoCAD написана не на языке Паскаль, а на языке Си. Что само по себе вносит уже некоторую специфику. В языке Си не подсчитывается число символов, содержащихся в строке, вместо этого для обозначения конца строки применяется специальный символ - признак конца строки. Символ, используемый как признак конца строки, является уникальным среди символов кода ASCII - это символ NULL ... ничего, двоичный ноль.

Для 32-символьной строки должно быть зарезервировано место для признака конца строки символа NULL, что в сумме дает 33 байта. Отметим, что в языке Си длина строки по сути не ограничивается, в отличие от строк в языке Паскаль (255 байтов). Однако существует физическое ограничение, определяемое длиной сегмента памяти для компьютеров, построенных на базе микропроцессоров семейства Intel 8086 - 64 Кбайта.

Следует соблюдать осторожность при резервировании фиксированной области памяти для строк. Если вы сами нарушите вами же установленные границы, пытаясь поместить строку длиной 40 символов в 32-байтовое поле, вы будете нести ответственность за переполнение других значений данных, заплатив за это полностью неработоспособной программой или, по крайней мере, ненадежной.

С другой стороны, я обращаюсь к вам с вопросом. Как часто вы используете 32-символьные имена для слоев или блоков? Чаще всего выбирают простые и короткие имена. Я использую имена, такие, как

PARTS, SIGNALS, L1, L2, DRILL.

Эти имена стали внутренними именами системы AutoCAD, и объем зарезервированной для них памяти является постоянным. Что произойдет с дополнительным пространством памяти в системе AutoCAD, т.е. с памятью, способной вместить 20 или более символов, оставшейся не использованной для каждого из заданных мной имен? Ничего! Это пространство будет просто потеряно. Однако если это приемлемо для системы AutoCAD, то это может быть приемлемо и для нас. Согласны? Если да, то вы можете пропустить оставшуюся часть этого раздела.

Остались со мной? Хорошо. Помните, ранее отмечалось, что для представления адресов отводится одинаковый объем памяти? Адреса могут занимать 16 или 32 бита, в зависимости от выбранной вами модели памяти. Я исхожу из той схемы, что все символьные строки хранятся в некотором укромном месте, а работа с ними осуществляется с помощью перемещения адресных указателей, вместо того чтобы каждый раз брать всю строку целиком.

Для представления адресов всегда требуется один и тот же объем памяти независимо от длины строки. Поэтому весьма просто включить их в сложную структуру данных и не беспокоиться о неиспользованной, устраивающей еще кого-либо памяти.

Если вы имеете адрес строки, вы просто производите выборку символа, расположенного по этому адресу, и продолжаете выполнять выборку остальных символов до тех пор, пока не встретится признак конца строки символ NULL.

В языке Си указатель на символ или на символьную строку объявляется следующим образом:

```
char *name_of_pointer;
```

Область памяти, в которой в действительности будут находиться символы, объявляется как массив символов:

```
char    name_of_string[81];
```

Указатель содержит адрес одного символа. Это необязательно должен быть первый символ строки или массива. Указатель может содержать адрес и 126-го символа.

Применение языка Си окажет нам помощь, поскольку в нем можно использовать строки и указатели. Если вы в программе пишете имя массива и указываете соответствующий конкретный элемент массива, как, например, `airplane[4]`, то в языке Си это обозначает использование значения данного элемента массива. Однако когда вы указываете только имя массива или строки, что безразлично, то в языке Си это подразумевает использование адреса массива. Представляете, как удобно получать адрес области памяти автоматически, только называя эту область.

Указав в программе имя `airplane[4]`, можно получить значение 5-го элемента (0-1-2-3-4) (видите, что это пятый элемент?) массива `airplane`. Но использование обозначения `airplane` в результате даст адрес этого массива. Кстати, это применимо только при работе с массивами. Если же вы, например, хотите получить адрес целочисленной переменной с именем `bike`, то вам следует использовать операцию извлечения адреса, обозначаемую с помощью символа `&`, что при записи будет выглядеть как `&bike`.

Предположим, что имеется область памяти с именем `words`, в которой для хранения символов зарезервировано 1024 байта. Тогда адрес области `words` может быть очень легко присвоен адресному указателю `next_word`. Присвоение значения (адреса массива `words`) осуществляется с помощью указания имени массива.

```
char    words[1024];  
char    *next_word = words;
```

Отметим, что переменной `next_word`, для того чтобы она действительно стала указателем на символы, должен быть присвоен адрес массива `words`. Переменная `next_word` предназначена для указания местонахождения следующего доступного символа или строки в массиве `words`. Если при выполнении программы требуется добавить новое слово в массив `words`, переменная `next_word` укажет, в какое место должно быть помещено это слово. В начале работы программы в массиве не содержится никаких слов, следовательно, переменная `next_word` будет содержать ссылку на начало массива.

"Что ж, все так хорошо и замечательно, - вероятно скажете вы, - но как я буду помещать слова в массив `words`?" На что получите банальный ответ: "Вы будете копировать имеющиеся строки в массив".

В языке Си имеется стандартная функция для копирования строк - `strcpy`. Когда этой функции передаются адреса двух строк (исходной и строки результата), она производит копирование символов исходной строки в строку, начинающуюся с указанного адреса. Для передачи строки, содержащейся в `airplane` в массив `words`, в программу могла быть вставлена следующая строчка:

```
strcpy(next_word,airplane);
```

Для функции `strcpy` требуются два адреса. Первый адрес является адресом результата. В качестве значения первого адреса мы используем адрес следующей доступной для занесения строки области массива `words`, этот адрес содержит переменная `next_word`. С помощью этой переменной адрес будет передан функции `strcpy`. Вторым адресом является адрес исходной строки. В данном примере при вызове функции указано имя исходной строки. Поскольку использовались только имена массивов, функции будет передан адрес массива `airplane`.

Функция `strcpy` осуществляет копирование символа из исходной строки в строку результата, после чего увеличивает значения адресных указателей, и так повторяется до тех пор, пока не будет обнаружен символ `NULL`.

Стандартная функция языка Си возвращает адрес строки результата, что позволяет выполнять копирование строк в цикле, добавлять строки одну к другой или сравнивать их между собой. Все стандартные библиотеки подпрограмм языка Си включают функцию `strcpy`, поэтому нет необходимости вставлять исходный текст этой функции в текст какой-либо из ваших программ.

```
char  *strcpy(d,s)
char  *d;
char  *s;
{
char  *t;
    t = d;      /* сохранить, как возвращаемый адрес */
    while (d++ = *s++); /* копирование строки */
    /* последним символом строки является символ NULL */
    return t;
}
```

Лист. 1.22. Функция копирования строки `strcpy`

Мы только что скопировали слово в следующую доступную для размещения слова область памяти, а это значит, что больше нет доступного места для какого-либо слова. Где же находится очередное доступное место для занесения нового слова? Если бы я находился в учебном классе, то выдержал бы здесь паузу, чтобы услышать ответ

от класса. Но иногда такая пауза может растянуться на несколько минут. Так как я не уверен, что смогу повторить паузу такой длительности в книге (прочитайте в это время несколько страниц, например из романа "Война и мир"), и мы продолжим.

Следующая доступная область памяти находится сразу после только что скопированного слова. Нам необходимо поместить в переменную `next_word` адрес следующего после новой строки байта. Новый адрес может быть равен значению старого адреса, содержавшегося в `next_word`, плюс длина добавленной новой строки и плюс один байт для признака конца строки символа `NULL`.

Кроме того, в стандартной библиотеке подпрограмм языка Си имеется функция определения длины строки - `strlen`. В качестве входного параметра функции `strlen` требуется адрес строки, функция возвращает число символов в строке без учета признака конца строки символа `NULL`.

Если добавляемая строка имеет имя `s`, то новое значение указателя `next_word` может быть вычислено следующим образом:

```
next_word += strlen(s) + 1;
```

На лист. 1.23 показано, что все это не страшно, когда организовано как отдельная функция `add_str`.

Функция `add_str` дает нам способ занесения символьных строк в центральную область памяти. В следующей главе мы рассмотрим извлечение строк из таблицы строк с помощью функции определения местонахождения строки.

```
char      words[2000];
char      *next_word = words;

char      *add_str(s)
char      *s;
{
char      *t;
    t = next_word;
    strcpy(next_word,s);
    next_word += strlen(s) + 1;
    return t;
}
```

Лист. 1.23. Функция `add_str`

Таблица имен

Можно предположить, что в сложных программах, таких, как система AutoCAD, входные и выходные файлы содержат множество данных разнообразных типов. Например, графические данные в формате DXF-файла представляются как числа с плавающей точкой (координаты, границы чертежа и ширина линий), целые числа (число вхождений и число шагов повторения), текст (значения атрибутов, текст и имена таблиц) и логические значения (флажки).

Если этот поток данных обслуживается систематически, то было бы замечательно составить таблицу, в которой бы содержалась вся необходимая информация для распознавания поступающих данных и для организации соответствующей их обработки. Такая таблица известна под названием таблицы имен. Она содержит всю необходимую информацию для обработки символьных значений.

Вы можете создать такую таблицу. Каждый элемент таблицы содержит информацию о поступающем значении. Минимальной необходимой для вас информацией является следующая: имя переменной; тип данных переменной; адрес, по которому данные могут быть занесены или получены.

Первым полем в записи таблицы является поле имени переменной. В DXF-файлах системы AutoCAD имя переменной заголовка, имя таблицы или тип графического объекта появляются перед соответствующим значением. Если мы будем осуществлять просмотр этих имен и искать соответствие для них в нашей таблице, то большая часть работы будет сделана. Чтобы избежать в таблице противоречий, связанных с внесением имен в поля имен таблицы, мы поместим образец правописания имени в некотором укромном месте (подобном таблице строк или области строковых констант), а в таблицу занесем адрес этой строки. Обнаружение имен в файлах системы AutoCAD и в этой таблице является предметом отдельного обсуждения и в другой главе.

Как уже отмечалось, данные, которые мы намереемся получить, перемешаны. Некоторые значения представляют собой целые числа, другие - числа с плавающей точкой (действительные числа). Второе поле таблицы указывает, какой тип данных должен использоваться для переменной. При обычном программировании мы объявляем переменные как переменные некоторого типа (int - целые, double - действительные двойной точности или char - символьные). Обозначения, применяемые при объявлении типа, обычно зарезервированы для использования компилятором. Мы не можем применять такие же обозначения, как при программировании. Поэтому, для указания типа данных в нашей таблице имен мы должны будем ввести некоторые наши собственные типы данных. В табл. 1.1 приведены некоторые допустимые имена и соответствующие им обозначения типов в языке Си. Отметим, что в этом списке присутствуют некоторые спе-

цифические имена. Как вы увидите, такая специфика добавится при последующей обработке.

Таблица 1.1. Типы данных в таблице имен

Типы в таблице имен	Типы в языке Си
INTEGER	int
CHARACTER	char
STRING	char *
REAL	double
COORD	double(x) double(y) double(z)
ON_OFF	----

Нет необходимости хранить наименование типа данных полностью. Требуется только сохранять некоторое кодовое число, которое будет обозначать некоторый тип данных. Поставив в соответствие символическому наименованию типа данных некоторое число, вы сможете работать с именами типов данных (как человек вы предпочтете работать с символьными обозначениями), а компьютеру предоставить работу с числовыми значениями.

Я предпочитаю держать все мои определения для программ на языке Си в одном файле. Этот файл называется `defines.h`. Кроме него, существует еще файл `types.h`. Используя директиву препроцессора `#include`, я могу подключать данный файл определений при компиляции всех моих программ.

Мы можем аккуратно закодировать имена наших типов данных с помощью директивы `#define`, как это показано на лист. 1.24. Добавьте эти определения в ваш файл `defines.h`.

```
#define INTEGER      1
#define REAL         2
#define CHARACTER    3
#define STRING       4
#define ON_OFF       5
#define COORD        6
```

Лист. 1.24. Определение типов данных для таблицы имен

Теперь об обработке значений переменных. Вспомните, одно из моих поучений состоит в том, что все типы указателей занимают одинаковый объем памяти. Дышите глубже! Больше указателей. Значения, на которые они указывают, могут занимать различный объем памяти, но указатели занимают один и тот же. Поэтому мы

можем построить таблицу из однородных элементов, которые будут ссылаться на неоднородные данные.

На лист. 1.25 приведена структура записи таблицы имен. Незвизрая на то, что некоторые значения не являются целочисленными, переменная — адресный указатель — объявляется как указатель на целое число. Для переключения между различными типами данных мы будем использовать операцию приведения типа.

```
typedef
struct
{
    char      *var_name;
    int       type_code;
    int       *var_addr;
}            sym;
```

Лист. 1.25. Определение типов для таблицы имен

Инициализация таблицы представляет довольно интересную часть работы. Данная таблица должна быть инициализирована перед началом обработки. Мы будем использовать значения в таблице для управления работой программы. На лист. 1.26 приведен весьма причудливый набор данных для инициализации таблицы. Адреса переменных должны быть приведены к целому типу `int`, для того чтобы компилятор не выдавал предупреждающие сообщения. Если в процессе обработки встретится переменная с именем ЯБЛОКО, то, используя данную таблицу, мы узнаем, что это переменная целого типа и связанное с ней значение должно быть присвоено переменной `apple`.

```
int      apple;
double   orange;
char     pear;
char     tomato[256];
sym      sym_table[] =
        {{"ЯБЛОКО", INTEGER, &apple},
         {"АПЕЛЬСИН", REAL, (int *)&orange},
         {"ГРУША", CHARACTER, (int *)&pear},
         {"ТОМАТ", STRING, (int *)&tomato},
         {NULL, 0, NULL}};
```

Лист. 1.26. Инициализация таблицы имен

Заключение

В данной главе были рассмотрены некоторые элементарные структуры данных и алгоритмы, нашедшие отражение в книге. Эти структуры данных встретятся и в следующей главе, так как они комбинируются с другими структурами данных и дают общее представление о формировании более сложных конструкций.

Глава 2

ДРУГИЕ МЕТОДЫ ПРОГРАММИРОВАНИЯ

Меня часто забавляет тот факт, что мы создаем чрезвычайно сложные программы из простых примитивных элементов. Мы имеем пакеты программ, такие, как различного рода системы автоматизированного проектирования, электронные таблицы, экспертные системы. А кирпичиками, из которых строятся эти программы, являются простые инструкции, такие, как инструкции сложения, вычитания, сдвига, пересылки и др. Конечно, комбинирование этих простых инструкций миллионами способов позволяет создавать сложные операции. Операции реализуются с помощью алгоритмов; алгоритмы воплощаются в процедуры и функции. Пласты и наложения функций становятся программами.

Цель данной главы - продолжить наш обзор методов программирования, применяемых в книге. Я предполагаю, что вы можете уже быть знакомы с рассматриваемым здесь материалом. Если это так, то пропустите эту часть и перейдите к разделу, посвященному программированию для систем автоматизированного проектирования.

В главе будут рассматриваться следующие вопросы: распознавание слов; трансляция слов во внутреннее представление; распознавание последовательностей.

В данной главе будет представлено много аппаратно-зависимых текстов программ. Текст программ написан на языке Си, для компиляции применялся компилятор Microsoft C версии 5.0 (фирма Microsoft), работающий в среде операционной системы MS-DOS 3.1 на компьютерах типа IBM PC или MS-DOS, совместимых компьютерах. Неплохой заменой могут быть компиляторы Quick C фирмы Microsoft или Turbo C фирмы Borland. Если предполагается использовать приведенные программы на рабочих станциях типа Sun или Apollo или на компьютерах серии MicroVAX, то тогда могут потребоваться некоторые изменения.

Основными факторами, оказывающими влияние на разработку программ, являются следующие:

- получение информации от некоторого источника;
клавиатура, коммуникационный порт или файл;
- обработка данных;
- вывод преобразованной информации по указанному назначению.

При разработке программ я обнаружил, что ввод данных и их преобразование при вводе существенно труднее, чем вывод значений уже известных данных. Суть или значение данных могут настолько зависеть от контекста, что программа при анализе объекта должна полностью знать все данные, составляющие его окружение.

Если в процессе сбора поступающих входных данных мы распознаем, что они несут какую-то смысловую нагрузку, то мы можем задавать операции, которые должна выполнять программа. Действия программы и их состояния могут дать нам некоторое понимание о контексте данных. Иначе говоря, если программе известно, на каком

этапе обработки она находится, то должна существовать некоторая специфическая цепочка данных, которая привела программу в это состояние. На основе данных и контекста мы можем принимать некоторые осмысленные решения. В этой главе объясняется, какие средства имеются для этого.

Когда автоматическое средство (еще одна программа или блок аппаратуры) выполняет обработку данных, обычно легче найти некоторую логическую связь (или распознаваемые образцы), чем когда эти данные вводятся человеком. Человек может быть так непредсказуем. К счастью, мы будем иметь дело с машинно-генерируемыми данными. Плохо же то, что данные, которые формируют вывод, вероятно, тоже созданы некоторым человеком. Это означает, что образцы данных будут просты для распознавания, но их структура может быть столь неоднозначна, что вы ничего не сможете сделать даже с известными вам данными.

Распознавание слов

Первым ключом к чтению графической информации системы AutoCAD является распознавание присутствующих в ней различных слов и признаков. Получить одиночное слово не очень трудно, это может быть выполнено с помощью функции `fscanf` языка Си. Функция `fscanf` вызывает функцию преобразования для замены последовательности символов кода ASCII числами с плавающей точкой (действительными числами), целыми числами или данными других типов. Поскольку сейчас нас интересуют только необработанные строки (без преобразования), то для ввода данных я выбрал функцию `fgets`, она отличается тем, что не обращается к функциям преобразования. Другие программы, как было отмечено, поддерживают преобразование.

Первый шаг при распознавании слов несложен - ввести слово, для чего используются стандартные функции моего любимого языка программирования Си.

На лист. 2.1 представлен исходный текст функции `fgets`. Эта функция включается в большинство библиотек при реализации языка Си, поэтому нет необходимости использовать данный листинг. Но, если вы захотите модифицировать функцию ввода согласно вашим требованиям, этот текст программы может пригодиться. Функция `fgets` считывает символы в символьную строку до тех пор, пока не встретится признак конца строки. Признаком конца строки может быть либо символ возврат каретки ('\r'), либо символ перевода строки ('\n'), что зависит от конкретной реализации. Теперь надо осознать, что же мы имеем. Существует ряд подходов к решению поставленной задачи.

Первый подход - это весьма примитивное решение. Метод состоит в том, что осуществляется вызов функции ввода, в результате чего слово помещается в некоторую переменную и далее все операции программы выполняются с этой переменной. Сравнивая значение этой переменной с различными вариантами выбора, вы определяете,

что надо делать. На лист. 2.2 представлен фрагмент программы, иллюстрирующий этот примитивный метод распознавания слов.

```
int fgets(f,s)
FILE      *f;
char      *s;
{
    while (TRUE)
    {
        c = getc(f);
        if ((c == '\n') || (c == '\r')) return;
        *s++ = c;
        *s = '\0';
    }
}
```

Лист. 2.1. Функция fgets

```
fgets(f,s);
if (strcmpi(s,"START") == 0)
    start_process();
else
    if (strcmpi(s,"SEQUENCE") == 0)
        sequence_process();
    else
        printf("Встречена неизвестная команда : %s\n",s);
```

Лист. 2.2. Примитивный набросок программы распознавания слов

Рассматриваемый метод связан с очень длительным процессом линейного поиска. Такой поиск занимает много времени и памяти, требуемой для хранения имен образцов и выполнения сравнений. Этот метод не освобождает вас от применения линейного поиска в вашей программе. Но иногда такое примитивное решение блестяще подходит для простых ситуаций.

Помимо прочего, каскад управляющих конструкций типа if-then-else, появляющийся при использовании такого подхода, весьма не легко сопровождать и модифицировать. Если вы будете делать очень большие отступы при оформлении блоков конструкции if-then-else, то довольно быстро сможете превысить ширину листа бумаги.

Избежать многочисленных повторений if-then-else можно с помощью управляющей конструкции типа вариант, реализуемой в языке Си операторами switch и case. Используемое в операторе switch дискретное значение (целые числа имеют дискретные, а числа с плавающей точкой — непрерывные значения) служит для выбора

одного из выполняемых блоков операторов, как это показано на лист. 2.3:

```
switch(value)
{
    case 0 : /* выполняется, если value == 0 */
        break;
    case 1 : /* выполняется, если value == 1 */
        break;

    .... и т.д. ....
}
```

Лист. 2.3. Управляющая конструкция switch в программе

Значение, указываемое в операторе switch, должно быть одиночным дискретным значением. Одиночный символ подходит в качестве управляющего в операторе switch, но вводимые строки могут состоять из нескольких символов и могут не являться дискретными типами данных. Например, строка в команде TEXT системы AutoCAD может быть очень длинной. Тем не менее для переключения процесса обработки могут потребоваться именно такие строки. Что же предпринять в данной ситуации?

Стандартный прием - это поставить в соответствие строке какое-либо дискретное значение. Поскольку такое дискретное значение представляет значение другого объекта (строки), мы назовем это значение лексемой (token). Как автобусный талон является эквивалентом денег, которые требуется заплатить за проезд на автобусе, так лексема - это простое дискретное представление слова или фразы, которые имеют для нас какое-либо значение.

Одно из основных преимуществ лексем состоит в том, что все они занимают один и тот же объем памяти. Хорошо, но автобусный билет имеет не тот размер, как лексема. Автомат для продажи талонов принимает только один тип монет или банкнот. Создатели автомата не предполагали осуществлять распознавание монет различного размера или определять достоинство банкнот. Вы используете билеты одного размера для совершения поездки. Аналогично все лексемы имеют один и тот же "размер" (целое число). И хотя длина строк может изменяться от 1 до 32 или более символов, лексемы, представляющие эти строки, одинаковы по "размеру". Легче и быстрее передать двухбайтовое целое число, чем каждый раз передавать целую строку через всю программу.

Для генерации лексем обратимся снова к примитивному методу. Начнем с просмотра всех известных программе слов или, иначе, слов, которые предположительно могут быть введены. При обнаружении одного из таких слов программой будет сгенерировано целое

число. Это целое число становится значением нашей лексемы. В основном для таких операций применяется уже рассматривавшаяся последовательность конструкций типа if-then-else, образующих как бы отдельную замкнутую ветвь программы. Мы увидим, что преобразование ключевого слова в целое число, используемое для его представления, будет выполняться в нескольких местах программы. Разумно было бы выделить эти операции в отдельную подпрограмму, на лист. 2.4 показано, как могла бы выглядеть такая подпрограмма.

```
#define FIRSTWORD      1
#define SECONDWORD     2
#define LASTWORD       3

int generate_token(s)
char      *s;
{
    if (strcmpi(s,"FIRSTWORD") == 0) return FIRSTWORD;
    if (strcmpi(s,"SECONDWORD") == 0) return SECONDWORD;
    if (strcmpi(s,"LASTWORD") == 0) return LASTWORD;
    return 0;
}
```

Лист. 2.4. набросок программы генерации лексем

Как уже говорилось, такое примитивное решение не отличается элегантностью и эффективностью. Но с другой стороны, если список слов постоянен и не очень велик, то, по крайней мере, вы будете знать, что программа будет работать и не нужно волноваться.

Давайте сделаем шаг в сторону повышения эффективности. Избавимся от последовательных вызовов функции `strcmpi` в теле программы, поместив вызов этой функции в цикл. Сперва занесем список слов в массив, который мог бы использоваться для нахождения слов в словаре, см. лист. 2.5.

```
char      *word_list[12] = {"FIRSTWORD",
                           "SECONDWORD",
                           "LASTWORD"};
```

Лист. 2.5. Фрагмент программы генерации лексем

Для разбора функции `strcmpi` или просто ради любопытства проведите следующий эксперимент. Напишите небольшую программу (лист. 2.6), в которой функции `strcmp` и `strcmpi` применяются для обработки одной и той же строки. Если в строке присутствует знак подчеркивания, вы можете получить некоторые поразительные результаты. Если полученные результаты совпадают, то вы можете поменять местами функции сравнения двух строк. Остерегайтесь применения этих функций одновременно в одной программе, если резуль-

таты отличаются (как показывает наш тест). Теперь мы можем сравнивать строки в цикле (лист. 2.7).

Повторим, функция возвращает значение 0, если соответствие не найдено. Поскольку в языке Си первому элементу массива соответствует индекс 0, то для компенсации этого, чтобы всегда в случае найденного соответствия возвращаемое значение отличалось от 0, в функции к текущему индексу прибавляется единица.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

main()
{
    int    result_1;
    int    result_2;

    result_1 = strcmp("ABCDEFGH","ABC_EFG");
    result_2 = strcmpi("ABCDEFGH","ABC_EFG");
    printf("Результат работы strcmp -- %d\n",result_1);
    printf("Результат работы strcmpi -- %d\n",result_2);
}
```

Лист. 2.6. Маленькая занимательная программа

```
int generate_token(s);
char    *s;
{
    int    i;

    for(i=0;i<12;i++)
        if (strcmpi(s.word_list[i])!= 0) return i+1;
    return 0;
}
```

Лист. 2.7. Улучшенный вариант генерации лексем

Программа нормально работает, если известен размер массива word_list и он является статическим массивом. Однако если вы решите расширить ваш список слов, то вам потребуется изменить диапазон индексов массива, а также верхний предел для переменной цикла i. Это можно быстро сделать с помощью директивы #define, определив в ней значение индекса массива. На лист. 2.8 приведен пример такого использования независимой верхней границы массива.


```
#define WORD_LIST_SIZE 12
```

```
char *word_list[WORD_LIST_SIZE] = { ..
```

```
for(i=0; i<WORD_LIST_SIZE; i++)
```

Лист. 2.8. Централизованное задание верхней границы

К счастью, существуют простые способы перехода к использованию расширяемых структур, позволяющие не беспокоиться о сохранении корректности всех значений.

Для начала мы исключим значение диапазона индекса, указанное в массиве `word_list`. Если размер массива не задан явно, то компилятор языка Си определит его автоматически, так, чтобы в массив вместились все слова, используемые при его инициализации (лист. 2.9). Выражаясь точнее, слова по сути сами являются массивами символов, а в массиве `word_list` хранятся только указатели (адреса) на эти слова - строки символов.

```
char *word_list[] = {"FIRSTWORD",  
                     "SECONDWORD",  
                     "LASTWORD"};
```

Лист. 2.9. Массив с неявно заданным размером

Теперь мы не знаем действительный размер массива, что на самом деле для нас не имеет никакого значения. Для проверки на соответствие скорее важно знать, сколько слов имеется в массиве. Но поскольку мы остановились именно на нахождении соответствия, то общее число слов тоже не так важно. Вместо подсчета числа слов в списке почему бы нам не пометить конец списка каким-либо специальным символом, который привлечет наше внимание? Чем-либо таким, что означает НИЧЕГО. Вспомним, что массив `word_list` является списком указателей на строки (определить, где находятся эти строки, - задача компилятора). Поместим значение указателя `NULL` в список, состоящий из нескольких указателей, имеющих любые значения, отличные от значения `NULL`. На лист. 2.10 показана программа — генератор лексем, в которой используется список слов, завершаемый пустой ссылкой `NULL`.

Присвоим адрес первого элемента списка указателю `p` (переменная `p` является указателем на строку в вашем списке слов). После чего с помощью адресной ссылки можно передать функции сравнения строк (`strcmp`) содержимое элемента массива для проверки на равенство. При неравенстве программа перемещает указатель на позицию следующего слова в списке. Можно упростить работу с указателем, если использовать целочисленную переменную `i` для задания смещения относительно указателя. Обнаружение пустого элемента массива (ссылки `NULL`) означает, что поиск должен быть завершен, поскольку он оказался безрезультатным.

```

char    *word_list[] = {"FIRSTWORD", "SECONDWORD",
                        "LASTWORD",
                        NULL};

int generate_token(s)
char    *s;
{
char    *p;
int     i;

    i = 0;
    p = word_list;
    while(*(p+i))
    {
        if (strcmpi(s,*(p+i)) == 0) return i+1;
        i++;
    }
    return 0;
}

```

Лист. 2.10. Список слов, завершаемый символом NULL, и программа генерации лексем

Теперь мы можем расширить наш список слов, не боясь забыть об изменении всех нужных значений. Для этого надо просто вставить новое слово в конец списка перед признаком конца списка NULL и перекомпилировать основную программу. Такая обработка подходит только для статических списков. Все слова должны находиться в списке до начала выполнения программы.

До сих пор мы имели дело со статическими списками. Предположительно это почти все, что будет использоваться в наших прикладных программах автоматизированного проектирования. Тем не менее остались еще некоторые вопросы. Например, что будет, если мы захотим пополнить наш список слов новыми словами, добавляемыми из входного потока данных. В качестве примеров можно привести имена LAYER, BLOCK или STYLE, которые должны распознаваться в различных приложениях. Нам нужен способ расширения списка слов для включения в него этих новых имен.

Для выполнения такой процедуры надо совершить следующие действия:

1. Найти место, в которое надо поместить новое имя.
2. Добавить новое имя в список слов.
3. Не дублировать одно и то же имя в списке слов.

Лист. 2.11 служит иллюстрацией выбранной нами стратегии.

Первая часть алгоритма - найти слово, уже была рассмотрена. Мы будем продолжать с того места алгоритма, где обычно нам воз

вращалось значение 0, в случае появления неизвестного программе имени.

Сначала нам потребуется место для занесения вновь найденного слова. Для этого мы создадим таблицу строк. Таблица строк представляет собой большой символьный массив, в котором мы будем хранить имена. Только накопленный опыт поможет вам решить, сколько памяти должно резервироваться под эту таблицу. Я склонен выделять пространство для хранения от 4000 до 8000 символов. Эти цифры получены на основе моего опыта разработки компиляторов.

```
Найти слово в списке слов - массив word_list и
возвратить соответствующее слову значение
Если слово не найдено
    добавить слово в таблицу строк
    поместить указатель в список слов word_list
    присвоить (по индексу массива) целочисленное значение
    возвратить присвоенное значение
```

Лист. 2.11. Алгоритм распознавания слов

Кроме того, нам потребуется сохранить указатель на этот массив, что поможет определить местонахождения следующего слова. Этим указателем является значение, помещенное в список слов, позднее это значение поможет нам найти образец написания слова. На лист. 2.12 представлен пример объявления таблицы имен. С объявлениями такого рода мы уже знакомы. В заключение применим на практике некоторые методы, рассмотренные в последней главе.

```
char    string_table[4000];
char    *next_word = string_table;
```

Лист. 2.12. Объявление таблицы имен

Ввиду того что в пустой таблице очередным или следующим словом является первое слово, мы инициализируем указатель на следующее слово, присваивая ему адрес начала таблицы строк.

Для того чтобы занести новую строку в таблицу строк, надо скопировать ее на следующее свободное в таблице место и изменить значение указателя на следующее слово так, чтобы он был подготовлен для выполнения следующей операции вставки. На лист. 2.13 показано, как просто это делается.

```
strcpy(next_word, new_word);
next_word += strlen(new_word)+1;
```

Лист. 2.13. Копирование строки в таблицу имен

Прибавление дополнительной единицы к значению, возвращаемому функцией `strlen`, вводит поправку на наличие неучтенного в конце строки байта `NULL` - признака конца строки. Мы по-прежнему должны помнить, что необходимо сохранить значение указателя `next_word` перед его увеличением, в результате чего указатель будет содержать ссылку на следующее свободное место. Теперь перейдем к списку слов.

Когда мы в последний раз занимались списком слов, мы устранили значения диапазона индексов для того, чтобы размер массива вычислялся сам автоматически исходя из объема памяти, требуемого для хранения инициализирующих его значений. Это не оставляло места для расширения списка! Возвратимся к той точке, где мы должны либо заранее определить размер массива, либо добавить в массив ряд значений `NULL`, чтобы позволить его расширение. Если вы укажете размер массива, то компилятор заполнит неиспользуемую часть массива значениями `NULL`. В другом случае вы будете вынуждены сами генерировать последовательность значений `NULL`, чтобы сохранить место для указателей на новые слова.

```
char    string_table[4000];
char    *next_word = string_table;
char    *word_list[] = {"FIRSTWORD", "SECONDWORD",
                        "LASTWORD",
                        NULL,
                        NULL,
                        NULL,
                        NULL};

char    **next_list;

int generate_token(s)
char    *s;
{
    char    *p;
    int     i;

    i = 0;
    p = word_list;
    while(*(p+i))
    {
        if (strcmpi(s,*(p+i)) == 0) return i+1;
        i++;
    }
    /* не найдено -- должно быть новое слово */
    /* переставить указатель на свободную позицию */
    *(p+i) = next_word;
    /* сохранить новое слово в таблице строк */
    strcpy(next_word,s);
```

```

next_word += strlen(s) + 1;
/* вернуть новое число */
return i+1;
}

```

Лист. 2.14. Объединенные вместе данные и функции

Пока при добавлении указателей к концу допустимого списка вы не выйдете за последний пустой элемент `NULL`, то в списке будет присутствовать завершающий список элемент `NULL` и алгоритм поиска будет по-прежнему правильно работать. Лист. 2.14 иллюстрирует вышесказанное.

Даже хотя мы и добавляем слова в таблицу строк, нет необходимости заносить оригинальные слова в таблицу строк. В массиве `word_list` содержатся указатели на символы и этого вполне достаточно для указания строк, находящихся в таблице, или строк, которые компилятор для удобства спрятал. Указатель на символы и есть указатель на символы.

Все это хорошо до тех пор, пока вы не поддерживаете большой список имен и не можете добавлять неопределенное число слов. Вы на самом деле познали ненависть к ускользающему концу массива `word_list`, когда вы добавляете на одно имя больше, чем дозволено. И вы в равной степени познали отвращение к объявлению нереально больших массивов, чтобы избежать переполнения. Это пожирает память быстрее, чем быстро растущая электронная таблица.

Давайте отойдем от статического управления памятью и рассмотрим использование динамической памяти. Динамическая память - эта такая память, которую вы можете запрашивать и использовать по мере необходимости. Приходится, конечно, "расплачиваться" за ее использование, но она позволяет вам использовать только объем памяти, необходимый для специальной цели, избегая резервирования очень большого или очень маленького объема памяти. Вы запрашиваете динамическую память, когда она вам нужна.

Функции языка Си `malloc` (выделить память) и `calloc` (очистить выделенную память) являются основными функциями для работы с динамической памятью. Вы сообщаете этим программам, сколько памяти вы хотите иметь, и они возвращают указатель на область выделенной памяти. Вы должны сохранить этот указатель (адрес области памяти) в доступном месте. Если вы этого не сделаете, то можете случайно освободить память в пределах этой области во время работы программы. Все происходящее можно сравнить с судном, которое сорвалось со своих якорей и дрейфует как угодно. Без экстраординарных средств вы не сможете вернуть этот корабль обратно. Нам требуется как бы привязать к якорю выделенную область динамической памяти, чтобы знать ее местонахождение, потянув якорь.

Только что мы собирались спросить, для чего нужна динамическая память? Очевидная вещь - иметь достаточно памяти для хранения указателя на символьную строку. Символьная строка представ-

ляет собой имя, которое мы нашли. Указатель - это адрес строки в таблице имен.

Кроме того, не следует уделять большое внимание, чтобы привязать каждый модуль памяти в массиве элементов (хотя мы и нуждаемся в такой привязке), поскольку это именно то, от чего мы старались уйти. Пока мы привязываем один модуль памяти к известному якорю, могут быть найдены все другие модули. Мы строим цепочку модулей. Каждый модуль содержит связующую ссылку на другой. Таким образом может быть построена и привязана к единственной переменной цепочка модулей памяти. На лист. 2.15 показаны типичные структуры данных, содержащие связующие ссылки, а также поясняется их роль при генерации лексем.

```
typedef
struct a
{
    struct a    *next;    /* ссылка на следующее слово */
    char        *spelling; /* адрес слова */
} element;
element        *anchor;
element        *tail;

int generate_token(s)
char          *s;
{
    element    *p;
    int        i;
    i = 0;
    p = anchor;
    while (p)
    {
        if (!strcmp(s,p->spelling)) return i;
        p=p->next;
        i++;
    }
    p = (element *)calloc(1,sizeof(element));
    if (!anchor) anchor = p;
    if (tail) tail->next = p;
    tail = p;
    p->spelling = next_word;
    strcpy(next_word,s);
    next_word += strlen(s)+1;
    return i;
}
```

Лист. 2.15. Связный список указателей на слова

Переменная `tail` содержит текущую ссылку на конец цепочки. Это позволит вам добавить новые модули к концу списка. Тогда, когда вы продвигаетесь по списку, используя целочисленное значение `i`, нумерация модулей остается постоянной.

Существует способ избавиться от потребности иметь указатель конца списка. Это (еще раз!) означает большие накладные расходы, но облегчает выполнение вставок в список. Если вы решили добавить новые модули в начало списка, постоянство нумерации будет потеряно при второй вставке в список. Вы можете обойти эту проблему, сохраняя номер лексемы в модуле с указателем на слово. В этой ситуации не имеет значение, каким извилистым может стать список, в нем сохраняется идентичная слову лексема. На лист. 2.16 показано добавление идентификатора лексемы в список элементов.

```
typedef
struct a
{
    struct a    *next;
    char        *spelling;
    int         id;
}              element;

int            next_word_value = 1;

int generate_token(s)
char          *s;
{
    element    *p;

    p = anchor;
    while(p)
    {
        if(!strcmp(s,p->spelling)) return p->id;
        p=p->next;
    }
    p=(element *)calloc(1,sizeof(element));
    p->next = anchor;
    anchor = p;
    p->id = next_word_value++;
    p->spelling = next_word;
    strcpy(next_word,s);
    next_word += strlen(s)+1;
    return p->id;
}
```

Лист. 2.16. Добавление номера лексемы

Вы могли бы уничтожить все сохраняемые имена, идентификационные номера и связанные списки, если бы существовал способ прямо преобразовывать имя в уникальный номер. Такой метод называется хешированием (далее перемешивание) - как блюдо типа мясоовощной смеси. Оригинальные символьные данные "отбиваются", "шинкуются", "нарезаются" в форме кубиков и переупорядочиваются, чтобы сформировать число. При использовании одной последовательности действий одно и то же имя должно всегда в результате давать одно и то же число. Ненадежное и изменчивое свойство этого процесса должно давать уникальные и предсказуемые числа для каждого вводимого слова. Как-то не верится, что это найдет практическое применение. Используя обычные способы, очень легко для двух вариантов ввода сгенерировать одни и те же числа, что является конфликтом.

```

element      *hash_table[64];

int generate_token(s)
char          *s;
{
    element      *p;
    int          h;

    p = hash_table[h=hash(s)];
    while(p)
    {
        if (!strcmp(s,p->spelling)) return p->id;
        p=p->next;
    }
    p=(element *)calloc(1,sizeof(element));
    p->next_table[h];
    hash_table[h] = p;
    p->id = next_word_value++;
    p->spelling = next_word;
    strcpy(next_word,s);
    next_word += strlen(s)+1;
    return p->id;
}

```

Лист. 2.17. Применение перемешивания для нахождения списка слов

Нельзя отрицать, что для достаточно большой выборки слов мы можем осуществить быстро и просто некоторое перемешивание и сгенерировать "класс" числовых слов. Например, в совокупности из 1000 слов мы можем перемешать эти слова и образовать 128 групп или классов в среднем по 8 слов в группе. Или мы могли бы сгенерировать 64 группы из 16 слов или любые другие группы, которые мы захотим. Что это означает для нас? Если мы можем перемешать слова, которые нам встретились, мы сравниваем это слово только с

несколькими другими словами одного класса вместо того, чтобы искать соответствие среди нескольких сотен слов.

Я надеюсь, что вы уже уловили идею. Если вы желаете что-нибудь облегчить в одном месте, то значит, где-то существует расплата за что-либо другое. Стоимость перемешивания слов для формирования классов из слов складывается из:

- 1) подпрограммы перемешивания;
- 2) отдельного списка и ссылок для каждого класса перемешивания.

Однако чем больше классов перемешивания используется, тем меньше требуется сравнений.

В основном мы применяем методы, уже рассмотренные ранее, за исключением того, что мы берем ссылку из массива ссылок в соответствии со значением числа перемешивания. На лист. 2.17 приведен текст программы для перемешивания слова, чтобы найти соответствующий список слов.

Отметим, что диапазон индексов в массиве `hash_table` равен 64. Я предпочитаю строить таблицы перемешивания с числом элементов, кратным степени числа 2, для упрощения генерации кода перемешивания. Рекомендуется выбирать такие числа, как 32, 64, 128, 256.

Функция перемешивания должна генерировать ключи - числа в диапазоне 0 — 63, чтобы выполнить правильную индексацию массива из 64 элементов. Для таблицы из 128 элементов вам потребуются значения 0 — 127.

Сразу приходят на ум два способа сделать это. Первый, если разделить любое положительное число на 64, то можно вычислить остаток от деления, имеющий значения 0 — 63. Реализовать это можно весьма просто, используя операцию языка Си - деление по модулю (%). Второй способ не требует выполнения деления, если выбраны числа, являющиеся степенями числа 2. Двоичным представлением числа 64 является 0010 0000, а для числа 63 - 0001 1111. Если мы выполним операцию логического умножения любого целого числа с числом 63, результатом будет число из диапазона 0 — 63. Поскольку младшие 5 бит содержат единицы, операция логического умножения сохраняет любые значения в указанном диапазоне и всегда очищает любые старшие биты. Для выполнения такой простой операции я и выбрал числа, кратные степени числа 2, для определения размера таблицы перемешивания.

Как мы получаем числа для перемешивания? Все символы кода ASCII представляются двоичными значениями.

```
int hash(s)
char *s;
{
    int i;

    i = 0;
```

```

        while(*s) i = (i + (*s++)) & 63;
        return i;
    }

```

Лист. 2.18. Типичная функция перемешивания

Обычно я складываю двоичные значения символов в строке и выполняю над результатом логическое умножение таким образом, что всегда получаю числа в заданном диапазоне. На лист. 2.18 представлена типичная функция перемешивания. Теперь мы имеем все части системы распознавания слов и кодирования. На лист. 2.19 представлен окончательный вариант этих программ.

```

    typedef
    struct          token
    {
        struct token  *next;
        char          *spell;
    }                token;

    token            tok[8000];
    short            next_token = 0;
    token            *hash_table[64];
    char             name_table[32000];
    char             *next_word = name_table;

    int hash(s)
    char  *s;
    {
        int    j;

        j = 0;
        while (*s) j = (j + *s++) & 63;
        return j;
    }

    char  *locate(s)
    char  *s;
    {
        char  *c;
        token *p;
        int    h;

        if (s==NULL || *s == '\0') return NULL;

        if (p = hash_table[hash(s)])
            while (p)
            {
                if (p->spell == s) return s;

```

```

        if (strcmp (s,p->spell) == 0) return p->spell;
        p = p->next;
    }
    h = hash(s);
    p = &tok[next_token++];
    p->next = hash_table[h];
    hash_table[h] = p;
    p->spell = next_word;
    strcpy(p->spell,s);
    next_word += strlen(s) + 1;
    return p->spell;
}

```

Лист. 2.19. Окончательный вариант функции locate()

Трансляция во внутреннее представление

Не секрет, что программы автоматизированного проектирования используют не только символьные строки для представления своих внутренних данных. Однако с точки зрения считывания входных файлов с информацией в коде ASCII все является символьными строками. Большая часть данной книги посвящена считыванию DXF-файлов системы AutoCAD, которые содержат исключительно информацию в коде ASCII. В данном разделе затрагиваются вопросы систематического преобразования строк символов в коде ASCII в соответствующее внутреннее представление данных.

Вся входная информация считывается представленными в книге программами как строки символов в коде ASCII. Поскольку рассматриваются разнообразные наборы данных самых различных типов, то проще сначала ввести информацию в программу, а за тем отсортировать, какие данные нужны и как они должны представляться внутри программы.

В предыдущем параграфе для ввода строк в программу применялась функция fgets или ее эквивалент. Функция fscanf имеет возможность преобразовывать строки символов в коде ASCII в требуемые типы данных при условии, что мы знаем, какие это должны быть типы данных. Однако в произвольной точке ввода мы можем не знать ввода какого типа данных следует ожидать.

В последующих главах мы узнаем, какие данные содержатся в DXF-файлах. Одной из основных секций файлов типа DXF является секция заголовка HEADER. Эта секция содержит символьные строки, действительные и целые значения. При чтении файла мы захотим получить наиболее важные части данных по мере их обнаружения. Порядок, в котором они могут находиться в файле, не обязательно известен. Присмы обнаружения специальных значений в файле рассматриваются позднее в этой главе и в гл. 6. Сейчас мы

хотим взглянуть на способы хранения символьной информации в коде ASCII, в соответствующем внутреннем формате представления.

DXF-файлах могут находиться следующие данные: \$EXTMIN, \$EXTMAX, \$MENU и \$ORTHOMODE. \$EXTMIN - действительные числа, представляющие минимальные координаты X и Y области чертежа, занимаемой элементами чертежа (область изображения). \$EXTMAX - также действительные числа, представляющие максимальные координаты X и Y области изображения. \$MENU представляет собой символьную строку - имя текущего меню чертежа. \$ORTHOMODE - это целое число, представляющее текущее значение параметра ORTHO. \$PLINEWID - действительное число, определяющее установленную по умолчанию ширину полилинии. Если бы эти значения должны были появиться вместе в DXF-файле, то они появились бы в том порядке, как показано на лист. 2.20. Точное значение чисел на данный момент несущественно.

```

          9
$EXTMIN
          10
          1.00
          20
          1.00
          9
$EXTMAX
          10
          12.00
          20
          15.50
          9
$MENU
          2
ACAD
          9
$ORTHOMODE
          70
          1
          9
$PLINEWID
          40
          0.0125
```

Лист. 2.20. Пример фрагмента DXF-файла

То, что я хочу сделать с этими числами, - это присвоить их каким-либо переменным для дальнейшего использования в программе. Для этого я должен буду создать некоторые переменные, которые будут содержать данные значения. Эти переменные должны иметь тип данных, соответствующий запоминаемым в них данным. Числа,

расположенные между именами и значениями в распечатке DXF-файла, являются маркерами (tag). Они сообщают, какой тип данных следует за ними. Вообще говоря имеется множество нюансов, которые будут обсуждаться в дальнейшем. Маркеры 10 и 20 помечают действительные числа. Маркер 2 используется для имени. Маркер 70 помечает целочисленные значения. Для работы с этими данными мы создаем переменные. На лист. 2.21 даны определения переменных.

Следующим для каждой переменной является то, сколько байтов она занимает. Отметим (еще раз), что все эти переменные занимают разный объем памяти. Минимум забот, когда вы имеете дело с одинаковыми переменными. Тем не менее нашим следующим шагом будет построение небольшой таблицы, в которой будет указываться местонахождение DXF-информации по мере ее нахождения. Каждый элемент таблицы должен иметь один и тот же размер, поскольку мы не будем вносить в таблицу сами значения.

```
double      extmin_x;      /* 8 байт */
double      extmin_y;      /* 8 байт */
double      extmax_x;      /* 8 байт */
double      extmax_y;      /* 8 байт */
char        *menu;         /* 4 байта */
int          ortho;        /* 2 байта */
double      plinewid;      /* 8 байт */
```

Лист. 2.21. Пример объявления переменных

Не требуется слишком большого воображения, чтобы понять, что в таблице будут находиться указатели на различные переменные. Указатели занимают одинаковое место в памяти и позволяют нам сохранять ясными структуры данных в таблице.

Наша таблица преобразований будет иметь следующие поля: имя элемента DXF-файла; тип элемента DXF-файла; указатель на область памяти.

Как уже говорилось, используются три типа данных - действительные числа, целые и символьные строки. Однако существует одна небольшая проблема. Значение области изображения чертежа представляют два действительных числа, заданные в виде пары координат. Мы назовем этот тип данных COORD вместо REAL, что будет сигнализировать программе об использовании таблицы для хранения двух значений вместо обычных переменных. На лист. 2.22 показана часть таблицы преобразований.

```
struct
{
    char      *name;
    int       data_type;
    void      *mem_location;
} convert_table;
```

```

double      *tmp_real;
double      *tmp_x;
double      *tmp_y;
int         *tmp_i;
char        *tmp_s;
char        *y;
int         t;
convert_table *ctp;

convert_table *ct = {{"$EXTMIN" , COORD, &extmin_x},
                     {NULL      , COORD, &extmin_y},
                     {"$EXTMAX" , COORD, &extmax_x},
                     {NULL      , COORD, &extmax_y},
                     {"$PLINEWID", REAL , &plinewid},
                     {"$MENU"   , STRING, &menu},
                     {"$ORTHOMODE", INTEGER,
                                     &orthomode},
                     {NULL      , 0      , 0}};

```

Лист. 2.22. Таблица преобразования типов

Каждому значению в DXF-файле предшествует маркер. Маркер сообщает о том, данные какого типа следуют за ним. С помощью этой дополнительной информации мы можем задать для строки символов в коде ASCII корректное программное преобразование. Когда преобразование будет сделано, мы используем указатель в таблице преобразований, чтобы сохранить значение для дальнейшего использования. На лист 2.23 приведен полный алгоритм разбора формата DXF-файлов с использованием таблицы преобразований.

Цикл

Получить значение маркера из DXF-файла

Получить следующее соответствующее маркеру значение

Если маркер "9"

Найти запись в таблице преобразований, используя имя поля

Если в поле data_type таблицы преобразований находится значение INTEGER

Присвоить вспомогательному указателю на целое - значение mem_location

Если в поле data_type таблицы преобразований находится значение STRING

Присвоить вспомогательному указателю на строку - значение mem_location

Если в поле data_type таблицы преобразований находится значение REAL

Присвоить вспомогательному указателю на действительное - значение mem_location

Если в поле `data_type` таблицы преобразований находится значение `COORD`

Присвоить вспомогательному указателю на координату X - значение `mem_location`

Присвоить вспомогательному указателю на координату Y - адрес следующей ячейки памяти после `mem_location`

Если маркер "2"

Найти и запомнить (если требуется) значение в таблице строк (с помощью функции `locate()`) и сохранить адрес строки по адресу, ссылка на который содержится во вспомогательном указателе на строку

Очистить значение указателя

Если маркер "10"

Преобразовать значение в действительное число (с помощью функции `atof()`) и сохранить его по адресу, ссылка на который содержится во вспомогательном указателе на координату X

Очистить значение указателя

Если маркер "20"

Преобразовать значение в действительное число (с помощью функции `atof()`) и сохранить его по адресу, ссылка на который содержится во вспомогательном указателе на координату Y

Очистить значение указателя

Если маркер "40"

Преобразовать значение в действительное число (с помощью функции `atof()`) и сохранить его по адресу, ссылка на который содержится во вспомогательном указателе на действительное

Очистить значение указателя

Если маркер "70"

Преобразовать значение в целое число (с помощью

функции `atoi()`)

и сохранить его по адресу, ссылка на который содержится во вспомогательном указателе на целое

Очистить значение указателя

Лист. 2.23. Алгоритм простой программы разбора формата DXF-файла

Теперь, когда мы имеем соображения о том, в каком направлении нам двигаться, взглянем на лист. 2.24, который содержит текст программы, реализующей наш алгоритм. Как можно увидеть, программа проще, чем могло показаться, взглянув на запись ее алгоритма.

```

while (TRUE)
{
    t = get_tag(dxf_file);
    v = get_value(dxf_file);
    switch(t)
    {
        case 9 : if ((ctp = find_name_in_ct(v)) != NULL)
                    switch (ctp->data_type)
                    {
                        case INTEGER : tmp_i =
                                        ctp->mem_location;
                                        break;
                        case STRING  : tmp_s =
                                        ctp->mem_location;
                                        break;
                        case REAL    : tmp_r =
                                        ctp->mem_location;
                                        break;
                        case COORD   : tmp_x =
                                        ctp->mem_location;
                                        tmp_y =
                                        (ctp+1)->mem_location;
                                        break;
                    }
                    break;
        case 2 : *tmp_s = locate(v); tmp_s = NULL;
                    break;
        case 10 : *tmp_x = atof(v);
                    tmp_x = NULL;
                    break;
        case 20 : *tmp_y = atof(v);
                    tmp_y = NULL;
                    break;
        case 40 : *tmp_r = atof(v);
                    tmp_r = NULL;
                    break;
        case 70 : *tmp_i = atoi(v);
                    tmp_i = NULL;
                    break;
    }
}

```

Лист. 2.24. Реализация программы разбора формата DXF-файла

Чтобы компиляция текста программы завершилась соответствующим образом, при присвоении значения указателю в секции "9" для согласования типов данных должна быть выполнена операция приве-

дения типа. В этот раз ничего не будет сказано о новых неизвестных нам функциях. Они будут обсуждаться в последующих главах.

Цель данного упражнения - образовать таблицу, которая давала бы информацию о преобразовании строк символов в коде ASCII к соответствующему типу данных. Программа имеет открытую структуру, что позволяет вводить в нее новые типы данных и распознаваемые имена.

Программная реализация конечных автоматов

Это некоторые рассуждения о состояниях. Под термином "состояние" я понимаю условие, при котором что-то существует. Я понимаю, что Калифорния является штатом, и при некоторых обстоятельствах состоянем, в котором находитесь вы сами, но это отдельный вопрос. Речь идет о логическом состоянии программы.

Рассмотрим конторский стол. Стол, у которого имеются выдвижные ящики. Одним состоянием стола является ситуация, когда все ящики закрыты, другим состоянием, когда открыт один из двух ящиков (ящик А). Еще одно состояние, когда открыт другой ящик (ящик Б). Может быть и еще состояние, когда открыты оба ящика. Это маленький стол. Чтобы провести аналогию, предположим, что со столом в один момент времени может произойти только одно событие. Мы собираемся открывать и закрывать ящики, и я хочу, чтобы была недопустима ситуация, когда два ящика открываются одновременно. Согласны?

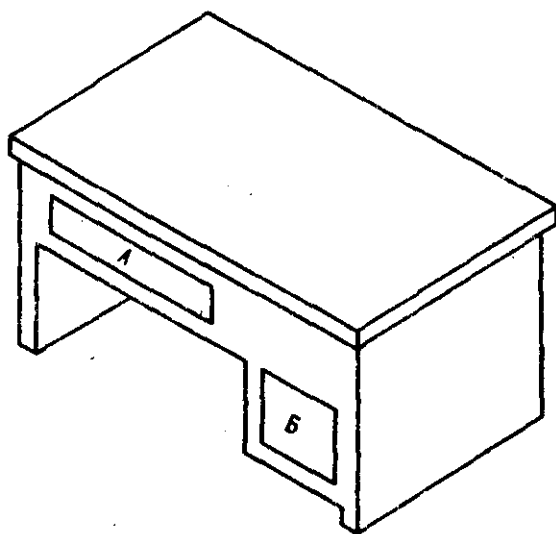


Рис. 2.1. Стол

Начнем с первого состояния - ящики закрыты. На рис. 2.1 показано это состояние. Теперь произошло какое-то событие, которое изменило состояние стола. Хорошо! Дядюшка Ван Лайнз переслал стол в Нью-Йорк. Неверно! Вы открыли один из ящиков. Какой из ящиков? (Я не знаю ... третий в тумбе.) Не имеет значения; состояние изменилось. Наше дело - отобразить изменение состояния. Диаграмма показывает известные состояния и предполагаемые события. Состояния изображаются кружками. В наших примерах, как и в программе, состояния являются статическими. Если программа находится в заданном состоянии, только некоторые события могут изменить

состояние программы. Стрелки показывают направление изменения состояния. Стрелки представляют события. Чтобы изобразить диаграмму системы, нам необходимо знать, какие альтернативные варианты возможны и важны в нашей модели. В данном примере нас касается ситуация, когда ящик открывается или закрывается. Если кто-то выбьет ножки стола, это не имеет для нас значения до тех пор, пока сила удара при сотрясении не откроет ящик или пока новый наклон стола не оставит ящики в задвинутом положении.

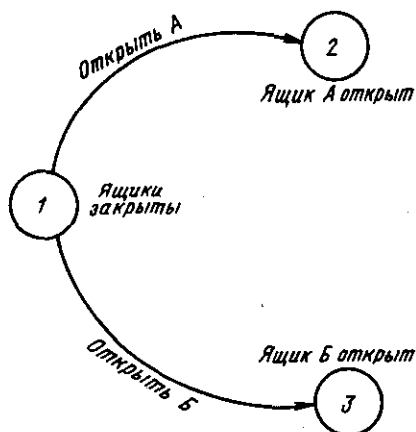


Рис. 2.2. Диаграмма состояний с переходами из состояния 1

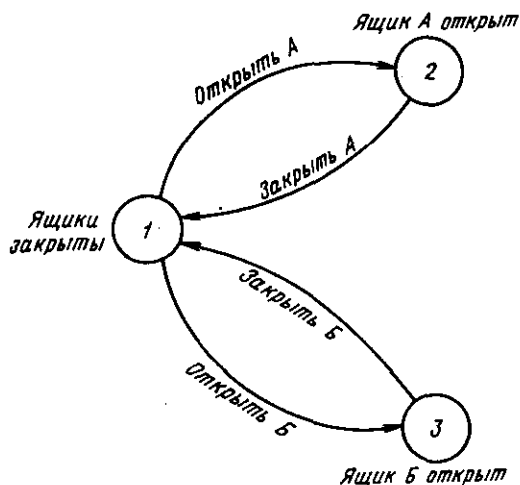


Рис. 2.3. Диаграмма состояний с возвратом к состоянию 1

В действительности, это не будет даже иметь значения, поскольку у вас больше не будет стола. Вы будете иметь обломки. А данный конечный автомат работает только

со столами. На рис. 2.2 показана диаграмма событий, которые могут изменить состояние стола, первоначально находящегося в состоянии 1. Первым событием является следующее:

Сила приложена к ящику А, в результате чего он открылся

Данное событие приводит к переходу в состояние 2. Еще одним событием является то, что приложенная к ящику Б сила открывает ящик. Состояние 3 является результатом данного события.

Перед переходом в состояние 4 оба ящика открыты, рассмотрим событие закрытия ящиков. Здесь есть еще одно событие, происходящее между переходом в состояния 1 и 2 или между 2 и 3.

В нашу диаграмму добавляются события, связанные с закрытием ящика. Состояния не изменились, а события добавились. На рис. 2.3 показано добавление событий для возврата в состояние 1.

Предположим, что мы находимся в состоянии 2. Как мы можем перейти в состояние 3? Нет прямой связи между состояниями 2 и 3, и одиночное событие не позволит осуществить этот переход. Вам может не понравиться ответ, но есть неограниченное число путей, чтобы перейти из состояния 2 в состояние 3. Только один путь экономичен и эффективен. Этим наилучшим путем, как показано на диаграмме состояний, является:

1. Закрыть ящик А - это переводит нас в состояние 1

2. Открыть ящик Б - это переводит нас в состояние 3.

А где же другие миллиарды способов перейти в такое состояние? В состоянии 1 имеются два варианта выбора - открыть ящик Б или снова открыть ящик А. Если ваш вариант принятия решения на практике чем-то короче (например, бросить монетку), вы могли бы выбрать открытие ящика А. Это решение приведет вас обратно к состоянию 2. Теперь вы имеете только один вариант выбора пути для перехода в состояние 3 - закрыть ящик А. Повторим еще раз, находясь в состоянии 1, вы должны решить, какое событие произойдет. Вы могли бы вновь открыть ящик А. (Проклятье! Опять все сначала.) Сколько раз вы могли бы открыть и закрыть ящик А перед тем, как открыть ящик Б? Фактически неограниченное число раз. Это маловероятно, но это могло бы произойти.

Осталось одно не рассматривавшееся нами состояние, состояние 4. В состоянии 4 открыты оба ящика. На рис. 2.4 показана новая диаграмма состояний. Единственный способ открыть оба ящика - надо находиться в одном из состояний, в котором один из ящиков открыт, и открыть другой ящик.

Не откладывая надолго, добавим события, заключающиеся в закрытии одного ящика. Это позволит нам перейти из состояния, когда открыты оба ящика, к состоянию, когда открыт один ящик.

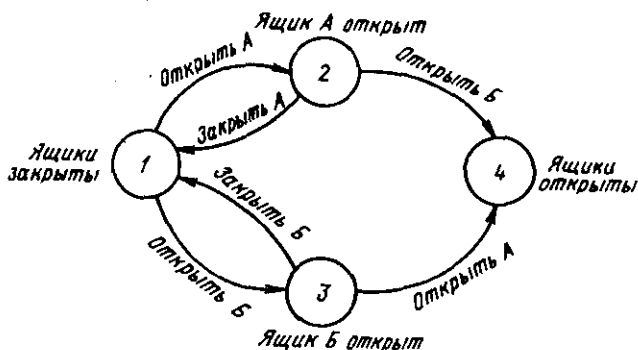


Рис. 2.4. Диаграмма с добавленным состоянием 4

На рис. 2.5 приведена полная диаграмма состояний. Все возможные и допустимые действия отображены на этой диаграмме. Нет ни одного изменения состояния, приводящего от состояния, в котором открыты оба ящика в состояние, когда оба ящика закрыты. Вы должны закрыть один ящик, перед тем как закрыть другой.

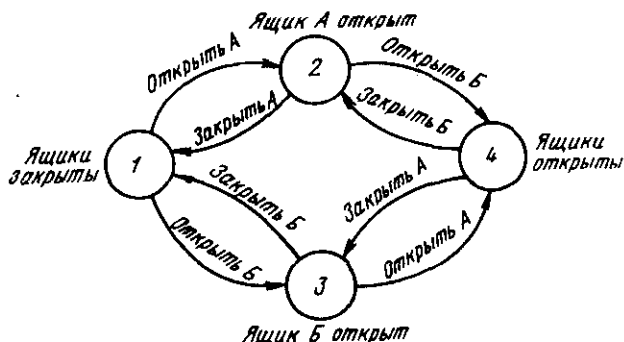


Рис. 2.5. Полная диаграмма состояний

Когда вы знаете, в каком состоянии находитесь, вы знаете, какие события являются допустимыми, а какие недопустимы. Необходимо, чтобы сообщалось о возникновении недопустимых событий либо они игнорировались.

Мы можем преобразовать диаграмму состояний в таблицу, в которой будет указываться, что делать при совершении какого-либо события. Когда происходит допустимое событие, таблица подскажет нам, в какое новое состояние осуществляется переход. Такая таблица называется таблицей переходов.

Таблица переходов - это двумерная таблица. На двух сторонах таблицы представлено:

1. Состояния, отображенные на диаграмме.
2. Возможные события.

Чтобы использовать таблицу переходов, вы выбираете строку таблицы, представляющую текущее состояние системы. Столбец таблицы определяется исходя из природы события. События, которые не отражены на диаграмме, игнорируются или о них сообщается как об ошибке. На пересечении строки текущего состояния и столбца события таблицы указывается новое состояние. Это новое состояние, в которое происходит переход при возникновении указанного события.

Если событие не влияет на изменение состояния, изменений не делается. Вы не можете открыть ящик А, если он уже открыт. Вы останетесь в прежнем состоянии. На лист. 2.25 приведен исходный текст программы - конечный автомат состояний стола.

```
/* названия состояний*/
#define ALL_CLOSED 0
/* Все ящики закрыты*/
#define DRAWER_A_OPEN 1
/* Ящик А открыт */
#define DRAWER_B_OPEN 2
/* Ящик Б открыт */
#define ALL_OPEN 3
/* Все ящики открыты*/
/* названия событий */
#define OPEN_A 0
/* Открыть ящик А */
#define CLOSE_A 1
/* Заккрыть ящик А */
#define OPEN_B 2
/* Открыть ящик Б */
#define CLOSE_B 3
```

```

/* Закрыть ящик Б */
int desk_transition[4][4] =
    {{DRAWER_A_OPEN, ALL_CLOSED, DRAWER_B_OPEN, ALL_CLOSED},
     {DRAWER_A_OPEN, ALL_CLOSED, ALL_OPEN, DRAWER_A_OPEN},
     {ALL_OPEN, DRAWER_B_OPEN, DRAWER_B_OPEN, ALL_OPEN},
     {ALL_OPEN, DRAWER_B_OPEN, ALL_OPEN, DRAWER_A_OPEN}};

int current_state = 0;
operate_desk()
{
    current_state = 0;
    while (TRUE)
    {
        current_state = desk_transition[current_state][get_event()];
        switch(current_state)
        {
            case ALL_CLOSED : printf("Все ящики закрыты\n");
                               break;
            case DRAWER_A_OPEN : printf("Ящик А открыт\n");
                               break;
            case DRAWER_B_OPEN : printf("Ящик Б открыт\n");
                               break;
            case ALL_OPEN : printf("Все ящики открыты\n");
                           break;
        }
    }
}

```

Лист. 2.25. Конечный автомат - состояния стола

Переходы между состояниями, в которых находится стол, не всегда требуются для управления действиями конечного автомата состояний. Да, события по-прежнему нужны. Однако автомат может управляться из программы, если переход всегда (или почти всегда) приводит к тому же самому состоянию. Это применяется в конечных автоматах с небольшим числом состояний.

На лист. 2.26 представлен исходный текст программы, реализующей конечный автомат для считывания DXF-файлов. Все детали этой программы будут разобраны в последних главах книги. Тем не менее сейчас рассмотрим, как происходит изменение состояний. В программе нет таблицы переходов для управления изменениями состояний. В этом случае основные изменения происходят из состояния 1 и вызываются событиями "9", "10", "20" и "0". Переход в состояние 4 происходит при обнаружении соответствующего маркера и пары значений. Изменение всех других состояний вызывает возврат к состоянию 1.

```

while (!quit)
{
    t = get_tag_token(c = get_tag(f));
    switch (state)
    {
        case 1: state = 6;
                if (t == NINE) state = 3;

```

```

        if (t == ZERO) state = 2;
        if (t == TEN) state = 4;
        if (t == TWENTY) state = 5;
        break;
    case 2: if (t == ENDSEC) quit = TRUE;
            state = 1;
            break;
        :
        :
        :
    }
}

```

Лист. 2.26. Пример реализации конечного автомата

Заключение

Вооружившись различными методами программирования, мы можем двигаться дальше, чтобы рассмотреть некоторые реальные ситуации, в которых эти методы найдут свое применение.

Разбирая примеры программирования, вы поймете, что слова представляют основу общения. Преобразование слов в управляемую форму (лексемы) становится неизбежным для однородных потоков информации в сложной системе.

Вы также увидите, что большинство наиболее часто используемых методов представляет собой реализацию конечных автоматов. Мы будем иметь дело с входным потоком информации (DXF-файлы), который не всегда является полным и в котором не соблюдается какой-либо фиксированный порядок данных.

Глава 3

ПРОГРАММНОЕ ОКРУЖЕНИЕ СИСТЕМЫ AUTOCAD

В этой главе продолжается изучение основной и вспомогательной информации, которая позволит вам разработать программы, дополняющие программы автоматизированного проектирования для системы AutoCAD. Предыдущие две главы знакомят с методами программирования, рассматриваемыми в книге. Данная глава и следующая за ней (данные и структура файлов) посвящена тем аспектам системы AutoCAD, с которыми вы столкнетесь как программист.

В частности, в главе рассматриваются основные принципы и соглашения, используемые в системе AutoCAD. Мы познакомимся с набором команд системы AutoCAD и соглашениями, применяемыми при настройке среды операционной системы MS-DOS.

Последующие разделы книги разделяются в зависимости от характера взаимодействия программ с системой AutoCAD. Система AutoCAD может использоваться для создания данных, которые служат для управления другой программой. Этот раздел называется "Система AutoCAD как управляющая система". В другом разделе показывается, как могут быть написаны программы, поставляющие системе AutoCAD графическую информацию. Второй раздел имеет название "Система AutoCAD как исполнительная система".

Раздел "Система AutoCAD как исполнительная система" в основном посвящен работе с пакетными файлами системы AutoCAD. Пакетными файлами называются файлы, содержащие команды системы AutoCAD. Команды интерпретируются системой так же, как если бы они были введены с клавиатуры. Сейчас мы рассмотрим структуру команд системы AutoCAD. Эти подробности необходимо знать для создания для системы AutoCAD файлов с корректной структурой.

Я собираюсь разделить команды системы AutoCAD на несколько категорий. На самом деле это можно сделать несколькими способами. Команды можно разделить по функциональному признаку, т. е. на команды, которые используются совместно. В этом случае можно сгруппировать команды вычерчивания графических объектов, таких, как CIRCLE (КРУГ)¹, LINE (ОТРЕЗОК), ARC (ДУГА), POINT (ТОЧКА). В то время как вспомогательные команды, такие, как SNAP (ШАГ), GRID (СЕТКА) и ORTHO (ОРТО), отнести к другой группе. Можно было расположить команды просто в алфавитном порядке, что явилось бы прекрасным справочником для пользователя. Однако я собираюсь разделить команды в зависимости от их структуры. Команды по своей структуре могут быть разделены на группы в зависимости от числа и типа параметров. Я выбрал именно такой подход.

¹ В скобках приведены названия команд, используемых в русском варианте системы AutoCAD версии 10. - *Прим. ред.*

поскольку применяемые методы программирования могут быть аналогично подстроены под структуру команд.

Цель данной главы — дать обзор команд системы AutoCAD, чтобы приведенные в последних главах программы, в которых используются эти команды, были более понятными. Список команд включает только те из них, которые обычно могут появляться в пакетных файлах. Подробную информацию о командах можно найти в документации по системе AutoCAD.

Команды без параметров

Команды без параметров наиболее простые, в них указывается только имя команды. Поскольку они не обладают достаточной гибкостью, таких команд не очень много.

В пакетных файлах параметры и команды отделяются друг от друга пробелами или символами "возврат каретки". Пробелы в конце строки очень трудно заметить, поэтому я советую использовать пробелы только в середине строки.

END (КОНЕЦ) - завершает текущий этап редактирования и сохраняет чертеж. По команде END система AutoCAD записывает представление вашего чертежа в файл чертежа (DWG-файл). В файле чертежа содержатся различные стертые объекты, измененные объекты и неиспользованные блоки. Чтобы избавиться от этого дополнительного багажа, вам потребуется начать новый этап редактирования. Обычное завершение второго сеанса редактирования по команде END уменьшит длину DWG-файла, так как система AutoCAD в начале своей работы удаляет стертые объекты. Для полной очистки DWG-файла от ненужной информации используется команда PURGE (УДАЛИ).

GRAPHSCR (ГРАФЭКР) - данная команда применяется для перевода системы AutoCAD в графический режим работы. Если до выполнения команды система находилась в текстовом режиме (например, были введены команды LIST (СПИСОК) или HELP (ПОМОЩЬ)), то после ее выполнения система перейдет обратно в графический режим, являющийся ее обычным рабочим режимом.

TEXTSCR (ТЕКСТЭКР) - команда служит для перевода системы AutoCAD в текстовый режим работы. Обычно для работы в текстовом режиме отводится область в нижней части экрана, вмещающая две или три строчки текста. В этой области появляются имена и параметры вводимых пользователем команд. По команде TEXTSCR область ввода-вывода текста займет весь экран целиком, на экране будут отображаться введенные ранее команды, а также сообщения системы.

REDRAW (ВСЕОСВЕЖ) - команда применяется для перерисовки изображения в области вывода графической информации экрана. При этом изображение очищается от ненужных деталей, появляющихся при добавлении и стирании графических объектов.

Команда **REDRAW**, помещенная в конце пакетного файла, позволяет получать обновленное изображение.

REGEN (ВСЕРЕГЕН) - по команде осуществляется пересчет графических объектов, находящихся в базе данных, и полностью заново строится изображение на экране. Эту команду не следует часто использовать в пакетных файлах, поскольку пересчет и вывод вновь сгенерированного изображения занимает относительно много времени.

RESUME (ПРОДОЛЖИ) - команда служит для продолжения выполнения пакетного файла, которое было прервано из-за какой-либо ошибки или по инициативе пользователя (например, нажата клавиша Backspace - "шаг назад"). Маловероятно, что вы будете использовать эту команду в пакетном файле. Однако, она вам пригодится для возобновления выполнения пакетного файла.

DBLIST (БДСПИСОК) - по данной команде выводится в виде списка информация о всех объектах, находящихся в базе данных чертежа. Текст выводится на экран очень быстро, поэтому рекомендуется осуществлять вывод на принтер.

OOPS (ОЙ) - по команде восстанавливаются на экране последние либо просто стертые объекты, либо стертые после их объединения в блок.

RSCRIPT (ВПАКЕТ) - помещенная в конец пакетного файла команда **RSCRIPT** возобновляет выполнение пакетного файла с самого начала, что может применяться для непрерывного показа слайдов или различного рода демонстраций.

Команды с одним параметром, выполняющим роль переключателя

Команды этого типа изменяют режимы работы с системой AutoCAD. Помещенные в пакетный файл, используемый для инициализации системы, они позволят настроить рабочую среду системы для вашего приложения. Каждая команда имеет один параметр, указываемый сразу после имени команды. Для отделения параметра от команды служит пробел или новая строка. Все эти команды воспринимают простые параметры, такие, как **ON (Вкл)** и **OFF (Откл)** или **YES (Да)** и **NO (Нет)**. Несколько команд этого типа также воспринимают рабочие значения или ключевое слово **AUTO**.

BLIPMODE (МАРКЕР) - при вводе координат, во время работы с системой AutoCAD, для обозначения местоположения точки, определяемой заданными координатами, на экране появляется специальная вспомогательная отметка. Эта отметка называется маркером (**BLIP**). Команда **BLIPMODE** предназначена для управления генерацией вспомогательных отметок. Задание в команде параметра **YES** разрешает отображать отметки маркера.

Если указывается параметр **NO**, отметки не отображаются. При выполнении команды **REDRAW** все вспомогательные отметки удаляются.

GRID (СЕТКА) - в процессе работы над чертежом удобно иметь на экране дисплея некоторые визуальные ориентиры, позволяющие оценивать расстояния. Команда **GRID** служит для переключения режима отображения координатной сетки на экране дисплея. Указание в команде параметра **ON** включает отображение координатной сетки, а указания параметра **OFF** - отменяет такое отображение. Например, можно задать шаг координатной сетки, указав число 0.5, что будет эквивалентно половине установленной единицы измерения расстояния. В типичных пакетных файлах шаг координатной сетки будет устанавливаться в соответствии с наиболее удобным масштабом. Задание шага координатной сетки включает отображение на экране точек этой сетки.

SNAP (ШАГ) - использование команды **SNAP** дает уверенность, что координаты указываемых точек будут попадать в четко определенный интервал. Вследствие ограниченной разрешающей способности экрана иногда может казаться, что один отрезок соприкасается с другим отрезком, но нельзя быть уверенным, что координаты отрезков совпадут, если только не использовать какой-либо из вариантов команды **SNAP**. Команда **SNAP** служит для установки или отмены заданной разрешающей способности при привязке объектов. При указании в команде параметра **ON** устанавливается ранее заданное значение разрешающей способности, а при задании параметра **OFF** - отменяется. Можно директивно установить интервал разрешения, задав шаг координат, например число 0.5, что будет эквивалентно половине принятой единицы измерения расстояния. В типичных пакетных файлах интервал разрешения будет устанавливаться в соответствии с наиболее удобным масштабом. Задание интервала разрешения (или иначе разрешающей способности) активизирует новое значение.

AXIS (ОСИ) - команда **AXIS** позволяет отображать оси координат вдоль границ графической области экрана. Отображение осей включается, если в команде **AXIS** указан параметр **ON**. Указание параметра **OFF** отключает отображение осей. Можно задать отображение делений на осях координат, для чего надо указать цену деления, например число 0.1, что соответствует одной десятой принятой единицы измерения расстояния. В типичных пакетных файлах цена деления будет устанавливаться в соответствии с наиболее удобным масштабом. При задании цены деления включается отображение осей координат.

ORTHO (ОРТО) - команда **ORTHO** управляет режимом "ортогонального" рисования. В этом режиме разрешается задавать координатные точки только по ортогональным осям (под углом 90°) относительно заданной точки. В обычном режиме на задание местоположения какого-либо объекта не накладывается никаких

ограничений. В команде **ORTHO** параметр **ON** служит для включения режима "ортогонального" рисования, а параметр **OFF** - для его отмены.

REGENAUTO (РЕГЕНАВТО) - для некоторых команд, чтобы отобразить сделанные в чертеже изменения на экране, требуется последующее выполнение команды **REGEN**, другие команды выполняют регенерацию автоматически. Команда **REGENAUTO** позволяет выполнять автоматическую регенерацию чертежа, когда это необходимо. Однако регенерация может занимать много времени. Пользователь может управлять этой возможностью, отменяя автоматическую регенерацию. Заданный в команде **REGENAUTO** параметр **ON** включает, а параметр **OFF** отменяет режим автоматической регенерации. Если в команде **REGENAUTO** был указан параметр **OFF**, то при выполнении команды после которой требуется регенерация чертежа, выводится предупреждающее сообщение, что должна быть выполнена регенерация. В качестве такого примера можно привести команду **ZOOM EXTENTS (ПОКАЖИ границы)**. Для продолжения работы после появления сообщения пользователь должен ответить на запрос системы, подтверждая или отказываясь от выполнения регенерации.

QUIT (ПОКИНЬ) - данная команда завершает текущий сеанс редактирования, все изменения, сделанные во время сеанса, не запоминаются. Для завершения сеанса работы по команде **QUIT** требуется подтверждение —

это равнозначно указанию в команде параметра **YES**. Команда **QUIT** может использоваться для завершения сеанса и в пакетных файлах, где для сохранения блоков чертежа применяются команды **WBLOCK (ПБЛОК)** или **SAVE (СОХРАНИ)**. Если команда **QUIT** была введена при работе над новым чертежом, система AutoCAD не образует файл чертежа.

FILL (ЗАКРАСЬ) - при закрашке областей экрана, занимаемых такими объектами, как полосы (**TRACE**), фигуры (**SOLID**) и полилинии (**POLYLINE**), выполнение этой операции требует определенного увеличения вычислений. Команда **FILL** служит для управления режимом закрашки. Параметр **ON** данной команды, указывает системе AutoCAD, что объекты на экране должны быть закрашены. Параметр **OFF** указывает, что на экране будут отображаться только контуры объектов. Для того чтобы изображение объектов на экране было приведено в соответствие с текущим режимом, установленным по команде **FILL**, необходимо воспользоваться командой **REGEN**.

QTEXT (КТЕКСТ) - текст является одним из объектов, генерация которых на экране отнимает немало времени. Для изображения букв может потребоваться большое число штрихов. Команда быстрого отображения текста **QTEXT** позволяет быстро изображать на экране текст, показывая только место расположения

текста и область, которую он занимает, не вычерчивая сами буквы. Параметр ON служит для переключения в режим быстрого отображения текста. Параметр OFF используется для переключения системы AutoCAD в режим обычного отображения текста.

Команды с одним параметром - именем файла

В командах данной группы в качестве параметра указывается имя файла. При создании пакетного файла с помощью вашей прикладной программы для этих команд следует обеспечить имя файла. В некоторых случаях вы можете использовать имя файла (имя чертежа) по умолчанию. Если в команде параметр не указан, то выбирается имя файла по умолчанию.

DXFIN (ИМПОРТА) - эта команда дает указание системе AutoCAD ввести файл обмена описаниями чертежей в формате DXF, содержащий дополнительную графическую информацию. Имя файла (как можно увидеть в каталоге на диске) должно иметь расширение .DXF. В системе AutoCAD предполагается, что файлы обмена описаниями чертежей в формате DXF имеют расширение имени файла DXF. Файлы типа DXF содержат символьную информацию - информацию в коде ASCII.

DXBIN (ИМПОРТД) - команда дает указание системе AutoCAD ввести файл обмена описаниями чертежей в формате DXB, содержащий дополнительную графическую информацию. Имя файла (как можно увидеть в каталоге на диске) должно иметь расширение DXB. В системе AutoCAD предполагается, что файлы обмена описаниями чертежей в формате DXB имеют расширение имени файла DXB. Файлы типа DXB содержат информацию в двоичном формате.

DXFOUT (ЭКСПОРТА) - по данной команде система AutoCAD создает файл обмена описаниями чертежей и записывает в него подробную информацию о текущем чертеже. Имя созданного файла (как можно увидеть в каталоге на диске) будет иметь расширение DXF. В системе AutoCAD предполагается, что файлы обмена описаниями чертежей в формате DXF имеют расширение имени файла DXF. Второй параметр команды устанавливает число значащих цифр для чисел, содержащихся в файле. Вы можете использовать значение по умолчанию, включив дополнительную пустую строку в пакетный файл. Более подробную информацию о выборе графических объектов, описание которых помещается в DXF-файл, можно найти в справочном руководстве системы AutoCAD.

SCRIPT (ПАКЕТ) - команда дает указание системе AutoCAD считать содержимое пакетного файла для ввода дополнительной графической информации. Имя файла (как можно увидеть в каталоге на диске) должно иметь расширение SCR. В системе

AutoCAD предполагается, что пакетные файлы имеют расширение имени файла SCR. Файлы типа SCR содержат графическую информацию, представленную в формате команд системы AutoCAD. Вы можете из одного пакетного файла системы AutoCAD запустить на выполнение другой. Это позволяет объединять несколько пакетных файлов вместе. При вызове одного пакетного файла из другого, после того как было считано содержимое второго файла, управление первому файлу обратно не передается.

SAVE (СОХРАНИ) - команда позволяет во время сеанса редактирования записать текущий чертеж в файл чертежа. Именем файла может быть имя текущего чертежа (по умолчанию) или любое другое имя. Выходной файл будет иметь расширение имени файла DWG. В команде SAVE не требуется явно указывать расширение имени файла.

LOAD (ЗАГРУЗИ) - по команде LOAD система AutoCAD загружает графическую информацию из заданного файла описания графических символов. Указываемое в качестве параметра имя файла должно иметь расширение SHX.

MENU (МЕНЮ) - команда MENU служит для загрузки файла меню в систему AutoCAD. Если вы подготовили специальное меню с набором команд для вашего приложения, то по команде MENU, встретившейся при выполнении пакетного файла, меню будет вызвано на экран без дополнительного напоминания пользователю о выполнении загрузки. Когда пользователь сохраняет чертеж, в файле чертежа сохраняется и имя меню. Обычно расширением имени для файла меню является MNU. Однако система AutoCAD после компиляции меню присваивает файлу расширение имени MNX.

VSLIDE (СЛАЙД) - команда СЛАЙД позволяет вывести на графический экран содержимое файла слайда. Система AutoCAD считывает данные слайда из файла, имя которого указано в команде в качестве параметра.

MSLIDE (ДСЛАЙД) - по команде MSLIDE система AutoCAD преобразует видимое изображение на экране и сохраняет его в файле слайда. Система AutoCAD записывает данные слайда в файл с именем, указанным в качестве параметра команды. Этот файл имеет расширение имени файла SLD.

SHELL (ДОС) - команда SHELL прерывает обычный процесс работы системы AutoCAD для выполнения внешней программы. Текущий процесс системы AutoCAD может заключаться в подготовке данных для внешней программы, выполняющей некоторые вычисления, которые не реализуются на языке АвтоЛИСП или обычными командами системы. Система AutoCAD после ввода команды SHELL запрашивает имя программы. Если программа с указанным именем доступна, то она запускается на выполнение. Система AutoCAD запустит на выполнение указанную

программу, если она зарегистрирована в файле описания параметров внешних команд - PGP-файле.

Другие команды с одним параметром

Эти команды имеют широкое применение. Однако в них тоже указывается только один-единственный параметр. При использовании этих команд в пакетных файлах они могут быть заданы одним из двух способов. Первый заключается в том, что имя команды и параметр могут отделяться друг от друга одним пробелом. Второй способ состоит в том, что команда и ее параметр записываются на разных строках. В каждой из этих строк дополнительные пробелы не нужны.

APERTURE (АПЕРТУРА) - эта команда относится к командам установки параметров рабочей среды. Команда определяет размер прицельного окна для привязки объектов, применяемого в случаях, когда при построении объектов был задан разный интервал разрешения. Данная команда не влияет на размер прицельной рамки, используемой для выбора объектов при включении их в набор объектов. Параметром этой команды является размер прицельного окна, который задается числом точек раstra. Команда может зависеть от конкретного типа устройств отображения графической информации.

COLOR (ЦВЕТ) - цвет объектов обычно устанавливается в соответствии с цветом, заданным для слоя. Однако возможно установить цвет для отдельных объектов, даже если они находятся на одном слое. Команда **COLOR** изменяет текущий цвет объекта. Параметр представляет собой номер цвета (от 1 до 255) или особое зарезервированное имя цвета (**BLUE** - голубой). Вы можете также указать одно из двух специальных имен цвета. Имя цвета **BYLAYER** (по слою) устанавливает цвет объекта по цвету текущего слоя. Имя цвета **BYBLOCK** (по блоку) назначает цвет объектов соответственно с цветом, определенным для блока, в который они входят.

DELAY (ЗАДЕРЖИ) - команда **DELAY** предназначена для замедления показа слайдов. Я не могу найти каких-либо других соображений, для чего надо замедлять работу системы AutoCAD. Параметром команды **DELAY** является число, приблизительно кратное одной миллисекунде, определяющее время задержки обработки.

LTSCALE (ЛМАСШТАБ) - команда **LTSCALE** устанавливает приемлемые пропорции размера штрихов в линиях различного типа. В системе AutoCAD в описании типа линии длина штриха приведена к величине, соответствующей (или приблизительно равной) одной единице измерения. Если линейные размеры объектов на чертеже велики или, наоборот, малы, что выражается либо очень большим

числом единиц измерения, либо, наоборот, их долями, то штриховые линии на чертеже могут быть потеряны. Параметр в команде **LTSCALE** представляет собой масштабный множитель (действительное число), устанавливающий разрешающую способность при изображении линий различного типа.

Команды построения простых объектов

Я собираюсь разделить графические объекты на простые и сложные. Простые объекты являются графическими построениями с фиксированным набором параметров. Если программа генерирует пакетный файл для простого объекта, то единственное, что остается сделать, - это подставить на место параметров известные или вычисленные данные. Сложные объекты не должны состоять из фиксированного числа элементов. Построение сложного объекта продолжается, пока для него вводятся данные. Примером построения сложного объекта является команда **LINE (ОТРЕЗОК)**. Команда **LINE** не будет завершена до тех пор, пока не будут перечислены все точки, которые вы хотите указать. В результате вы получите ломаную линию, соединяющую все заданные точки.

Следующие далее команды попадают в разряд команд построения простых графических объектов. Отметим, что команды **ARC (ДУГА)** и **TEXT (ТЕКСТ)** имеют некоторые расширения, которые могут использоваться и для построения сложных объектов. Тем не менее эти расширения команд обычно не используются в пакетном файле.

POINT (ТОЧКА) - команда **POINT** служит для задания точки. Координаты точки определяются параметрами команды. Параметрами команды является пара координат по осям X, Y. Числа отделяются друг от друга запятой без пробелов. Данный объект является простейшим графическим объектом. На лист. 3.1 показана команда **POINT**.

POINT 5.5.3.12

Лист. 3.1. Команда **POINT**

CIRCLE (КРУГ) - команда **CIRCLE** предназначена для вычерчивания окружности в соответствии с заданными координатами и числами. Существует несколько способов задания места и размера вычерчиваемой окружности. В справочном руководстве системы AutoCAD перечислено пять способов. Вы легко можете применять четыре из них в пакетных файлах. Метод **TTR (ККР)** - построение по двум касательным и радиусу, не подходит для пакетных файлов. Этот способ сопряжен с подбором различных вариантов и требует дополнительных расчетов, перед тем как его можно было бы использовать в пакетном файле. На лист. 3.2 - 3.5 показаны четыре способа задания окружностей.

CIRCLE 1.2.3.4 0.8

Лист. 3.2. Команда CIRCLE - построение окружности, заданной координатами центра и радиусом

CIRCLE 1.2.3.4 D 1.6

Лист. 3.3. Команда CIRCLE - построение окружности, заданной координатами центра и диаметром

CIRCLE 3P 2.4.4.1 5.6.6.8 4.56.5.0

Лист. 3.4. Команда CIRCLE - построение окружности по трем точкам

CIRCLE 2P 5.5 3.5 /

Лист. 3.5. Команда CIRCLE - построение окружности по двум точкам (точки диаметра)

ARC (ДУГА) - команда ARC применяется для построения дуги (части окружности) в соответствии с заданными параметрами. Подобно команде CIRCLE имеется несколько способов задания дуги. В справочном руководстве системы AutoCAD перечислены восемь способов задания дуги. Одним из способов является продолжение предыдущей дуги или линии. По своей природе данная команда могла быть отнесена и к командам построения сложных объектов. На лист. 3.6 - 3.13 показаны различные примеры команды ARC.

ARC 2.4.4.1 5.6.6.8 4.56.5.0

Лист. 3.6. Команда ARC - построение дуги по трем точкам

ARC 1.3 C 3.3 5.3

Лист. 3.7. Команда ARC - построение дуги по начальной точке, центру и конечной точке

ARC 1.3 C 3.3 A 135

Лист. 3.8. Команда ARC - построение дуги по начальной точке, центру и центральному углу

ARC 1.3 C 3.3 L 2.134

Лист. 3.9. Команда ARC - построение дуги по начальной точке, центру и длине хорды

ARC 1.3 E 6.5 R 10 0

Лист. 3.10. Команда ARC - построение дуги по начальной точке, конечной точке и радиусу

ARC 1,3 E 6,5 A 125

Лист. 3.11. Команда ARC - построение дуги по начальной точке, конечной точке и центральному углу

ARC 1,3 E 6,5 D 90

Лист. 3.12. Команда ARC - построение дуги по начальной точке, конечной точке и начальному направлению

ARC

2,3,4,5

Лист. 3.13. Продолжение предыдущей линии или дуги

TEXT (ТЕКСТ) - команда TEXT является основным средством для вставки текста в чертеж. Команда ATTDEF также позволяет отображать текст, но имеет дополнительные параметры. В команде TEXT задается правило выравнивания текста. Система AutoCAD воспринимает пять различных типов выравнивания текста. Две дополнительные формы команды позволяют устанавливать стиль начертания текста (шрифт) и продолжать текст, начатый в предыдущих командах. Параметры команды определяют местоположение, высоту и поворот текста. Сама текстовая строка помещается в конце команды. Если в описании шрифта высота символов фиксирована, то задающий высоту текста параметр команды может быть опущен. На лист. 3.14 - 3.20 показаны различные варианты команды TEXT.

TEXT 1,2,2,0 0,08 0,0 Как поживает коричневая корова

Лист. 3.14. Команда TEXT - текст начинается справа от указанной точки

TEXT C 5,0,3,6 0,125 90 Вертикальный отцентрированный текст

Лист. 3.15. Команда TEXT - текст отцентрирован относительно указанной точки

TEXT F 1,25,2,5 5,50,2,5 0,10 Теперь текст горизонтальный

Лист. 3.16. Команда TEXT - текст написан между двумя заданными точками

TEXT M 2,0,4,0 0,125 0,0 Очень похож на отцентрированный текст

Лист. 3.17. Команда TEXT - текст располагается так, чтобы его середина находилась в указанной точке

TEXT R 3,0,5,25 0,25 0,0 Текст выровнен по правой границе

Лист. 3.18. Команда TEXT - текст заканчивается справа в указанной точке

TEXT S HELVET C 2.0,3.0 0.30 0.0
Оцентрированный текст, тип шрифта Helvetica

Лист. 3.19. Команда **TEXT** - установка типа шрифта

TEXT

Вам потребовалось продолжить текст с новой строки

Лист. 3.20. Команда **TEXT** - для продолжения ранее начатого текста

SHAPE (ФОРМА) - команда **SHAPE** применяется для вставки в чертеж графических символов. Графические символы хранятся в файлах графических символов (**SHP**-файлах). Эти файлы подобны файлам текстовых шрифтов. В отличие от шрифтов система AutoCAD не предоставляет для графических символов такое многообразие вариантов их модификации. Файл графических символов должен быть загружен перед использованием графических символов. В команде **SHAPE** указывается имя графического символа, его местоположение, высота символа и угол поворота. На лист. 3.21 приведен типичный пример команды **SHAPE**.

SHAPE A_SHAPE 2.25,3.33 1.0

Лист. 3.21. Команда **SHAPE**

INSERT (ВСТАВЬ) - команда **INSERT** является, может быть, наиболее мощной командой системы AutoCAD. Эта команда позволяет вставлять в текущий чертеж простые изображения или даже целый чертеж. Большинство других систем автоматизированного проектирования предоставляют пользователям такую возможность, но выполняют это существенно тяжелее, чем могли бы сделать внешние программы. Применяя команду **INSERT** в пакетном файле, вы можете легко манипулировать с библиотеками графических символов. Команда **INSERT** имеет ряд параметров, которые включают: имя вставляемого блока, его местоположение, масштабные множители и угол поворота. Если вставляемый блок имеет атрибуты, то в пакетный файл требуется вставить соответствующие значения этих атрибутов. В системе AutoCAD отсутствуют команды для вставки значений атрибутов. На лист. 3.22 показана вставка блока с тремя значениями атрибутов.

INSERT A_BLOCK 4.5,7.65 1.0 1.0 0.0

12-342L

BLUE SPECKS

JONES MANUFACTURING

Лист. 3.22. Команда **INSERT**

Команды построения сложных объектов

В системе AutoCAD сложные объекты определяются в виде списка, составленного из связанных между собой простых объектов. В файлах формата DWG и DXF такие определения объектов задаются в более строгом виде. Согласно этим определениям, единственно сложными объектами являются объекты, построение которых осуществляется с помощью команд POLYLINE и INSERT (с вводом значений атрибутов). Если говорить о формате указанных файлов, то полилинию образуют индивидуальные объекты, рассматриваемые как узлы списка сложного объекта. Список объектов завершает запись SEQEND. В противоположность этому вставленный блок с атрибутами представляет собой цепочку объектов, которая начинается с записи INSERT, далее следует запись ATTRIB, а завершает цепочку запись SEQEND.

Однако с точки зрения организации пакетного файла команды, в которых допустимо задание неограниченного числа точек или значений, отличаются от рассмотренных команд построения простых объектов и являются более сложными. В этом параграфе рассматриваются используемые в пакетных файлах команды построения сложных объектов. Эти команды могут иметь большое число аргументов.

Поскольку не накладывается ограничений на число аргументов, которые могут иметь команды построения сложных объектов, то должно быть средство ограничения и завершения последовательности аргументов команды. В качестве разделителей между аргументами система AutoCAD воспринимает пробелы и новые строки. Пакетный файл становится более выразительным и легким для чтения, если каждый аргумент (обычно координаты) помещается на отдельной строке. Список аргументов завершается при добавлении второго пробела или переходе на новую строку в том месте, где система AutoCAD ожидает обнаружить аргумент. При работе с редактором текста очень трудно увидеть пробелы, находящиеся в конце строки. Редактирование файла, содержащего пробелы в концах строк может быть кошмарным.

LINE (ОТРЕЗОК) - поскольку команда LINE может иметь неограниченное число аргументов, она относится к командам построения сложных объектов. Сами по себе аргументы не являются сложными. Они представляют собой просто координаты следующей точки отрезка линии. LINE 1.0,2.0

1.0,5.5

1 34.6.76

REDRAW

Лист. 3.23. Команда LINE

На лист. 3.23 типичная команда LINE. Чтобы показать наличие в конце пакетного файла пустой строки, я добавил команду REDRAW.

TRACE (ПОЛОСА) - в начале развития системы AutoCAD, пока не была введена полилиния, команда TRACE была основной командой для вычерчивания линий с заданной шириной. Весьма подходящая для работы с чертежами печатных плат команда TRACE стала арматурой в фундаменте системы AutoCAD. Графический объект TRACE (полоса) легко вычерчивается, но трудно редактируется. Поэтому с появлением полилинии необходимость в нем отпала, что снизило популярность рассматриваемой команды. По своей форме команда TRACE подобна команде LINE. Единственное различие представляет то, что первым аргументом команды является значение ширины полосы. На лист. 3.24 показана типичная команда TRACE.

```
TRACE 0.015
```

```
2.5.3.321
```

```
4.0,4.0
```

```
4.0,5.5
```

```
6.3,5.5
```

```
REDRAW
```

Лист. 3.24. Команда TRACE

SOLID (ФИГУРА) - для закраски больших областей чертежа каким-либо цветом или чернилами применяется команда SOLID. Так как с помощью данной команды можно строить фигуры только типа треугольников и четырехугольников, закрашка областей сложной формы основана на заполнении их этими примитивными фигурами. Вручную это может потребовать много работы. Но с помощью программы для вычисления границ закрашенной области может быть сделано эффективно и точно. Тем не менее, как и при использовании команды в ручном режиме, вы должны соблюдать осторожность с порядком следования аргументов, чтобы не создать ужасную фигуру, напоминающую галстук-бабочку. Если вы еще не сталкивались с этим, то на самом деле пропустили одну из наиболее таинственных особенностей системы AutoCAD. Поскольку в команде SOLID допускается большое число аргументов и одна фигура может строиться на базе другой фигуры, я отнес эту команду к командам построения сложных объектов. SOLID

```
1.0,1.0
```

```
1.0,3.0
```

```
4.0,1.0
```

```
4.0,3.0
```

```
REDRAW
```

Лист. 3.25. Применение команды SOLID для закраски простого прямоугольника

На лист. 3.25 показана типичная последовательность данных для команды **SOLID**. Эта фигура не похожа на галстук-бабочку.

POLYLINE (ПЛИНИЯ) - это базовая команда для построения сложных объектов. Имеется так много различных вариаций данной команды, что я даже затрудняюсь перечислить все, не опасаясь пропустить некоторые из них. Руководствуясь основным списком координат, определяющих непрерывные сегменты линии, вы можете изменять внешний вид полилинии с помощью модификаторов. В отличие от чисел система AutoCAD распознает эти модификаторы, идентифицируя их по имени (или, по крайней мере, по букве). При записи команды в формате пакетного файла модификаторы являются наилучшим средством для изменения ширины линии или задания дуг. Вы можете в дальнейшем изменить полилинию с помощью команды **REDIT (ПОЛРЕД)**. Эта команда сложна для работы в составе пакетного файла, так как она включает множество вариантов выбора и требует вмешательства человека. На лист. 3.26 показано описание полилинии в виде последовательности строк пакетного файла. В начале данной последовательности задается ширина полилинии. Поэтому в данном примере имеются два значения, которым предшествует буква **W**.

```
PLINE 3.0,2.5  
W 0.010 0.010  
3.25,2.5  
3.25,5.0  
4.0,5.0
```

REDRAW

Лист. 3.26. Команда **POLYLINE**

Команда **LAYER** и ее модификаторы

Команда **LAYER (СЛОЙ)** удовлетворяет всем критериям, которые позволяют ее отнести к сложным командам системы AutoCAD, но и сама имеет достаточно различий, чтобы посвятить ей целый параграф. В рассмотренных перед этим командах построения сложных объектов одни координаты соотносились с другими координатами, но все они были частью одного и того же объекта.

Команда **LAYER** не имеет дело с построением графических объектов. Но тем не менее между командами имеется некоторая схожесть. Команды завершаются дополнительной пустой строкой. В командах не ограничивается число параметров. Подобно тому как графические команды служат для построения объектов, команда **LAYER** применяется для построения элементов таблицы. Другие команды строят один объект или группу связанных объектов - хотя

эти объекты не обязательно запоминаются как один объект в рабочей структуре данных системы AutoCAD. Напротив, при одном вызове команды **LAYER** можно построить или модифицировать несколько элементов таблицы. Это основная отличительная особенность команды, которая будет обсуждаться.

Программа обработки команды описания слоя, вызывается по команде **LAYER**. Работа с данной программой, если используются модификаторы команды, будет продолжаться до тех пор, пока не будет введена завершающая команду пустая строка. Большинство выполняемых над слоем операций включают имя операции и имя слоя, над которым выполняется данная операция. Исключение из правила составляют модификаторы **COLOR** (цвет) и **LTYPE** (тип линии), в которых требуется дополнительный параметр - цвет для модификатора **COLOR**, и имя типа линии для модификатора **LTYPE**.

Вместо имени одного слоя вы можете указать несколько имен слоев. Имена слоев отделяются запятыми. Если в качестве параметра допускается использовать несколько имен слоев, то также разрешается указывать символ звездочки "*" (универсальный шаблон), воспринимаемый системой AutoCAD. Поскольку в каждый момент только один слой может быть активным, модификаторы **MAKE** и **SET** требуют имя только одного слоя.

В команде **LAYER** допустимы следующие модификаторы:

NEW (НОВЫЙ) - применяется для создания нового слоя с заданным именем слоя. При выполнении данной операции вы можете указать сразу несколько имен слоев.

SET (УСТАНОВИ) - делает заданный слой текущим активным слоем. Все описываемые после установки активного слоя графические объекты будут располагаться на этом слое. Для данной операции вы можете указать только одно имя слоя.

MAKE (СОЗДАЙ) - этот модификатор позволяет комбинировать действия, выполняемые при обработке модификаторов **NEW** и **SET** в одной строке.

OFF (ОТКЛ) - убирается (но не удаляется) изображение слоя с экрана дисплея. Вы можете указать несколько слоев. Эта операция помогает очистить экран, забитый ненужной графической информацией.

ON (ВКЛ) - операция, противоположная по своему действию операции, выполняемой при задании модификатора **OFF**. Перечисленные слои снова отображаются на экране. Вы можете указать несколько слоев.

COLOR (ЦВЕТ) - выполняется операция установки цвета, используемого по умолчанию для графических объектов на конкретном слое. Данный модификатор имеет два параметра. Первым параметром является цвет. Цвет задается либо номером цвета - число в диапазоне 1 — 255, либо одним из зарезервированных названий цвета, распознаваемых системой

AutoCAD: RED (Красный), YELLOW (Желтый), GREEN (Зеленый), BLUE (Голубой), CYAN (Синий), MAGENTA (Фиолетовый) или WHITE (Белый). Вторым параметром являются имена слоев. По умолчанию для слоя используется цвет WHITE.

LTYPE (ТИПЛИНИИ) - данная операция устанавливает тип линии, используемый по умолчанию для графических объектов на заданном слое. Модификатор имеет два параметра. Первым параметром является имя типа линии. Вторым - имена слоев, для которых назначается этот тип линии. Тип линии, используемый по умолчанию, - CONTINUOUS (Непрерывный). На лист. 3.27 в формате пакетного файла показано несколько операций над слоем, выполнение которых задается с помощью модификаторов. В примере создаются несколько слоев и назначается цвет, используемый по умолчанию на слое.

```
LAYER
NEW Lyr_1,FLOOR,WING,SURFACE
COLOR RED Lyr_1
COLOR BLUE FLOOR
COLOR 14 WING
COLOR 4 SURFACE
LTYPE DASHED SURFACE
ON *
SET WING

REDRAW
```

Лист. 3.27. Команда LAYER

Соглашения операционной системы MS-DOS

Чтобы придать некоторую гибкость работе программ, разработчики часто позволяют пользователям самостоятельно устанавливать значения нескольких рабочих параметров.

В некоторых случаях для этого применяется файл конфигурации. Система AutoCAD использует файл конфигурации для установки типов имеющихся драйверов аппаратуры, начальных параметров чертежа и основных редко изменяемых значений.

Тем не менее имеются рабочие параметры, которые могут влиять на работу программы в зависимости от конкретного приложения. Они представляют собой значения, которые должны быстро и легко изменяться, чтобы отразить потребности программы в специальных средствах. Например, библиотеки графических символов в соответствии с их назначением могут храниться в нескольких каталогах. Для системы AutoCAD может потребоваться указание, какую библиотеку следует использовать.

Обычно в операционной системе MS-DOS файлы текущего каталога доступны программе без каких-либо дополнительных указаний в конструкции маршрута поиска. Кроме того, по заданному маршруту поиска операционная система обеспечивает доступ к каталогу, содержащему оригинальный, выполняющийся в данный момент файл типа .EXE. В интересующем нас случае маршрут к выполняющейся программе помещается в аргументах argc и argv функции main. Аргумент argv[1] обычно представляет собой указатель на первый параметр активизированной командной строки. Это первое, что обычно просматривает программа. Однако если компилятор обеспечивает такую возможность (как делается в компиляторе Microsoft C), то указателем на маршрут к данной программе является аргумент argv[0]. Этот аргумент включает имя программы. Чтобы получить маршрут, необходимо убрать имя программы. Для удаления имени программы поместите символ NULL после последней обратной косой черты в строке, на которую содержит ссылку аргумент.

Команда PATH операционной системы MS-DOS не предоставляет информацию о маршруте к каталогам, содержащим файлы данных. Команда PATH используется только для нахождения выполняемых файлов и пакетных файлов ОС MS-DOS.

Маршрут к любому другому библиотечному каталогу должен быть получен из другого источника. Система AutoCAD решает задачу нахождения библиотечного каталога с помощью команды SET операционной системы MS-DOS.

Одной из особенностей ОС MS-DOS является область межзадачной связи, известная как операционная среда. Она представляет собой область памяти, зарезервированную для обмена короткими сообщениями между пользователем, MS-DOS и прикладной программой. Команда SET дает пользователю возможность обращаться к этой области. Программы также могут получить доступ к среде, чтобы просмотреть, оставлено ли что-нибудь для них в этой области.

Команда SET обычно применяется для установки значения параметра. Формат команды следующий:

```
SET имя_параметра=значение_параметра
```

Имя параметра, так же как и его значение, весьма произвольно. Разнообразные фирмы-производители программного обеспечения выбирают имена параметров так, чтобы они соответствовали их назначению, и, кроме того, были уникальны, чтобы исключить конфликт с именами других параметров.

Попав в операционную среду, параметр становится общедоступным. Зная имя параметра, программа может запрашивать доступ к этому параметру операционной среды.

Используя функцию `getenv` компилятора Microsoft C (и, по-видимому, также и других компиляторов), программа может запрашивать значение конкретного параметра операционной среды. Функция `getenv` имеет единственный аргумент. Этот аргумент представляет собой строку (или, точнее, адрес строки), содержащую "имя_параметра". Функция `getenv` возвращает адрес строки, содержащей связанное с заданным параметром "значение_параметра". Например, пользователь ввел следующую команду операционной системы MS-DOS:

```
SET ABCD=C:\LIBRARY
```

На лист. 3.28 показан фрагмент программы, в котором выполняется проверка операционной среды на наличие указанного значения. Результатом работы программы будет

```
C:\LIBRARY
```

```
char s[64];  
s = getenv("ABCD");  
printf("Значение параметра ABCD - %s\n",s);
```

Лист. 3.28. Применение функции `getenv`

Система AutoCAD использует некоторые из сообщений, находящихся в операционной среде, чтобы позволить пользователю внести рабочие коррективы между сеансами редактирования. Большую проблему составляет то, что операционная среда должна изменяться извне системы AutoCAD. Когда система AutoCAD запускается, она проверяет адресованные ей сообщения среды, воспринимает их и повторно к параметрам среды не возвращается. Применение команды SHELL системы AutoCAD для модификации параметров среды не производит желаемых изменений в работе системы.

Система AutoCAD распознает следующие сообщения:

ACAD - данный параметр среды должен быть задан так же, как маршрут в операционной системе MS-DOS. Этот маршрут ведет к вспомогательной библиотеке чертежей (блоков). Система AutoCAD будет искать связанный с указанным маршрутом поиска каталог, чтобы попытаться удовлетворить запрос на вставку блока, если такой блок не найден внутри чертежа. Например, если вы имеете каталог, в котором содержатся символы деталей трубопроводов, то они могут находиться в каталоге с именем PIPES. Для работы с этими символами вы не должны использовать, в качестве вашего основного каталога рабочих чертежей каталог PIPES. Никогда не следует использовать в таких целях и каталог ACAD (или каталог, который вы назначили вместо него). Если на приглашение операционной системы MS-DOS вы ввели команду SET ACAD=PIPES, то система AutoCAD будет искать запрашиваемый блок в каталоге \PIPES. Обратная косая черта "\" означает, что каталог присоединен к корневому каталогу текущего диска. При удалении обратной косой черты система AutoCAD будет искать каталог с именем PIPES, присоединенный к текущему рабочему каталогу. Это, вероятно, не то, что вам надо, поэтому не забывайте указывать ведущую обратную косую черту в команде SET.

Предполагается, что если диск явно не указан, то маршрут определен для текущего диска. Если для хранения библиотек отводится секционированный на разделы диск,

или несколько физических дисков, или организованный в оперативной памяти электронный диск, то в команду должен быть включен идентификатор диска:

SET ACAD=D:\PIPES

ACADCFG - этот параметр среды сообщает системе AutoCAD, где находится текущий файл конфигурации. Когда вы задаете для системы AutoCAD рабочие ограничения, система запрашивает, следует ли сохранить эти изменения в текущей конфигурации. Информация о конфигурации сохраняется в этом файле. Файл конфигурации считывается каждый раз при запуске системы AutoCAD, чтобы помочь настроить работу системы. Сперва поиск файла конфигурации осуществляется в текущем каталоге; затем, поиск производится в каталоге ACAD (или соответствующем ему); и наконец, поиск производится в каталоге, который указан в параметре ACADCFG. Каждая конфигурация могла бы храниться в отдельном каталоге. Тогда, установив значение этого параметра среды перед запуском системы AutoCAD, можно получить доступ к весьма различным конфигурациям системы AutoCAD. Это может иметь смысл, если желательно сконфигурировать ваш графопостроитель несколькими способами, в зависимости от типа чертежа, с которым вы работаете. Хотя конфигурация монитора и планшета может оставаться постоянной, для графопостроителя (в разных случаях) могло бы сохраняться различное назначение перьев, типов линий и т.д.

ACADFREERAM - значение этого параметра среды дает указание системе AutoCAD зарезервировать некоторый объем памяти для какой-либо цели. Поскольку без этой установки для большинства чертежей любого солидного размера возникают проблемы, я не имею ни малейшего понятия, почему это значение должен задавать пользователь. Установите параметр ACADFREERAM равным 24 (т.е. 24 000).

LISPSTACK — данное значение дает системе AutoCAD указание зарезервировать некоторый объем памяти под рабочий стек для интерпретатора языка АвтоЛИСП. Если ваша программа на языке АвтоЛИСП имеет высокую степень рекурсии или если вы передаете структуры данных типа массивов, это значение должно быть большим. В обычных случаях вполне устраивает устанавливаемое для этого параметра значение 8000. Сумма значений параметров LISPSTACK и LISPHEAP не должна превышать 45 000.

LISPHEAP - значение параметра среды указывает системе AutoCAD зарезервировать некоторый объем памяти для динамической области памяти, требуемой для программ на языке АвтоЛИСП. Эта область предназначена для глобальных структур данных. Для большинства приложений значение данного параметра является большим из значений параметров LISPSTACK и LISPHEAP. Вполне подходящим для рассматриваемого параметра представляется значение 37 000. Сумма значений параметров LISPSTACK и LISPHEAP не должна превышать 45 000, что уже нам знакомо.

Глава 4

СТРУКТУРА ФАЙЛОВ СИСТЕМЫ AUTOCAD

Мы рассмотрели внутреннее представление данных системы AutoCAD, и теперь настало время изучить внешнее представление данных. Цель книги - дать информацию о написании прикладных программ для системы AutoCAD. Поэтому мы должны близко познакомиться с внешними структурами данных этой системы.

Цель главы - представить читателю файлы разнообразной структуры, создаваемые или воспринимаемые системой AutoCAD. В последних главах структура этих файлов будет обсуждаться более подробно. Я разделил главы так, чтобы показать происхождение и назначение этих файлов. В гл. 5 и 6 рассматриваются файлы, создаваемые системой AutoCAD и считываемые прикладными программами. В гл. 7 и 8 рассматриваются входные файлы системы AutoCAD.

Система AutoCAD различает файлы по расширению имени файла или, иначе, по типу файла. Расширение имени файла представляет собой короткое обозначение (не более 3 букв), указываемое после имени файла. Имя файла и его расширение отделяются друг от друга точкой.

Ниже приводится краткая характеристика файлов системы AutoCAD.

DXF - файл обмена описаниями чертежей. Файл содержит информацию в коде ASCII. Структура этого файла эквивалентна структуре файла чертежа (DWG), имеющего двоичный формат. Поскольку формат DWG-файлов защищен авторскими правами, я не буду обсуждать его структуру. (Не думайте, что я хочу избежать этой темы. Я разработал коммерчески распространяемую программу, которая декодирует DWG-файлы и работает с ними. Но, чтобы сохранить корректные отношения с фирмой Autodesk, разработчиком системы AutoCAD, я решил не опубликовывать структуру DWG-файлов.) В гл. 5 и 8 рассматриваются ввод и вывод информации при работе с DXF-файлами - файлами обмена описаниями чертежей.

DXB - файл обмена описаниями чертежей (двоичный формат). Файл содержит информацию в двоичном виде. В отличие от файлов типа DXF или DWG, в нем отсутствует бесполезная информация. Преобразование значений в коде ASCII во внутренний формат может занимать относительно продолжительное время. Формат DXB-файлов позволяет быстро вводить закодированную в двоичном виде информацию, не требуя дополнительных преобразований из символьного представления в числовое. В гл. 8 можно найти более подробные сведения о формате DXB-файлов.

TXT - файл описания шаблонов атрибутов. Файл описания шаблонов атрибутов создается пользователем. Он идентифицирует атрибуты, определяет формат атрибутов и их значений, записываемых в этот файл, когда применяется команда извлечения информации об атрибутах - **ATTTEXT**.

TXT - файл вывода значений атрибутов. Существуют два типа файлов с расширением имени файла TXT, что вносит некоторую путаницу. Второй тип файла

с расширением имени файла TXT представляет собой файл вывода файла атрибутов, создаваемого по команде ATTEXT. Эти файлы различаются по имени файла. Файлу шаблонов атрибутов обычно присваивается имя, не зависящее от имени чертежа. Файл вывода значений атрибутов получает свое имя по имени чертежа. Пользователь может изменить это имя во время использования команды ATTEXT.

Вы можете определить структуру файла вывода значений атрибутов одним из трех способов. Если ваши прикладные программы работают с таким файлом, вам следует выбрать наиболее подходящую для вас структуру. Могут быть заданы три различных формата:

1. CDF - формат с использованием запятых в качестве разделителей. В этом формате все значения атрибутов отделяются друг от друга запятыми. Это обычный формат, применяемый некоторыми программами управления базами данных. Символьные строки, которые могут содержать среди прочих символов запятые, заключаются в двойные кавычки. Система AutoCAD для каждого объекта создаст одну строку информации об атрибутах. В строке будет содержаться, по крайней мере, одно значение определенного пользователем атрибута, указанного в файле шаблона атрибутов. Если объект содержит не все атрибуты из перечисленных в файле шаблона атрибутов, то система AutoCAD оставит в строке пустое поле. Пустое поле обозначают две смежные запятые.

2. SDF - формат с фиксированными полями. Хотя первоначальное его название - формат с полями, разделяемыми пробелами, может навести на мысль, что в качестве символа разделителя используется пробел, это не так. Этот формат определяет, сколько места отводится под каждое поле, что весьма схоже с заданием формата ввода-вывода в программах на Фортране.

3. DXF - формат обмена описаниями чертежей. Структура генерируемого в этом формате файла подобна структуре всех DXF-файлов. В файл записываются только объекты, содержащие заданные атрибуты (присутствует хотя бы один определенный пользователем атрибут), причем они записываются в формате DXF-файлов. Система AutoCAD представляет значения атрибутов, связанные с ее внутренними данными, как обычные поля DXF-формата (т.е. блоки, коэффициенты масштабирования, угол поворота и т.д.). Более подробно формат DXF-файлов рассматривается в гл. 6.

SCR - пакетный файл. Файлы этого типа дают простейшую возможность управлять работой системы AutoCAD извне. Пакетный файл содержит точно такие же команды системы AutoCAD и данные, которые вы могли бы ввести с клавиатуры в интерактивном режиме работы. Система AutoCAD временно переключает ввод с планшета или клавиатуры на ввод из файла, что по сути равносильно очень быстрому вводу команд обычным способом. Могут быть задействованы все команды системы AutoCAD. Однако средства идентификации отдельных объектов несколько отличаются. Поскольку отсутствует возможность непосредственного указания объекта на чертеже, выбор объекта должен осуществляться посредством задания окна, или точным указанием координат, или с помощью команд выбора из предложенных наборов LAST и PREVIOUS. Поскольку пакетный файл, вероятно, генерируется какой-либо программой, то этой программе следует знать местонахождение добавляемого объекта. Это означает, что программа может идентифицировать их местонахождение для последующего составления набора объектов. Остальные подробности будут рассмотрены в гл. 7.

PGP - файл описания параметров внешних программ. Внешние по отношению к системе AutoCAD программы (при условии, что они описаны в файле описания параметров внешних программ) могут выполняться так, как если они были бы командами этой системы. Значения, указываемые в записи PGP-файла сообщают системе AutoCAD об имени программы, требованиях по памяти, а также имени внутренней команды, по которому вызывается эта программа. Программа, описанная в PGP-файле, может быть связана с входными DXB-файлами, что позволяет осуществлять автоматическое включение в чертеж графических объектов. Более подробную информацию можно найти в гл. 8.

LSP - файл исходного текста программы на языке АвтоЛИСП. Язык АвтоЛИСП представляет собой реализованный в системе AutoCAD вариант языка Лисп. Наличие в системе встроенного языка позволяет программистам и пользователям добавлять собственные команды в набор команд системы AutoCAD. Расширение LSP в наименовании файла означает, что этот файл содержит исходный текст программы на языке АвтоЛИСП.

MNU - файл исходного текста меню. Возможность настройки системы AutoCAD под специальное приложение делает этот продукт очень гибким. Добавление меню к системе, обеспечивающее легкий доступ к специальным комбинациям команд, требуемых для выполнения какой-либо задачи, упрощает для пользователей работу с системой, а также облегчает поддержку своей разработки для автора.

MNX - файл скомпилированного меню. Для ускорения ответа на выбор из меню система AutoCAD выполняет компиляцию меню. После компиляции меню сохраняется в файле типа MNX. Система AutoCAD создает файл с расширением MNX, если для файла с расширением MNU такого файла не существует или если файл с расширением MNU образован позже, чем имеющийся файл MNX (предполагается, что было сделано изменение).

SHP - файл исходного описания графических символов и шрифта. Система AutoCAD позволяет создавать в специальном формате новые шрифты и часто используемые графические символы. Формат файла - это сокращенный графический язык для быстрого воспроизведения изображения на экране. Расширение имени файла SHP используется для файлов, содержащих исходный текст описания.

SHX - файл графических символов и шрифта. Система AutoCAD сохраняет скомпилированные файлы исходного описания графических символов или шрифта (SHP) в соответствующих файлах с расширением SHX. Этот файл содержит информацию в двоичном виде и представляет собой эквивалент SHP-файла.

Глава 5

ФОРМАТ DXF-ФАЙЛОВ

Перед тем как углубиться в подробности синтаксического разбора формата файлов обмена описаниями чертежей (DXF-файлы), будет дано краткое описание того, с чем нам придется встретиться. Файлы обмена описаниями чертежей содержат почти всю информацию, требуемую для воспроизведения чертежа. Это основной источник документированной графической информации. Первичный источник информации представляют файлы чертежей (DWG). Однако структура этих файлов не подлежит публичному оглашению.

Одним необходимым элементом, отсутствующим в DXF-файлах, является описание графических символов и шрифтов. Эти описания могут быть найдены в файлах графических символов и шрифта (SHX-файлы).

Файлы типа DXF разделяются на четыре отдельные секции. Когда вы создаете DXF-файл сами, то в него можно было бы включить только те секции, которые важны для воспроизведения вашего чертежа. Однако когда DXF-файл образуется при работе с системой AutoCAD, то в него включаются все секции, и вам требуется предусмотреть обработку всех этих секций.

Файлы типа DXF содержат следующие секции:

- секция заголовка (HEADER);
- секция таблиц (TABLE);
- секция блоков (BLOCKS);
- секция объектов (ENTITIES).

Секция заголовка (обзор)

Секция заголовка содержит значения, определяющие рабочие параметры чертежа и его окружения. Более подробный список этих параметров приведен в приложении Б. Для пояснения можно назвать некоторые из значений, которые содержатся в секции заголовка, например:

- номер версии системы AutoCAD;
- координаты габаритов чертежа;
- координаты границ чертежа;
- параметры размеров;
- имя текущего меню.

Полезность этой информации зависит от вашего конкретного приложения. Я обычно использую координаты габаритов чертежа и координаты границ чертежа, игнорируя остальные значения. Эти значения я применяю для определения коэффициентов масштабирования.

ния при преобразовании чертежей, полученных с помощью системы AutoCAD для вывода на фотокоординатограф.

Еще одним важным значением, содержащимся в DXF-файле, является номер версии системы. Запись с номером версии является ключевой по отношению к другим значениям, содержащимся в файле типа DXF. Программе преобразования формата DXF-файлов последних версий в формат DXF-файлов более ранних версий (для сохранения преемственности с форматом предыдущих версий системы) необходимо найти запись, содержащую номер версии системы AutoCAD, а затем опускать любые значения в DXF-файле, неиспользуемые в старых версиях системы.

Секция таблиц (обзор)

Если в секции заголовка содержатся простые рабочие значения, то в секции таблиц информация разбита по группам, представляющим более сложные рабочие значения чертежа, созданного в системе AutoCAD. Секция таблиц разделяется на четыре подсекции. В каждой подсекции может быть несколько записей. Ниже перечислены эти подсекции:

- тип линий (LINETYPE);
- слой (LAYER);
- шрифт (STYLE);
- вид (VIEW).

В каждой подсекции содержится столько записей, сколько необходимо для поддержки всей имеющейся графической информации. Если в чертеже вы используете 10 слоев (включая слой 0), то в таблице слоев будет 10 записей.

Таблица типов линий LINETYPE задает типы линий, используемые в вашем чертеже. Тип линии определяется с помощью команды LINETYPE. Общепринятыми являются следующие типы линий: пунктирная (DOT), штриховая (DASHED), тонкая (PHANTOM) и штрихпунктирная (DOTDASH). По умолчанию используется непрерывная линия (CONTINUOUS); этот тип линии всегда включается в DXF-файл. Подсекция типа линии — LINETYPE — содержит пример представления линии данного типа на чертеже. В этом примере изображение типа линии выполняется условно, с помощью знаков символического кода ASCII.

Таблица слоев LAYER содержит определения для каждого слоя чертежа. Эта структура данных является основным источником получения информации о цвете, типе линий и их видимости. Создание слоев и присвоение им наименований осуществляется с помощью команды LAYER/NEW. Система AutoCAD всегда включает описание слоя 0 (слой, используемый по умолчанию). Наименование слоя 0 должно обрабатываться как символическая строка, а не как целочисленное значение. Это делает легким сравнения и позволяет

сочетать числовые и символьные имена. В первых версиях системы, такой, как, например, AutoCAD версия 1.4, имена слоев были числовыми. Изменение в наименовании слоев произошло в AutoCAD, версия 2.0.

Таблица шрифтов **STYLE** содержит информацию о текстовых шрифтах и графических символах. Используемые по умолчанию характеристики указываются в каждой записи таблицы. Имя шрифта может совпадать с именем файла описания шрифта. Поскольку на основе одного шрифта возможно получить несколько стилей начертания шрифта, в записи таблицы шрифтов имя стиля начертания шрифта сохраняется отдельно. Система AutoCAD включает в DXF-файл информацию о стиле начертания шрифта, используемого по умолчанию (txt).

Таблица видов **VIEW** содержит список поименованных видов и точек наблюдения, добавленных во время сеанса работы с системой AutoCAD.

Секция блоков (обзор)

Секция блоков DXF-файла содержит подробную информацию об объектах для всех определенных в чертеже блоков. Эти блоки могут быть образованы при выполнении операций нанесения размеров, штриховки, вставки изображений из графических библиотек или просто быть новыми блоками, созданными во время сеанса работы с системой AutoCAD. В секции блоков остаются даже те блоки чертежа, которые были стерты с экрана дисплея при работе с системой AutoCAD.

Данные внутри секции блоков организованы следующим образом: каждый блок начинается с имени блока и зафиксированной точки привязки данного блока, затем следует список объектов, входящих в этот блок.

В секции блоков каждый блок определяется индивидуально. Хотя при вставке блока возможно, чтобы определение этого блока входило в определение другого блока, ни один из блоков не определяется внутри других. В DXF-файлах отсутствуют вложенные определения блоков.

Секция объектов (обзор)

В секции объектов содержится список всех активных графических объектов. В секции сохраняются все подробности, необходимые для воспроизведения чертежа. Поскольку имеются постоянные ссылки на секции блоков и таблиц, данные секции уже определены и готовы к тому, чтобы быть использованными в отдельных графических объектах. Каждый объект имеет свой собственный формат описания. Фор-

мат графических объектов подробно рассматривается в последних разделах главы.

Синтаксический разбор DXF-файлов

Теперь мы имеем общее представление о структуре DXF-файлов и можем перейти к изучению частных вопросов, включая ввод данных из этих файлов и их синтаксический разбор.

В целом DXF-файл построен как структура данных, в которой используется соотношение маркер — значение. Маркер представляет собой целочисленное значение, обозначающее тип (целый, действительный или символьная строка) и основное назначение информации (имя, координаты или флажки), содержащейся в структуре данных. Зная тип следующего за маркером значения данных, вы можете передавать обработку соответствующей программе преобразования. Все значения вводятся как символы в коде ASCII. Преобразование этих значений в соответствующий внутренний формат представления зависит от ваших потребностей. В DXF-файле маркер и соответствующее ему значение помещаются на отдельных строках.

Формат записи маркера соответствует применяемому в языке Фортран формату I3. Это означает, что маркер является целым числом, занимающим три символьных позиции и выравненным по правой границе поля. Например, если маркер состоит из одной цифры, то перед этой цифрой должны быть два пробела. Этот стандарт используется при выводе информации в системе AutoCAD, а также в некоторых прикладных программах. Большинство прикладных программ не придерживаются строго данного стандарта, поскольку в таких языках, как Паскаль и Си ведущие пробелы всегда опускаются. В системе AutoCAD при вводе информации из DXF-файлов, созданных при работе других прикладных программ, также не требуется строгое соблюдение этого стандарта.

В AutoCAD применяется весьма строгая система маркеров, определяющих тип данных, что делает достаточно простым извлечение только тех порций данных, которые необходимы для последующей обработки. Но это также означает, что ваши программы должны чрезвычайно корректно выполнять считывание и запись информации при работе с DXF-файлами.

В качестве примера рассмотрим файл с некоторой упрощенной структурой, подобной структуре DXF-файлов, основанной на использовании соотношений маркер-значение, а также исходный текст программы, осуществляющей синтаксический разбор этого файла. На лист. 5.1 приведен пример такого простого DXF-файла.

```
2
APPLE
10
5.126
```

Лист. 5.1. Простой DXF-файл

Файл содержит две группы данных—имя APPLE и действительное число 5.216. Перед каждым из значений помещен маркер. В данном примере число 2 является маркером, помечающим, что за ним следует имя—символьная строка. Число 10 используется как маркер для действительного числа.

Фрагмент программы, предназначенный для распознавания такой структуры файлов, представляет собой довольно простой цикл. Цикл типа WHILE выполняется до тех пор, пока не будет достигнут конец файла. В начальной части цикла выполняется считывание информации из входного файла с целью извлечь значение маркера. Мы будем иметь большие неприятности, если при считывании строки, вводимой с помощью первой в цикле функции fscanf, не встретится один из ожидаемых маркеров (в том числе, признак конца файла EOF). Исходный текст программы для ввода данных нашего простого примера приведен на лист. 5.2.

```
read_dxf(dxf)
FILE      *dxf;
{
    int      tag;
    char      name[256];
    char      garbage[256];
    double    real;

    while(fscanf(dxf,"%d",&tag) != EOF)
        switch (tag)

            case 2 : fgets(dxf,name,256);
                     break;
            case 10 : fscanf(dxf,"%f",&real);
                     break;
            default : fgets(dxf,garbage,256);
                     break;
}
```

Лист. 5.2. Простая программа ввода DXF-файла

Если бы маркер был пропущен, то мы могли бы потерять управление в программе и результат был бы непредсказуем. Для большей уверенности вы можете добавить в программу проверку корректности значений маркера. К счастью, если файл был создан системой AutoCAD или некоторыми другими правильно работающими программами, структура такого файла должна быть корректной. Ошибки этого рода могут просочиться в случае кодирования данных вручную. Но для нашего простого примера подобных проблем не предвидится.

Теперь рассмотрим, как работает эта маленькая программа. Считав значение маркера и преобразовав его в целочисленную форму (строка программы с оператором WHILE), мы используем маркер для выбора способа ввода информации и типа требуемого преобразования значения. Это осуществляется в операторе выбора switch. Для ввода строк я предпочитаю применять функцию fgets. Эта функция помещает всю строку целиком в буфер. Буфер описан как локальная переменная. С другой стороны, функция fscanf могла бы приостановить считывание входного потока данных при обнаружении первого пробела. Функция fscanf хорошо подходит для получения одиночного имени (APPLE) из входной строки. Для более сложных строк (типа—AN APPLE A DAY) это приведет к катастрофе, поскольку функция fscanf считывает только первое слово в строке (AN). Второе слово (APPLE) будет оставлено во входном потоке данных и будет обрабатываться как следующий маркер, что нарушит последовательность обработки входного потока. Произойдет катастрофа. Используемые в системе AutoCAD специальные маркеры позволяют определять, ожидается ли ввод одиночного имени или сложной строки.

В табл. 5.1 и 5.2 приведены значения маркеров, используемых в системе AutoCAD. Маркеры имеют специфические назначения, и, чтобы быть правильными, они должны появляться в нужном контексте. Отсутствие допустимого маркера вызовет сообщение об ошибке системы AutoCAD и прекращение ввода информации из DXF-файла.

Таблица 5.1. Маркеры используемые в DXF-файлах (упорядоченные по значению)

0 - 9	Строки (имена, атрибуты, текст)
10 - 59	Действительные числа (координаты, углы, расстояния)
60 - 70	Целые числа (флажки, счетчики, варианты выбора)

Таблица 5.2. Маркеры, используемые в DXF-файлах

0	Идентифицирует начало раздела в DXF-файле. Используется как начало секций, для секций таблиц TABLES и объектов ENTITIES. Следующее после маркера значение представляет строку без пробелов
1	Основной текст. Используется для таких инструкций, как TEXT, ATTDEF или ATTRIB. Текст помещается после данного маркера и представляет собой строку, которая может включать пробелы
2	Имя. После данного маркера следует имя блока или маркер атрибута. В инструкции INSERT этот маркер используется для наименования вставляемого блока. Следующее за маркером значение - строка, которая

может содержать пробелы при использовании с инструкцией ATTRIBUTE. Если строка представляет имя блока, то в ней не должно быть пробелов

- 3-5 Прочие текстовые значения. После этих маркеров можно обнаружить значения атрибутов по умолчанию или текст приглашения ко вводу. Маркер 3 также применяется для обозначения имени шрифта в записи стиля начертания шрифта. Следующее за маркером значение представляет собой строку, в которой могут содержаться пробелы
- 6 Имя типа линии. Следующее после маркера значение представляет собой строку без пробелов
- 7 Имя стиля начертания текста. Следующее после маркера значение – строка без пробелов
- 8 Имя слоя. Один из наиболее распространенных маркеров в DXF-файлах. Следующее после маркера значение представляет собой строку без пробелов
- 9 Применяется с секции заголовка HEADER для обозначения имени переменной. Соответствующим маркеру значением является строка без пробелов
- 10 Основная координата по оси X. Соответствующее маркеру значение является действительным числом, представленным в виде строки цифр в коде ASCII
- 11-18 Прочие координаты по оси X. Используются для идентификации второго конца отрезка, углов для полосы или фигуры, а также вершин для трехмерных объектов (3DFACE). Соответствующим маркеру значением является действительное число, представленное в виде строки цифр в коде ASCII
- 20 Основная координата по оси Y. Соответствующее маркеру значение является действительным числом, представленным в виде строки цифр в коде ASCII
- 21-28 Прочие координаты по оси Y. Используются так же, как и прочие координаты по оси X. Соответствующим маркеру значением является действительное число, представленное в виде строки цифр в коде ASCII
- 30 Основная координата по оси Y. Это пока еще не стало распространенным. Соответствующее маркеру значение

- является действительным числом, представленным в виде строки цифр в коде ASCII
- 31-36 Прочие координаты по оси Z. Используются совместно с прочими координатами по осям X и Y. Соответствующим маркеру значением является действительное число, представленное в виде строки цифр в коде ASCII
- 38 Уровень возвышения объекта (если не равен 0.0). Соответствующее маркеру значение является действительным числом, представленным в виде строки цифр в коде ASCII
- 39 Высота (толщина) объекта (если не равна 0.0). Соответствующее маркеру значение является действительным числом, представленным в виде строки цифр в коде ASCII
- 40-48 Прочие действительные числа. Соответствующее маркеру значение является действительным числом, представленным в виде строки цифр в коде ASCII
- 49 Разнообразные значения. Маркер используется в определениях типа линии, для задания длины отдельных видимых и невидимых отрезков в штриховых и пунктирных линиях. Соответствующее маркеру значение является действительным числом, представленным в виде строки цифр в коде ASCII
- 50-58 Углы (градусы). Соответствующее маркеру значение является действительным числом, представленным в виде строки цифр в коде ASCII
- 62 Номер цвета. (Маркер встречается в описаниях слоя LAYER и в описаниях отдельных объектов, если их цвет отличается от цвета слоя.) Соответствующим маркеру значением является символьная строка, представляющая номер цвета. Также могут использоваться значения BYLAYER и BYBLOCK
- 66 После маркера следует дополнительная информация. Данный маркер обычно применяется, чтобы показать, что далее следуют атрибуты объекта. Кроме того, маркер также появляется в описании полилинии, чтобы показать, что далее следуют описания ее вершин
- 70-78 Целочисленные значения Соответствующее маркеру

значение является целым числом, представленным в виде строки цифр в коде ASCII

Мы ознакомились с идеей применения пар маркер — значение для передачи программе указания, как должны интерпретироваться вводимые данные. Файлы DXF-формата целиком строятся из таких простых пар. На следующем уровне сложности система AutoCAD комбинирует вместе имя группы и сопутствующее ему значение. Каждое имя группы и связанное с ним значение представляют собой пару маркер — значение. Например, объект LINE имеет, по крайней мере, пять пар маркер — значение для группы линий. На лист 5.3 приведена простая группа из объектов LINE.

Я сказал "Простая группа линий", поскольку типичная группа линий может содержать информацию о слое, цвете, ширине и угле наклона. В рассматриваемом примере, маркер 0 помечает начало объекта. Маркер 0 также используется для обозначения начала блоков, секций и таблиц, но это пока не рассматривается. Следующее за маркером 0 значение даст информацию о типе объекта, описанного в оставшейся части группы. Обнаружив инструкцию LINE, программа вызывает подпрограмму для обработки объектов такого типа. Приведенная в примере линия проведена между точками с координатами 1.0, 1.5 и 6.32, 7.233. Оставшиеся пары маркер — значение дают информацию об этих координатах. Подробности будут рассмотрены в дальнейшем. Сейчас я только намечаю варианты того, что может встретиться в DXF-файлах.

```
0
LINE
10
1.0
20
1.5
11
6.32
21
7.223
```

Лист. 5.3. Группа LINE в файле типа DXF

Считывание файлов формата DXF

Перед тем как приступить к детальному разбору каждой из секций DXF-файлов, взглянем на полную структуру всего файла. Изучая эту структуру, мы постепенно придем к рассмотрению деталей различных секций.

До сих пор мы рассматривали общие стратегические особенности структуры DXF-файлов. На очереди изучение деталей формата DXF-файлов.

Фирма Autodesk разработала структуру DXF-файлов таким образом, чтобы одна центральная программа управляла активизацией программ синтаксического разбора секций. Программа синтаксического разбора представляет собой такую программу, которая "разбивает на части" или выделяет части структуры так, чтобы можно было получить нужную информацию.

Когда вы разбираете какое-либо предложение по частям (Вспомнили школьные упражнения?), то вы разбиваете его на части подлежащее — сказуемое — фраза. Синтаксический разбор DXF-файлов легче, чем разбор предложения на естественном языке. Известно, что в DXF-файлах все секции начинаются с пары: маркер секции и метка. На лист. 5.4 показано, как это выглядело бы в файле.

Когда ваша программа обнаруживает одну из этих пар, ей надо начать поиск типа секции. Эта информация следует сразу после такой пары. Ей предшествует маркер 2. На лист. 5.5 представлен заголовок, используемый для объявления секции HEADER.

0
SECTION

Лист. 5.4. Заголовок SECTION

2
HEADER

Лист. 5.5. Заголовок секции HEADER

Когда программа распознает конкретную секцию - встречен помеченный в начале секции маркер 2, то она вызывает для обработки соответствующую секцию программы синтаксического разбора. В рассматриваемом примере программа вызывает программу разбора секции заголовка (HEADER). Информация, следующая после имени секции, показывает некоторую последовательность, имеющую значение для специальных подпрограмм синтаксического разбора.

По установленным для DXF-формата правилам в файле также помечается конец секции. Признак конца секции служит для обрабатываемой программы сигналом передачи управления обратно центральной программе синтаксического разбора. После чего центральная программа подготавливается к распознаванию очередной секции и вызову соответствующей программы разбора секций типа HEADER, TABLE, BLOCK или ENTITIES.

Конец секции помечается парой маркер и метка. В качестве маркера используется число 0. На лист. 5.6 показана последовательность данных, обозначающих конец секции.

Я хотел бы познакомить читателя с совокупностью подпрограмм, которые постоянно используются в программах обработки DXF-файлов. Эти программы обеспечивают нижний уровень обработки и выполняют обработку маркеров и значений, используемых в DXF-файлах.

Основные подпрограммы и структуры данных для разбора DXF-файлов

В текстах программ в оставшейся части главы (или книги, если затрагиваются эти вопросы) будет неоднократно использоваться совокупность подпрограмм для ввода информации из DXF-файлов. В последующих многочисленных фрагментах программ используется информация, полученная и в некоторых случаях немного преобразованная, только с помощью двух подпрограмм - `get_tag` и `get_tag_token`.

Подпрограмма `get_tag` (лист. 5.7) представляет собой непосредственный интерфейс с DXF-файлами. Она выполняет довольно простую работу - считывает некоторые символы в коде ASCII из файла и удаляет избыточные пробелы, символы табуляции и перевода строки. После завершения работы программа возвращает указатель на начало введенного текста. Данная подпрограмма чрезвычайно примитивна и даже не производит распознавание признака конца файла (EOF), когда он встречается. В нашем примере программа "предполагает", что DXF-файл содержит всю необходимую информацию, и автоматически включает запись, содержащую признак EOF, в конец введенного текста. Читатель может ввести более строгие ограничения и породить специальный маркер и значение для обозначения конца файла, если тот будет встречен раньше времени.

```
char *get_tag(dxf)
FILE      *dxf;
{
    int      i;

    /*                                     */
    /* Считать строку текста из DXF-файла */
    /* Длина строки ограничивается 255 символами */
    /* Удалить в конце строки символ перевода строки */
    /* Найти первый символ, следующий после пробелов */
    /* и символов табуляции */
    /*                                     */
}
```



```

/*
fgets(tag_str,256,dxf);
if (tag_str[strlen(tag_str)-1] == '\n')
    tag_str[strlen(tag_str)-1] = '\0';
i = strstrn(tag_str," \t");
return &tag_str[i];
}

```

Лист. 5.7. Функция `get_tag`

С другой стороны, функция `get_tag_token` представляет собой интерфейс для распознавания маркеров и трансляции синтаксических единиц (лексем) для остальных частей программы. Получив с помощью функции `get_tag` имя маркера, для сокращения числа имен лексем, рассматриваемая функция использует код перемешивания, который должен применяться при поиске, для нахождения указанного имени по запросу. При обнаружении имени, которое соответствует имени маркера, применяемому в формате DXF-файлов, функция возвращает связанное с этой лексемой число. Функция возвращает значение 0, если соответствие не найдено, что бывает достаточно часто в случае обнаружения в DXF-файлах имен и значений, которые не являются лексемами. Исходный текст функции `get_tag_token` приведен на лист. 5.8.

```

int get_tag_token (s)
char      *s;
{
    tok_tab      *t;

    /* Вычислить код перемешивания для слова -- s --,
    /* чтобы определить к какому классу относится это слово */
    /* Получить указатель -- t -- на список слов
    /* Просмотреть каждое слово в списке (используя ссылку
    /* NEXT для получения следующего слова)
    /* Если написание слов совпадает (strcmp == 0),
    /* вернуть соответствующее внутреннее значение
    /* Вернуть значение 0, если слово не найдено

    t = tok_hash[hash(s)];
    while (t)
    {
        if (!strcmp(t->tok_name,s)) return t->tok_value;
        t = t->next;
    }
    return 0;
}

```

Лист. 5.8. Функция `get_tag_token`

Для инициализации таблицы имен лексем имеется функция `init_token_table`. Данная функция должна выполняться в начале основной программы. Эта функция делает доступными при выполнении синтаксического разбора имена, встречающиеся в формате DXF в полях значений. На лист. 5.9 представлен исходный текст функции `init_token_table`. Текст функции содержит ссылку на таблицу лексем `tok_table`. Еще один листинг (см. лист. 5.12) включает определения для этой таблицы.

```
init_tok_table()
{
    tok_tab      *t;
    int          j;

    /* Начальные значения для таблицы лексем, включают */
    /* образцы написания имен и внутренние значения */
    /* Значение NULL используется в качестве значения */
    /* ссылки NEXT для связи слов, имеющих одинаковое */
    /* значение кода перемешивания */
    /* */
    /* Для каждого значения в списке имен лексем */
    /* (пока имя не равно NULL) */
    /* */
    /* Вычислить код перемешивания для определения класса */
    /* Связать имя, с помощью ссылки NEXT, с записью */
    /* в таблице перемешивания, соответствующей данному */
    /* классу */
    /* */
    /* */

    t = tok_table;
    while (t->tok_name)
    {
        j = hash(t->tok_name);
        t->next = tok_hash[j];
        tok_hash[j] = t;
        t++;
    }
}
```

Лист. 5.9. Инициализации таблицы лексем

Поскольку список определений для целочисленных значений лексем довольно большой, то приведенные в нем соответствия между именами маркеров и значениями больше нигде в книге повторяться не будут. Используемые в определениях значения, предназначенные для внутреннего применения, являются произвольными, за исключением значения `NONE`, которое всегда должно быть 0. При создании этой таблицы я взял все обычно встречающиеся слова и име-

на (числам я тоже дал имена), расположил их в алфавитном порядке и присвоил значения лексем.

Во всех программах в книге используются эти определения. Если я попытался бы изменить внутренние значения, то единственное, что при этом мне надо было сделать, это скорректировать файл, содержащий определения (defines.h), и после вновь скомпилировать всю программу целиком. На лист. 5.10 приведены определения значений лексем, на лист. 5.11 приведено объявление типа данных, а на лист. 5.12 представлены объявления для переменных.

#define FALSE	0	#define POLYLINE	33
#define TRUE	1	#define SECTION	34
		#define SEVEN	35
#define NONE	0	#define SEVENTY	36
#define ARC	1	#define SEVENTY1	37
#define ATTDEF	2	#define SEVENTY2	38
#define ATTRIB	3	#define SEVENTY3	39
#define BLOCK	4	#define SHAPE	40
#define BLOCKS	5	#define SIX	41
#define CIRCLE	6	#define SIXTY2	42
#define EIGHT	7	#define SIXTY6	43
#define ELEVEN	8	#define SOLID	44
#define ENDBLK	9	#define STYLE	45
#define ENDSEC	10	#define TABLE	46
#define SEQEND	11	#define TABLES	47
#define ENDTAB	12	#define TEN	48
#define ENTITIES	13	#define TEXT	49
#define EOFILE	14	#define THIRTEEN	50
#define FIFTY	15	#define THIRTY1	51
#define FIFTY1	16	#define THREE	52
#define FORTY	17	#define TRACE	53
#define FORTY1	18	#define TWELVE	54
#define FORTY2	19	#define TWENTY	55
#define FORTY3	20	#define TWENTY1	56
#define FORTY4	21	#define TWENTY2	57
#define FORTY5	22	#define TWENTY3	58
#define FORTY9	23	#define TWO	59
#define FOUR	24	#define VERTEX	60
#define HEADER	25	#define VIEW	61
#define INSERT	26	#define ZERO	62
#define LAYER	27	#define LIMMIN	63
#define LINE	28	#define EXTMIN	64
#define LTYPE	29	#define DIMENSION	65
#define NINE	30	#define LIMMAX	66
#define ONE	31	#define EXTMAX	67
#define POINT	32		

Лист. 5.10. Определения значений лексем

```

typedef
struct    z
{
    struct z  *next;
    char      *tok_name;
    int       tok_value;
}           tok_tab;

```

Лист. 5.11. Объявление типа для структуры данных лексемы

```

tok_tab      tok_table[] = { {NULL, "0", ZERO},
                              {NULL, "1", ONE},
                              {NULL, "10", TEN},
                              {NULL, "11", ELEVEN},
                              {NULL, "12", TWELVE},
                              {NULL, "13", THIRTEEN},
                              {NULL, "2" ,TWO},
                              {NULL, "20", TWENTY},
                              {NULL, "21", TWENTY1},
                              {NULL, "22", TWENTY2},
                              {NULL, "23", TWENTY3},
                              {NULL, "3" ,THREE},
                              {NULL, "32", THIRTY1},
                              {NULL, "4" ,FOUR},
                              {NULL, "40", FORTY},
                              {NULL, "41", FORTY1},
                              {NULL, "42", FORTY2},
                              {NULL, "43", FORTY3},
                              {NULL, "44", FORTY4},
                              {NULL, "45", FORTY5},
                              {NULL, "49", FORTY9},
                              {NULL, "50", FIFTY},
                              {NULL, "51", FIFTY1},
                              {NULL, "6", SIX},
                              {NULL, "62", SIXTY2},
                              {NULL, "66", SIXTY6},
                              {NULL, "7", SEVEN},
                              {NULL, "70", SEVENTY},
                              {NULL, "71", SEVENTY1},
                              {NULL, "72", SEVENTY2},
                              {NULL, "73", SEVENTY3},
                              {NULL, "8" ,EIGHT},
                              {NULL, "9", NINE},
                              {NULL, "ARC", ARC},
                              {NULL, "ATTDEF", ATTDEF},
                              {NULL, "ATTRIB", ATTRIB},
                              {NULL, "BLOCK", BLOCK},
                              {NULL, "BLOCKS", BLOCKS},

```

```

{NULL, "CIRCLE", CIRCLE},
{NULL, "DIMENSION", DIMENSION},
{NULL, "ENDBLK", ENDBLK},
{NULL, "ENDSEC", ENDSEC},
{NULL, "$EXTMIN", EXTMIN},
{NULL, "$EXTMAX", EXTMAX},
{NULL, "SEQEND", SEQEND},
{NULL, "ENDTAB", ENDTAB},
{NULL, "ENTITIES", ENTITIES},
{NULL, "EOF", EOF},
{NULL, "HEADER", HEADER},
{NULL, "INSERT", INSERT},
{NULL, "LAYER", LAYER},
{NULL, "LINE", LINE},
{NULL, "$LIMMIN", LIMMIN},
{NULL, "$LIMMAX", LIMMAX},
{NULL, "LTYPE", LTYPE},
{NULL, "ONE", ONE},
{NULL, "POINT", POINT},
{NULL, "POLYLINE", POLYLINE},
{NULL, "SECTION", SECTION},
{NULL, "SHAPE", SHAPE},
{NULL, "SOLID", SOLID},
{NULL, "STYLE", STYLE},
{NULL, "TABLE", TABLE},
{NULL, "TABLES", TABLES},
{NULL, "TEXT", TEXT},
{NULL, "TRACE", TRACE},
{NULL, "VERTEX", VERTEX},
{NULL, "VIEW", VIEW},
{NULL, NULL, 0}};

```

```

double      limit_x = 0.0;
double      limit_y = 0.0;
double      extent_x = 0.0;
double      extent_y = 0.0;
double      limit_mx = 0.0;
double      limit_my = 0.0;
double      extent_mx = 0.0;
double      extent_my = 0.0;
char        tag_str[40];

```

Лист. 5.12. Объявления лексем и переменных

Теперь о том, как это все работает. Предположим, что выполняется поиск отрезка - геометрического объекта, выражающего кратчайшее расстояние между двумя точками. В DXF-файле отрезок идентифицируется ключевым словом **LINE**. Первое, что надо сделать для

поиска отрезка, это получить очередной маркер. На лист. 5.13 приведена строка исходного текста программы, в которой производится операция выборки маркера из DXF-файла.

```
tag = get_tag(dxf);
```

Лист. 5.13. Вызов функции `get_tag`

Имея маркер (указатель на символьную строку), мы хотим получать целочисленное значение для этого маркера. Эту работу выполняет функция `get_tag_token`. Она сравнивает значение строки, в которой содержится маркер, со всеми предполагаемыми именами маркеров. Она использует при этом функцию перемешивания, чтобы ограничить число проверяемых слов. При обнаружении соответствия функция возвращает внутреннее значение как лексему (лист. 5.14).

```
token = get_tag_token(tag);
```

Лист. 5.14. Вызов функции `get_tag_token`

Поскольку имя "LINE" связано с определенным в таблице лексем целочисленным значением LINE, то при нахождении слова "LINE" в DXF-файле, функция `get_tag_token` должна возвратить целочисленное значение LINE. В исключительной ситуации, когда осуществляется поиск только отрезков (LINE) в DXF-файле, можно вставить в текст программы следующую строку (лист. 5.15)

```
if (get_tag_token(get_tag(dxf)) == LINE)
```

Лист. 5.15. Использование функции получения лексемы для нахождения объекта LINE

Имеется маркер 0, который предшествует значению LINE. Программе следует распознавать этот маркер 0 перед проверкой на наличие слова LINE. Этот контроль позволит избежать затруднений при нахождении слоя (LAYER) с именем LINE или текста, содержащего слово LINE. Оба эти варианта допустимы и являются достаточным общим случаем.

Более логичным является изменение потока управления в программе соответственно с тем, как это определяется значением лексемы. Тогда программа сможет распознавать и обрабатывать объекты нескольких типов. Пример, иллюстрирующий это, представлен на лист. 5.16.

```
switch(get_tag_token(get_tag(dxf)))
{
    case LINE      : line_processor(dxf);
                    break;
    case CIRCLE    : circle_processor(dxf);
                    break;
```

```

case ARC      : arc_processor(dxf);
                break;
case POINT    : point_processor(dxf);
                break;
default       : break;
}

```

Лист. 5.16. Простая программа синтаксического разбора объектов

Центральная управляющая программа синтаксического разбора

Центральная управляющая процедура представляет собой небольшую программу, которая по мере необходимости вызывает на выполнение различные программы детального разбора получаемой из DXF-файла информации. Конечно, программы детального синтаксического разбора должны взаимодействовать так, чтобы управление, как обычно, возвращалось центральной управляющей программе. На рис. 5.1 представлена диаграмма конечных состояний для программы синтаксического разбора. Поскольку на этой стадии нашей работы взят не очень сложный по структуре DXF-файл, программа разбора не делает многого. На лист. 5.17 приведен центральный управляющий модуль. Программа после открытия DXF-файла запускает "мяч в игру", вызывая центральную процедуру синтаксического разбора и передавая ей ссылку на DXF-файл (лист. 5.18). В конце концов, после того как DXF-файл будет полностью прочитан, управление будет возвращено основной программе. В приведенном тексте программы обнаружение значения EOF (конец файла) приведет к тому, что программа завершит свой цикл.

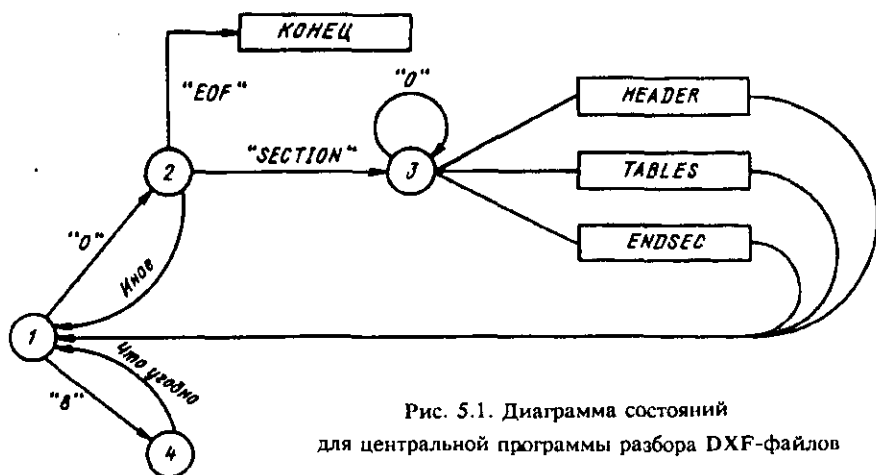


Рис. 5.1. Диаграмма состояний для центральной программы разбора DXF-файлов

```

dxfin(f)
FILE      *f;
{
char      buf[132];
int       state;
int       quit;
int       t;
matrix    m1;
matrix    m2;

quit = FALSE;
state = 1;
while (!quit)
{
t = get_tag_token(get_tag(f));
switch (state)
{
case 0: quit = TRUE;
break;

case 1: if (t == ZERO) state = 2;
if (t != EIGHT) state = 4;
break;

case 2: state = 1;
if (t == SECTION) state = 3;
if (t == EOF) quit = TRUE;
break;

case 3: switch(t)
{
case HEADER : header(f); state = 1;
break;

case TABLES : tables(f); state = 1;
break;

case ENDSEC : state = 1;
break;

}
break;

case 4: state = 1;
break;

}
}
}

```

Лист. 5.17. Центральный управляющий модуль


```

strcpy (fn_dxf.path);
strcat (fn_dxf, ".dxf");
init_tok_table();
if (fd = fopen (fn_dxf, "r"))
{
    dxf_in (fd);
    fclose(fd);
}
else
{
    printf (" DXF-файл <%s> не найден\n", fn_dxf);
    printf (" Для продолжения, нажмите любую клавишу\n");
    c = getch();
}

```

Лист. 5.18. Начало ввода DXF-файла в основной программе

Чтение секции заголовка HEADER

Секция заголовка (HEADER) содержит специфические рабочие значения для чертежа. Некоторая информация из этой секции может оказаться полезной для вас, а некоторая не очень. Существуют условия, когда секция HEADER может быть проигнорирована. Это довольно просто и часто использовалось в моих программах. Рассмотрим, как это можно сделать. Чтобы проигнорировать обработку секции HEADER, просмотрим все маркеры, используемые в этой секции, до появления значения последнего - маркера конца секции ENDSEC. Прекратим выполнение программы, когда будет найдено значение ENDSEC. На лист. 5.19 описан алгоритм для реализации такой стратегии.

Повторять

Получить маркер из DXF-файла

Получить значение из DXF-файла

Если маркер ZERO, а значение ENDSEC --> Завершить!

До завершения

Лист. 5.19. Стратегия чтения секции заголовка DXF-файла

Необходимо всегда получать следующее после маркера значение. Ввод значений каждого поля так, как если бы это было поле маркера, может привести к неприятностям. В данном примере мы осуществляем поиск маркера ZERO. Если мы считываем каждое поле, то вполне возможно найти значение ZERO (показывающее установку флажка или слой 0) и случайно обработать его как маркер. Тогда при попытке обнаружить следующее после неверно считанного

ZERO значение **ENDSEC** считывается маркер. Это не будет катастрофой, если маркер, который был ошибочным для следующего значения, не является маркером **ZERO** для значения **ENDSEC**. При этом вы потеряете синхронизацию и программа не будет работать корректно.

На лист. 5.20 приведен исходный текст программы для считывания и пропуска секции **HEADER**. Центральная управляющая процедура вызывает данную процедуру, передавая ей в качестве аргумента ссылку на **DXF**-файл. Процедура выполняет чтение **DXF**-файла до тех пор, пока не встретится первая запись, содержащая **ENDSEC**. Это значение **ENDSEC** помечает конец секции заголовка.

```
header (f)
FILE      *f;
{
    int      quit;
    int      state;
    int      t;
    char      *c;

    quit = FALSE;
    state = 1;
    while (!quit)
    {
        t = get_tag_token(c = get_tag(f));
        switch (state)
        {
            case 1:    state = 3;
                       if (t == ZERO) state = 2;
                       break;
            case 2:    if (t == ENDSEC) quit = TRUE; state = 1;
                       break;
            case 3:    state = 1;
                       break;
        }
    }
}
```

Лист. 5.20. Структура программы для чтения секции **HEADER** **DXF**-файла

Имеется ряд случаев, когда желательно знать некоторое специфическое значение в **DXF**-файле. Если это так, то вы захотите найти и получить это специфическое значение. Мы опишем конечный автомат для распознавания полей в **DXF**-файле. Поскольку нельзя быть уверенными в том, какое из значений будет в действительности присутствовать в **DXF**-файле, каков порядок появления значений или форма (тип данных), в которой они будут представлены, то использование конечного автомата является нашим наиболее практичным

методом, обеспечивающим обработку при всех непредвиденных обстоятельствах.

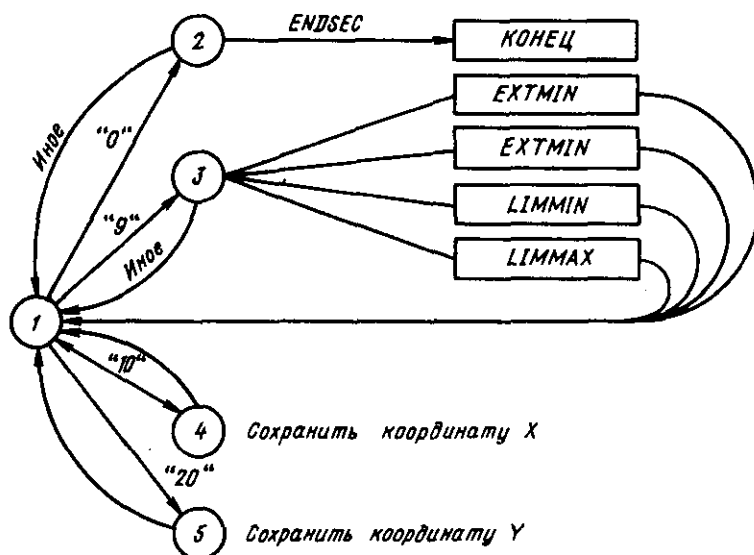


Рис. 5.2. Диаграмма состояний для программы разбора секции HEADER

На рис.5.2 приведена диаграмма конечных состояний программы синтаксического разбора секции заголовка. В действительности мы знаем форму, в которой данные будут появляться в DXF-файлах. Это данные в коде ASCII. Однако рабочие значения данных могут находиться в некоторой другой внутренней форме представления - целые числа, символы или числа с плавающей точкой. На лист. 5.21 представлен алгоритм, описывающий стратегию нахождения значений в секции HEADER DXF-файла. На лист. 5.22 приведен исходный текст программы, реализующей этот алгоритм.

Получить маркер из DXF-файла

Получить значение из DXF-файла

Если маркер NINE и значение выбрано из нужного поля

установить соответствующие указатели на введенные значения

Иначе

Если маркер один из следующих:

10 - сохранить координату X (действительное число)

20 - сохранить координату Y (действительное число)

30 - сохранить координату Z (действительное число)

50 - сохранить угол (действительное число)

Лист. 5.21. Стратегия нахождения значений в секции HEADER DXF-файла

```

header (f)
FILE    *f;
{
int      quit;
int      state;
int      t;
char     *c;
double   p2;

p1 = NULL;
p2 = NULL;
quit = FALSE;
state = 1;
while (!quit)
{
t = get_tag_token(c = get_tag(f));
switch (state)
{
case 1:
state = 6;
if ( t == NINE) state = 3;
if ( t == ZERO) state = 2;
if ( t == TEN)  state = 4;
if ( t == TWENTY) state = 5;
break;
case 2: if ( t == ENDSEC) quit = TRUE;
state = 1;
break;
case 3: state = 1;
switch(t)
{
case EXTMIN : p1 = &extent_x;
p2 = &extent_y;
break;
case EXTHAX : p1 = &extent_mx;
p2 = &extent_my;
break;
case LIMMIN : p1 = &limit_x;
p2 = &limit_y;
break;
case LIMMAX : p1 = &limit_mx;
p2 = &limit_my;
break;
}
break;
}
}
break;

```

```

case 4:  if ( p1 ) p2 = atof(c);
        p2 = NULL;
        state = 1;
        break;
case 6:  state = 1;
        break;
}
}
}

```

Лист. 5.22. Нахождение значений в секции HEADER DXF-файла

Чтение секции таблиц TABLES

Секция таблиц TABLES содержит параметры рабочего окружения для чертежей системы AutoCAD. Записи в последней секции DXF-файла имеют множество ссылок на эти таблицы, что необходимо для определения формы отображения графических объектов. В этих таблицах можно найти информацию о цвете, шрифте, типе линии и расположении точки наблюдения для вида. Секция таблиц состоит из четырех частей. Каждая часть содержит специфическую информацию, определяющую представление графических объектов на чертеже.

Секция таблиц в DXF-файлах начинается со своего собственного заголовка. В качестве признака начала заголовка таблицы используется сочетание маркера 0 и значения TABLES. Отметим, что ключевое слово TABLES находится во множественном числе. Каждая частная таблица начинается с ключевого слова TABLE (в единственном числе). На лист. 5.23 показаны строки DXF-файла, которые обозначают начало и конец секции таблиц TABLES. Центральная управляющая программа синтаксического разбора должна распознавать лексему TABLES и вызывать для дальнейшей обработки программу синтаксического разбора секции таблиц.

```

0
SECTION
2
TABLES
:
:
:
0
ENDSEC

```

Лист. 5.23. Секция TABLES в DXF-файле

Задачей программы синтаксического разбора секции таблиц является определение момента и порядка вызова отдельных программ

для разбора каждой из таблиц. Все таблицы в DXF-файле начинаются специальной последовательностью. На лист. 5.24 показана структура секции TABLES в DXF-файле, остальные секции файла для наглядности пропущены. В этот листинг включены маркеры начала отдельных таблиц. Детальное описание таблиц можно найти в последующих разделах книги.

```
0
SECTION
2
TABLES
0
LTYPE
:
:
0
ENDTAB
0
TABLE
2
LAYER
:
:
0
ENDTAB
0
TABLE
2
STYLE
:
:
0
ENDTAB
0
TABLE
2
VIEW
:
:
0
ENDTAB
0
ENDSEC
```

Лист. 5.24. Структура секции TABLES в DXF-файле

Программа синтаксического разбора, выполняющая разбор на уровне секции таблиц TABLES, должна предвидеть и распознавать

все маркеры и значения, показанные в листинге. Поскольку нет жестких требований к тому, чтобы какая-либо из таблиц обязательно присутствовала в DXF-файле, то алгоритм, основанный на использовании конечного автомата, позволит обнаружить и разобрать любые части DXF-файла. На рис.5.3 показана диаграмма конечных состояний для синтаксического разбора. На лист. 5.25 представлена реализация этой диаграммы. Каждая таблица секции TABLES имеет свою собственную структуру и назначение. В последующем будет рассмотрена каждая таблица.

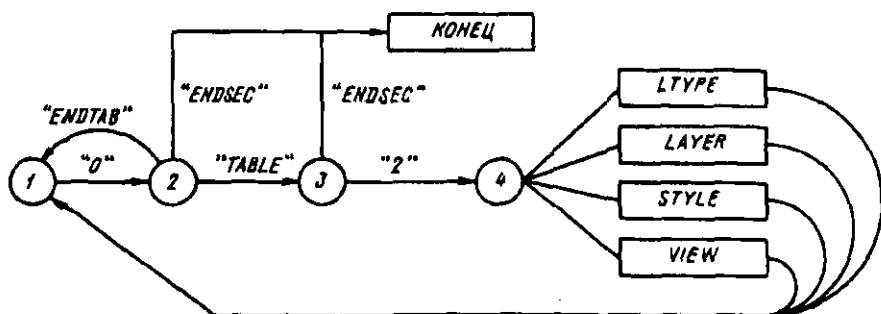


Рис. 5.3. Диаграмма состояний для программы разбора секции HEADER

```

tables (f)
FILE      *f;
{
char      quit;
int       state;
int       t;

quit = FALSE;
state = 1;
while (!quit)
{
t = get_tag_token(get_tag(f));
switch (state)
{
case 1:  if (t == ZERO) state = 2;
         break;
case 2:  state = 1;
         if (t == ENDSEC) quit = TRUE;
         if (t == ENDTAB) state = 1;
         if (t == TABLE ) state = 3;
         break;
case 3:  state = 1;
         if (t == ENDSEC) quit = TRUE;

```

```

        if (t == TWO) state = 4;
        break;
    case 4: state = 1;
        switch(t)
        {
            case LTYPE : ltype(f);
                        break;
            case LAYER : layer(f);
                        break;
            case STYLE : style(f);
                        break;
            case VIEW  : view(f);
                        break;
        }
        break;
    }
}
}

```

Лист. 5.25. Управление вводом секции TABLES

Таблица типов линий LINETYPE

В таблице типов линий LINETYPE определяется представление на чертеже линий различного типа. В данной таблице впервые в DXF-файле появляется графическая информация. Поскольку описание типа линий связано с представлением графической информации на слоях, в DXF-файлах определение типов линий предшествует описанию слоев. Это обеспечивает возможность использования ссылок на типы линий при описании слоев.

Объем информации о типе линий не так уж велик. Основное назначение этой информации - дать системе AutoCAD указание о правильном представлении линий. Не все линии на чертеже только сплошные или непрерывные, некоторые могут быть пунктирными, некоторые штриховыми, а некоторые представлять комбинацию двух последних.

Если вас на самом деле интересует использование линий разных типов, то вы непременно захотите получить соответствующую информацию. Но более вероятно, что эта информация вам не нужна и вы захотите как можно скорее пропустить данную таблицу. Именно это и сделаем на первом шаге - считаем таблицу типов линий и затем ее пропустим.

Ключевым для быстрого пропуска таблицы типов линий является поиск значения ENDTAB (конец таблицы). На лист. 5.26 представлен исходный текст программы, позволяющей без особого труда пропускать таблицу типов линий. Диаграмма конечных состояний этой программы довольно проста, она приведена на рис. 5.4. Определение

типа линии состоит из двух частей. Для того чтобы понять, какие это части, рассмотрим пример DXF-файла, представленный на лист. 5.27.

```

ltype(dxf)
FILE      *dxf;
{
char      quit;
int       state;
char      *s;
int       t;

quit = FALSE;
state = 1;
while (!quit)
{
t = get_tag_token(s = get_tag(dxf));
switch (state)
{
case 1: state = 3;
        if (t == ZERO) state = 2;
        break;
case 2: state = 1;
        if (t == ENDTAB) quit = TRUE;
        break;
case 3: state = 1;
        break;
}
}
}

```

Лист. 5.26. Быстрый пропуск таблицы LINETYPE

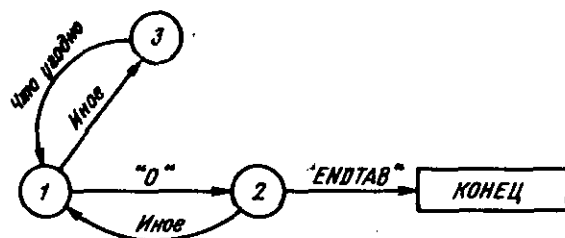


Рис. 5.4. Диаграмма конечных состояний для программы пропуска таблицы LINETYPE

```

0
LTYPE
2
CENTER
70
64
3
-----
72
65
73
4
40
2.000000
49
1.250000
49
-0.250000
49
0.250000
49
-0.250000

```

Лист. 5.27. Пример таблицы LINETYPE в DXF-файле

Основная часть определения содержит такие данные, как имя, описание и характеристики линейного масштаба. Вторая часть содержит математическое определение отображаемой на чертеже линии.

Линии состоят из видимых сегментов линии, невидимых сегментов линии и точек. Стандартным типом линии является **CONTINUOUS** (непрерывная линия) - это сплошная линия, не включающая при начертании каких-либо пунктиров или штрихов. Описание каждого типа линии, для определения ее начертания, содержит список элементов, составляющих линию. Описание этих элементов располагается после основной части определения типа линии. Элементы представлены в виде действительных чисел, определяющих длину сегмента линии. Положительные числа используются для представления видимых, а отрицательные для невидимых сегментов линии. Значения 0.0 применяются для обозначения точек. Сумма абсолютных значений сегментов линии должна быть равна значению параметра длины линии, помещаемого в основной части определения типа линии. Поскольку число определяемых сегментов может быть различным для линий разных типов, в описании типа линии отсутствует какая-либо явно выделенная запись, применяемая для определения сегментов линии. Я предпочитаю использовать для сегментов связный список значений. На лист. 5.28 показано определение

типа для построения такого списка сегментов. Новый тип данных называется `dash_type`, поскольку он может использоваться только при описании штриховых линий. На рис. 5.5 показана структура данных в виде списка, состоящая из определений штрихов и их отношений к остальным данным структуры определения типа линии.

```
typedef
struct c
{
    struct c  *next;
    double    length;
}            dash_type;
```

Лист. 5.28. Определение типа данных `dash_type`

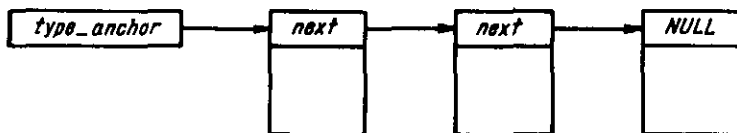


Рис. 5.5. Список определений сегментов линии, составляющих полное определение типа линии

Основная часть определения типа линии также имеет определение типа. Определение типа включает всю собранную информацию о типе линии. В определение типа также включается указатель на рассмотренный выше список определений штрихов. Как только информация о типе линии считывается из DXF-файла и преобразуется во внутреннее представление, программа включает определение типа линии в список, состоящий из других определений типов линий, уже обнаруженных в DXF-файле. Определения типов связываются с помощью ссылки - переменная `lt_anchor`. На лист. 5.29 представлено определение типа данных для типа линии (`ltype_type`). На рис. 5.6 показан связный список из определений типов линий.

```
typedef
struct d
{
    struct d  *next;
    char      *name;
    char      *descript;
    int       flags;
    int       alignment;
    int       dashes;
    double    length;
    dash_type *dash_length;
}            ltype_type;
```

Лист. 5.29. Определение типа данных для типа линии

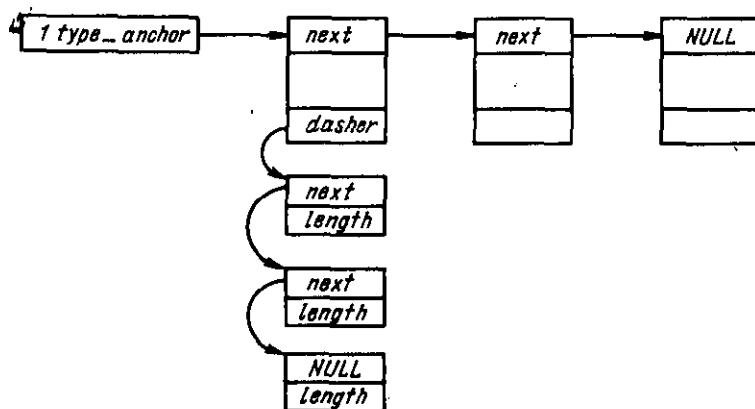


Рис. 5.6. Список определений типов линий, входящих в таблицу типов линий

Поскольку эта информация извлекается из DXF-файла, то в файле она представляется в виде пары маркер — значение. В структуре данных типа линии каждому полю соответствует уникальный маркер. В табл. 5.3 представлены маркеры, встречающиеся в определениях типа линии. Эти значения помогут разобраться в тексте программы синтаксического разбора.

Таблица 5.3. Маркеры, применяемые в секции LINETYPE DXF-файлов

2	STRING	Имя типа линии
3	STRING	Обозначение типа линии в коде ASCII
72	INTEGER	Выравнивание
73	INTEGER	Число штриховых элементов
40	REAL	Общая длина цепочки
49	REAL	Значение длины штриха
70	INTEGER	Флажки

Полная программа синтаксического разбора таблицы типа линий вначале считывает маркер из DXF-файла, дальнейшая логика ее работы определяется в зависимости от значения маркера. Находясь в соответствующем состоянии, программа получает значения и сохраняет их, преобразуя во внутреннее представление. На лист. 5.30 приведен исходный текст программы полного синтаксического разбора таблицы типов линий. На рис. 5.7 показана диаграмма конечных состояний этой программы.

```

ltype(dxf)
FILE      *dxf;
{
char      quit;

```

```

int      state;
char     *s;
ltype_type *l;
dash_type *d1;
int      t;
int      used_flag;

/* Получить элемент списка типа линии - ltype */
/* и добавить его к списку */
/* Установить значения по умолчанию */
l = NULL;
quit = FALSE;
state = 1;
while (!quit);
{
    t = get_tag_token(s = get_tag(dxf));
    switch (state);
    {
        case 1: state = 1;
                switch(t)
                {
                    case ZERO      : state = 2;
                                    break;
                    case TWO       : state = 3;
                                    break;
                    case THREE     : state = 4;
                                    break;
                    case SEVENTY2  : state = 5;
                                    break;
                    case SEVENTY3  : state = 6;
                                    break;
                    case FORTY     : state = 7;
                                    break;
                    case FORTY9    : state = 8;
                                    break;
                    case SEVENTY   : state = 9;
                                    break;
                }
                break;
        case 2: state = 1;
                l = NULL;
                if (t == ENDTAB) quit = TRUE;
                break;
        case 3: state = 1;
                if (l = (ltype_type *)
calloc(1,sizeof(ltype_type)))
                {

```

```

        l->next = lt_anchor;
        lt_anchor = l;
        l->name = locate(s);
    }
    else
        printf("Не хватает памяти -- LINETYPE\n");
        break;
case 4: state = 1;
        if (!l) l->descript = locate(s);
        break;
case 5: state = 1;
        if (!l) l->alignment = atoi(s);
        break;
case 6: state = 1;
        if (!l) l->dashes = atoi(s);
        break;
case 7: state = 1;
        if (!l) l->length = atof(s);
        break;
case 8: state = 1;
        if (!l) break;
        d1 = l->dash_length;
        if (!d1)
        {
            l->dash_length = (dash_type *)
                calloc(1, sizeof(dash_type));
            l->dash_length->length = atof(s);
        }
        else
        {
            while (d1->next) d1 = d1->next;
            d1->next = (dash_type *)
                calloc(1, sizeof(dash_type));
            d1->next->length = atof(s);
        }
        break;
case 9: state = 1;
        used_flag = atoi(s);
        if (!l) l->flags = used_flag;
        break;
    }
}

```

Лист. 5.30. Полная программа разбора таблицы LINETYPE

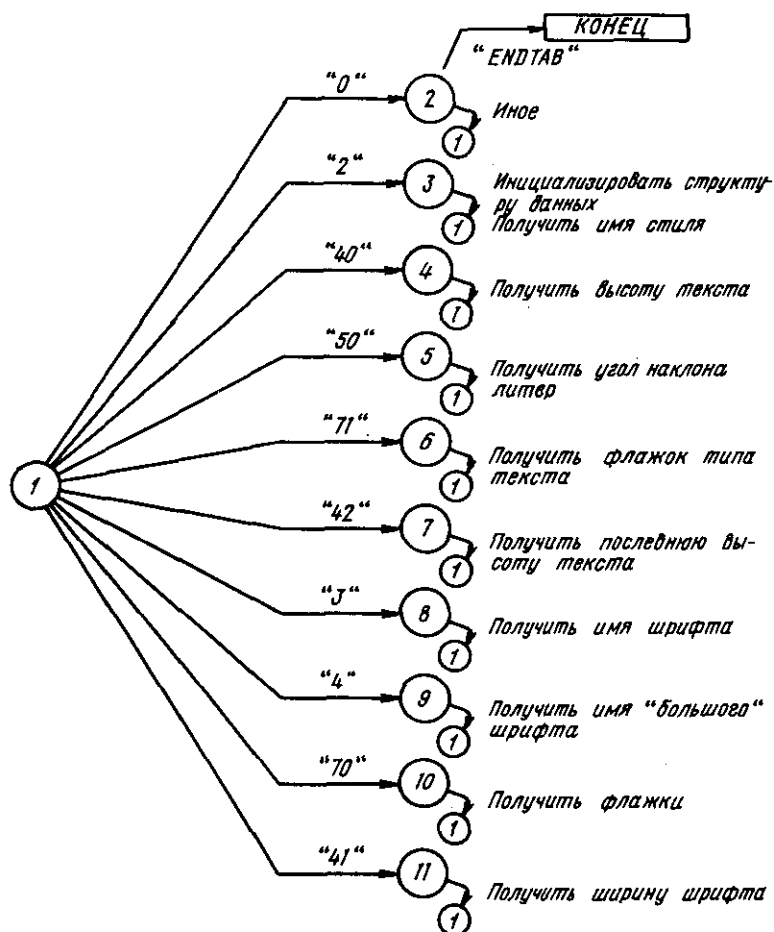


Рис. 5.7. Диаграмма конечных состояний для программы разбора таблицы LINETYPE

Следующее обсуждение не относится к синтаксическому разбору таблицы типов линий. Оно касается нахождения информации о типе линии после того, как будет полностью считан DXF-файл. В рассмотренных ранее фрагментах программ полученная из DXF-файла информация о типе линий запоминалась в связанном списке, содержащем определения различных типов линий. Иногда в вашей программе вы можете захотеть получить доступ к этой информации. Если вы не хотите использовать информацию о типе линий, то примените программу для пропуска таблицы типа линий. На лист. 5.31 показано, как можно в программе получить данные о типе линии, используя в качестве аргумента для запроса имя типа линии.

```

ltype_type *find_ltype(s)
char *s;
{
    ltype_type *l;
    l = lt_anchor;
    while (l)
    {
        if (!strcmpi(s, l->name)) return l;
        l = l->next;
    }
    printf("Тип линии не найден\n");
    return NULL;
}

```

Лист. 5.31. Нахождение данных о типе линии

Программа выполняет поиск по списку (в качестве связующей ссылки используется переменная `lt_anchor`), производя сравнения имени искомого типа линии с именами типов линий, содержащихся в списке. При совпадении двух имен программа возвращает указатель на найденную структуру данных - элемент списка. Если для запрашиваемого типа линии в списке нет соответствующих данных, то программа выдаст сообщение об ошибке и возвратит значение - указатель со значением ссылки `NULL` (пустая ссылка).

Таблица слоев LAYER

Как элемент структуры DXF-файла таблица слоев LAYER определяет характеристики каждого слоя чертежа, подготовленного в системе AutoCAD. Для каждого слоя в таблице содержится информация, касающаяся цвета, типа линий и видимости. Эту информацию легко извлечь из DXF-файла.

В отличие от таблицы LINETYPE, данные которой могут быть проигнорированы, к таблице LAYER может быть много ссылок из разных частей DXF-файла. Каждый элемент чертежа, описанный в секции ENTITIES, содержит ссылку на слой. На лист. 5.32 показан типичный вид таблицы LAYER в DXF-файле.

```

1
6
CONTINUOUS
70
64
0
ENDTAB

```

Лист. 5.32. Пример типичной записи таблицы LAYER в DXF-файле

В рассматриваемом примере описывается слой с именем A_LAYER. На слое по умолчанию используется красный цвет RED(1). Тип линии на слое A_LAYER - непрерывный (CONTINUOUS).

Поскольку таблица слоя содержит минимальную информацию по сравнению с другими таблицами, ее легче разобрать. Это так, и это является ее отличительной особенностью как наиболее активно используемой таблицы. В табл. 5.4 перечислены маркеры, которые можно встретить внутри определения слоя.

Таблица 5.4. Маркеры, используемые в таблице LAYER

2	STRING	Имя слоя
62	INTEGER	Номер цвета
		отрицательное = слой выключен (OFF)
6	STRING	Имя типа линии
70	INTEGER	Флажки
		1 = заморожен

На лист. 5.33 показана структура данных, предназначенная для хранения информации о слое. Подобно структуре данных для типа линии, структура данных для слоя разработана так, чтобы ее также можно было включить в список, каждый элемент которого несет информацию о конкретном слое. Это позволяет по мере необходимости получать доступ к отдельному элементу списка. Каждое поле в структуре данных соответствует одной из пар маркер-значение, которые можно встретить в таблице слоев.

```
typedef
struct e'
{
    struct e    *next;
    char        *name;
    int         flags;
    int         color;
    char        *lt;
                layer_type;
}
```

Лист. 5.33. Структура данных для сохранения информации о слое

Исходный текст программы для пропуска секции типа линии можно модифицировать и использовать для пропуска секции таблицы слоев. Нет необходимости повторять текст программы, подобной уже рассмотренной прежде. Тем более что для большинства приложений нежелательно пропускать информацию о слоях, которая нужна. На рис. 5.8 показана диаграмма конечных состояний программы разбора

таблицы слоев. На лист. 5.34 приведен исходный текст программы, реализующей эту диаграмму.

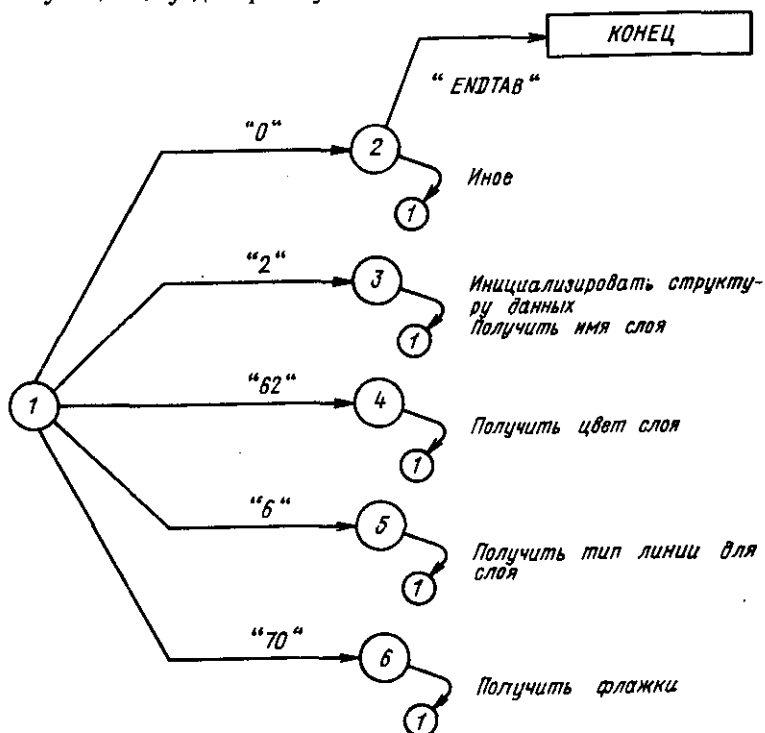


Рис. 5.8. Диаграмма конечных состояний для программы разбора таблицы LAYER

```

layer(dxf)
FILE      *dxf;
{
char      quit;
int       state;
char      *s;
layer_type *l;
int       t;
char      *c;
int       used_flag;

l = NULL;
quit = FALSE;
state = 1;
while (!quit)
{
t = get_tag_token(s = get_tag(dxf));
switch (state)

```

```

{
    case 1: state = 1;

        switch(t)
        {
            case ZERO      : state = 2;
                           : break;

            case TWO       : state = 3;
                           : break;

            case SIXTY2    : state = 4;
                           : break;

            case SIX       : state = 5;
                           : break;

            case SEVENTY   : state = 6;
                           : break;

        }
        break;
    case 2: state = 1;
        l = NULL;
        if ( t == ENDTAB) quit = TRUE;
        break;
    case 3: state = 1;
        if (l=(layer_type *)
            calloc(1,sizeof(layer_type)))
        {
            l->next = ly_anchor;
            ly_anchor = l;
            l->name = locate(s);
        }
        else
            printf("Не хватает памяти -- LAYER\n");
        break;
    case 4: state = 1;
        if (l) l->color = atoi(s);
        break;
    case 5: state = 1;
        if (l) l->lt = locate(s);
        break;
    case 6: state = 1;
        used_flag = atoi(s);
        if (l) l->flags = used_flag;
        break;
    }
}
}

```

Лист. 5.34. Программа разбора таблицы LAYER

Есть одно важное замечание, относящееся к значению номера цвета в таблице слоев. Это значение обычно должно находиться в диапазоне 1 — 255. Отображаемое одновременно число цветов зависит от имеющейся у вас аппаратуры. Если значение номера цвета отрицательное, слой "выключен" (невидим). Номер цвета определяет абсолютное значение отрицательного числа.

Кстати, о "выключенных" слоях: в DXF-файлах в описании слоя имеется специальный флажок. В файле перед флажком помещается маркер 70. Вообще говоря, флажки показывают некоторые условия. Если значение флажка равно 1, то это означает, что слой заморожен. Такая индикация сообщает системе AutoCAD, что надо пропустить обработку некоторых элементов чертежа при выполнении операции REGEN (регенерация изображения). Для нас это еще один способ указать, что слой выключен.

Поскольку мы включаем таблицы слоев в связный список для обработки последующих на них ссылок, необходимо иметь способ нахождения требуемого слоя. Текст программы, представленной на лист. 5.35, аналогичен программе для поиска заданного типа линии. В данном случае для связи между элементами списка используется указатель `ly_anchor`. Реализован следующий алгоритм: с помощью ссылок осуществляется переход от одного описания слоя к другому; при каждой смене слоя выполняется сравнение имени текущего слоя с именем, указанным в запросе. Когда обнаруживается соответствие, программа через переменную-указатель возвращает адрес найденного описания слоя. В противном случае программа возвращает значение указателя NULL, обозначающее отсутствие соответствия. Вы можете захотеть внести изменения в программу для того, чтобы она возвращала ссылку на слой 0, тогда корректная информация о слое будет доступна даже при наличии некоторых несоответствий в DXF-файле.

```
layer_type *find_layer(s)
char      *s;
{
    layer_type      *l;
    l = ly_anchor;
    while (l)
    {
        if (!strcmp(s, l->name)) return l;
        l = l->next;
    }
    printf("Слой не найден\n");
    return NULL;
}
```

Лист. 5.35. Функция нахождения слоя

Таблица стиля начертания STYLE

В таблице стиля начертания **STYLE** описывается форма, в которой система AutoCAD должна отображать специальные графические символы или шрифты. Поскольку графические символы (формы) по сути идентичны шрифтам и повторяют их внутреннюю структуру, эта таблица также поддерживает информацию о файлах форм.

Для создания различных стилей начертания, отдельные литеры могут быть скручены, иметь зеркальное отображение и масштабироваться различными способами. Каждый стиль описывается в таблице стилей начертания **STYLE**. Может быть несколько стилей начертания, базирующихся на единственном исходном начертании.

Таблица стиля - вторая наиболее сложная таблица. Без сомнений, таблица типов линий является наиболее сложной. На лист. 5.36 показана типичная для DXF-файлов последовательность данных, входящих в таблицу стиля.

```
0
STYLE
2
SIMPLEX
70
64
40
0.000000
41
1.000000
50
0.000000
71
0
42
0.160000
3
simplex
4
```

Лист. 5.36. Пример таблицы **STYLE** в DXF-файле

В табл. 5.5 перечислены маркеры, которые можно встретить в таблице стиля. Для описания графических символов не требуется наличие всех указанных полей, необходимым является только имя файла. Хотя в DXF-файле могут быть и другие поля, ваша программа

может их проигнорировать. Использование файла форм помечается в формате DXF-файла установкой значения 1 в поле флажков, для обозначения этого поля используется маркер 70.

Таблица 5.5. Маркеры, используемые в таблице STYLE

2	STRING	Имя стиля
70	INTEGER	Флажки 1 = файл формы (SHAPE) 2 = вертикальный шрифт
40	REAL	Фиксированная высота текста 0 = не фиксированная
41	REAL	Ширина
50	REAL	Угол наклона литер
71	INTEGER	Флажок типа текста
42	REAL	Последняя использованная высота текста
3	STRING	Основной файл шрифта
4	STRING	"Большой" файл шрифта пробел = нет "большого"

Для образования стиля начертания на самом деле могут использоваться два файла определения шрифта. Признак 3 помечает основной файл шрифта. В системе AutoCAD графическая информация из этого файла применяется для начертания обычных символов кода ASCII. Информация для начертания расширенного набора символов (как, например, требуется для японской катаканы) содержится в "большом" файле шрифта. Коды для расширенного набора шрифта соответствуют 16-битовой кодировке, используемой на японских персональных компьютерах. Если бы вы вывели на экран такого компьютера содержимое DXF-файла, включающее символы "большого" шрифтового набора (например, с помощью команды TYPE операционной системы MS-DOS), то вы смогли бы увидеть символы катаканы, а не некоторые странные числовые значения. В DXF-файле всегда имеется описание, по крайней мере, одного стиля начертания шрифта. Этот стиль начертания называется STANDART (стандартный), в нем обычно используется шрифт txt.

В описании стиля имеется много полей. На лист. 5.37 показана структура данных для хранения этой информации. Каждый элемент этой структуры соответствует паре маркер — значение DXF-файла.

Следующее, что должно быть представлено по традиции, — это диаграмма состояний для программы разбора таблицы STYLE (рис. 5.9). На лист. 5.38 приведен исходный текст этой программы.

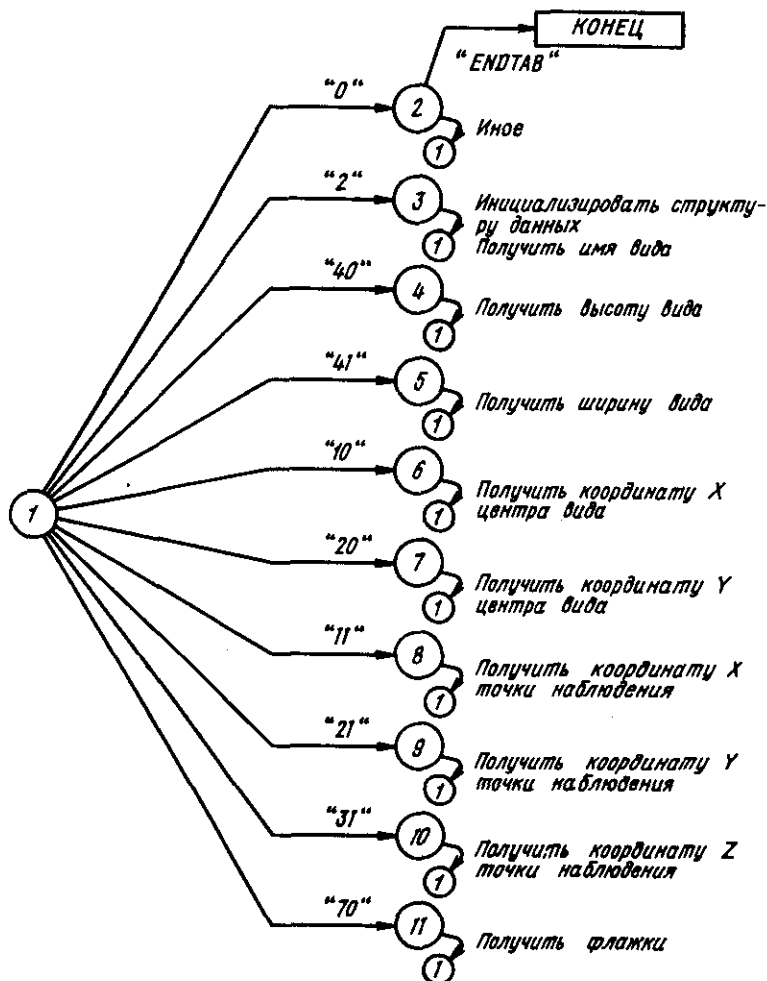


Рис. 5.9. Диаграмма конечных состояний для программы разбора таблицы STYLE

```

typedef
struct f
{
    struct f    *next;
    char        *name;
    int         flags;
    int         gen_flags;
    double      txt_scale;
    double      height;
    double      oblique;

```

```

double      last_height;
char        *font;
char        *big_font;
}           style_type;

```

Лист. 5.37. Структура данных для элемента таблицы STYLE

```

style(dx)
FILE      *dx;
{
char      quit;
int       state;
char      *s;
style_type *l;
int       t;
char      *c;
int       used_flag;

l = NULL;
quit = FALSE;
state = 1;
while (!quit)
{
t = get_tag_token(s = get_tag(dx));
switch (state)
{
case 1: state = 1;
switch(t)
{
case ZERO      : state = 2;
break;
case TWO       : state = 3;
break;
case FORTY     : state = 4;
break;
case FORTY1    : state = 11;
break;
case FIFTY     : state = 5;
break;
case SEVENTY1  : state = 6;
break;
case FORTY2    : state = 7;
break;
case THREE     : state = 8;
break;
case FOUR      : state = 9;
break;
case SEVENTY   : state = 10;

```



```

        break;
    }
    break;
case 2: state = 1;
        l = NULL;
        if (t == ENDTAB) quit = TRUE;
        break;
case 3: state = 1;
        if (l = (style_type*)calloc(1, sizeof(style_type)))
        {
            l->next = s_anchor;
            s_anchor = l;
            l->name = locate(s);
        }
        else
            printf("Не хватает памяти -- Text Style\n");
        break;
case 4: state = 4;
        if (!l) l->height = atof(s);
        break;
case 5: state = 1;
        if (!l) l->oblique = atof(s);
        break;
case 6: state = 1;
        if (!l) l->gen_flags = atoi(s);
        break;
case 7: state = 1;
        if (!l) l->last_height = atof(s);
        break;
case 8: state = 1;
        if (!l) l->font = locate(s);
        break;
case 9: state = 1;
        if (!l) l->big_font = locate(s);
        break;
case 10: state = 1;
        used_flag = atoi(s);
        if (!l) l->flags = used_flag;
        break;
case 11: state = 1;
        if (!l) l->txt_scale = atof(s);
        break;
    }
}
}

```

Поскольку для разрешения последующих ссылок таблицы стилей объединяются в список, необходимо иметь способ нахождения требуемого стиля начертания. Текст программы на лист. 5.39 очень похож на тексты программ для поиска типа линии и слоя. В данном случае в качестве ссылки на последующий элемент списка используется переменная `s_anchor`. Реализован следующий алгоритм: с помощью ссылок осуществляется переход от одного описания стиля к другому и выполняется сравнение имени текущего стиля с именем стиля, указанным в запросе. При обнаружении соответствия программа возвращает адрес найденного описания стиля как значение указателя. В противном случае программа возвращает значение указателя `NULL`, обозначающее, что соответствие не найдено. Вы можете захотеть несколько изменить программу, так, чтобы она возвращала ссылку на стиль `STANDART`, тогда даже при наличии некоторых несоответствий в `DXF`-файле будет доступна корректная информация о стиле начертания.

```

style_type *find_style(s)
char      *s;
{
    style_type    *t;
    t = s_anchor;
    while (t)
    {
        if (!strcmpi(s,t->name)) return t;
        t = t->next;
    }
    return NULL;
}

```

Лист. 5.39. Нахождение информации о стиле начертания

Таблица видов VIEW

Вероятно, что наименее часто используемой таблицей является таблица видов `VIEW`. В этой таблице содержится информация об именах и точках наблюдения, установленных для чертежа. До тех пор, пока не будет реализован хороший пакет программ трехмерной графики, данная таблица будет оставаться без применения. На лист. 5.40 показано типичное представление в `DXF`-файле таблицы видов. В табл. 5.6 перечислены маркеры, встречающиеся в таблице видов.

```

0
VIEW
2
ONE
70
0

```

40
 3.994192
 10
 4.372212
 20
 7.490436
 41
 5.480400
 11
 0.000000
 21
 0.000000
 31.
 1.000000

Лист. 5.40. Пример таблицы VIEW в DXF-файле

Таблица. 5.6. Маркеры, используемые в таблице VIEW

2	STRING	Имя вида
40	REAL	Высота вида
41	REAL	Ширина вида
10	REAL	Координата X центра вида
20	REAL	Координата Y центра вида
11	REAL	Координата X точки наблюдения
21	REAL	Координата Y точки наблюдения
31	REAL	Координата Z точки наблюдения

В таблице VIEW имеется несколько полей, но они все довольно обычные, нет никаких сюрпризов. На лист. 5.41 показана структура данных, применяемая для представления этой информации. Каждый элемент структуры соответствует паре маркер — значение в DXF-файле. Нет точек наблюдения, используемых по умолчанию. Эта таблица может быть пустой, а может быть и совсем пропущена. На рис. 5.10 приведена диаграмма состояний программы разбора таблицы VIEW. На лист. 5.42 приведен исходный текст этой программы.

```

typedef
  struct g
  {
    struct g    *next;
    char        *name;
    int         flags;
    double      v_height;
    double      v_width;
    double      v_ctr_x;
    double      v_ctr_y;
  }

```

```

double    v_dir_x;
double    v_dir_y;
double    v_dir_z;
}          view_type;

```

Лист. 5.41. Пример таблицы VIEW в DXF-файле

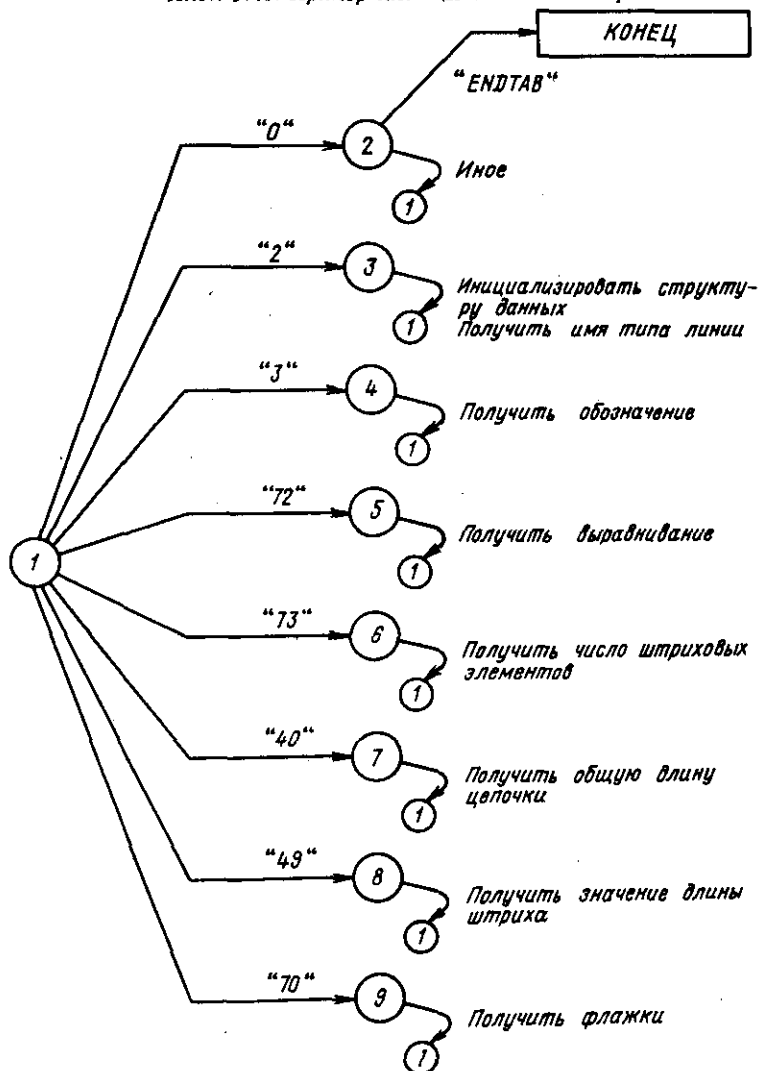


Рис. 5.10. Диаграмма конечных состояний для программы разбора таблицы VIEW

```

view(dx)
FILE      *dx;
{
char      quit;
int       state;
char      *s;
view_type *l;
int       t;
char      *c;
int       used_flag;

l = NULL;
quit = FALSE;
state = 1;
while (!quit)
{
t = get_tag_token(s = get_tag(dx));
switch (state)
{
case 1: state = 1;
switch(t)
{
case ZERO : state = 2;
break;

case TWO : state = 3;
break;

case FORTY : state = 4;
break;

case FORTY1 : state = 5;
break;

case TEN : state = 6;
break;

case TWENTY : state = 7;
break;

case ELEVEN : state = 8;
break;

case TWENTY1 : state = 9;
break;

case THIRTY1 : state = 10;
break;

case SEVENTY : state = 11;
break;

}
break;
case 2: state = 1;
l = NULL;
if (t == ENDTAB) quit = TRUE;

```

```

        break;
case 3:  state = 1;
        if (l = (view_type *)calloc(1,sizeof(view_type)))
        {
            l->next = v_anchor;
            v_anchor = l;
            l->name = locate(s);
        }
        else
            printf("Не хватает памяти -- VIEW\n");
        break;
case 4:  state = 1;
        if (l) l->v_height = atof(s);
        break;
case 5:  state = 1;
        if (l) l->v_width = atof(s);
        break;
case 6:  state = 1;
        if (l) l->v_ctr_x = atof(s);
        break;
case 7:  state = 1;
        if (l) l->v_ctr_y = atof(s);
        break;
case 8:  state = 1;
        if (l) l->v_dir_x = atof(s);
        break;
case 9:  state = 1;
        if (l) l->v_dir_y = atof(s);
        break;
case 10: state = 1;
        if (l) l->v_dir_z = atof(s);
        break;
case 11: state = 1;
        used_flag = atoi(s);
        if (l) l->flags = used_flag;
        break;
    }
}
}

```

Лист. 5.42. Программа разбора таблицы VIEW

Поскольку для разрешения последующих ссылок таблицы видов объединяются в список, необходимо иметь способ нахождения требуемого вида. Текст программы, представленной на лист. 5.43, очень схож с текстами программ поиска типа линии, слоя и стиля начертания. Обычно я храню все эти программы разбора в одном файле, что облегчает переход от одного приложения к другому. В данной про-

грамме в качестве ссылки на последующий элемент списка используется переменная `v_anchor`. Реализован следующий алгоритм: с помощью ссылок осуществляется переход от одного описания вида к другому и выполняется сравнение имени текущего вида с именем вида, указанного в запросе. При обнаружении соответствия программа возвращает адрес найденного описания вида как значение указателя. В противном случае программа возвращает значение указателя `NULL`, обозначающее, что соответствие не найдено.

```
view_type *find_view(s)
char      *s;
{
    view_type      *v;

    v = v_anchor;
    while (v)
    {
        if (!strcmpi(s, v->name)) return v;
        v = v->next;
    }
    return NULL;
}
```

Лист. 5.43. Нахождение информации о виде

Чтение секции объектов ENTITIES

Секция объектов **ENTITIES** DXF-файла представляет собой часть файла, в которой описываются элементы реального чертежа. Секция содержит определения объектов, отображаемых на экране или вычерчиваемых на бумаге.

Обычно обсуждение секции блоков **BLOCKS** начинается в этом месте, так как она является следующей секцией в DXF-файле. Однако секция **BLOCKS** по своей конструкции очень схожа со структурой секции **ENTITIES**. Поэтому, дабы сохранить все в логическом и последовательном порядке, с точки зрения программирования секция **BLOCKS** (или то, что о ней еще можно будет сообщить) рассматривается в следующем параграфе.

Сердцевина любой программы, осуществляющей чтение DXF-файла, строится на основании информации об отдельных объектах. Именно, исходя из этого, мы до сих пор выполняли синтаксический разбор и определяли вспомогательную информацию для чертежа. Получив информацию о значениях **LIMITS** (о габаритах чертежа) и **EXTENTS** (границах чертежа) из секции заголовка, мы знаем, насколько велика область чертежа. Из таблицы слоев мы знаем, какие используются цвета и с какими слоями чертежа они связаны. Из таблицы типов линий мы знаем, как выглядят линии, которые бу-

дут появляться на чертеже. Но то, что мы не знаем, - это как будет выглядеть сам чертеж. Теперь мы постараемся изменить ситуацию.

Как и в случаях с любыми другими секциями данных, мы отмеча-
ли, что секция начинается со специальной последовательности, со-
стоящей из маркера и следующего за ним значения. В начале сек-
ции объектов мы обнаруживаем то же самое. На лист. 5.44 показаны
предшествующие данной секции маркеры, за которыми следует опи-
сание объектов.

```

0
SECTION
2
ENTITIES
:
:
:
0
ENDSEC

```

Лист. 5.44. Маркеры заголовка секции и значения для секции ENTITIES

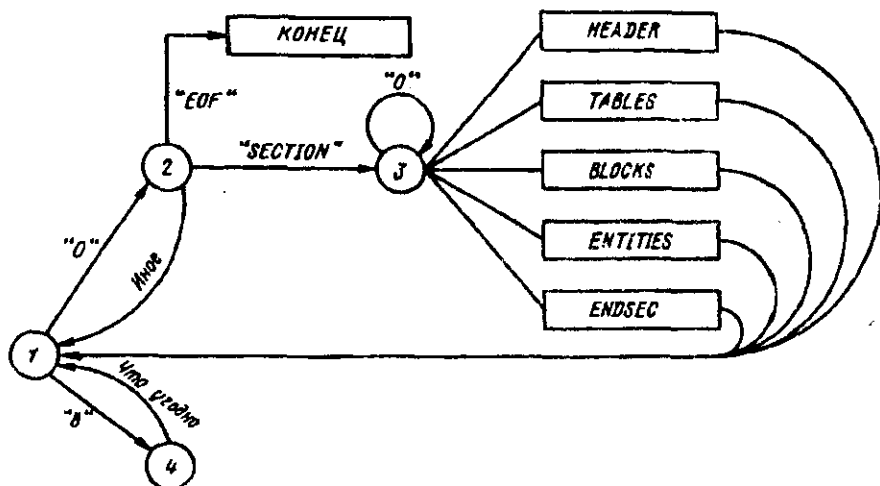


Рис. 5.11. Диаграмма конечных состояний для центральной программы синтаксического разбора формата DXF-файла

Поскольку первые строки в секции ENTITIES в точности такие же, как и во всех других секциях, мы можем переложить задачу распознавания этих строк на уже имеющуюся процедуру распознавания — центральный управляющий модуль. Единственное отличие

заключается в том, что надо добавить распознавание последовательности маркеров, относящихся непосредственно к секции ENTITIES. Как только эти строки будут найдены, центральная программа вызовет программу синтаксического разбора секции объектов. На рис. 5.11 показана диаграмма конечных состояний для распознавания объектов. На лист. 5.45 приведен исходный текст модифицированной центральной управляющей программы синтаксического анализа, которая передаст управление программе разбора секции объектов ENTITIES.

```
switch(state)
{
    case 1 : state = 2;
            if (t == ZERO) state = 3;
            break;
    case 2 : state = 1;
            break;
    case 3 : state = 1;
            if (t == SECTION) state = 4;
            break;
    case 4 : state = 2;
            if (t == TWO) state = 5;
            break;
    case 5 : state = 1;
            switch(t)
            {
                case HEADER : header_parser(f); state = 3;
                             break;
                case TABLES : tables_parser(f); state = 3;
                              break;
                case BLOCKS : blocks_parser(f); state = 3;
                             break;
                case ENTITIES : entity_parser(f, NULL); state = 3;
                               break;
            }
            break;
}
```

Лист. 5.45. Модифицированная центральная управляющая программа синтаксического разбора

Каждый объект идентифицируется по предшествующей ему паре маркер — значение. Мы используем данную пару для вызова специальной программы обработки объекта конкретного типа. Следующие за идентификатором объекта данные являются уникальными для этого объекта.

Ниже приводится стратегия, которой вы можете придерживаться при разборе данных объекта:

1. До распознавания типа объекта активизируйте функцию обработки поступающих последовательностей данных.

2. В процессе специальной обработки выполняйте выборку всей информации об объекте, заполняя соответствующие поля структуры данных. Эта структура данных может сохраняться для последующего использования, когда будет получена и структурирована дополнительная информация, или сразу целиком передаваться некоторой обрабатывающей программе.

3. Результатом, возвращаемым специальной функцией обработки, является указатель на структуру данных, содержащую характеристики объекта.

Таблица 5.7. Объекты в DXF-файле

LINE	ENDBLK
POINT	INSERT
CIRCLE	ATTDEF
ARC	ATTRIB
TRACE	POLYLINE
SOLID	VERTEX
TEXT	SEQEND
SHAPE	DIMENSION
BLOCK	3DLINE
	3DFACE

В DXF-файле каждому объекту предшествует пара маркер — значение. Используемый при этом маркер всегда нулевой. Значение идентификатора представляет собой символьную строку — имя объекта. В табл. 5.7 приведен список имен объектов.

Детали каждой конкретной программы синтаксического разбора объекта могут изменяться. Тем не менее в каждой программе используется довольно схожая стратегия обработки. Поэтому я останавлиюсь на подробном обсуждении стратегии и постараюсь дать некоторые иллюстрации. В качестве иллюстрации будет приведена разработка специальной обрабатывающей программы, которая послужит моделью для построения других программ синтаксического разбора. При обсуждении стратегии вы обнаружите ряд таблиц. В каждой таблице приводится структура какого-либо объекта. Таблица содержит следующую информацию:

имя поля;

маркер, используемый для распознавания поля;

тип данных поля;

значение для поля по умолчанию.

Плотно придерживаясь структуры и логики модели, используя информацию из таблицы объектов, вы можете извлечь всю доступную информацию о каждом объекте. Что вы потом будете делать с этой информацией, зависит от вас. При обнаружении конкретного объекта должны быть выполнены некоторые действия, необходимые для

подготовки программы синтаксического разбора к обработке детальной графической информации.

Я отношусь весьма неравнодушно к организации всей доступной информации в структуру данных. Вы могли бы получать и обрабатывать информацию по частям, но в этом случае вы можете потерять некоторое контекстное окружение объекта. Имея всю информацию в одном центральном месте, я могу передать эту информацию в виде блока как параметр для последующей обработки — используя в качестве аргумента указатель вместо нескольких числовых аргументов. Таким же образом можно сохранить информацию для последующей обработки.

Принимая во внимание тактику "использования в последующем", я пропускаю секцию блоков **BLOCKS**, отдавая предпочтение секции объектов **ENTITIES**. В дальнейшем мы вернемся к рассмотрению секции **BLOCKS**. Это потому, что секция **BLOCKS** на 90% идентична секции **ENTITIES**. Единственным ее отличием является то, что вы можете найти в секции **BLOCKS** элементы **BLOCK**, **ENDBLK** и **ATTDEF**, которые не появляются в секции **ENTITIES**. Поскольку блоки сами по себе не являются объектами — они представляют собой определение широко используемых групп объектов — информация об объектах, включенных в определение блока, должна быть сохранена так, чтобы компоненты блока были доступны, если, например, по команде **INSERT** выполняется вставка блока в чертеж. Итак! Подробности об этом вы узнаете позже.

Теперь, после того как мы наметили на возможности сохранения графической информации в структуре данных, первым шагом при создании программы разбора графической информации будет разработка структуры данных. Все элементы структуры данных довольно различны. Используйте подробную таблицу, чтобы определить, какие поля данных нужны для ваших целей. Во всех структурах данных потребуются два одинаковых поля. Первое из этих полей — ссылка, позволяющая связывать между собой несколько элементов.

Второе поле — поле идентификатора объекта. При работе программы синтаксического разбора вы, несомненно, знаете, какой объект обрабатывается. Однако когда объект связан с другими объектами (как в случае определения блока), вы быстро теряете возможность его отождествления. Отсюда следует, что надо иметь способ для отметки, какого типа объект вы имеете. Рассмотренные выше два поля должны присутствовать в каждой структуре данных объекта, предполагается, что вы намерены сохранять информацию об объектах.

Несколько объектов различного типа могут входить в сложный блок. В этом случае может быть удобно иметь родовой тип данных, в котором все ссылки на следующий указатель и идентичные поля могут быть приведены к соответствующему типу данных. (Операция приведения типа реализована в языке Си для отмены проверки допустимости использования типа данных компилятором.) Такой при-

ем вводит компилятор в "заблуждение", но тем не менее работает достаточно хорошо. На лист. 5.46 приведено определение родового типа данных.

```
typedef
struct a
{
    struct a    *next;
    int         ident;
}              generic;
```

Лист. 5.46. Определение родового типа данных

На лист. 5.47 показано возможное практическое использование родового типа данных. В этом фрагменте программы осуществляется просмотр связанного списка объектов и соответственно с идентификацией объекта вызываются процедуры для последующей обработки. В примере выполняемая операция приведения типа удовлетворяет контролируемому типу данных, объявленному в прототипе любой функции, которую вы могли бы иметь.

```
generic    *e;
/* Указатель представляет собой ссылку */
/* на родовой тип данных */
e = first_entity;
while(e)
{
    /* В этом месте определяется тип объекта */
    switch (e->ident)
    {
        /* приведение указателя к соответствующему типу */
        case LINE : line_processor((line_entity *)e);
                    break;
        case ARC  : arc_processor((arc_entity *)e);
                    break;
    }
    e = e->next;
}
```

Лист. 5.47. Использование родового типа данных

Вне зависимости от указанного истинного типа данных рассмотренный способ будет работать, если структуры данных построены согласованно. Ключевым словом является здесь "согласованность". Все структуры должны иметь связующую ссылку и поле идентификатора, расположенные в одном и том же месте внутри каждой структуры данных.

Перейдем к специфическим примерам, в качестве такого примера я выбрал объект TEXT. Этот пример дает богатый набор типов данных и различных ситуаций, встречающихся при его обработке. На

лист. 5.48 представлено определение типа данных для объекта **ТЕХТ**. Сравните эту структуру с таблицами определений данных для формата DXF-файлов.

```
typedef
struct tex
{
    generic      *next;
    int          ident;
    double       x;
    double       y;
    double       angle;
    double       oblique;
    double       height;
    double       x_scale;
    short        generation_flag;
    short        justification;
    layer_node   *layer;
    char         *words;
    style_node   *font;
}
text_entity;
```

Лист. 5.48. Определение типа данных для объекта **ТЕХТ**

Следующее, что необходимо сделать, возвращаясь к обсуждению синтаксического разбора элементов чертежа, — это рассмотреть объявление функций. Поскольку при различных обстоятельствах может потребоваться сохранение данных, весьма резонным является возврат значения указателя на обрабатываемые данные и структуры этих данных.

Рассмотрим также входные аргументы, передаваемые программе синтаксического разбора. Первым и самым важным является ссылка на файл, содержащий графические данные в формате DXF. То, как вы назовете этот тип данных, зависит от вас. В своих примерах я применяю указатель, имеющий тип **FILE**.

В качестве еще одного возможного аргумента может быть задан слой, на котором находятся объекты. Взглянув на реальный DXF-файл, вы увидите, что описанию объекта предшествует объявление слоя (маркер 8). Итак, существует ситуация, когда мы знаем, на каком слое находится объект, но нам ничего не известно о самом объекте. Действие очевидно — сохранить информацию о слое и передать ее программе обработки объекта, в которой она может быть использована. На лист. 5.49 представлено объявление программы обра-

ботки объекта TEXT. На лист. 5.50 показан прототип этой функции в стандарте ANSI.

```
text_entity *text_parser(f, lyr)
FILE *f;
char *lyr;
```

Лист. 5.49. Прототип программы обработки объекта TEXT

```
text_entity *text_parser(FILE *f, char *lyr);
```

Лист. 5.50. Прототип в стандарте ANSI, программы обработки объекта TEXT

Следующее, что надо рассмотреть, — это выделение блоков памяти для хранения графических данных. Для этого с помощью функции malloc просто выполняется запрос блока из области динамической памяти.

Дальнейшее рассмотрение должно коснуться вопроса инициализации выделенного блока памяти. Недостаточно с помощью функции malloc получить блок памяти и при этом предположить, что все поля когда-нибудь сами заполнятся. Функция malloc предоставляет неинициализированные блоки памяти, содержащие абсолютно ненужную информацию - "мусор". В DXF-файле вовсе не обязательно содержатся данные для всех возможных параметров объекта. В файле содержится информация только о тех полях описания объекта, данные в которых были непосредственно использованы в чертеже. Значения, используемые по умолчанию, в DXF-файл обычно не включаются.

Также было бы некорректным установить значения всех полей равными ZERO. Хотя это более безопасно, чем оставлять всякий "мусор" в этих полях (то, что было возвращено функцией malloc), но это может не отвечать требованиям по их обработке, предъявляемым в системе AutoCAD.

Нежелательно иметь неверные значения в структуре данных - значение 0 для коэффициента масштабирования может оказаться катастрофическим — мы будем осторожны и инициализируем все поля структуры данных сами. Во-первых, необходимо убедиться, что значением указателя на следующий элемент будет пустая ссылка NULL. Данные в программе достаточно сложны и без того, чтобы еще иметь и неинициализированные указатели.

Во-вторых, необходимо установить идентификатор объекта. Значения для идентификаторов объектов определяются программой генерации лексем. Используйте те же значения, это облегчит вашу задачу.

Следующее, несколько полей в структуре данных объекта имеют по умолчанию значение, отличное от значения ZERO. Это значение, которое обычно занесено в поле и не изменяется, если только оно

явно не будет задано с помощью пары маркер — значение в DXF-файле. Значения по умолчанию не всегда ZERO. Например, значение коэффициента масштабирования по умолчанию 1.0. Коэффициент масштабирования ZERO мог бы привести к тому, что элемент был бы вообще не видим. Итак, давайте попробуем выполнить инициализацию для объекта TEXT. На лист. 5.51 показана инициализация структуры данных для объекта TEXT. Вы увидите этот фрагмент программы еще раз, когда, в конце концов, все части программы синтаксического разбора будут сложены вместе. Теперь же вы видите, как в игру вводятся две новые процедуры find_layer (найти слой) и find_style (найти стиль).

```
text_entity      *te;

if ((te = (text_entity *)malloc(sizeof(text_entity))) == NULL)
    return NULL;
te->next = NULL;
te->ident = TEXT;
te->x = 0.0;
te->y = 0.0;
te->angle = 0.0;
te->oblique = 0.0;
te->height = 1.0;
te->x_scale = 1.0;
te->generation_flag = 0;
te->justification = 0;
te->layer = find_layer((lyr)?lyr:"0");
te->words = NULL;
te->font = find_style("STANDARD");
```

Лист. 5.51. Инициализация структуры данных объекта TEXT

Теперь переходим к трудной части. Когда программа обнаруживает каждый маркер, относящийся к объекту TEXT, она должна преобразовывать считанные в коде ASCII значения в соответствующий тип данных, и сохранять полученные значения в соответствующем поле структуры данных. Поскольку отсутствуют требования на то, чтобы поля данных объекта располагались в DXF-файле в каком-либо определенном порядке (в DXF-файлах, создаваемых системой AutoCAD, поля следуют в установленном порядке, но нет требований на обязательное выполнение этих условий), мы можем разработать для такого неупорядоченного появления данных еще один конечный автомат. На рис. 5.12 показана диаграмма состояний конечного автомата для разбора объекта TEXT. Тот же самый принцип может легко быть приложен для обработки и других объектов.

Требуется опасаться одной особенности этого автомата. Ранее я подчеркивал, что в DXF-файле описанию каждого объекта предшествует появление маркера 0. Это единственный случай появления

маркера 0 в секции объектов ENTITIES. Поэтому, обнаружение вашей программой маркера 0 обозначает, что найдено начало описания нового объекта. Но это также обозначает конец описания текущего объекта и то, что настало время для обработки накопленных данных. Поскольку последующие разнообразные данные в DXF-файле являются идентификаторами следующих объектов, управление синтаксического разбора объекта должно быть оставлено в состоянии обнаружения идентификатора следующего объекта.

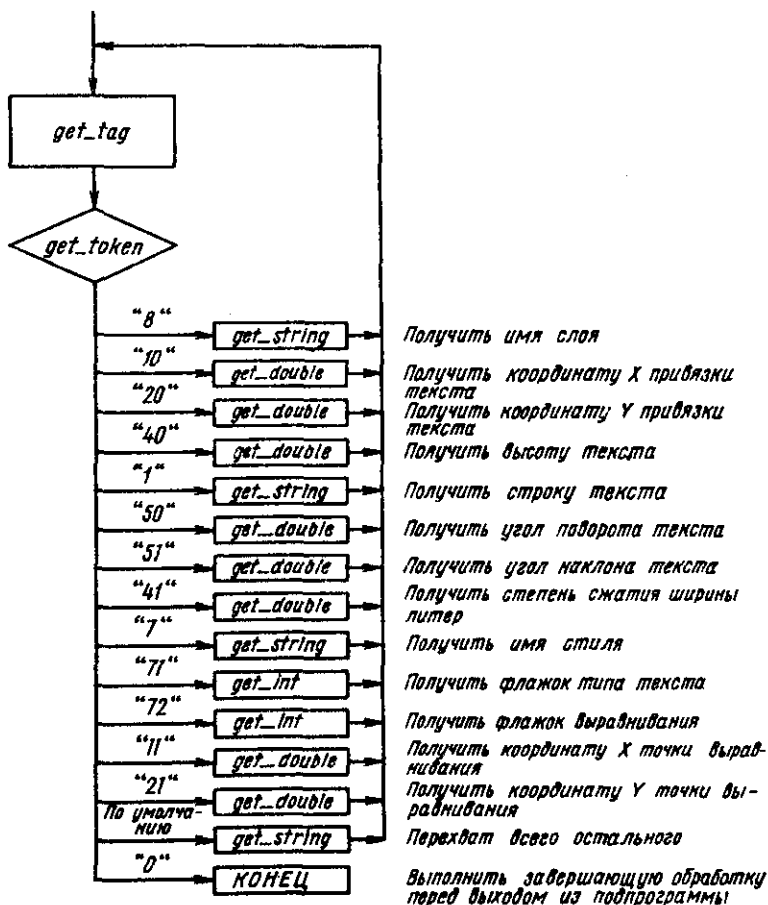


Рис. 5.12. Диаграмма состояний для программы

разбора объекта TEXT

```
quit = FALSE;
while(!quit)
{
    t = get_tag_token(c = get_tag(f));
    switch(t)
    {
```



```

case ZERO      : if (te->lyr == NULL) te->lyr = lyr;
                  if (te->text && *(te->text)) te->lt = NULL;
                  quit = TRUE;
                  break;
case EIGHT     : te->lyr = get_string(f);
                  break;
case TEN       : te->ix = get_double(f);
                  break;
case TWENTY    : te->iy = get_double(f);
                  break;
case FORTY     : te->height = get_double(f);
                  break;
case ONE       : te->text = get_string(f);
                  break;
case FIFTY     : te->angle = get_double(f);
                  break;
case FORTY1    : te->x_scale = get_double(f);
                  break;
case FIFTY1    : te->oblique = get_double(f);
                  break;
case SEVEN     : t_font = find_style(ts = get_string(f));
                  if (t_font)
                      te->font = t_font;
                  else
                      printf ( "Шрифт STANDARD используется
                               вместо %s шрифта\n", ts);
                  break;
case SEVENTY   : te->att_flags = get_int(f);
                  break;
case SEVENTY1  : te->gen_flags = get_int(f);
                  break;
case SEVENTY2  : te->justify = get_int(f);
                  break;
case ELEVEN    : te->ax = get_double(f);
                  break;
case TWENTY1   : te->ay = get_double(f);
                  break;
}

```

Лист. 5.52. Программа синтаксического разбора для объекта TEXT

На лист. 5.52 представлена программная реализация конечного автомата синтаксического разбора для объекта TEXT. Заменяя значения и поля структуры данных на найденные в других таблицах, вы сможете разработать программу синтаксического разбора для любого объекта.

Листинг содержит обращения к функциям преобразования типов: `get_double()` — получить действительное; `get_string()` — получить

строку; `get_int()` — получить целое. Перед вызовом указанных функций программа уже декодировала встреченный в DXF-файле маркер. Эти функции считывают очередную строку DXF-файла, в которой находится значение. Рассматриваемые функции отвечают за преобразование в соответствующий тип данных символьной информации в коде ASCII; после завершения своей работы они возвращают преобразованное значение. При преобразовании в символьную строку функция возвращает указатель, значением которого является ссылка на таблицу строк, в которой находится строка. На лист. 5.53-5.55 приведены исходные тексты этих функций.

```
double get_double(f)
FILE      *f;
{
    return atof(get_tag(f));
}
```

Лист. 5.53. Функция `get_double()`

```
char *get_string(f)
FILE      *f;
{
    char    s[256];

    fgets(s, 255, f);
    if (s[strlen(s)-1] == '\n') s[strlen(s)-1] = '\0';
    return locate(s);
}
```

Лист. 5.54. Функция `get_string()`

```
int get_int(f)
FILE      *f;
{
    return atoi(get_tag(f));
}
```

Лист. 5.55. Функция `get_int()`

Когда программа находится в состоянии, в которое она переходит при обнаружении маркера 0, наступает момент сделать что-либо с полученными данными объекта ТЕХТ. Одно из возможных предложений — обработать текст каким-либо образом. Для этого надо просто передать заполненную структуру данных специальной процедуре. После завершения работы процедуры программе больше не нужны данные объекта ТЕХТ и вы можете освободить зарезервированный блок памяти. Обработка объекта завершена.

Я провел эту часть нашего обсуждения в стиле "снизу вверх". Мы уже увидели изящные детали программ синтаксического разбора и структуры данных, но еще не рассматривали управляющие факторы. Поскольку основным в данной главе является синтаксический разбор DXF-файлов, я чувствую, что внимание привлекают именно эти подробности.

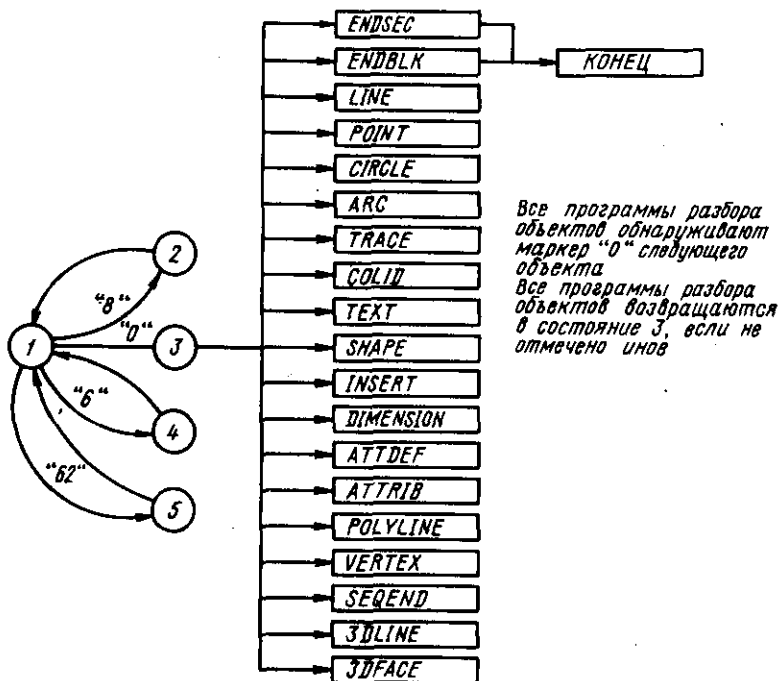


Рис. 5.13. Диаграмма конечных состояний для центральной программы разбора секции объектов

Программы синтаксического разбора объектов объединяет центральный управляющий модуль разбора секции объектов. Каждая другая из основных секций программы имела свою управляющую секцию, и теперь настал момент рассмотреть управление разбором секции объектов. Подобно другим управляющим модулям, о которых мы говорили в этой главе, я разработал на базе конечного автомата управляющую процедуру синтаксического разбора секции объектов. Я исходил из того, что работа программы, осуществляющей считывание данных из DXF-файла, не может зависеть от появления объектов любой секции в том порядке, в котором они показаны в справочном руководстве системы AutoCAD. Это, кстати, явно указывается и в руководстве. Отлично, то же самое, но только в большей степени мы собирались увидеть и в секции объектов ENTITIES. Мо-

жете вы представить DXF-файл, в котором описание всех объектов LINE (отрезок) появляется раньше, чем описание объектов POINT (точка) или любых других объектов (и т.д.)? Действительно, мы не можем ожидать появления объектов в каком-либо определенном порядке или того, что какой-нибудь конкретный объект вообще появится!

На рис.5.13 показана модель центральной программы синтаксического разбора секции объектов, в которой в качестве событий, вызывающих переход конечного автомата в новое состояние, используется список объектов, представленный в табл. 5.7. На лист. 5.56 приведена программная реализация этой модели. Представленная программа будет модифицирована в следующем параграфе для обработки определений блоков. На данном листинге (лист. 5.56) программа, вызываемая в очередном состоянии конечного автомата, полностью выполняет синтаксический разбор и обработку данных.

```

entities(f,blk)
FILE      *f;
block_type  *blk;
{
    int      t;
    int      quit;
    char     *c;
    int      state;
    char     *lyr;
    generic  *en;
    line_type *vl;
    int      closed_pline; /* делается глобальным */

    pl = NULL;
    state = 1;
    quit = FALSE;
    lyr = NULL;
    while(!quit)
    {
        t = get_tag_token(c = get_tag(f));
        switch (state)
        {
            case 1 : if (t == ZERO) state=3;
                     if (t == SIX)  state=4;
                     if (t == EIGHT) state=2;
                     break;
            case 2 : state = 1;
                     lyr = locate(c);
                     break;
            case 3 : state=1;
                     en = NULL;

```

```

switch(t)
{
    case ENDOSEC :
    case ENDBLK : quit = TRUE;
                  break;
    case LINE    : state = 3;
                  en = (generic *)line_parser(dxf, lyr);
                  break;
    case POINT   : state = 3;
                  en = (generic *)point_parser(dxf, lyr);
                  break;
    case CIRCLE  : state = 3;
                  en = (generic *)circle_parser(dxf, lyr);
                  break;
    case ARC     : state = 3;
                  en = (generic *)arc_parser(dxf, lyr);
                  break;
    case TRACE   : state = 3;
                  en = (generic *)trace_parser(dxf, lyr);
                  break;
    case SOLID   : state = 3;
                  en = (generic *)solid_parser(dxf, lyr);
                  break;
    case TEXT    : state = 3;
                  en = (generic *)text_parser(dxf, lyr);
                  break;
    case SHAPE   : state = 3;
                  en = (generic *)shape_parser(dxf, lyr);
                  break;
    case INSERT  : state = 3;
                  en = (generic *)insert_parser(dxf, lyr);
                  break;
    case DIMENSION : state = 3;
                  en = (generic *)dimen_parser(dxf, lyr);
                  break;
    case ATTDEF  : state = 3;
                  en = (generic *)attdef_parser(dxf, lyr);
                  break;
    case ATTRIB  : state = 3;
                  en = (generic *)attrib_parser(dxf, lyr);
                  break;
    case POLYLINE : state = 3;
                  pl = pline_parser(dxf, lyr);
                  break;
    case VERTEX  : state = 3;
                  vertex_parser(dxf, pl, lyr);
                  break;
}

```

```

case SEQEND : state = 1;
              if (pl) en = pl;
              break;
            }
          }
        }
      }
    }
  }
}

```

Лист. 5.56. Центральная управляющая программа разбора секции объектов

Чтение секции блоков BLOCKS

Вспомним, какие два действия мы могли бы совершить над объектом. Первое - это обработать его сразу, второе - сохранить информацию о нем. Такая ситуация возникает при синтаксическом разборе секции блоков BLOCKS.

Блок представляет собой поименованную группу объектов, которые могут появляться вместе на чертеже любое число раз. Каждое появление блока на чертеже является отдельным вхождением образца блока. Образец блока создается на чертеже по команде INSERT (вставить). Включенные в блок объекты формируют заготовку, которая в очередной раз дублируется системой AutoCAD.

Программа синтаксического разбора секции BLOCKS заимствует отдельные свои части у программы разбора секции TABLE и целиком использует программу разбора секции объектов для организации своего собственного описания объектов.

Такое двойственное состояние наполовину таблицы, наполовину объекта частично иллюстрирует природу блока. Описания блока сведены в табличную структуру данных. В этой структуре мы найдем имя блока, слой, на котором расположен блок и набор базовых координат. Такого типа информация могла быть найдена и в других таблицах. Тем не менее в блоке содержится нечто такое, что полностью отличает его от других структур данных. В блоке содержатся объекты - любого типа и в любом количестве. Как раз в этом случае к нам на помощь придет программа синтаксического разбора объектов. Мы должны понять, что блок подобен объекту, и затем сохранить эти объекты для последующего использования (например, для создания образца). На лист. 5.57 показано типичное описание блока, встречающееся в DXF-файле.

```

0
BLOCK
8
0
2
A_BLOCK
70

```

```

64
10
0.000000
20
0.000000

```

Лист. 5.57. Пример описания блока в DXF-файле

В данном случае описан блок с именем **A_BLOCK**. Блок создан на слое 0. Блоки созданные на слое 0, обладают большой гибкостью при их вставке в чертеж. В табл. 5.8 приведены маркеры, встречающиеся внутри описания блока.

Таблица 5.8. Маркеры, используемые при описании блока

2	STRING	Имя блока
70	INTEGER	Флажки блока
		1 = анонимный (штриховка или размер)
10	REAL	Базовая координата X
20	REAL	Базовая координата Y

На лист. 5.58 показана структура данных для хранения информации о блоке. Подобно таблице объектов, я разработал структуру данных для блока, которая может включаться в связанный список блоков. Это позволяет по мере необходимости обеспечивать доступ к отдельному элементу списка. Каждое поле в структуре данных соответствует полю описания блока, определяемого парой маркер — значение. На рис. 5.14 приведена диаграмма состояний для программы разбора секции блока. На лист. 5.59 показана программная реализация этой диаграммы.

```

struct k
{
    struct k      *next;
    int           what;
    char          *name;
    int           flags;
    struct e      *lyr;
    struct d      *ltype;
    double        base_x;
    double        base_y;
    int           *ent;
}
block_type:

```

Лист. 5.58. Структура данных для блока

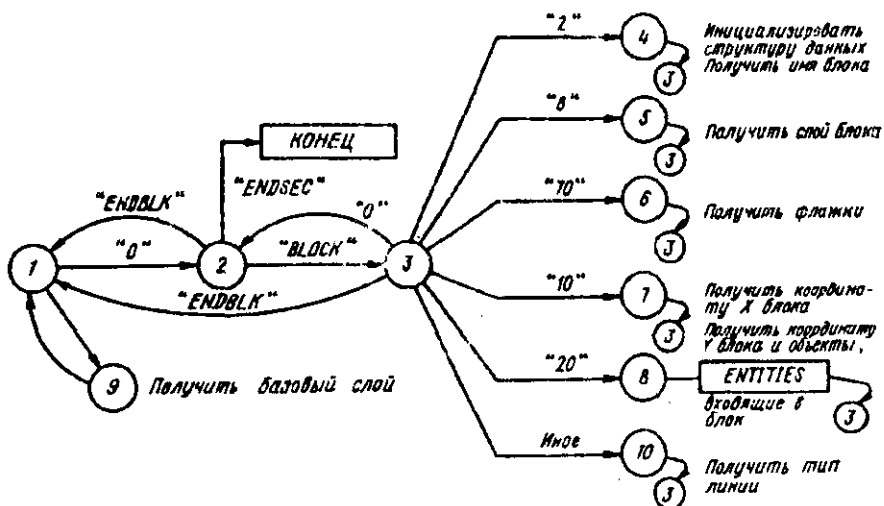


Рис. 5.14 Диаграмма состояний для программы разбора секции блоков

```

blocks (f)
FILE      *f;
{
  int      t;
  int      quit;
  char     *c;
  int      state;
  block_type *b;
  layer_type *ly;
  int      used_flag;

  b = NULL;
  ly = NULL;
  state = 1;
  quit = FALSE;
  while(!quit)
  {
    t = get_tag_token(c = get_tag(f));
    switch (state)
    {

```



```

case 1 : if (t == ZERO) state = 2;
        if (t == EIGHT) state = 9;
        b = NULL;
        break;
case 2 : state = 1;
        if (t == BLOCK) state = 3;
        if (t == ENDBLK) state = 1;
        if (t == ENDSEC) quit = TRUE;
        break;
case 3 : state = 1;
        switch(t)
        {
            case ZERO : state = 2;
                        break;
            case TWO : state = 4;
                       break;
            case EIGHT : state = 5;
                        break;
            case SEVENTY : state = 6;
                          break;
            case TEN : state = 7;
                      break;
            case TWENTY : state = 8;
                        break;
            case ENDBLK : state = 1;
                        break;
            default : state = 10;
                    break;
        }
        break;
case 4 : state = 3;
        if (b = (block_type *)calloc(1,sizeof(block_type)))
        {
            b->next = b_anchor;
            b_anchor = b;
            b->name = locate(c);
            b->flash = NULL;
            b->ent = NULL;
            if (ly)
            {
                b->lyr = ly;
                ly = NULL;
            }
        }
        else
            printf("Не хватает памяти -- BLOCKS\n");
        break;

```

```

case 5 : state = 3;
        if (b && !b->lyr)
            b->lyr = find_layer(c);
        else
            ly = find_layer(c);
        break;
case 6 : state = 3;
        used_flag = atoi(c);
        if (b) b->flags = used_flag;
        break;
case 7 : state = 3;
        if (b) b->base_x = atof(c);
        break;
case 8 : state = 3;
        if (b)
        {
            b->base_y = atof(c);
            entities(f,b);
        }
        else
            printf ("Нет связанных с блоком объектов\n");
        break;
case 9 : state = 1;
        break;
case 10 : state = 3;
        if (b && !b->ltype) b->ltype = find_ltype(c);
        break;
    }
}
}

```

Лист. 5.59. Программа разбора секции блоков

Следующая часть описания блока в DXF-файле представляет собой объекты, определяющие заготовку блока. Они имеют упорядоченную структуру. Последней в описании отдельного блока является запись ENDBLK. В тексте программы вы увидите, что назначение записи ENDBLK такое же, как и у записи ENDSEC (конец секции). Обе записи отмечают конец совокупности элементов. Запись ENDBLK помечает конец описания блока, а запись ENDSEC — конец описания графической секции. Секция блоков BLOCKS также включает запись ENDSEC, но, когда встречается эта запись, программа разбора секции объектов ENTITIES не должна находиться в активном состоянии.

Повторим еще раз: основное отличие между объектами, входящими в блок, и объектами, входящими в чертеж, состоит в том, что объекты, включенные в блок, должны сохраняться для последующего их восстановления на чертеже (представьте себе, что это подобно

воскресению). Мы воспользуемся этой особенностью, обеспечив место для сохранения объектов блока. В тексте примера таким местом является поле `ent` структуры данных. Когда программа синтаксического разбора секции **BLOCKS** вызывает программу разбора объектов, то последней передается в качестве аргумента ссылка на поле `ent`. Но, когда выполняется синтаксический разбор секции **ENTITIES**, аналогичная ссылка программе не передается.

Логика проста. Если имеется ссылка на блок, то вы сохраняете объект в списке; ссылка на начало этого списка содержится в поле `ent`. Если такой ссылки нет, то объект надо сразу же обработать.

Существует несколько способов выполнения такой обработки. Можно реализовать проверку ссылки на блок и затем последовательно вызывать программы обработки для каждого объекта. На лист. 5.60 показана реализация такого способа для объекта **TEXT**. Этот вариант программной реализации представляет собой фрагмент исходного текста основного управляющего модуля программы синтаксического разбора секции объектов (см. лист. 5.56). Программы обработки других объектов могут иметь аналогичную организацию.

```
case TEXT      state = 3;
               en = (generic *)text_parser(dxfl.yr);
               if (blk)
               {
                   (generic *)en->next = (generic *)blk->ent;
                   (generic *)blk->ent = (generic *)en;
               }
               else
               {
                   text_processor(en);
                   free(en);
               }
               break;
```

Лист. 5.60. Привязка объекта к блоку после синтаксического разбора

Еще один способ выполнения рассматриваемой обработки представляют отложенная проверка блока и вызов обрабатывающей программы, пока поток управления после завершения работы всех программ синтаксического разбора не объединится вновь. Это весьма близко к управляющей конструкции типа переключателя (**switch**). Теперь, чтобы привязать объект к блоку, нам нужна только одна неповторяющаяся последовательность операторов исходного текста программы. Приведение всех структур данных к родовому типу данных облегчает образование связей. Однако для того чтобы отделить друг от друга обрабатывающие программы для различных объектов, вам будет необходим еще один переключатель. Можно отметить, что при обработке различных объектов могут использоваться одни и те же программы. Например, объекты **SOLID** и **TRACE** в данный мо-

мент идентичны и для их обработки можно использовать одну и ту же программу. На лист. 5.61 показано образование связей между блоками с помощью ссылок, а также конструкция второго переключателя. Привязка объекта к блоку, в который входит данный объект, завершает формирование блока.

```

if (blk)
{
    (generic *)en->next = (generic *)blk->ent;
    (generic *)blk->ent = (generic *)en;
}
else
{
    switch(t)
    {
        case LINE      : line_processor(en);
                        break;
        case POINT     : point_processor(en);
                        break;
        case CIRCLE    : circle_processor(en);
                        break;
        case ARC       : arc_processor(en);
                        break;
        case TRACE     :
        case SOLID     : solid_processor(en);
                        break;
        case ATTDEF    :
        case ATTRIB    :
        case TEXT      : text_processor(en);
                        break;
        case SHAPE     : shape_processor(en);
                        break;
        case INSERT    : insert_processor(en);
                        break;
        case DIMENSION : dimen_processor(en);
                        break;
        case SEQEND    : if (en) pline_processor(en);
                        break;
    }
    if (en->ident == POLYLINE) pline_free(en);
    free(en);
}

```

Лист. 5.61. Привязка объекта к блоку в одной общей точке программы

Вы можете поинтересоваться, что же делать дальше с сохраненными в структуре данных блока объектами блока. Детали блоков становятся необходимы, когда вашей программе во время синтаксического разбора секции объектов ENTITIES встретится

инструкция INSERT (вставить). Все инструкции INSERT, обнаруживаемые в секции BLOCKS, сохраняются, так же как и все другие элементы. Программа обработки (не программа разбора) инструкции INSERT представляет собой нечто особенное. Программа обработки инструкции INSERT должна выполнить для вставки заданного блока в чертеж следующие действия:

- найти блок;
- получить список входящих в блок объектов;
- вызвать программу обработки для каждого объекта.

Поскольку мы для последующего использования поместили объекты блока в связанный список, то необходимо найти требуемый блок в этом списке. Исходный текст программы на лист. 5.62 аналогичен тексту программы для поиска в таблице. В данной программе в качестве ссылки на следующий элемент списка используется переменная `bl_anchor`. Реализован следующий алгоритм: с помощью ссылок осуществляется переход от одного блока к другому и выполняется сравнение имени блока с именем, указанным в запросе. При обнаружении соответствия программа возвращает адрес найденного блока как значение указателя. В противном случае программа возвращает значение указателя NULL обозначающее, что соответствие не найдено.

```
block_type *find_block(s)
char      *s;
{
    block_type  *b;

    b = b_anchor;
    while (b)
    {
        if (!strcmp(s,b->name)) return b;
        b = b->next;
    }
    return NULL;
}
```

Лист. 5.62. Поиск блока

На лист. 5.63 приведен текст программы, которая определяет адрес начала списка входящих в блок объектов и вызывает соответствующие программы для обработки каждого объекта. Если для вашего приложения требуются геометрические и тригонометрические преобразования, то вам нужно при вычислениях использовать координаты точки привязки (вставки), указанные в инструкции INSERT, и координаты базовой точки блока.

```
block_type  *b;
generic     *this;
b = find_block(inserted->block_name);
this = (generic *)b->ent;
while (this)
{
    switch (this->what)
    {
```

```

case LINE      : line_processor(this);
                  break;
case POINT     : point_processor(this);
                  break;
case CIRCLE    : circle_processor(this);
                  break;
case ARC       : arc_processor(this);
                  break;
case TRACE     :
case SOLID     : solid_processor(this);
                  break;
case TEXT      : text_processor(this);
                  break;
case SHAPE     : shape_processor(this);
                  break;
case INSERT    : insert_processor(this);
                  break;
case ATTRIB    : attrib_processor(this);
                  break;
case ATTDEF    : attdef_processor(this);
                  break;
case POLYLINE  : pline_processor(this);
                  break;
case DIMENSION : dimen_processor(this);
                  break;
default        : printf
                  ("В вставленном блоке найден неизвестный объект\n");
                  break;
}
this = (generic *)this->next;
}

```

Лист. 5.63. Обработка инструкции INSERT

Таблицы описания компонентов DXF-файлов

Приведенные ниже таблицы содержат обещанные ранее описания всех компонентов DXF-файлов. Создаваемые программы для разбора описания любого компонента должны выполнять выборку номера маркера и включение его в конечный автомат, осуществляя далее соответствующее преобразование данных и сохранение полученного результата в структуре данных.

Всегда по отношению к чему-либо имеется исключение. В данном случае исключением являются полилинии. Полилинии - это сложные объекты. Они имеют три входящих в них компонента: POLYLINE (полилиния), VERTEX (вершина) и SEQEND (конец последовательности). Компонент POLYLINE является основной частью объекта. С точки зрения программы обработки он представляет собой объект. С этим компонентом связываются объекты VERTEX. Это место, где проявляется основ-

ная сложность. При обнаружении объектов VERTEX должен быть произведен синтаксический разбор (как показано в таблицах) и данные присоединены к объекту POLYLINE. Это объясняет почему программе разбора объекта VERTEX передается дополнительный параметр. Этим параметром является ссылка на объект POLYLINE, с помощью которой к нему может быть присоединен объект VERTEX. Таким образом образуется список вершин. Добавляемая к списку вершина попадает в конец списка. Инструкция SEQEND помечает конец описания объекта POLYLINE. При обнаружении этой инструкции может быть вызвана программа обработки полилинии.

В первую таблицу (табл. 5.9) включены маркеры и значения, которые можно встретить в описании любого объекта. Это необязательные и излишние значения. Они не появляются в каждом объекте, но следует предусмотреть их возможную обработку.

Остальные таблицы (табл. 5.10 - 5.27) содержат описания объектов.

Таблица 5.9. Специальные маркеры

6	STRING	Имя типа линии (по умолчанию BYLAYER)
38	REAL	Уровень возвышения (по умолчанию = 0)
39	REAL	Высота (толщина) (по умолчанию = 0)
62	INTEGER	Цвет (по умолчанию BYLAYER)
8	STRING	Имя слоя

Таблица 5.10. Маркеры, используемые при описании объекта LINE

10	REAL	Координата X начальной точки
20	REAL	Координата Y начальной точки
11	REAL	Координата X конечной точки
21	REAL	Координата Y конечной точки

Таблица 5.11. Маркеры, используемые при описании объекта POINT

10	REAL	Координата X
20	REAL	Координата Y

Таблица 5.12. Маркеры, используемые при описании объекта CIRCLE

10	REAL	Координата X центра
20	REAL	Координата Y центра
40	REAL	Радиус

Таблица 5.13. Маркеры, используемые при описании объекта ARC

10	REAL	Координата X центра
20	REAL	Координата Y центра
40	REAL	Радиус

50	REAL	Угол начала
51	REAL	Угол конца

Таблица 5.14. Маркеры, используемые при описании объекта TRACE

10	REAL	Координата X первого угла
20	REAL	Координата Y первого угла
11	REAL	Координата X второго угла
21	REAL	Координата Y второго угла
12	REAL	Координата X третьего угла
22	REAL	Координата Y третьего угла
13	REAL	Координата X четвертого угла
23	REAL	Координата Y четвертого угла

Таблица 5.15. Маркеры, используемые при описании объекта SOLID

10	REAL	Координата X первого угла
20	REAL	Координата Y первого угла
11	REAL	Координата X второго угла
21	REAL	Координата Y второго угла
12	REAL	Координата X третьего угла
22	REAL	Координата Y третьего угла
13	REAL	Координата X четвертого угла
23	REAL	Координата Y четвертого угла

Таблица 5.16. Маркеры, используемые при описании объекта TEXT

10	REAL	Координата X точки вставки
20	REAL	Координата Y точки вставки
40	REAL	Высота
1	STRING	Текст
50	REAL	Угол поворота текста (по умолчанию = 0)
41	REAL	Линейный масштаб по оси X (по умолчанию = 1)
51	REAL	Угол наклона литер (по умолчанию = 0)
7	STRING	Имя стиля начертания (по умолчанию = STANDARD)
71	INTEGER	Флажки генерации текста (по умолчанию = 0) 2 = Зеркальное отображение по оси X 4 = Перевернутое отображение
72	INTEGER	Флажки выравнивания текста (по умолчанию = 0) 0 = Выровненный влево 1 = Отцентрированный 2 = Выровненный вправо 3 = Выровненный относительно указанной точки 4 = Середина в указанной точке 5 = Вписанный

11	REAL	Координата X точки выравнивания
21	REAL	Координата Y точки выравнивания

Таблица 5.17. Маркеры, используемые при описании объекта SHAPE

10	REAL	Координата X точки вставки
20	REAL	Координата Y точки вставки
40	REAL	Размер
1	STRING	Имя графического символа (формы)
50	REAL	Угол поворота (по умолчанию = 0)
41	REAL	Линейный масштаб по оси X (по умолчанию = 1)
51	REAL	Угол наклона (по умолчанию = 0)

Таблица 5.18. Маркеры, используемые с инструкцией ENDBLK

Нет маркеров

Таблица 5.19. Маркеры, используемые с инструкцией INSERT

66	INTEGER	Присутствуют атрибуты
2	STRING	Имя блока
10	REAL	Координата X точки вставки
20	REAL	Координата Y точки вставки
41	REAL	Линейный масштаб по оси X (по умолчанию = 1)
42	REAL	Линейный масштаб по оси Y (по умолчанию = 1)
43	REAL	Линейный масштаб по оси Z (по умолчанию = 1)
50	REAL	Угол поворота (по умолчанию = 0)
70	INTEGER	Число столбцов (по умолчанию = 1)
71	INTEGER	Число рядов (по умолчанию = 1)
44	REAL	Шаг между столбцами (по умолчанию = 1)
45	REAL	Шаг между рядами (по умолчанию = 1)

Таблица 5.20. Маркеры, используемые с инструкцией ATTDEF

10	REAL	Координата X точки вставки текста
20	REAL	Координата Y точки вставки текста
40	REAL	Высота текста
1	STRING	Значение по умолчанию
3	STRING	Приглашение ко вводу
2	STRING	Имя атрибута
70	INTEGER	Флажки атрибута
		1 = Невидимый
		2 = Неизменяемый
		3 = Невидимый и неизменяемый
73	INTEGER	Длина поля (по умолчанию = 0)
50	REAL	Угол поворота текста (по умолчанию = 0)

41	REAL	Линейный масштаб по оси X (по умолчанию = 1)
51	REAL	Угол наклона литер (по умолчанию = 0)
7	STRING	Имя стиля начертания (по умолчанию = STANDART)
71	INTEGER	Флажки генерации текста (по умолчанию = 0) 2 = Зеркальное отображение по оси X 4 = Перевернутое отображение
72	INTEGER	Флажки выравнивания текста (по умолчанию = 0) 0 = Выровненный влево 1 = Отцентрированный 2 = Выровненный вправо 3 = Выровненный относительно указанной точки 4 = Середина в указанной точке 5 = Вписанный
11	REAL	Координата X точки выравнивания
21	REAL	Координата Y точки выравнивания

Таблица 5.21. Маркеры, используемые с инструкцией ATTRIB

10	REAL	Координата X точки вставки текста
20	REAL	Координата Y точки вставки текста
40	REAL	Высота текста
1	STRING	Значение
2	STRING	Имя атрибута
70	INTEGER	Флажки атрибута 1 = Невидимый 2 = Неизменяемый 3 = Невидимый и неизменяемый
73	INTEGER	Длина поля (по умолчанию = 0)
50	REAL	Угол поворота текста (по умолчанию = 0)
41	REAL	Линейный масштаб по оси X (по умолчанию = 1)
51	REAL	Угол наклона литер (по умолчанию = 0)
7	STRING	Имя стиля начертания (по умолчанию = STANDART)
71	INTEGER	Флажки генерации текста (по умолчанию = 0) 2 = Зеркальное отображение по оси X 4 = Перевернутое отображение
72	INTEGER	Флажки выравнивания текста (по умолчанию = 0) 0 = Выровненный влево 1 = Отцентрированный

		2 = Выравненный вправо
		3 = Выравненный относительно указанной точки
		4 = Середина в указанной точке
		5 = Вписанный
11	REAL	Координата X точки выравнивания
21	REAL	Координата Y точки выравнивания

Таблица 5.22. Маркеры, используемые при описании объекта POLYLINE

70	INTEGER	Флажки полилинии
		1 = Замкнутая полилиния
		2 = Вершины соединены кривыми
66	INTEGER	Присутствуют сложные объекты
40	REAL	Начальная ширина
41	REAL	Конечная ширина

Таблица 5.23. Маркеры, используемые при описании объекта VERTEX

10	REAL	Координата X
20	REAL	Координата Y
40	REAL	Начальная ширина (по умолчанию = 0)
41	REAL	Конечная ширина (по умолчанию = 0)
42	REAL	Выпуклость
70	INTEGER	Флажки вершин
		1 = Вершины соединены кривыми
50	REAL	Угол касательной к кривой

Таблица 5.24. Маркеры, используемые с инструкцией SEQEND

Нет маркеров

Таблица 5.25. Маркеры, используемые с инструкцией DIMENSION

2	STRING	Имя блока изображения размера
10	REAL	Координата X базовой выносной линии
20	REAL	Координата Y базовой выносной линии
11	REAL	Координата X середины текста размера
21	REAL	Координата Y середины текста размера
12	REAL	Координата X точки вставки размерной линии
22	REAL	Координата Y точки вставки размерной линии
70	INTEGER	Тип размера
		0 = Повернутый, горизонтальный или вертикальный
		1 = Параллельный
		2 = Угловой

		3 = Диаметр
		4 = Радиус
1	STRING	Заданный пользователем текст
13	REAL	Координата X первой выносной линии
23	REAL	Координата Y первой выносной линии
14	REAL	Координата X второй выносной линии
24	REAL	Координата Y второй выносной линии
15	REAL	Координата X точки задания диаметра
25	REAL	Координата Y точки задания диаметра
16	REAL	Координата X точки задания дуги
26	REAL	Координата Y точки задания дуги
40	REAL	Длина выносной линии для диаметра
50	REAL	Угол поворота

Таблица 5.26. Маркеры, используемые при описании объекта 3DLINE

10	REAL	Координата X начальной точки
20	REAL	Координата Y начальной точки
30	REAL	Координата Z начальной точки
11	REAL	Координата X конечной точки
21	REAL	Координата Y конечной точки
31	REAL	Координата Z конечной точки

Таблица 5.27. Маркеры, используемые при описании объекта 3DFACE

10	REAL	Координата X первого угла
20	REAL	Координата Y первого угла
30	REAL	Координата Z первого угла
11	REAL	Координата X второго угла
21	REAL	Координата Y второго угла
31	REAL	Координата Z второго угла
12	REAL	Координата X третьего угла
22	REAL	Координата Y третьего угла
32	REAL	Координата Z третьего угла
13	REAL	Координата X четвертого угла
23	REAL	Координата Y четвертого угла
33	REAL	Координата Z четвертого угла
70	INTEGER	Флажки видимости кромки поверхности

Бит установлен	Значение
0	Первая кромка невидима
1	Вторая кромка невидима
2	Третья кромка невидима
3	Четвертая кромка невидима

Глава 6

АТТРИБУТЫ

Наиболее важным усовершенствованием в системе AutoCAD, с точки зрения разработчиков пакетов прикладных программ, явилось введение в системе AutoCAD версии 2.0 атрибутов. Однако в связи с этим замечанием мне следует немного перестраховаться, поскольку язык АвтоЛИСП придает особую функциональность системе AutoCAD, что уже описывалось неоднократно. Скажем, что с точки зрения организации внешних программ атрибуты придают им большую гибкость, что отсутствует в других графических пакетах программ. Накопленный опыт привел к констатации факта, что применение атрибутов может завести настолько далеко, что ограничит перенос некоторых прикладных программ, написанных для системы AutoCAD, для их использования с другими системами автоматизированного проектирования.

Что же такое атрибуты? Атрибуты — это определяемые пользователем специфические данные, которые присоединяются к описанию блока в чертеже.

Почему же атрибуты вызывают такой повышенный интерес? То, что введение атрибутов в систему AutoCAD позволяет извлекать из графической базы данных информацию для выпуска спецификаций применяемых материалов, не является весомым аргументом. Это основа для примеров, описываемых в руководстве пользователя. Данные, присоединенные к различным блокам, содержащим графическую информацию о мебели, описывают разнообразные характеристики этих изделий. В блоке может быть несколько атрибутов. Каждый атрибут содержит какой-то один "кусочек" информации. Несколько атрибутов описывают неграфические свойства мебели. К таким свойствам могут быть отнесены следующие:

- артикул изделия;
- имя производителя;
- складской номер;
- имеющиеся цвета;
- время поставки;
- цена;
- процент скидки.

Эти значения не обязательно должны отображаться на экране так, чтобы их видел каждый. Хранение некоторой информации в скрытом виде может явиться хорошим преимуществом. Кроме того вся эта информация может загромождать экран, заняв все свободное

место. Стул легко можно пометить с помощью видимого атрибута "Рабочий стул" или "Обеденный стул". Прочая информация осталась бы по-прежнему доступной, но была бы просто скрыта.

После завершения работы над чертежом, содержащим атрибуты, конструктор имеет для передачи внешней программе приготовленную системой AutoCAD информацию об атрибутах. Эта передача выполняется по команде **ATTTEXT**. Название команды **ATTTEXT** означает **ATTRIBUTE EXTRACTION** (извлечение атрибута). Руководствуясь заданным пользователем шаблоном вывода, система AutoCAD просматривает свои активные базы данных, осуществляя поиск блоков, содержащих одно или более полей, установленных в шаблоне. После этого система AutoCAD выводит форматированные записи в выходной файл.

Это, несомненно, вызывает некоторые вопросы. Одним из них должен быть такой, что находится в файле описания шаблонов атрибутов? Еще одним вопросом мог быть: каков формат выходного файла?

В файле описания шаблонов атрибутов содержатся имена атрибутов, собранные во время выполнения операции извлечения атрибутов (команда **ATTTEXT**), и данные, которые представляют индивидуальные атрибуты. Эти данные по своей природе являются либо числовыми (**NUMERIC**), либо символьными (**CHARACTER**). Файл описания шаблонов атрибутов - это список записей. Каждая запись содержит имя атрибута и относящиеся к нему данные. Имя атрибута - это родовое название атрибута. Имя не является значением, присваиваемым атрибуту (т.е. то, что вы хотите извлечь). Именем атрибута может быть нечто подобное:

ITEM_NAME	(наименование изделия)
MANUFACTURER	(производитель)
STOCK_NUMBER	(складской номер)
PATTERN_NAME	(имя шаблона)

В другом поле записи шаблона атрибута указывается формат представления данных. Поле формата имеет три подполя:

тип данных;

число разрядов целой части числа;

число разрядов дробной части числа.

На рис. 6.1 показана эта структура данных

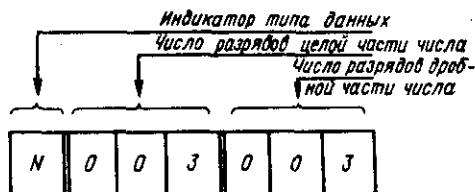


Рис. 6.1. Символьные обозначения в шаблоне атрибута

Поле типа данных имеет достаточно простой формат. В поле содержится либо буква C (Character) для символьного, либо буква N (Numeric) для числового типа. Любой атрибут, который всегда будет характеризоваться числовой информацией, может использовать формат N. Тем не менее если атрибут время от времени может изменяться, более практичным может быть использование символьной формы. Например, для атрибута "цена" можно предположить, что выражением этого атрибута может быть только какое-либо число, но для кого-либо не составит труда ввести в это поле вместо числового значения замечание типа NO CHARGE или NC (бесплатно), которое все перепутает. При обнаружении такого смещения данных система AutoCAD выдает предупреждающие сообщения. Пользователи пугаются предупреждающих сообщений. Напуганные пользователи обращаются за помощью к группе технического сопровождения. Работа группы сопровождения потребует и времени, и денег. Усвоив эту истину, я хотел бы предложить, чтобы следующие атрибуты были числовыми: атрибуты, значение которых константа (значение присваивается во время формирования блока), атрибуты, которые создаются автоматически другой программой, и атрибуты, являющиеся внутренними для системы AutoCAD (наиболее общий пример - координаты). Выполнение операции извлечения всех других атрибутов следует производить в символьном режиме. Введенные пользователем пробелы даже перед правильным числом могут быть достаточной причиной того, чтобы возникли проблемы с системой AutoCAD.

Следующие три символьные позиции во втором поле шаблона указывают число цифр или разрядов, расположенных слева от десятичной точки числа. Поскольку символьная строка и, следовательно, целая часть числа смещаются к левой границе поля, это означает, что в поле должны быть только значимые разряды числа. В этом подполе должны быть три цифры. Незадействованные разряды заполняются нулями. В табл. 6.1 показано описание символьного и целочисленных полей в шаблоне.

Таблица 6.1. Описание полей символьного и целочисленного формата в шаблоне

C008000 -- символьная строка из 8 символов (макс.)

C060000 -- символьная строка из 60 символов (макс.)

N003000 -- целое число в диапазоне от -99 до 999

Последние три символьные позиции показывают число значащих цифр после десятичной точки в числе. Это поле используется в случае представления действительных чисел. Значениями внутренних атрибутов, таких, как координаты, коэффициенты масштабирования

и углы поворота, являются действительные числа (числа с плавающей точкой).

Форма третьего подполя такая же, как и у второго поля шаблона. Все правила по заполнению предыдущего поля относятся и к рассматриваемому; поле должно вмещать три цифры. В табл. 6.2 показаны описания двух полей действительных чисел в шаблоне.

Таблица 6.2. Описание формата полей действительных чисел в шаблоне

N003003 -- действительное число типа xxxx fff

N004001 -- действительное число типа xxxx.f

Руководствуясь перечисленными правилами, мы можем создавать файл описания шаблонов атрибутов. Файл описания шаблонов атрибутов представляет собой список имен атрибутов и описаний форматов шаблонов. На лист. 6.1 представлен пример записей шаблонов атрибутов. Этот список может выглядеть несколько некрасиво, поскольку его поля не были выравнены по одной границе с помощью табуляции. Однажды я попробовал сделать такое выравнивание, но системе AutoCAD это чем-то "не понравилось". С тех пор я отказался от подобного наведения красоты. По сути я не знаю, допускают ли новые версии системы AutoCAD наличие лишних пробелов или табуляций в файлах описания шаблонов. Вы нашли что-то работающее и полагаетесь на это. Сказанное не означает, что вы должны прекратить поиск новых и лучших путей решения, но я стараюсь избегать старых ошибок.

```
BL:NAME C016000
BL:X N008004
BL:Y N008004
BL:LAYER C016000
BL:ORIENT N008004
BL:XSCALE N008004
BL:YSCALE N008004
STOCK_NUMBER C012000
FABRIC C020000
COLOR C012000
```

Лист. 6.1. Записи шаблонов атрибутов

Порядок записей в файле описания шаблонов атрибутов крайне важен. Порядок записей в выходном файле в точности повторяет порядок записей в файле описания шаблонов. Поскольку нет смысла рассматривать все записи не используя их, давайте исходить из предположения, что в вашей программе ожидается появление полей в подобном порядке. Я ввел суровые предупреждения в мое руководство для пользователей против изменения порядка следования атри-

бутов. Вы не можете гарантировать правильную работу вашей программы, если перепутаны входные данные.

Имеется несколько атрибутов, которые являются внутренними по отношению к системе AutoCAD. Вы можете иметь к ним доступ при выполнении операции извлечения атрибутов, если вы включили их описание в файл описания шаблонов атрибутов. В начале описания внутренних атрибутов помещается префикс BL. Эти атрибуты ссылаются непосредственно на значения, относящиеся к вставляемому блоку. В табл. 6.3 показаны такие атрибуты.

Таблица 6.3. Внутренние атрибуты и их шаблоны

BL:LEVEL	Nwww000
BL:NAME	Cwww000
BL:X	Nwwwfff
BL:Y	Nwwwfff
BL:LAYER	Cwww000
BL:ORIENT	Nwwwfff
BL:XSCALE	Nwwwfff
BL:YSCALE	Nwwwfff

Давайте рассмотрим описанные выше данные и сформулируем, что же мы хотим с ними сделать. Мы подготовили шаблоны для выполнения операции извлечения атрибутов. Все описания шаблонов содержатся в файле. Файл должен иметь расширение имени файла (или, иначе, тип файла) — TXT. Пусть файл описания шаблонов имеет имя SAMPLE.TXT. Предположительно мы также имеем активный чертеж в системе AutoCAD. То — что мы хотим сделать, — это извлечь информацию об атрибутах из блоков чертежа и поместить данную информацию в выходной файл для последующей обработки. Эта обработка может быть связана с обновлением базы данных или подготовкой отчета.

При последующих обсуждениях, в примерах программирования, в качестве формата выходного файла атрибутов будет применяться формат с разделением полей запятыми. Вы можете модифицировать алгоритм и по желанию применить формат с разделением полей пробелами.

Для того чтобы извлечь информацию об атрибутах из чертежа, воспользуйтесь командой ATTEXT. Введите в командной строке сис-

темы AutoCAD команду ATTEXT. Первым сообщением системы, которое вы увидите, будет:

CDF, SDF, or DXF Attribute extract (or Entities)? <C>

(CDF, SDF или DXF формат извлечения атрибутов (или Объектов)?)

Что это, загадочная команда или нечто иное? Ответ на сообщение вводом буквы **C** применяется для выходного формата с разделением полей запятыми. Значение команды, выполняемой по умолчанию, указанное в скобках, задает формат с разделением полей запятыми.

Ответ на сообщение вводом буквы **S** применяется для формата с разделением полей пробелами. Поля имеют фиксированный размер, что может не позволить вставить пробелы между полями. Если вы знакомы с языком программирования Фортран, то этот формат соответствует жесткому входному формату программ, написанных на Фортране.

Ответ на сообщение вводом буквы **D** применяется для DXF- формата выходного файла. В этом режиме записывается модифицированный DXF-файл для блоков, содержащих ссылки на атрибуты. Следующее основное обсуждение посвящено извлечению информации об атрибутах из DXF-файлов. Если вы будете придерживаться рассматриваемых при этом приемов, то вам не потребуется использовать данный способ извлечения атрибутов.

Ответ на сообщение вводом буквы **E** применяется для дополнительного выбора блоков, из которых будут извлекаться атрибуты перед выполнением обработки, в результате которой создается выходной файл. Команда ATTEXT позволяет производить выбор из ряда методов, чтобы сократить диапазон поиска при извлечении атрибутов. Когда выбор сделан, вы снова увидите сообщение с приглашением ко вводу для форматов типа CDF, SDF или DXF. Если ранее был задан формат типа CDF или SDF, то система AutoCAD потребует указать для выполнения операции извлечения атрибутов файл описания шаблонов атрибутов. Сообщение системы AutoCAD имеет следующий вид:

Template file <default>:

(Файл описания шаблонов <имя по умолчанию>:)

По умолчанию файла имя описания шаблонов атрибутов - это последнее используемое имя такого файла. Введите имя файла описания шаблонов атрибутов. Нет необходимости указывать расширение имени файла TXT. После этого система AutoCAD запросит имя выходного файла. Сообщение будет следующим:

Extract file name <drawing name>:

(Имя файла вывода значений атрибутов <имя чертежа>:

Именем по умолчанию для файла, содержащего извлеченные значения атрибутов, является имя чертежа. Расширение имени этого файла - TXT. Вы можете указать любое имя, которое бы вы хотели использовать для файла вывода значений атрибутов. Однако в связи с тем, что файлу будет присвоено расширение имени файла TXT независимо от того, захотите ли вы ему дать имя, используемое по умолчанию, или нет, не присваивайте файлу имя, совпадающее с именем файла описания шаблонов атрибутов. Вы потеряете файл описания шаблонов, когда система AutoCAD запишет в этот файл информацию об извлеченных атрибутах. Старайтесь использовать имя по умолчанию и не присваивайте вашему чертежу имя, такое же, как и у файла описания шаблонов атрибутов.

На самом деле, как только система AutoCAD записала файл атрибутов, не имеет значения, что вы будете делать дальше. Вы можете продолжить сеанс редактирования чертежа; вы можете завершить сеанс с сохранением или без сохранения измененной во время сеанса информации. Файл вывода значений атрибутов не прекратит свое существование, если вы закончите сеанс по команде QUIT (без сохранения измененной информации). На лист. 6.2 приведен типичный пример содержимого файла вывода значений атрибутов после выполнения команды ATTEXT.

"CHAIR",12.5,24.0,"FLOOR",0.0,1.0,1.0,"BR-549","M15-6","RED"

"COUCH",23.5,17.9,"FLOOR",90.0,1.0,1.0,"ML-123","M1-23","AQUA"

Лист. 6.2. Записи в файле вывода значений атрибутов

В оставшейся части обсуждения мы будем иметь дело с программой синтаксического разбора записей, содержащих информацию об извлеченных атрибутах. Позвольте мне сделать некоторые замечания по структуре записей в файле вывода значений атрибутов.

1. Каждая запись имеет одно и то же число полей.
2. Эти поля в точности соответствуют записям-шаблонам в файле описания шаблонов атрибутов.
3. Имеются пустые поля для атрибутов, не найденных в блоке
4. Поля располагаются в том же порядке, в каком они были описаны в файле описания шаблонов атрибутов.
5. Значения во всех полях представляют собой символьную информацию в коде ASCII.
6. В формате с разделением полей запятыми все поля отделяются друг от друга запятыми.
7. Символьные строки заключаются в кавычки. Запятые внутри кавычек не являются разделителями.

На рис. 6.2 показана часть записи извлеченных атрибутов. На этом рисунке приведена побайтовая иллюстрация структуры этой записи. Программа сразу считывает всю запись целиком в буфер ввода. Мы не будем выполнять считывание по одному полю, поскольку это требует много работы.

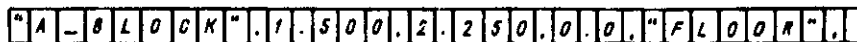


Рис. 6.2. Детали записи атрибута

Так как содержимое каждого поля представляет собой строку символов в коде ASCII и имеются дополнительные байты, содержащие запятые, которые не включаются в информационные строки, довольно просто увидеть, что при замене запятых символом NULL, вы получите несколько рядов коротких, отлично сформированных в стиле языка Си символьных строк. На рис. 6.3 показано выполненное преобразование.

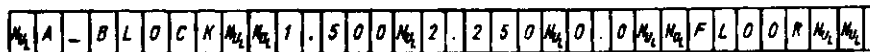


Рис. 6.3. Запись атрибутов с добавленными символами NULL

Наша задача завершится, когда мы для каждого поля сформируем переменную-указатель. И чтобы сделать жизнь еще прекрасней, мы поместим указатели в массив указателей, в котором их индекс будет соответствовать позиции в строке. Приступим к осуществлению этого.

Исходя из структуры файла описания шаблонов атрибутов, на лист. 6.3 приведены определения констант, которые мы будем использовать при обращении к таблице указателей. Для имен констант я заимствовал имена шаблонов атрибутов.

```
#define NUMBER_OF_FIELDS      10
#define BL_NAME                0
#define BL_X                   1
#define BL_Y                   2
#define BL_LAYER               3
#define BL_ORIENT              4
#define BL_XSCALE              5
#define BL_YSCALE              6
#define STOCK_NUMBER           7
#define FABRIC_STYLE            8
#define FABRIC_COLOR           9
```

Лист. 6.3. Определения индексов

На лист. 6.4 показан массив указателей для адресации полей в буфере ввода. Также показано объявление массива, являющегося буфером ввода. Поскольку эти массивы требуются для всех частей программы, лучше объявить их как глобальные переменные. На рис. 6.4 проиллюстрировано соотношение между рассмотренными данными. На лист. 6.5 представлено описание стратегии ввода информации об атрибутах, которой должна следовать программа.

```
char          *attribs[NUMBER_OF_FIELDS];
char          in_buf[1024];
```

Лист. 6.4. Переменные для нахождения атрибутов

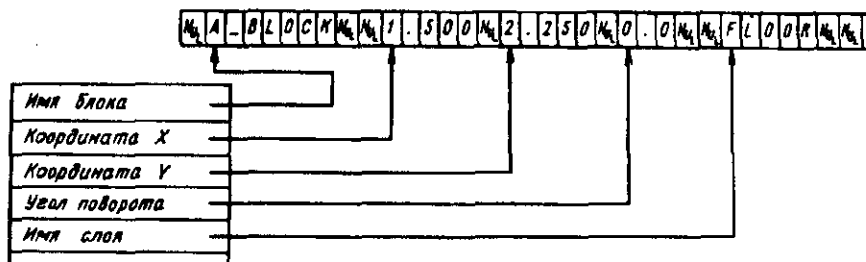


Рис. 6.4. Установление ссылок на запись атрибутов

До тех пор, пока есть входные записи, считать одну запись из файла вывода значений атрибутов, выделить в записи отдельные поля, установить указатели на каждое поле, используя указатели, получить данные из записи.

Лист. 6.5. Стратегия получения информации об атрибутах

Верхний уровень программы не намного сложнее. На лист. 6.6 показан исходный текст для верхнего уровня программы. Алгоритм выделения полей из записи будет рассмотрен подробно далее.

```
void read_attributes(fi)
FILE *fi;
{
    char in_buf[1024]; /* на ваш выбор -- локальный или глобальный */
    while (fgets(in_buf, 1023, fi) != NULL)
    {
        separate_fields(in_buf);
        process_findings(in_buf);
    }
}
```

Лист. 6.6. Ввод информации из файла вывода значений атрибутов

Процесс выделения полей связан с нахождением запятых и символов перевода строки и последующей их заменой на символ NULL. В момент, когда найдена запятая, следующий после запятой символ является первым символом следующей строки данных. Адрес этого символа должен быть сохранен в очередном элементе массива указателей. При нахождении каждой запятой вы должны одновременно идентифицировать следующее поле записи.

Другим символом, который должен привлечь внимание, являются двойные кавычки ("). Двойные кавычки применяются для выделения текстовых атрибутов. Всегда используется пара кавычек. Первые кавычки сообщают вам, что следует начать поиск вторых кавычек. Замените первые кавычки символом NULL и передвиньте соответствующий указатель в массиве на первый символ строки данных. Замените вторые кавычки символом NULL. Так как вторые кавычки завершают строку данных, никакой другой дополнительной обработки в этом случае не требуется. Когда вы снова перейдете к поиску запятых, все будет в полном порядке. В записи может встретиться символ перевода строки, который определяет конец строки. Замените этот символ на символ NULL.

Поиск прекращается, как только в конце записи будет встречен символ NULL. Если вы сохраняли соответствие между адресами строк и указателями, то вам следует также иметь список указателей на строки с информацией об атрибутах.

Первое поле начинается с первого символа в буфере ввода. Будем считать, что в байте перед этим символом была запятая. На лист. 6.7 показан исходный текст программы выделения полей.

```
void separate_fields(b)
char      *b;
{
    int      i;
    char      *s;

    /* установить указатель на первый атрибут */
    i = 0;
    attribs[i] = b;
    s = b;
    while (*s) /* для всех символов в буфере */
    {
        switch(*s)
        {
            case ',' : *s++ = '\0'; /* заменить на NULL */
                       attribs[++i] = s; /* следующий адрес
                                           таблицы */
                       break;
            case '\n' : *s++ = '\0'; /* заменить на NULL */
                       break;
        }
    }
}
```

```

        case '\\' : *s++ = '\\0';    /* заменить на NULL */
                attrs[i]++;
                /* установить указатель после кавычек */
                /* искать вторые кавычки */
                while (*s && (*s != '\\')) s++;
                /* заменить вторые кавычки на NULL */
                if (*s) *s++ = '\\0';
                break;
        default : *s++;    /* ничего не делать */
                break;
    }
}
}

```

Лист. 6.7. Выделение полей атрибутов

Как только программа завершила выделение полей, значения указателей устанавливаются в соответствии с расположением полей. Все поля представляют собой строки. В ходе дальнейшей обработки вы можете использовать программы преобразования для числовых полей, чтобы облегчить выполнение вычислений. На лист. 6.8 приведен фрагмент программы вывода данных, применяемой для отображения результатов выделения полей.

```

printf("Имя блока : %s\n", attrs[BL_NAME]);
printf("Точка вставки блока: %s.%s\n", attrs[BL_X],
        attrs[BL_Y]);
printf("Складской номер : %s\n", attrs[STOCK_NUMBER]);

```

Лист. 6.8. Использование информации об атрибутах

Перед тем как считать очередную запись, вы должны что-нибудь сделать с данными, содержащимися в буфере ввода. Следующая операция ввода уничтожит данные, находившиеся в буфере, и установленные ранее значения указателей станут неверными.

Применение данного способа извлечения атрибутов предполагает определенное доверие к пользователю при определении значений атрибутов, которые должны соответствовать типу и формату данных, описанных в шаблоне. Также предполагается, что никто до момента обработки не изменял порядок данных или сами данные в файле описания шаблонов атрибутов. Зная, что подобное никогда не случится, использование команды АТТЕХТ обеспечивает ясный и легкий (для программы) способ извлечения данных атрибутов.

Поскольку я пользовался командой АТТЕХТ для подготовки данных для прикладных программ, исходя из своего опыта, я могу сказать, что вы даже не можете представить себе, какие бывают фантастические случаи. О, будьте уверены, подавляющее число пользователей будут автоматически придерживаться инструкций, файл опи-

сания шаблонов отражает их стиль работы. Но всегда найдется несколько заблудших, которым потребуется просто немного другой подход к поставленной задаче. Их данные должны быть обязательно символьными, в то время как для всех других они числовые. Для подсчета числа апельсинов им нужны вместо целых чисел — действительные. Вы можете представить, к чему это приводит. Использование файла описания шаблонов при извлечении информации об атрибутах из DXF-файлов игнорируется, и вам передаются напрямую необработанные данные. В DXF-файлах все эти данные являются символьной информацией в коде ASCII. Нет никаких разделителей запятых (за исключением присутствующих в тексте). Нет никаких кавычек. То, что вы должны сделать, — это всего-то разобрать DXF-файл и сохранить значения атрибутов, когда они вам встретятся. Вздохи и стоны: "Вы всего-то должны разобрать ... D ... X ... F". Взгляните, если это у вас вызывает отвращение, то вернитесь обратно к извлечению атрибутов. Настоящие Программисты используют DXF-файлы (и только мазохисты работают с файлами чертежей — DWG-файлами).

При работе с DXF-файлами вы получаете информацию об атрибутах из двух источников. Первым является запись определения атрибута (ATTDEF) в секции описания блока. Вторым источником — запись атрибута (ATTRIB), которая следует после записи вставки блока (INSERT). Для каждого переменного (непостоянного) определения атрибута должна существовать одна запись об атрибуте.

Определения атрибутов

Запись определения атрибутов в DXF-файле находится только в секции блоков. Эта запись находится среди описаний линий и окружностей в каком-либо отдельном блоке. Не требуется, чтобы определения появлялись обязательно в начале или конце описаний других объектов.

Структура определения атрибута довольно схожа со структурой объекта текста (из разумных соображений). В обеих структурах указываются позиционное положение, ориентация, размер, шрифт и сам текст. Если эти данные используются в вашей программе, то вам уже известно, как осуществить их выборку (см.гл.5). Определение атрибута (и сам атрибут) отличают три поля, задающие правила написания текста определения атрибута. Интересующие поля перечислены ниже:

- маркер атрибута или имя;
- строка приглашения к вводу;
- значение по умолчанию;
- флажки атрибута.

Поле маркера атрибута служит для задания атрибуту имени. Это поле фиксировано и не содержит никакой другой информации кроме

идентификации атрибута. Имя повторяется во вставленных блоках - в записях атрибутов. В DXF-файлах в качестве маркера атрибута используется число 2.

Поле строки приглашения к вводу появляется только в определении атрибута и лишь для изменяемых (переменных) атрибутов. Значение изменяемых атрибутов удовлетворяется, например, когда блок, в котором они находятся, вставляется в чертеж. Защищая пользователя от довольно запутанной обработки атрибута, система AutoCAD выводит для пользователя приглашение к вводу, сигнализирующее, что необходимо ввести соответствующие данные. При просмотре DXF-файлов пусть вас не смущают строки, в которых содержится приглашение к вводу, сопровождаемое значением маркера атрибута. Маркеры атрибута должны быть уникальными. (Я не знаю, что произойдет, если это будет не так!) Приглашение к вводу необязательно должно быть уникальным. Однако если приглашение к вводу уникально, это, конечно, помогает пользователю решить, какой надо дать ответ. В DXF-файлах для обозначения строк приглашений к вводу используется маркер 3.

```
0
ATTDEF
8
0
10
1.2345
20
2.3456
40
0.125
1
Статуя Свободы
3
Введите название монумента
2
Монумент
70
0
50
90.0
7
HELWETIC
```

Лист. 6.9. Типичное определение атрибута в DXF-файле

Поле значения по умолчанию содержит значение атрибута. Это как раз те данные, которые извлекаются по команде **ATTTEXT**. Для неизменяемых (постоянных) атрибутов это значение является константой. Для определений неизменяемых атрибутов не существует записи атрибута (**ATTRIB**). Для изменяемых атрибутов это поле определяет значение атрибута по умолчанию. Таким образом, если пользователь в ответ на приглашение к вводу не введет никаких данных, то значением атрибута станет его значение по умолчанию. В DXF-файлах для обозначения значения по умолчанию используется маркер 3. На лист. 6.9 представлено типичное определение атрибута, встречаемое в DXF-файлах. Данный пример мы будем использовать в своей работе над программами.

Атрибуты и структура данных

Запись атрибута следует после операции включения блока в чертеж с помощью команды **INSERT**. В записи **INSERT** содержится флажок, значение которого показывает наличие последующих атрибутов. Это маркер 66. Значение, сопровождающее маркер 66, всегда единица. Если атрибуты, соответствующие записи **INSERT**, отсутствуют, маркер 66 также будет отсутствовать.

Как и запись определения атрибута, запись атрибута подобна объекту текста. В ней содержится вся информация, такая, как местоположение, тип шрифта, высота текста, значения углов поворота. Но, в отличие от записи определения атрибута, в ней представлены только маркер атрибута или имя и значения атрибута. Больше не требуются строка приглашения к вводу и значения по умолчанию. Маркер атрибута является обратной ссылкой на определение атрибута. В действительности, запись о вставке блока идентифицирует явление базового блока, а маркер атрибута - соответствующие определения атрибутов. Инструкция **ATTDEF** присоединяется к базовому блоку. Поскольку тот же маркер атрибута может быть присоединен к нескольким определениям блоков (Сколько раз вы можете произнести "Складской номер?") - это обязывает к тому, чтобы вы идентифицировали блок прежде, чем попытаетесь установить соответствие между атрибутом и его определением. Подобно сказанному ранее наиболее важными в записи атрибута являются: маркер атрибута или имя; значение атрибута.

Маркер атрибута или имя служит для идентификации атрибута. Это поле представляет собой обратную логическую связь на определение атрибута.

Используйте имя блока в записи вставки блока для нахождения блока, а маркер атрибута — для нахождения определения атрибута. Как и в записи **ATTDEF**, в DXF-файле для выделения данного поля

применяется маркер 2. Это необходимо для нахождения значения отдельного атрибута, присвоенного вставленному блоку. Значением атрибута может быть значение, введенное пользователем или прикладной программой, либо значение атрибута по умолчанию. В DXF-файлах значению атрибута предшествует маркер 1.

Теперь, когда названы составные части структуры атрибута в DXF-файле, мы можем рассмотреть то, как их использовать. Следующее обсуждение посвящено применяемому мной методу. Этот метод обеспечивает хорошо структурированное окружение для работы с атрибутами. Это открытый для модификации метод и в основном в большинстве случаев применимый. Метод основывается на том, что вам при многих обстоятельствах необходимо только осуществить выборку маркера атрибута и его значения. В некоторых необычных ситуациях нужно получить дополнительную информацию, такую, как тип данных. На лист. 6.10 показана основная структура данных.

```
struct
{
    char      *att_tag;
    char      *att_value;
}            att_entry;
```

Лист. 6.10. Основная структура данных для атрибута

Поскольку как маркер атрибута, так и его значение представлены в символьной форме в коде ASCII (и имеют переменную длину), они могут быть сохранены в таблице строк. Структура данных содержит только указатели на реальные значения, помещенные в таблице.

Если необходимо иметь информацию о типе данных, то эта информация может быть добавлена к рассмотренной структуре данных. В этой ситуации хорошо подходит применение перечисляемого типа данных - enum или констант, определенных оператором #define. На лист. 6.11 приведена структура, аналогичная первой структуре данных, содержащая поле определения типа данных. Список, сформированный на основе этих структур, представляет отличную таблицу, содержащую всю имеющую отношение к делу информацию об атрибутах.

```
#define REAL      1
#define STRING    2
#define INTEGER   3

struct
{
    char      *att_tag;
    int       att_type: /* REAL, STRING или INTEGER */
    char      *att_value;
}            att_entry;
```

Лист. 6.11. Внутренняя структура данных для атрибута с определениями типа данных

Если бы вы проследили, какие данные сохраняются в этой структуре, то увидели бы, что вся информация об атрибутах представляет собой строки, извлеченные из DXF-файла. Да, но как в эту структуру данных попадут ссылки на действительные и целые числа? Существуют некоторые внутренние атрибуты, предоставляемые системой AutoCAD, которые не появляются в записях атрибутов. Внутренние атрибуты - это не обязательно символьные строки. В табл. 6.4 показаны внутренние атрибуты системы AutoCAD и их реальные типы данных. Зарезервируйте место при построении таблицы атрибутов для сохранения внутренних значений.

Таблица 6.4. Предопределенные типы данных атрибутов

Имя блока	STRING
Точка вставки	REAL
Угол поворота	REAL
Коефф.масштабир. по оси X	REAL
Коефф.масштабир. по оси Y	REAL
Имя слоя	STRING

Я предлагаю рассмотреть конструкцию таблицы, в которой содержатся все идентификаторы и значения всех требуемых для блока атрибутов. Эта таблица представляет собой массив элементов типа `att_entry`. Число элементов в таблице полностью зависит от числа определений атрибутов в блоке и числа внутренних атрибутов, которые вы пожелаете добавить. В целях обсуждения мы будем использовать все внутренние атрибуты, кроме атрибута `BL: LEVEL`. Атрибут `BL: LEVEL` относится к уровню вложенности, с которого осуществляется ссылка на резидентный блок. Этот атрибут применяется в некоторых случаях для организации рекурсии при программировании и динамического изменения размера таблицы. Поскольку здесь я не хочу переходить на этот уровень сложности (хотя сам я реализовал такой механизм), мы остановимся на первом уровне обработки.

В этом месте мы сделаем следующее:

1. Сформируем заготовку определения атрибутов и присоединим ее к определению блока.
2. При обнаружении вставляемого блока скопируем заготовку в рабочую область памяти.
3. Заполним заготовку относящимися к атрибуту значениями, взятыми из инструкции `INSERT` и соответствующих описаний атрибутов.
4. Передадим таблицу вызывающей программе для обработки.

При построении заготовок, обработка секции блоков `BLOCKS` в DXF-файле осуществляется дважды. Первый раз - при завершении обработки описания блока. Вместо того чтобы просто выйти из цикла конечного автомата при обнаружении записи `ENDBLK`, перед выходом из цикла выполняется процедура построения заготовки. Второй раз - в конце секции `BLOCKS`. В цикле, который охватывает

обработку всех блоков, включенных в цепочку определений блоков, создается заготовка для каждого блока в цепочке.

Независимо от последовательности действий, которую вы выберете для построения заготовки, процедура одна и та же: дается ссылка на описание блока, формируется и присоединяется к описанию блока заготовка.

Первое, что делается при создании заготовки таблицы, - это выделяется достаточный для хранения таблицы объем памяти. Подсчитайте число определений атрибутов, связанных с блоком, и прибавьте к нему число внутренних атрибутов, которые желательно использовать. Добавьте к таблице еще один элемент, как признак конца таблицы. На лист. 6.12 представлена функция `count_attdef`, которая возвращает как результат своей работы число определений атрибутов, содержащихся в блоке.

```
int count_attdef(b)
block_node *b;
{
    int count;
    generic *e;

    count = 0;
    e = b->ent;
    while(e)
    {
        if (e->ident == ATTDEF) count++;
        e = e->next;
    }
    return count;
}
```

Лист. 6.12. Подсчет числа определений атрибутов

На лист. 6.13 показано суммирование числа определений атрибутов с числом внутренних атрибутов и завершающего таблицу элемента `NULL`. Функция `calloc` возвращает ссылку соответствующего размера на очищенный блок памяти для заготовки таблицы. Суммирование можно было проделать и при вызове функции `calloc`, но в данном примере это выполняется отдельно.

```
void get_template(b)
block_node *b;
{
    int n;

    n = count_attdef(b) + 7 + 1;
    b->template = (att_entry *)calloc(n, sizeof(att_entry));
}
```

Лист. 6.13. Присоединение шаблона к блоку

На следующем шаге производится заполнение построенной заготовки маркерами или именами внутренних атрибутов. В нашем примере задействованы 7 из них. Инициализацию полей для каждого атрибута лучше выполнять отдельно. Имена внутренних атрибутов отчасти произвольны, поскольку они будут поддерживаться согласованно внутри всей программы. Я буду использовать имена из секции АТТЕХТ так, чтобы вы могли проследить за моими действиями. Для вашего удобства эти поля еще раз приведены в табл. 6.5.

Таблица 6.5. Внутренние атрибуты

BL:NAME	BL:XSCALE
BL:X	BL:YSCALE
BL:Y	BL:LAYER
BL:ORIENT	

Чтобы облегчить чтение исходных текстов программ, я буду использовать во всех последующих листингах определения констант, приведенные ранее на лист. 6.3. На лист. 6.14 представлена программа инициализации для внутренних атрибутов. В этой программе не присваиваются реальные значения атрибутов. Вы можете добавить значения по умолчанию.

```
void set_internal_attributes(b)
block_node *b;
{
    b->template[BL_NAME].att_tag = locate("BL:NAME");
    b->template[BL_NAME].att_type = STRING;
    b->template[BL_NAME].att_value = NULL;
    b->template[BL_X].att_tag = locate("BL:X");
    b->template[BL_X].att_type = REAL;
    b->template[BL_X].att_value = NULL;
    b->template[BL_Y].att_tag = locate("BL:Y");
    b->template[BL_Y].att_type = REAL;
    b->template[BL_Y].att_value = NULL;
    b->template[BL_ORIENT].att_tag = locate("BL:ORIENT");
    b->template[BL_ORIENT].att_type = REAL;
    b->template[BL_ORIENT].att_value = NULL;
    b->template[BL_XSCALE].att_tag = locate("BL:XSCALE");
    b->template[BL_XSCALE].att_type = REAL;
    b->template[BL_XSCALE].att_value = NULL;
    b->template[BL_YSCALE].att_tag = locate("BL:YSCALE");
    b->template[BL_YSCALE].att_type = REAL;
    b->template[BL_YSCALE].att_value = NULL;
    b->template[BL_LAYER].att_tag = locate("BL:LAYER");
    b->template[BL_LAYER].att_type = STRING;
    b->template[BL_LAYER].att_value = NULL;
}
```

Лист. 6.14. Инициализация внутренних атрибутов

Другие элементы таблицы остаются неинициализированными, они предназначены для данных, получаемых из определений атрибутов, содержащихся в цепочке определений блоков. Процедура осуществляет проверку каждого элемента. Если встречается запись **ATTDEF**, программа заносит имя атрибута и значения по умолчанию в следующее свободное место таблицы. Тип атрибута — всегда строка. Если пожелаете, вы можете поэкспериментировать со смешанными типами данных. На лист. 6.15 приведен исходный текст этой процедуры.

```
void add_attdefs(b)
block_node *b;
{
    generic *e;
    int next_att;

    next_att = 7;
    e = b->ent;
    while(e)
    {
        if (e->ident == ATTDEF)
        {
            b->template[next_att].att_tag = locate(e->name);
            b->template[next_att].att_type = STRING;
            b->template[next_att++].att_value = locate(e->value);
        }
        e = e->next;
    }
}
```

Лист. 6.15. Добавление определений атрибутов

На рис. 6.5 показана структура таблицы. Последний элемент таблицы пустой, он обозначает конец таблицы. Таблица подготовлена для работы, которая начнется, когда программа считает секцию объектов **ENTITIES**.

Теория будет реализована на практике, когда программа встретит запись **INSERT**. С обнаружением этой записи начинают поступать ожидаемые атрибуты и реализуются наши приготовления. Но сначала мы должны сделать следующее. Если заготовка для атрибутов уже имеется, мы должны выделить область памяти для вставляемого дубликата блока. Нам необходимо скопировать заготовку в рабочий блок. Это означает, что мы должны найти ссылки блока и получить указатели на заготовку. Во фрагменте программы на лист. 6.16 выполняется выборка указателя на заготовку. Имя блока мы получаем из записи **INSERT**. Зная имя блока, мы можем получить указатель на адресуемый блок-прототип. Блоки-прототипы содержат ссылки на заготовку. Надежно связанные с заготовкой, мы можем легко вклю-

чать значения для внутренних атрибутов. Все эти значения берутся из записи INSERT.

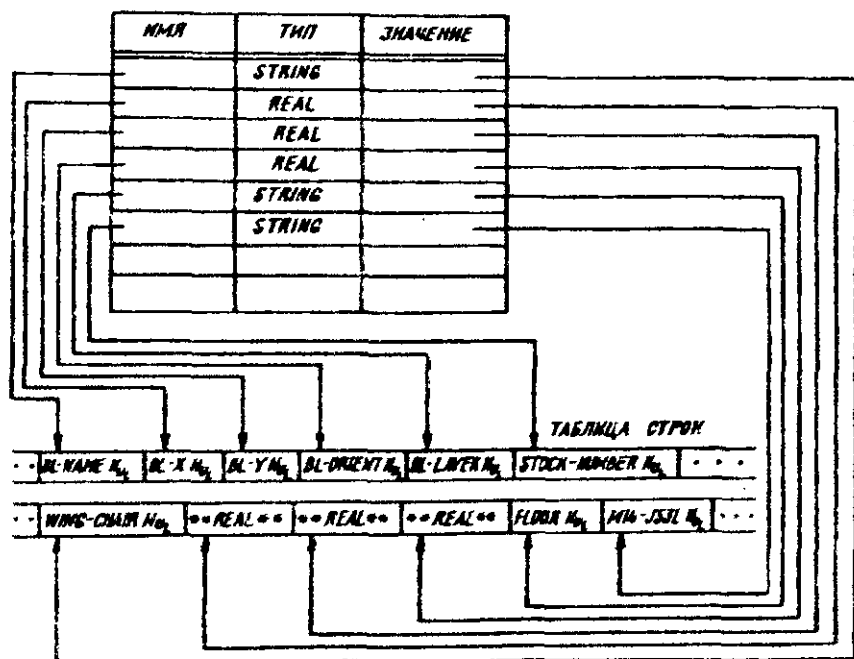


Рис. 6.5. Структура атрибута в DXF-файле

```
b = find_block(e->block_name);
tmpit = b->template;
```

Лист. 6.16. Извлечение ссылки на заготовку

Здесь имеется одно коварное место. Некоторые внутренние атрибуты не являются строками. Они являются действительными или целыми числами. Нет проблем поместить целое 16-битовое число в таблицу, где хранятся 32-битовые указатели. Но поместить 64-битовое число с плавающей точкой в 32-битовое поле — это воистину тяжелая работа.

Вместо вслипываний, растяжения и сжатия я предлагаю вам использовать существующую структуру и применять указатель на целые и действительные числа. Но где в этом случае хранятся значения? Вы не можете хранить значения с помощью локальных пере-

менных, поскольку область локальных переменных исчезнет сразу, как только процедура вернет управление вызвавшей ее программе и освободит стек. Вы не можете положиться и на глобальные переменные, поскольку при появлении новой записи INSERT старые значения переменных будут перезаписаны новыми. А как насчет того, чтобы поместить эти числовые значения в таблицу строк? Таблица является статической областью, которая после изменения значения указателя `next_word` не перезаписывается. Я не хотел бы рекомендовать выполнение перемешивания для чисел с плавающей точкой; это может далеко завести. Следует посоветовать, что значения, помещаемые в следующее доступное место таблицы, и указатели должны соответствовать друг другу по размеру поля памяти, занимаемого переменной. Целые и действительные числа заносятся в таблицу с помощью функций `put_int` и `put_double`. На лист. 6.17 и 6.18 показаны данные функции. Если вы полагаете осуществлять раздельную компиляцию подпрограмм, эти функции связаны их расположением и порядком следования. Имея адрес заготовки, содержащийся в переменной `tmp1t`, и средства для занесения в заготовку данных любого рода, мы можем продолжить.

```
char *put_int(n)
int    n;
{
    char    *p;

    p = next_word;
    next_word += sizeof(int);
    *(int *)p = n;
    return p;
}
```

Лист. 6.17. Функция `put_int` для целочисленных значений

```
char *put_double(n)
double    n;
{
    char    *p;

    p = next_word;
    next_word += sizeof(double);
    *(double *)p = n;
    return p;
}
```

Лист. 6.18. Функция `put_double` для действительных чисел

Следующая операция — это получение рабочей копии заготовки, полностью совпадающей по размеру и содержимому с оригиналом.

Сначала мы определим размер. Существуют несколько способов получения размера заготовки блока. Один из них заключается в том, чтобы сохранять размер блока прототипа, при первом создании заготовки. Количество записей определений атрибутов, найденное с помощью маленькой программы подсчета, было бы достаточной для вас информацией, которую нужно сохранять для последующего вычисления размера.

Другой способ определения размера заготовки несколько неэкономичен, если вы ищете компактное решение. Некоторые компиляторы языка Си (в частности, Microsoft C) включают функцию `_msize`. По заданному указателю на блок функция `_msize` возвращает число байт в блоке. Поскольку такая функция есть не во всех библиотеках компиляторов и не является стандартной для языка Си (как определено в ANSI-стандарте на язык Си), лучше пропустить эти примеры программ. Третий способ определения размера мог заключаться в выполнении повторного подсчета определений атрибутов в блоке. Такая программа уже существует и ее легко использовать.

Еще одним способом мог быть подсчет элементов непосредственно в самой заготовке. В момент подсчета имена атрибутов уже полностью занесены в заготовку. Последнее имя — фиктивная ссылка `NULL`, которая играет роль ограничителя списка. Для того чтобы определить общий размер, надо подсчитать число нефиктивных указателей и прибавить единицу для учета последнего нулевого указателя. На лист. 6.19 показано, что я имел в виду.

```
count = 0;
s = tmp1t;
while(s->att_tag)
{
    count++;
    s++;
}
count++;
```

Лист. 6.19. Подсчет числа элементов в заготовке

Зная число элементов в таблице заготовок (или число байтов), можно получать блок памяти соответствующей длины. Как это делается, показано на лист. 6.20. Теперь мы скопируем заготовку в рабочую область. Это даст нам имена атрибутов, типы данных, а также значения неизменяемых атрибутов и значения, используемые по умолчанию, которые уже будут находиться в рабочей копии.

```
work = (att_entry *) calloc(count, sizeof(att_entry));
```

Лист. 6.20. Получение рабочей заготовки

Вы можете написать более эффективную программу, чем функция языка Си `memcpy` (копирование содержимого области памяти), но зачем нужны две такие программы? Эта функция выполняет копирование заданного числа байтов из одной области памяти в другую. На лист. 6.21 показана пересылка информации из заготовки в рабочую копию. В функции `memcpy` требуется указание числа байтов, которое определено в ходе предыдущих вычислений. Если вы тяготеете к неявным приемам, то строка программы на лист. 6.22 придется вам по душе.

```
memcpy(work,tmpl,t,count*sizeof(att_entry));
```

Лист. 6.21. Копирование заготовки в рабочую область

```
work = memcpy(malloc(_msize(tmpl)),tmpl,_msize(tmpl));
```

Лист. 6.22. Выполнение тех же операций неявным образом

Однако мы закончили операции, имея два указателя, содержащие ссылки на идентичные блоки памяти. Заготовка выполнила свое предназначение, и теперь мы можем о ней забыть. Но не отбрасывайте ее совсем, так как она понадобится при каждом новой вставке блока в чертеж. На рис. 6.6 отображено то, что мы сделали.

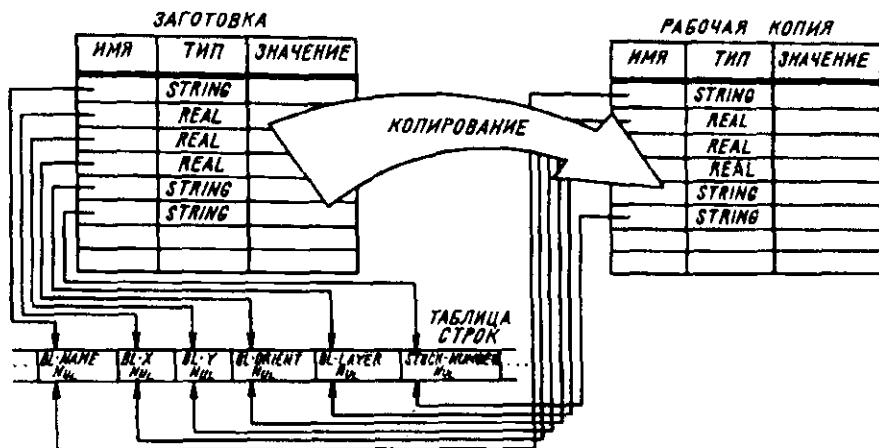


Рис. 6.6. Копирование заготовки в рабочую структуру данных

Внутренние атрибуты мы получили из записи `INSERT`. Они подготовлены для нас и помещаются в структуру данных, образованной после синтаксического разбора записи `INSERT`. Все, что мы должны сделать, - переслать эти значения в рабочий блок памяти. На лист. 6.23 приведен фрагмент программы, в котором осуществляется пересылка. В процессе последующей обработки мы будем заполнять блок значениями реальных атрибутов.

```

work[BL_NAME].att_value = locate(e->block_name);
work[BL_X].att_value = put_double(e->x);
work[BL_Y].att_value = put_double(e->y);
work[BL_ORIENT].att_value = put_double(e->orient);
work[BL_XSCALE].att_value = put_double(e->xscale);
work[BL_YSCALE].att_value = put_double(e->yscale);
work[BL_LAYER].att_value = locate(e->lyr);

```

Лист. 6.23. Выборка значений внутренних атрибутов

Если в записи **INSERT** имеются атрибуты, следующие после этой инструкции, то вы обнаружите в DXF-файле маркер 66 перед атрибутом и инструкцию **SEQEND** в конце данных, относящихся к атрибуту. Директива **SEQEND** является для вас сигналом, что следует остановить поиск атрибутов и передать указатель на рабочий блок атрибутов программе обработки. Единственными инструкциями, которые следуют непосредственно после инструкции **INSERT** в записи, содержащей атрибуты, являются инструкции **ATTRIB** и **SEQEND**.

В обсуждении, посвященном синтаксическому разбору графических объектов в гл. 5, может быть немного неясно, что же делать с атрибутами. Это потому, что я не знаю, что вы хотите с ними делать. И остаюсь в неведении и теперь. Тем не менее я могу детализировать программу синтаксического разбора для атрибутов и вставленных блоков. Текст программы на лист. 6.24 представляет собой часть программы синтаксического разбора, приведенной ранее на лист. 5.56. На данном листинге показаны только вызовы программ обработки вставки блоков и атрибутов.

```

case ATTDEF      :
case ATTRIB      :
case TEXT        : text_processor(en);
                  break;
case SHAPE       : shape_processor(en);
                  break;
case INSERT      : insert_processor(en);
                  break;
case DIMENSION   : dimen_processor(en);
                  break;
case SEQEND      : if (en) pline_processor(en);
                  break;

```

Лист. 6.24. Фрагмент программы синтаксического разбора (повторение)

На лист. 6.25 показан заменяющий предыдущий фрагмент текст программы с вызовами программ обработки атрибутов, описанных в этой главе. Программа обработки вставки возвращает указатель на рабочую копию заготовки. Программа обработки атрибутов использует этот указатель на таблицу как входные данные. Внутри этой программы производится заполнение таблицы значениями найденных атрибутов. Программа обработки конца последовательности данных переключает управление программы на обработку заполненной таблицы.

```

case ATTRIB      : attribute_processor(en,templt);
                  break;
case INSERT      : templt = insert_processor(en);
                  break;
case SEQEND      : if (en) pline_processor(en);
                  if (templt)
                  {
                      process_template(templt);
                      free(templt);
                      templt = NULL;
                  }
                  break;

```

Лист. 6.25. Модифицированный фрагмент программы синтаксического разбора

Давайте оценим, насколько проста программа обработки атрибутов. Поскольку данная программа не может переключаться до тех пор, пока она не найдет полностью все данные, относящиеся к инструкции **ATTRIB**, мы знаем, что мы получаем полную структуру этой записи. В этой структуре имеются две интересующие нас вещи — маркер атрибута (имя) и значение атрибута. На все другие поля можно не обращать внимание, если наша цель была найти только названные выше данные. Зная имя атрибута, мы можем найти соответствующий ему элемент в таблице. Имея значение атрибута, мы можем присвоить его соответствующему элементу таблицы. Это просто. На лист. 6.26 показана программа обработки атрибутов. Так как имя атрибута в записи **ATTRIB** и внутренняя таблица атрибутов получили значения для своих указателей от функции нахождения атрибута (*locate*), эти значения должны совпадать. Мы могли бы найти соответствие, сравнивая значения указателей. Однако как раз в данном случае вы не используете функцию *locate*; на листинге показано выполнение сравнения с помощью функции *strcmp*. Когда будет обнаружена инструкция **SEQEND**, вы будете иметь завершенную таблицу.

Одно из преимуществ выполняемой таким образом обработки состоит в том, что вам не надо беспокоиться о значениях по умолчанию и неизменяемых (постоянных) значениях. Неизменяемые значения никогда не появляются в инструкции **ATTRIB**, таким образом их оригинальное значение переносится в рабочую область при копировании. Программа обработки атрибутов не трогает эти значения. То же самое происходит и со значением по умолчанию, которое отсутствует в инструкции **ATTRIB** или не заменяется значением, вводимым в ответ на приглашение к вводу. Обнаружив признак, означающий, что данные не были введены — значение указателя **NULL**, вы просто возвращаете управление из программы обработки, не заменяя значение в рабочей области.

```

void attribute_processor(e,w)
att_entity      *e;
att_entry       *w;
{
    att_entry     *p;

```

```

/* нет записей -- оставить значения по умолчанию */
if (e->att_value == NULL) return;

p = w;
while (p->att_tag)
{
    if (strcmp(e->att_name,p->att_tag) == 0)
    {
        p->att_value = locate(e->att_value);
        return;
    }
    p++;
}
}

```

Лист. 6.26. Программа обработки атрибутов

Теперь, когда вы имеете весь материал, что вы собираетесь с ним делать? Я не имею никаких соображений, но я чувствую, что должен что-то сделать. Программа, представленная на лист.6.27, распечатывает имена и значения найденных атрибутов.

```

void print_attributes(w)
att_entry      *w;
{
    att_entry    *p;

    p = w;
    while (p->att_tag)
    {
        printf("%s ",p->att_tag);
        switch(p->att_type)
        {
            case INTEGER : printf("%d\n",*(int *)p->att_value);
                            break;
            case REAL     : printf("%f\n",*(double *)p->att_value);
                            break;
            case STRING   : printf("%s\n",p->att_value);
                            break;
        }
        p++;
    }
}

```

Лист. 6.27. Программа печати значений атрибутов

Это привело нас к концу дискуссии об атрибутах. Мы увидели, какую роль играют атрибуты в чертежах, выпущенных с помощью системы AutoCAD, для задания дополнительной неграфической информации. Мы также рассмотрели два способа извлечения такой информации, с тем чтобы использовать ее в прикладных программах.

Глава 7

УПРАВЛЕНИЕ СИСТЕМОЙ AUTOCAD С ПОМОЩЬЮ ПАКЕТНЫХ ФАЙЛОВ

Настало время рассмотреть, как наши программы могут порождать команды для управления системой AutoCAD. Конечно, есть смысл использовать чертежи, подготовленные в системе AutoCAD, как источник данных для программ анализа, синтеза или создания баз данных. Имея результаты работы этих программ, может быть разумным позволить системе AutoCAD готовить графическое представление результатов анализа и синтеза. В моей работе с системой AutoCAD программы выполняли расчет, каким должен быть рабочий чертеж, и затем сообщали системе AutoCAD, как нарисовать законченную структуру.

В этой и следующей главе обсуждается подготовка графических данных для их использования в чертежах, выпускаемых с помощью системы AutoCAD. В данной главе рассматривается применение пакетных файлов. В следующей главе затрагиваются вопросы создания файлов форматов DXF и DXB для ввода графической информации в систему AutoCAD.

Пакетные файлы. Введение

Пакетный файл системы AutoCAD представляет собой файл с символической информацией в коде ASCII. В этом файле содержатся команды системы AutoCAD, которые имеют точно такой же формат, как и команды, вводимые с клавиатуры. Когда система AutoCAD считывает пакетный файл, она приостанавливает ввод с клавиатуры и графического планшета до тех пор, пока не закончится или не будет прервано (единственное исключение при приостановке ввода с клавиатуры) выполнение команд из пакетного файла.

Вы можете прервать считывание пакетного файла, нажав на клавишу BACKSPACE. Продолжить считывание файла вы можете, введя с клавиатуры команду RESUME (возобновить). В промежутке между прерыванием и возобновлением выполнения ввод других графических команд и команд установки параметров рабочего окружения (среды) не влияет на состояние пакетного файла. При работе с пакетным файлом система AutoCAD считывает из него команды и выполняет их так же, как если бы они были введены с клавиатуры.

Пакетный файл может показаться не очень большим достижением, тем не менее данная особенность позволит разработчикам прикладных программ многое реализовать в системе AutoCAD легче, чем в других системах автоматизированного проектирования для персональных компьютеров.

Выполнение пакетного файла

Вы можете запустить пакетный файл на выполнение несколькими способами. Наиболее простой способ — начать выполнение пакетного файла одновременно с запуском системы AutoCAD. Это делается при вводе команды в ответ на приглашение к вводу операционной системы. Обычно при вызове системы AutoCAD в командной строке вы должны указать имя чертежа, который вы хотите отредактировать, как это показано на лист. 7.1.

```
ACAD my_dwg
```

Лист. 7.1. Команда запуска системы AutoCAD

Система AutoCAD стартует, и после некоторой загрузки и скрежета перед вами появляется основное меню системы. Для редактирования чертежа вы должны выбрать первый пункт меню (новый чертеж) или второй (редактирование старого чертежа). После чего система AutoCAD запрашивает имя чертежа. Хотя вы уже сообщили системе имя чертежа, она по-прежнему хочет его знать. Заданное вами в командной строке имя чертежа будет именем, используемым по умолчанию. При нажатии клавиши ENTER (возврат каретки) система будет использовать имя чертежа, заданное по умолчанию.

Применение пакетных файлов позволяет автоматизировать этот процесс. Конечно, пакетные файлы могут быть более сложными, чем рассматриваемый, но для запуска пакетного файла на выполнение это самое подходящее место.

Для начала изменим командную строку, которая вводится в ответ на приглашение к вводу операционной системы MS-DOS. После имени чертежа добавим имя пакетного файла. Новая командная строка показана на лист. 7.2. При запуске системы AutoCAD (как показано в примере) система AutoCAD будет искать файл с именем old_dwg.scr. Если пакетный файл с таким именем не будет найден, система завершит свою работу.

```
ACAD my_dwg old_dwg
```

Лист. 7.2. Указание имени пакетного файла в командной строке

Что же содержится в файле old_dwg.scr? Поскольку этот файл является пакетным файлом, в нем содержится то, что вы обычно могли бы ввести с клавиатуры. Для редактирования уже существующего чертежа вы сначала должны ввести число 2. Затем вы просто нажимаете клавишу ENTER, что служит указанием для использования имени файла по умолчанию. На лист. 7.3 показано содержимое файла old_dwg.scr. В примере имеется еще одна дополнительная строка. Так как трудно показать пустую строку (нажатие на

ENTER) как последнюю строку листинга, я добавил команду REDRAW. Эта команда не является обязательной и введена лишь для выделения пустой строки.

2

```
redraw
```

Лист. 7.3. Пакетный файл old_dwg

Если есть пакетный файл для редактирования старого чертежа, то также может быть пакетный файл и для создания нового чертежа. Единственным различием между этими двумя файлами является выбор другого пункта в основном меню системы AutoCAD. Для нового чертежа выбирается первый пункт меню. На лист. 7.4 показано содержимое пакетного файла new_dwg.scr.

1

```
redraw
```

Лист. 7.4. Пакетный файл new_dwg

Я буду первым, кто согласится, что усилия по дополнительному вводу символов в командной строке едва ли соизмеримы с нажатием клавиши с цифрой 1 или 2 и нескольких нажатий клавиши ENTER. Тем не менее мы начинаем с создания пакетного файла, который избавляет от необходимости набирать лишние символы на клавиатуре или делать выбор в меню.

Теперь о том, что мы собираемся сделать. Операционная система MS-DOS может проверить наличие какого-либо отдельного файла (в данном случае файла чертежа). В зависимости от условия, существует ли этот файл или нет, MS-DOS может выбрать различную последовательность выполнения команд в пакетном файле. Если файл чертежа уже существует, то при запуске системы AutoCAD может быть выбран пакетный файл old_dwg. Если файл чертежа отсутствует, то при запуске системы AutoCAD может использоваться пакетный файл new_dwg.

На лист. 7.5 приведен пакетный файл MS-DOS для запуска системы AutoCAD. Вы можете дать этому файлу любое имя. Единственное, что надо избегать, - это не следует давать файлу имя ACAD. Использование имени ACAD в качестве имени пакетного файла для MS-DOS может создать конфликт с другими пакетными файлами MS-DOS. В других пакетных файлах имя ACAD может использоваться для запуска программы ACAD (система AutoCAD). Вместо того чтобы непосредственно стартовать систему AutoCAD и возобновить выполнение пакетного файла MS-DOS, после завершения сеанса работы с этой системой первый пакетный файл мог бы вызвать на

выполнение пакетный файл ACAD. В этом случае при завершении работы системы AutoCAD пакетный файл ACAD закончит последовательность выполнения команд и может быть потеряна последовательность обработки в первом пакетном файле.

```
echo off
if exist %1.dwg goto old
acad %1 new_dwg
goto join
:old
acad %1 old_dwg
:join
cls
echo on
```

Лист. 7.5. Пакетный файл MS-DOS для запуска системы AutoCAD

Применяемое в пакетном файле обозначение %1 служит для MS-DOS сигналом, что вместо него следует подставить первый параметр, взятый из активизированной командной строки. Предположим, что мы назвали пакетный файл CAD.BAT. На лист. 7.6 показано, как запускается система AutoCAD с помощью пакетного файла.

ACAD my_dwg

Лист. 7.6. Использование пакетного файла для запуска системы AutoCAD

Программа обработки пакетных файлов операционной системы MS-DOS произведет подстановку имени, которое вы задали при вызове пакетного файла, заменяя на него символ %1 в пакетном файле. В пакетном файле запрашивается проверка на наличие файла чертежа. При запросе используется заданное вами имя файла чертежа. Если файл существует, то графический редактор системы AutoCAD запускается с помощью пакетного файла old_dwg. Если файл чертежа отсутствует, то система AutoCAD создает новый чертеж, присваивая ему введенное вами имя.

SCRIPT

Система AutoCAD ответит следующим приглашением к вводу:

Script file:

Имеется другой способ запустить пакетный файл системы AutoCAD. Вы можете сделать это во время сеанса работы с системой AutoCAD. Запуск на выполнение пакетного файла осуществляется с помощью команды SCRIPT (ПАКЕТ). В режиме ввода команд системы AutoCAD, введите:

Вы вводите имя пакетного файла. Нет необходимости указывать расширение имени файла .scr. Если пакетный файл вызывается из системы AutoCAD, то в нем должны находиться только команды редактирования системы AutoCAD. Наличие в начале этого файла записи для выбора пункта меню явится причиной ошибки работы редактора.

Если во время считывания пакетного файла возникнет ошибка, система AutoCAD остановит выполнение пакетного файла и перейдет в обычный режим ручного ввода. Если ошибка незначительна, такая, как опечатка или пропуск данных, то, воспользовавшись командой RESUME, можно продолжить выполнение пакетного файла.

Ошибки, возникающие в середине таких команд, как LINE или LAYER, останавливают выполнение пакетного файла на середине этих команд (и других). Использование в этом месте команды RESUME даст системе AutoCAD указание считать координаты для команды LINE или аргумент для команды LAYER. Не получив правильные данные или команды, система AutoCAD снова остановит выполнение пакетного файла. Нажатие клавиши ENTER заставляет систему AutoCAD повторять выполнение последней команды - команды RESUME. Ввод из пакетного файла выполняется еще раз. Если и в этот раз постигла неудача, продолжайте нажимать клавишу ENTER, чтобы преодолеть область пакетного файла, в которой возникли проблемы.

Генерация пакетных файлов

Создание небольшого пакетного файла для запуска системы AutoCAD — это прекрасное упражнение, позволяющее понимать основные идеи применения пакетных файлов. Однако реальное преимущество пакетных файлов проявляется при разработке пакетного файла, в котором определена последовательность действий системы AutoCAD для построения чертежа с самого начала.

Программы, выполняющие генерацию пакетных файлов для построения сложных графических элементов, становятся очень полезным инструментом профессионального конструктора. Вместо ввода большого объема графических данных вручную или с помощью меню программа поможет автоматизировать некоторые участки цикла конструирования.

Сейчас вы не можете представить себе программу, выполняющую творческую работу человека, но программа вполне может выполнять механическую и рутинную работу, что позволит конструктору более эффективно использовать свой творческий потенциал. Я обычно разделяю процесс генерации пакетных файлов на две части.

В первой части генерации пакетных файлов производится установка параметров графического окружения системы AutoCAD. Обычно это статическая часть программы. Настроив один раз рабочие па-

параметры так, как это устраивает вас или пользователя, вы можете копировать имеющийся фрагмент текста программы в другие программы. Настройка параметров графического окружения включает в себя следующее:

- создание слоев;
- установку цвета для слоя; типа линий для слоя;
- создание стилей начертания текста;
- установку границ чертежа; шага сетки и разрешающей способности;
- переключение режима (ORTHO, SNAP и COORDS);
- вычерчивание рамки чертежа;
- считывание меню;
- увеличение изображения, чтобы показать область чертежа.

В примере на лист. 7.7 все перечисленные действия не выполняются. Тем не менее показанный фрагмент текста программы участвует в процессе генерации пакетных файлов и служит для инициализации параметров рабочей среды.

```
fprintf(SCR_file, "LIMITS 0,0 12,12\n");
fprintf(SCR_file, "SNAP 0.125\n");
fprintf(SCR_file, "GRID 0.250\n");
fprintf(SCR_file, "ORTHO ON\n");
fprintf(SCR_file, "LAYER\n");
fprintf(SCR_file, "NEW FLOOR,WALL,SURFACE\n");
fprintf(SCR_file, "COLOR RED FLOOR\n");
fprintf(SCR_file, "COLOR BLUE WALL\n");
fprintf(SCR_file, "COLOR GREEN SURFACE\n");
fprintf(SCR_file, "ON *\n");
fprintf(SCR_file, "SET FLOOR\n\n");
fprintf(SCR_file, "ZOOM ALL\n");
```

Лист. 7.7. Инициализация пакетного файла

При генерации пакетных файлов проявляется одно несовершенство. Система AutoCAD воспринимает символы пробела и перевода строки как разделители для команд и их параметров. Возможно образовать пакетный файл, содержащий единственную строку, в которой в качестве разделителей будут использоваться только пробелы.

Более реально — использовать пробелы как разделители команд, а список параметров отделять в конце списка символом перевода строки. Это представляет более естественную форму вывода.

Однако в некоторых командах, в частности в командах POLYLINE и LINE, список параметров или координат может выйти за пределы некоторой установленной длины строки. В этих случаях может оказаться весьма трудным скорректировать пакетный файл (что иногда случается) с помощью простых редакторов текста. Для команд с переменным числом параметров хорошим стилем является разделение параметров или пар координат символами перевода стро-

ки, что позволяет вам легко просмотреть содержимое пакетного файла.

Очень важно при генерации пакетных файлов запомнить форматы различных команд. Несколько команд системы AutoCAD оканчиваются несколькими дополнительными нажатиями клавиши ENTER. Примером может служить команда LAYER. На лист. 7.8 показан пакетный файл для создания и установки параметров слоя FLOOR (этаж), на котором используется цвет BLUE (голубой). Программа обработки команды LAYER системы AutoCAD выполняет ввод данных для этой команды до тех пор, пока не встретится пустая строка (перевод строки или дополнительный пробел). Обратите внимание на наличие в исходном тексте программы дополнительного символа перевода строки (лист. 7.7), который используется для преодоления возможных неприятностей. Чтобы показать наличие пустой строки, команда REDRAW снова вставлена в текст пакетного файла. Вторая часть пакетного файла связана непосредственно с вычерчиванием чертежа. Теперь мы введем в игру графические команды.

```
LAYER NEW FLOOR  
COLOR BLUE FLOOR  
SET FLOOR
```

```
REDRAW
```

Лист. 7.8. Создание слоя FLOOR

Национальные варианты системы AutoCAD

Одна из целей учредителей фирмы Autodesk — сделать систему AutoCAD международным продуктом. Учредители из нескольких стран быстро подготовили материалы для этого. В настоящее время существуют национальные варианты системы AutoCAD на английском, французском, немецком, испанском и итальянском языках. Появились сообщения о создании китайского и русского варианта системы. В этих различных национальных вариантах системы AutoCAD команды и приглашения к вводу команд появляются на экране на родном языке. В других национальных вариантах системы, таких, как, например, шведский и японский, приглашения к вводу появляются на родном языке, однако для команд используется английский язык.

Следовательно, возможно сделать вашу программу пригодной для работы с национальными вариантами системы, имея возможность настройки программы на различные языки, и в этом параграфе рассказывается, как это можно реализовать.

Различные национальные варианты системы AutoCAD отличаются только написанием команд и сообщений системы. Структура этих команд и, конечно, их число одни и те же для всех национальных

вариантов системы. Единственное, что мы должны сделать, - это изменить написание каждой команды.

Для этого есть основа. На лист. 7.9 показан небольшой фрагмент программы для записи команды LINE системы AutoCAD. Функция записывает слово LINE и следующие за ним две пары координат, определяющих отрезок, в пакетный файл. Обратите внимание, что в функции fprintf строка спецификации формата оканчивается двумя символами перевода строки. Второй символ перевода строки завершает команду LINE. В этом листинге нет многого, но он прекрасно иллюстрирует принципы организации вывода национальных текстов.

```
output_line(scr,x1,y1,x2,y2)
FILE      *scr;
double    x1;
double    y1;
double    x2;
double    y2;
{
    fprintf(scr,"LINE %f.%f %f.%f\n\n",x1,y1,x2,y2);
}
```

Лист. 7.9. Пример формирования команды LINE в пакетном файле

Основное, из-за чего команды в пакетном файле пишутся на английском языке, - это трудность кодирования слова LINE в строке формата вывода. Но нет необходимости делать это именно так. Это слово может быть просто аргументом для функции вывода. Давайте попробуем изменить программу. На лист. 7.10 слово LINE вынесено из строки спецификации формата функции fprintf.

```
output_line(scr,x1,y1,x2,y2)
FILE      *scr;
double    x1;
double    y1;
double    x2;
double    y2;
{
    fprintf(scr,"%s %f.%f %f.%f\n\n","LINE",x1,y1,x2,y2);
}
```

Лист. 7.10. Второй пример формирования команды LINE

Еще одно изменение исходного текста и английское слово LINE будут полностью вынесены из программы. На лист. 7.11 показана замена слова LINE параметрической константой с тем же именем, определенной с помощью оператора #define. Во время компиляции параметрическая константа LINE в функции fprintf заменяется на строку LINE.

```
#define LINE          "LINE"

output_line(scr,x1,y1,x2,y2)
FILE          *scr;
double        x1;
double        y1;
double        x2;
double        y2;
{
    fprintf(scr,"%s %f,%f %f,%f\n\n",LINE,x1,y1,x2,y2);
}
```

Лист. 7.11. Простой способ формирования команды **LINE**

Теперь довольно просто заменить английское слово **LINE** на французское слово **LIGNE**. После повторной компиляции выводимая команда станет уже французской, а не английской. На лист. 7.12 показан вариант формирования команды **LINE** на французском языке.

```
#define LINE          "LIGNE"

output_line(scr,x1,y1,x2,y2)
FILE          *scr;
double        x1;
double        y1;
double        x2;
double        y2;
{
    fprintf(scr,"%s %f,%f %f,%f\n\n",LINE,x1,y1,x2,y2);
}
```

Лист. 7.12. Французский вариант команды **LINE**

Итак, хотите ли вы раз и навсегда оставить все это в таком виде? Добавив некоторые переключатели для препроцессора, мы можем изменить национальный язык "щелчком" переключателя. Как показано на лист. 7.13, мы можем сгенерировать английский вариант пакетного файла, если перед компиляцией программы мы определим значение **ENGLISH**. Также, определив перед компиляцией значение **FRENCH**, мы получим вывод для французского варианта системы AutoCAD. Большинство компиляторов позволяют определять значения на уровне командной строки **MS-DOS**. Например, в компиляторе Microsoft C для установки определений в командной строке используется ключ **-D**.

При компиляции для английского варианта могла быть введена команда:

```
cl -c -DENGLISH line_out.c
```

А для компиляции английского варианта — команда:

```
cl -c -DFRENCH line_out.c
```

```
#ifdef ENGLISH
#define LINE      "LINE"
#endif
#ifdef FRENCH
#define LINE      "LIGNE"
#endif

output_line(scr,x1,y1,x2,y2)
FILE      *scr;
double     x1;
double     y1;
double     x2;
double     y2;
{
    fprintf(scr,"%s %f,%f %f,%f\n\n",LINE,x1,y1,x2,y2);
}
```

Лист. 7.13. Формирование команды LINE на двух языках

Объектный код этих двух модулей, полученных при разных вариантах компиляции, должен храниться отдельно (вы знаете, как хорошо разграничены англичане и французы), каждый в своем каталоге. После компоновки с другими модулями, вызывающими программу генератор пакетных файлов, вы получаете соответствующий вариант программы для того языка, который вы хотите.

В приложении А дан листинг файла определений для создания пакетных файлов системы AutoCAD на различных языках. В одной из моих систем файл определений имеет название lang.h. Я храню его в том же каталоге (\include), где находятся все обычные файлы определений, такие, как stdio.h, stdlib.h и sturpe.h. Для того чтобы этот файл был считан, в программе должна присутствовать следующая строка:

```
#include <lang.h>
```

Для установки определений ENGLISH, FRENCH, ITALIAN, SPANISH и GERMAN я имею специальные пакетные файлы и файлы для утилиты MAKE, в которых в командной строке вызова компилятора используется ключ -D. Написание команд на разных язы-

как было взято из официальных справочников по системе AutoCAD для различных национальных вариантов системы.

Создание пакетных файлов с помощью редактора текстов

Основное назначение данной главы - продемонстрировать, как создаются пакетные файлы для управления системой AutoCAD. Внимание заостряется на применении программ для генерации этих файлов. Однако, используя редакторы текста и следуя рекомендациям данного обсуждения, вы можете создавать пакетные файлы сами вручную. Рассмотрим основные положения. Написанные вручную пакетные файлы имеют свои преимущества. Я применяю такой способ создания пакетных файлов для формирования библиотек символов.

С моей точки зрения, пакетные файлы обладают двумя преимуществами. Первое преимущество не особенно важно для меня, но для заказчиков оно является основным. Я могу предложить исходный текст для библиотек. Если кто-либо захочет понять, как я создаю символы, или захочет в корне изменить какой-нибудь символ, то он будет иметь перед собой стандартный образец, следуя которому он сможет провести свои изменения.

Основным требованием для создаваемых вручную символов является получаемая между ними согласованность. При построении нескольких библиотечных элементов часто между ними имеется некоторое сходство. Я использую такое сходство как преимущество. Конечно, обычно есть небольшие графические отличия, различающие разные символы между собой. Например, таким отличием может быть длина основной конструкции, различное число выносных позиционных меток. В любом случае я разрабатываю первый символ как описание для пакетного файла. Я могу задать минимальное расстояние, на котором пройдет линия. Не требуется подстраиваться под разрешающую способность и шаг сетки. Все получается точно. Я позволяю системе AutoCAD начерчивать символ, описанный в пакетном файле. Все ли нормально на чертеже? Хорошо ли он выглядит? Находится ли в правильном месте текст? Выдержаны ли все размеры?

Затем, воспользовавшись редактором текста, я вношу требуемые коррективы в пакетный файл. Я повторяю цикл - система AutoCAD, редактор текста - до тех пор, пока не буду удовлетворен тем, как выглядит символ.

Если символ содержит атрибуты, я вставляю этот символ в пустой чертеж, чтобы проконтролировать приглашения к вводу и расположение значений атрибутов. Если вы когда-нибудь в системе AutoCAD пытались модифицировать атрибуты, то вы знаете, что это может быть бесполезным. Система AutoCAD не отслеживает запрошенные атрибуты. Если ваши приглашения к вводу для атрибутов должны появляться в некотором порядке, модификация блока с ат-

рибутами может разрушить порядок. Порядок всегда сохраняется, когда вы управляете структурой блока сами. Кстати, при создании атрибутов с помощью пакетного файла используется принцип LIFO (первым вошел, первым обслужен).

Это может служить основной рекомендацией данного способа даже при создании одного символа, но в начале обсуждения было сказано, что я хотел иметь несколько символов. В этом месте и проявляются преимущества. Имея описание одного символа в пакетном файле, очень просто (будучи довольно ленивым, но не чрезмерно, я ищу простые и легкие способы выполнения повторяющейся работы) размножить этот символ любое число раз. Для этого используйте в вашем редакторе текста такие команды, как копирование (COPY) или команды вырезки и вставки текста (CUT и PASTE). При наличии рабочей заготовки вы делаете маленькие изменения, которые позволят вам создать различные элементы. Напомним, чтобы получить максимальное преимущество, символы должны быть схожи между собой. Возможность получить атрибуты в правильном порядке, даже для непохожих символов, может быть достаточным оправданием применения рассматриваемого способа.

Перед тем как приступить к программированию, я хотел бы показать другую сторону дела. Как правило, имеется одна программа, которая генерирует пакетные файлы. Иногда есть несколько таких программ. Должно существовать некоторое соглашение по выбору имени для пакетного файла. Это имя файла должно быть известным, так чтобы его можно было передать системе AutoCAD для ввода. На лист. 7.14 показано содержимое пакетного файла MS-DOS, в котором используется фиксированное имя для пакетного файла системы AutoCAD.

```
my_prog
acad %1 a_script
```

Лист. 7.14. Фиксированное имя пакетного файла

Чтобы программа генерации пакетного файла могла открыть требуемый файл, в ней следует придерживаться данного соглашения. На лист. 7.15 показана последовательность операторов для открытия файла.

```
FILE      *fo:

fo = fopen("a_script.scr","w");
```

Лист. 7.15. Открытие пакетного файла с фиксированным именем

В пакетном файле MS-DOS, с которым мы уже имели дело, вместо имени файла использован символ %1. При запуске на выполне-

ние пакетного файла MS-DOS программа обработки пакетных файлов производит подстановку заданного пользователем имени на место символа %1. Кроме того, программа подставляет и другие заданные пользователем имена, заменяя ими другие %-символы. Второе имя помещается на место символа %2, а третье имя - на место символа %3. При запуске на выполнение показанного в примере выше пакетного файла MS-DOS вы могли бы добавить в командную строку имя файла чертежа, чтобы операционная система могла заменить символ %1 на указанное имя и активизировать систему AutoCAD для этого чертежа.

Еще один способ ввода имени файла состоит в том, чтобы имя пакетного файла системы AutoCAD совпадало с именем чертежа. Имя файла также может использоваться как параметр, передаваемый программе генерации пакетных файлов. На лист. 7.16 показан новый пакетный файл MS-DOS. На лист. 7.17 показано новая последовательность действий для открытия файла в программе. Применяя подобные приемы в пакетных файлах MS-DOS и в ваших программах, вы можете связать между собой имена чертежей и пакетных файлов.

```
my_prog %1
acad %1 %1
```

Лист. 7.16. Применение одного имени для чертежа и пакетного файла

```
int      debug;
int      quiet;
FILE     *fo;

main(argc,argv)
int      argc;
char     *argv[];
{
char     basename[80];
char     filename[80];
int      i;

if (argc == 1)
{
printf("Нет имени файла в командной строке\n");
printf("Формат команды: \n");
printf("  my_prog имя_пакетного_файла\n");
exit(1);
}

for (i=0; i<=argc; i++)
/* Проверка наличия модификаторов в командной строке */
```

```

if (*argv[i] == '-')
    switch(toupper(*(argv[i]+1)))
    {
        case 'D' : debug = TRUE;
                    printf("Задан режим отладки - Debug\n");
                    break;
        case 'Q' : quiet = TRUE;
                    printf("Отменен вывод
                        сообщений - Quiet\n");
                    break;
    }
else /* Тип не обнаружено - должно быть имя файла */
    strcpy(basename,argv[i]);

/* Образовать имя пакетного файла и открыть файл */
strcpy(filename,basename);
strcat(filename, ".scr");
fo = fopen(filename, "w");

```

Лист. 7.17. Получение имени пакетного файла из командной строки

В тексте программы (функция `fopen`) показана буква `w` (от англ. `write` - писать). Эта буква информирует систему управления файлами о том, что программа будет осуществлять запись в файл. Система управления файлами устанавливает свой внутренний указатель позиции в файле на начало файла. Если пакетный файл существует под тем же самым именем, которое вы сообщили программе, его предыдущее содержимое будет потеряно. В большинстве случаев это вполне приемлемо. Однако если есть несколько программ, выполняющих запись в пакетный файл, то вам желательно иметь в файле всю информацию полностью, а не просто вывод последней программы. В этом случае для программы, которая будет работать первой, используйте в функции `fopen` символ `w`, а для всех последующих программ букву `a`. (от англ. `append` — добавлять). Использование буквы `a` в функции `fopen` информирует систему управления файлами о том, что необходимо установить внутренний указатель на конец файла так, чтобы вы смогли добавить информацию к концу файла.

Между тем вернемся к основной теме... Нет ничего, чтобы сдерживало вас, особо не задумываясь, генерировать команды для пакетного файла в произвольной последовательности. Это тоже имеет право на жизнь. Но тем не менее я буду стремиться подвести читателя к применению методического подхода при создании пакетных файлов. Файлы типа DXF (тоже относится и к DWG-файлам) имеют определенную логическую организацию графической информации. В связи с этим чертеж разделяется на несколько секций:

- заголовок (параметры графического окружения);
- таблицы;
- типы линий;

слои;
стили;
виды;
блоки;
объекты.

Если бы мы должны были создать пакетный файл с такой структурой, то, сделав это, мы могли бы найти в нем требуемые параметры в том месте, где они нам могут понадобиться. Кроме того, пакетный файл будет легче читать.

Наш основной интерес состоит в том, чтобы поместить символьные команды системы AutoCAD в файл. Для этого есть несколько способов. Большинство способов выходит за рамки обычного программирования. Некоторые способы могут быть типа "эй, ухнем". Я буду стоять на практических позициях и предлагаю два подхода. Для них требуется, чтобы вы имели подготовленные для вывода данные. Эти подходы базируются на способах вывода.

Первый подход — наиболее подходящий для большинства случаев. Он заключается в том, чтобы локализовать операции записи в файл в отдельных подпрограммах. Это позволит направить вывод данных одного рода через одну подпрограмму. Например, чтобы начертить линию, вы вызываете подпрограмму черчения линии, передавая ей как аргументы координаты концевых точек отрезков. Для вычерчивания окружности вызывается подпрограмма черчения окружности, в качестве аргументов которой передаются координаты центра окружности и радиус. Вы могли бы придумать и передать в команду черчения окружности системы AutoCAD код и предопределенное число действительных чисел. Код служил бы для определения, какой из способов черчения окружности должен применяться и как при этом используются заданные числа. Подробности вычерчивания окружностей описаны в гл. 3.

Второй подход состоит в том, чтобы выполнять запись в пакетный файл в любом месте программы, где вы хотите сделать это. В данном способе операторы записи (в нашем примере функция `fprintf`) вкрапляются во всю программу. Эти операторы весьма заметны в тексте программы. Данный способ привлекает меня как наиболее быстрый для реализации; способ, который вы могли бы использовать, чтобы сделать что-нибудь очень быстро, не беспокоясь об элегантности и эффективности. Если вы применяете этот способ и имеете многочисленные операции записи данных одного рода (таких, как запись команд `LINE`), то вы совсем не экономите ресурсы. Однако способ прекрасно подходит для записи одиночных параметров, таких, какие можно встретить в секции заголовка. Нет необходимости тщательно разрабатывать подпрограмму, чтобы установить значения разрешающей способности и шага сетки.

На самом деле наиболее удовлетворительным представляется смешение этих двух способов. Я буду использовать отдельные операции

вывода для записи одиночных параметров и централизованный вывод для команд.

Для того чтобы придать одноразовым операциям согласованный характер, внимательно взгляните на остальные программы вывода для пакетных файлов. Я написал для таких операций макроопределения, используя оператор `#define` языка Си. Эти макроопределения приведены на листингах программ для каждой секции.

Секция заголовка

В секции заголовка устанавливаются параметры рабочей среды для чертежа. Эти параметры предназначаются не только для пакетных файлов, но и для пользователей. Конструктору надо иметь возможность начать работу с программно сгенерированным чертежом, не выполняя какие-либо дополнительные требуемые для него установки параметров. Но поскольку каждый имеет свой стиль работы, то это может быть трудно осуществимо.

Один из способов угадать, чего же хочет конструктор (задачи неблагодарной), — это спросить, что он предпочитает. В настоящее время я вообще не нашел способ, чтобы полностью удовлетворить обработку многочисленных вопросов и ответов с помощью компьютера. Я не люблю писать подобные программы и не люблю находиться в положении отвечающего на вопросы. По этим соображениям мне нравятся файлы предпочтительных значений параметров.

Файлы предпочтительных значений параметров содержат символическую информацию в коде ASCII. Эти файлы создаются с помощью редактора текста, а установка предпочтительных значений в этих файлах подчиняется простым правилам. Единственное, что должен сделать конструктор, — это изменить только те параметры, значения которых отличаются от значений по умолчанию, уже установленных в программе. Если конструктора удовлетворяют значения по умолчанию (не рассчитывайте на это), то нет проблем и незачем иметь файл предпочтительных значений параметров.

На лист. 7.18 показано содержимое типичного файла предпочтительных значений параметров. Слева находятся различные параметры программы. Справа указаны значения параметров, которым отдает предпочтение конструктор. Несколько параметров обозначают имена слоев генерируемых программой, а соответствующим значением этих параметров является установленный для слоя цвет.

```
ORTHO = ON
SNAP = 0.125
GRID = 0.25
BASE = WHITE
LEVEL1 = RED
LEVEL2 = BLUE
```

Лист. 7.18. Файл предпочтительных значений параметров

Вы можете достичь высокой производительности, используя предпочтительные значения параметров. Конструктор может определить некоторый стиль начертания текста для размеров, примечаний, предостережений, предупреждений и подписей (в зависимости от их назначения); и в то же время задать цвета для этих вариантов текста (в зависимости от слоя). Вы можете достичь такой гибкости, приложив творческие силы. Внутри вашей программы надо иметь процедуру для считывания файла предпочтительных значений параметров. Вот то, что вы должны сделать.

В секции данных вашей программы необходимо иметь следующее:

1. Действующие переменные, представляющие параметры в файле значений предпочтительных параметров. Значения этих переменных должны быть установлены соответственно со значениями параметров по умолчанию.

2. Массив допустимых имен параметров. Если вы хотите разрешить вариации написания этих имен, удостоверьтесь, что все варианты перечислены в этом списке.

3. Ссылка на переменную, соответствующую каждому имени параметра. Ссылка либо входит в одну структуру данных с именами параметров, либо имеется отдельный список. Ссылки в списке должны следовать в том же порядке, что и имена параметров в списке имен.

4. Тип данных. Если имеются смешанные типы данных, вы должны будете знать, как справиться с каждым из них. Поскольку вначале все данные находятся в символьной форме (в коде ASCII), то, если вы оставляете всю информацию в символьной форме, вам не нужна дополнительная информация о типе данных.

На лист. 7.19 показан пример задания этих четырех различных групп данных. На лист. 7.20 три последние группы данных объединены в одну структуру. На лист. 7.21 поясняется стратегия нахождения и установки предпочтительных значений для параметров. На лист. 7.22 приведен исходный текст программы для считывания информации из файла предпочтительных значений параметров, в которой используется структура данных, показанная на лист. 7.19. На лист. 7.23 приведен текст аналогичной программы, но с использованием структуры данных, показанной на лист. 7.20.

Независимо от того, как вы получаете значения для графических данных, вы должны занести эти значения в пакетный файл. Фрагмент программы на лист. 7.24 применяется для генерации в вашем пакетном файле параметров рабочей среды. В нем используются значения переменных, имена которых известны по предыдущим примерам. Записываемые в файл данные относятся к новому чертежу, поскольку в первых строчках фрагмента программы для системы AutoCAD устанавливается режим ввода описания нового чертежа.

```

#define CHAR      1
#define INTEGER   2
#define REAL      3
#define STRNG     4
#define YN        5

char      *ortho = ON;
double    *snap = 0.500;
double    *grid = 1.000;
char      *base_color = WHITE;
char      *lyr_color[] = {WHITE,
                           RED,
                           BLUE,
                           YELLOW,
                           GREEN,
                           MAGENTA,
                           CYAN,
                           WHITE,
                           WHITE,
                           WHITE,
                           WHITE};

char      param_name[] = {"ORTHO",
                           "SNAP",
                           "GRID",
                           "BASE",
                           "LEVEL0",
                           "LEVEL1",
                           "LEVEL2",
                           "LEVEL3",
                           "LEVEL4",
                           "LEVEL5",
                           "LEVEL6",
                           "LEVEL7",
                           "LEVEL8",
                           "LEVEL9",
                           "LEVEL10",
                           NULL};

int      params_type[] = {STRNG,
                           REAL,
                           REAL,
                           STRNG,
                           STRNG,
                           STRNG,
                           STRNG,
                           STRNG};

```



```

                                STANG,
                                STANG,
                                STANG,
                                STANG,
                                STANG,
                                STANG,
                                STANG,
                                0};

void                            *param_value[] = {&ortho,
                                                  &snap,
                                                  &grid,
                                                  &base_color,
                                                  &lyr_color[0],
                                                  &lyr_color[1],
                                                  &lyr_color[2],
                                                  &lyr_color[3],
                                                  &lyr_color[4],
                                                  &lyr_color[5],
                                                  &lyr_color[6],
                                                  &lyr_color[7],
                                                  &lyr_color[8],
                                                  &lyr_color[9],
                                                  &lyr_color[10],
                                                  NULL};

```

Лист 7 19. Раздельные списки данных для программы синтаксического разбора параметров

```

#define CHAR                    1
#define INTEGER                 2
#define REAL                    3
#define STRNG                   4
#define YN                      5

typedef
    struct param
    {
        char                *param_name;
        short               param_type;
        void                **param_value;
    }
    param;

char                *ortho = ON;
double              *snap = 0.500;
double              *grid = 1.000;
char                *base_color = WHITE;

```

```

char          ^lyr_color[] = {WHITE,
                                RED,
                                BLUE,
                                YELLOW,
                                GREEN,
                                MAGENTA,
                                CYAN,
                                WHITE,
                                WHITE,
                                WHITE,
                                WHITE};

param          params[] = {{ "ORTHO", STRNG, &ortho},
                             {"SNAP", REAL, &snap},
                             {"GRID", REAL, &grid},
                             {"BASE", STRNG, &base_color},

{"LAYER0", STRNG, &lyr_color[0]},
{"LAYER1", STRNG, &lyr_color[1]},
{"LAYER2", STRNG, &lyr_color[2]},
{"LAYER3", STRNG, &lyr_color[3]},
{"LAYER4", STRNG, &lyr_color[4]},
{"LAYER5", STRNG, &lyr_color[5]},
{"LAYER6", STRNG, &lyr_color[6]},
{"LAYER7", STRNG, &lyr_color[7]},
{"LAYER8", STRNG, &lyr_color[8]},
{"LAYER9", STRNG, &lyr_color[9]},
{"LAYER10", STRNG, &lyr_color[10]},

                                {NULL, 0, NULL}};

```

Лист. 7.20. Комбинированные списки данных для программы синтаксического разбора параметров

Для установки предпочтительных значений параметров:

Считать строку из файла предпочтительных параметров,
Найти имя параметра в списке допустимых имен.

вывести предупреждение, если имя не найдено,
 Получить тип данных для параметра (индекс в массиве
 типов совпадает с индексом имени в массиве имен),
 Преобразовать значение параметра в значение
 соответствующего типа данных,
 Запомнить преобразованное значение в выделенной для
 этого переменной

Лист. 7.21. Алгоритм синтаксического разбора параметров

```
void set_params (f)
FILE      *f;
{
    short    quit;
    short    i;
    short    state;
    char      w[20];
    int      *v;
    double    *d;
    char      *c;

    printf("\nСчитывание файла параметров\n");
    quit = FALSE;
    state = 1;
    while (!quit)
    {
        if (fscanf (f, "%s", w) == EOF) state = 0;
       strupr(w);
        switch (state)
        {
            case 0 : quit = TRUE;
                     break;
            case 1 :
                     i = 0;
                     /* Поиск слова -- установить указатель */
                     /* поочередно на каждый элемент таблицы */
                     while (param_name[i])
                     {
                         /* Найдено ли нужное слово ? */
                         if(!strcmp(param_name[i], w))
                         {
                             /* перейти в состояние 2. если найдено */
                             state = 2;
                             break;
                         }
                         ++i;
                     }
                     /* Если соответствия не найдено -- выдать */
```

```

        /* предупреждение о возможной опечатке */
        if (!p) printf("Ожидалось имя параметра,
                        а найдено %s\n",w);
        break;
case 2 :
case 3 :
    /* Пропустить знак равенства*/
    if (w[0] == '=') break;
    state = 1;
    /* Получить из таблицы тип данных
       для параметра */
    /* Присвоить значение переменной
       соответствующего типа */
    switch(param_type[i])
    {
        case CHAR      : c = (char *) param_value[i];
                        *c = toupper(w[0]);
                        break;

        case INTEGER   : v = (int *) param_value[i];
                        *v = atoi(w);
                        break;

        case REAL      : d = (double *) param_value[i];
                        *d = atof(w);
                        break;

        case STRNG     : *(param_value[i]) = locate(w);
                        break;

        /* Определяются значения типа YES/NO */
        case YN        : v = (int *) param_value[i];
                        *v = (toupper(w[0]) == 'Y')?
                            TRUE:FALSE;
                        break;
    }
    break;
default : break;
}
}
}

```

Лист. 7.22. Программа синтаксического разбора параметров с отдельными списками данных

```

void set_params (f)
FILE    *f;
{

```

```

short      quit;
short      state;
char       w[20];
param      *p;
int        *v;
double     *d;
char       *c;

printf("\nСчитывание файла параметров\n");
quit = FALSE;
state = 1;
while (!quit)
{
    if (fscanf (f,"%s",w) == EOF) state = 0;
   strupr(w);
    switch (state)
    {
        case 0 : quit = TRUE;
                  break;
                  /* Поиск слова -- установить указатель */
                  /* поочередно на каждый элемент таблицы */
        case 1 :
                  p = param;
                  while (p->param_name)
                  {
                      /* Найдено ли нужное слово ? */
                      if (!strcmp(p->param_name,w))
                      {
                          /* перейти в состояние 2, если найдено */
                          state = 2;
                          break;
                      }
                      p++;
                  }
                  /* Если соответствия не найдено -- выдать */
                  /* предупреждение о возможной опечатке */
                  if (!p) printf("Ожидалось имя параметра,
                               а найдено %s\n",w);
                  break;
        case 2 :
        case 3 :
                  /* Пропустить знак равенства*/
                  if (w[0] == '=') break;
                  state = 1;
                  /* Получить из таблицы тип данных
                     для параметра */
                  /* Присвоить значение переменной

```

```

        соответствующего типа */
switch(p->param_type)
{
    case CHAR      : c = (char *) p->param_value;
                    *c = toupper(w[0]);
                    break;

    case INTEGER   : v = (int *) p->param_value;
                    *v = atoi(w);
                    break;

    case REAL      : d = (double *)p->param_value;
                    *d = atof(w);
                    break;

    case STRNG     : *(p->param_value) = locate(w);
                    break;

    /* Определяются значения типа YES/NO */
    case YN        : v = (int *) p->param_value;
                    *v = (toupper(w[0]) == 'Y')?
                        TRUE:FALSE;
                    break;

}
break;
default : break.
}
}
}

```

Лист. 7.23. Программа синтаксического разбора параметров с комбинированными списками данных

```

printf("Инициализация пакетного файла\n");

fprintf(SCR_file, "%s %s\n", ZOOM, ALL);
fprintf(SCR_file, "%s %f\n", GRID, grid);
fprintf(SCR_file, "%s %s\n", ORTHO, ortho);
fprintf(SCR_file, "%s\n", LAYER);
fprintf(SCR_file, "%s %s 0\n", COLOR, lyr_color[0]);
fprintf(SCR_file, "%s BASE %s %s BASE\n", NEW, COLOR,
        base_color);
for(i=1; i<11; i++);
    fprintf(SCR_file, "%s LEVEL%d %s %s LEVEL%d\n",
        NEW, i, COLOR, lyr_color[i], i);
fprintf(SCR_file, "\n");

```

Лист. 7.24. Генерация параметров среды

Секция таблиц

Тщательность проработки любых сторон программы генерации пакетных файлов зависит от требований, предъявляемых графическими объектами. В моих программах очень важными являются слои. До сего дня я редко работаю с типами линий и никогда не работаю с видами. Принятые в системе AutoCAD значения по умолчанию обычно удовлетворяют все мои запросы. Определите ваши потребности. Вам незачем определять значения подряд всех параметров.

Типы линий

Придерживаясь последовательности следования информации, как это принято в примере для DXF-файла, следующим будет рассматриваться тип линий. В DXF-файлах типы линий описываются очень подробно. И вы могли бы в действительности вставить такое описание в DXF-файл. Но вы могли бы создать новый тип линии, используя пакетный файл, хотя я не уверен, стоит ли это делать именно сейчас.

Имеется четыре модификатора, определяющих режимы работы с типами линий. Только три из них полезны для использования в пакетных файлах. Четвертый — запрос типов линий — позволяет получать список всех известных типов линий. Но при работе с пакетным файлом вы можете сделать совсем немного с таким списком. Однако другие три модификатора — CREATE, LOAD и SET могут оказаться полезными. Мы рассмотрим модификатор SET позже, и используем его для управления.

Модификатор CREATE позволяет нам определять свои собственные типы линий. Для создания типа линии нам необходимо:

- имя типа линии;

- имя файла для типа линии;

- короткое (до 80 символов) символьное описание;

- математическое описание линии.

Символьное описание предназначается для задания внешнего вида линии, того, как она выглядит. На лист. 7.25 показаны некоторые примеры описаний типов линий.

```
.....
-----
-----
-----
```

Лист. 7.25. Описание типов линий

Математическое описание представляет собой список длин сегментов, детализирующих определение линии. Положительные значения в данном описании обозначают длину видимых сегментов линий. Отрицательные значения — длину промежутков между сегментами.

Нуль служит для обозначения пунктира или точки. Точки не изменяют протяженность линии.

Пунктирная линия может иметь следующее математическое описание:

$A, 0, -0.25, 0, -0.25, 0, -0.25, 0, -0.25$

Начальная буква A применяется для выравнивания. Нет никаких других вариантов ее значения. На лист. 7.26 показана программа для создания нового типа линии. На лист. 7.27 - содержимое полученного в результате ее работы пакетного файла.

Легче загрузить описания уже известных типов линий. В пакетный файл включается тип линии и имя файла описания типа линии. На лист. 7.28 приведена программа для загрузки известного типа линии из стандартного файла. На лист. 7.29 - содержимое полученного в результате ее работы пакетного файла.

```
void create_linetype(SCR_file, ltype_name, ltype_file,
                    ASCII_description, math_description)
FILE      *SCR_file;
char      *ltype_name;
char      *ltype_file;
char      *ASCII_description;
char      *math_description;
{
    fprintf(SCR_file, "%s %s %s\n", LINETYPE, CREATE, ltype_name);
    fprintf(SCR_file, "%s\n", ltype_file);
    fprintf(SCR_file, "%s\n", ASCII_description);
    fprintf(SCR_file, "A, %s\n", math_description);
}
```

Лист. 7.26. Программа создания описания типа линии

```
LINETYPE C my_sos
my_lines
.....
A, 0, -.25, 0, -.25, 0, -.25, .25, -.25, .25, -.25, .25, 0, -.25, 0, -.25, 0
```

Лист. 7.27. Пакетный файл с описанием типа линии

```
void load_linetype(SCR_file, ltype_name, ltype_file)
FILE      *SCR_file;
char      *ltype_name;
char      *ltype_file;
```



```

{
    fprintf(SCR_file, "%s %s %s\n", LINETYPE, SET, ltype_name);
    fprintf(SCR_file, "%s\n", ltype_file);
}

```

Лист. 7.28. Программа загрузки описания типа линии

```

LINETYPE S my_sos
my_lines

```

Лист. 7.29. Пакетный файл загрузки описания типа линии

Слой

Установка параметров слоев, выполняемая с помощью программ, более широко применима, чем установка параметров для любых других таблиц. Просмотрев материал гл. 3, посвященный команде LAYER, вы отметите, что в этой команде имеются несколько параметров. Вы можете разработать программу любой сложности для определения требуемых для вас характеристик слоев.

На представленных в этом параграфе листингах показан способ генерации пакетных файлов с использованием макроопределений, заданных оператором #define. Команды установки параметров слоев могут быть очень короткими и удобными для работы, они могут быть вставлены в любом месте программы. Как я отмечал ранее, такой прием может увеличить объем вашей программы за счет происходящего расширения макросов. Если это создаст проблему, то для преобразования макросов в подпрограммы требуется только задать тип аргументов и добавить лишнии скобки в операторах вывода.

Макроопределения, приведенные на лист. 7.30, находятся в файле defines.h. Они определяют основные макросы для программы. Соответствующие им операторы появляются в тексте программы d в том случае, если в этом месте программы был указан макрос. Убедитесь, что флажок выбора языка определен так, чтобы макросы порождали команды системы AutoCAD на соответствующем национальном языке. На лист. 7.31 показано применение макросов. На лист. 7.32 показан фрагмент полученного в результате пакетного файла.

```

#define NEW_LAYER(f,n)  fprintf(f, "%s %s %s\n\n", LAYER, NEW, n)
#define SET_LAYER(f,n)  fprintf(f, "%s %s %s\n\n", LAYER, SET, n)
#define SET_COLOR(f,c,n) fprintf(f, "%s %s %s %s\n\n",
                                LAYER, COLOR, c, n)
#define SET_LTYPE(f,l,n) fprintf(f, "%s %s %s %s\n\n",
                                LAYER, LTYPE, l, n)
#define LAYER_ON(f,n)   fprintf(f, "%s %s %s\n\n", LAYER, ON, n)
#define LAYER_OFF(f,n)  fprintf(f, "%s %s %s\n\n", LAYER, OFF, n)

```

```
#define MAKE_LAYER(f,n) fprintf(f,"%s %s %s\n\n",LAYER,MAKE,n)
#define FREEZE(f,n) fprintf(f,"%s %s %s\n\n",LAYER,FREEZE,n)
#define THAW_LAYER(f,n) fprintf(f,"%s %s %s\n\n",LAYER,THAW,n)
```

Лист. 7.30. Макроопределения для команды **LAYER**

```
NEW_LAYER(SCR_file,"FLOOR");
NEW_LAYER(SCR_file,"SURFACE");
SET_COLOR(SCR_file,"RED","FLOOR");
SET_COLOR(SCR_file,"13","SURFACE");
LAYER_ON(SCR_file,"*");
SET_LAYER(SCR_file,"FLOOR");
fprintf(scr_file,"%s\n",REDRAW);
```

Лист. 7.31. Использование макросов в программе

```
LAYER NEW FLOOR

LAYER NEW SURFACE

LAYER COLOR RED FLOOR

LAYER COLOR 13 SURFACE

LAYER ON *

LAYER SET FLOOR

REDRAW
```

Лист. 7.32. Пакетный файл с командами **LAYER**

Стиль начертания текста

Установка стиля начертания текста с помощью пакетного файла является одной из наиболее регулярно выполняемых операций. Система AutoCAD для установки стиля начертания текста должна получить порцию данных с определенной структурой. Вы должны подготовить эти данные.

Для создания стиля начертания текста вы должны иметь следующие данные:

- имя стиля начертания текста;
- имя файла описания шрифта (.SHX);
- высоту текста (0 - высота не фиксирована);
- степень сжатия/растяжения (для текста со сжатым шрифтом);
- угол наклона литер;

расположение текста справа налево (флажок YES/NO);
 расположение текста сверху вниз (флажок YES/NO);
 вертикальное расположение (флажок YES/NO).

Располагая этой информацией, вы можете создавать пакетный файл для определения нового стиля. Программа достаточно проста. Вы начинаете с записи команды **STYLE** и далее вставляете все остальные данные, требуемые для системы AutoCAD. На лист. 7.33 показана программа для решения этой весьма нужной задачи. На лист. 7.34 представлен типичный фрагмент пакетного файла.

```
void create_text_style(SCR_file, style_name, font_name,
                      height, width, oblique,
                      backwards, upside_down)
FILE          *SCR_file;
char          *style_name;
char          *font_name;
double        height;
double        width;
double        oblique;
int           backwards;
int           upside_down;
{
    fprintf(SCR_file, "%s %s\n", STYLE, style_name);
    fprintf(SCR_file, "%s\n", font_name);
    fprintf(SCR_file, "%f\n", height);
    fprintf(SCR_file, "%f\n", width);
    fprintf(SCR_file, "%f\n", oblique);
    fprintf(SCR_file, "%s\n", (backwards)?YES:NO);
    fprintf(SCR_file, "%s\n", (upside_down)?YES:NO);
}
```

Лист. 7.33. Программа создания стиля начертания текста

```
STYLE MY_FONT
TXT
0.0
0.75
0.0
NO
NO
```

Лист. 7.34. Созданное описание стиля начертания текста

Виды

Задание вида в системе AutoCAD в действительности производится в два этапа. Во-первых, желаемый вид должен быть построен. Это становится возможным при задании точки наблюдения. Во-вторых, для наименования этого частного вида используется команда VIEW. Образование вида не является простой операцией, чисто механически выполняемой в пакетном файле, сначала вы должны убедиться, как будет выглядеть на экране заданный вид.

В вашей программе операции по построению и наименованию вида могут быть объединены в одной подпрограмме. Что касается самой программы, имя вида и точка наблюдения являются для нее исчерпывающей информацией. На лист. 7.35 приведен исходный текст подпрограммы. На лист. 7.36 показан полученный в результате работы этой программы пакетный файл.

```
void create_view(SCR_file,x,y,z,view_name)
FILE      *SCR_file;
double     x;
double     y;
double     z;
char       *view_name;
{
    fprintf(SCR_file,"%s %f,%f,%f\n",VPOINT,x,y,z);
    fprintf(SCR_file,"%s %s %s\n",VIEW,SAVE,view_name);
    fprintf(SCR_file,"%s 0.0,0.0,1.0\n",VPOINT);
}
```

Лист. 7.35. Программа создания описания вида

```
VPOINT 2.4,1.3,2.5
VIEW SAVE NICE_VIEW
VPOINT 0.0,0.0,1.0
```

Лист. 7.36. Созданное описание вида

Секции объектов и блоков

Представляется, что ненужно слишком многое на краю неизбежного. Секции блоков и объектов по-прежнему так схожи, что их обсуждение следует объединить. Мы обрабатываем две эти секции одинаково. Имея собранные вместе объекты, образующие единое графическое изображение, очень просто, используя команды BLOCK и WBLOCK, поместить эти объекты в блок.

В начале этого раздела книги мы рассмотрели два подхода к генерации пакетных файлов. Эти подходы заключались либо в централизации операций вывода, либо в их децентрализации. Децентрализованный способ вывода хорошо подходит для секций заголовков и отчасти для таблиц. Однако такой способ не очень хорош для секции объектов. Если в секции заголовка вы записывали команду **LIMITS** только один раз, то в секции объектов команда **LINE** может появляться любое число раз. Для облегчения управления и повышения эффективности вы должны локализовать вывод этих команд. Мы остановимся на подробностях разработки подпрограмм вывода.

Зависит от вас, какие данные и какие операторы использовать для вывода команд. В этом параграфе в пакетный файл будут помещаться ваши данные. Вы увидите почти идентичную обработку секции объектов еще раз в гл.8. В следующей главе формат вывода данных будет разным, но представление данных для вывода останется одним и тем же. Заранее извините меня, если я перенесу отсюда все в следующую главу и модифицирую применительно к обстоятельствам.

К этому моменту в вашей программе для записи пакетного файла уже установлены параметры рабочей среды, описания типов линий, слоев, стилей и видов, на которые вы ссылаетесь при создании блоков и объектов. Настало время появиться на вашем экране некоторым изображениям.

Но сначала давайте расставим все по местам. Обсудив секцию таблиц, я отложил в сторону использование модификаторов **SET** для типов линий, слоев, стилей начертания текста и видов. Я сделал это потому, что до сих пор они никогда не использовались в программе. Наиболее важными из модификаторов **SET** являются модификаторы для слоев и стилей начертания текста. Типы линий следует задавать при объявлении слоя. Не стоит уделять много внимания заданию типов линии целиком для всего слоя, когда вы находитесь на полпути к получению чертежа. Задание видов — это нечто такое, что, по моему мнению, следует предоставить конструктору. Я мог бы найти случаи, когда вы могли бы сами сконструировать что-нибудь на экране и использовать полученные виды для эффективного управления показом слайдов; но я не думаю, что это подходящее приложение сил, когда вы повышаете производительность конструктора, стараясь сами выполнить его работу. Конструировать виды — это прекрасно, но использовать их — не очень. На лист. 7.37 — 7.40 показаны некоторые применения модификаторов **SET** в пакетных файлах. Команда **REDRAW** применяется для того, чтобы показать пустые строки, следующие после модификатора **SET**, обозначающие значения по умолчанию или строки, завершающие команду. На лист. 7.41 — 7.44 приведены тексты программ, с помощью которых были созданы строки этих пакетных файлов. Частота появления представленных этих фрагментов в тексте программ зависит от применяемой организации вывода. Как можно заметить, в представленных фрагментах программ используются различные варианты организации вывода.

`LINESTYLE SET DASHED`

Лист. 7.37. Модификатор **SET** для задания типа линии

`LAYER SET LAYER_1`

`REDRAW`

Лист. 7.38. Модификатор **SET** для выбора слоя

`STYLE HELVETIC`

(имя файла по умолчанию)

(высота текста по умолчанию)

(степень сжатия/растяжения по умолчанию)

(угол наклона литер по умолчанию)

(обратная последовательность букв по умолчанию)

(расположение сверху вниз по умолчанию)

(вертикальное расположение по умолчанию)

`REDRAW`

Лист. 7.39. Модификатор **SET** для задания стиля начертания текста

`VIEW R VISTA`

Лист. 7.40. Модификатор **SET** для задания вида

```
#define SET_LINETYPE(f,l) fprintf(f,"%s %s %s\n",  
                                LINETYPE,SET,l);
```

Лист. 7.41. Текст программы — модификатор **SET** для типа линии

```
#define SET_LAYER(f,l) fprintf(f,"%s %s %s\n\n",LAYER,SET,l);
```

Лист. 7.42. Текст программы — модификатор **SET** для слоя

```
void set_style(dxfl, stylename)  
{  
    FILE *dxfl;  
    char *stylename;  
    fprintf(f,"%s %s\n",STYLE,stylename);  
    fprintf(f,"%s\n\n\n\n\n\n\n\n\n\n\n");  
}
```

Лист. 7.43. Текст программы — модификатор **SET** для стиля начертания текста

```
#define SET_VIEW(f,n)    fprintf(f,"%s %s %s\n",VIEW,V_R,n);
```

Лист. 7.44. Текст программы — модификатор SET для вида

Применяя команду CHANGE системы AutoCAD, вы можете изменять слой, тип линии, уровень возвышения или высоту (толщину) некоторых объектов. Данная команда имеет так много приложений, комбинаций и вариаций, что я не собираюсь тратить на нее время. Если вы точно знаете, что вы хотите изменить, то, воспользовавшись способом точного указания объекта, который хотите изменить, вы сможете использовать команду CHANGE на практике. Вы имеете хороший шанс воспользоваться этой командой, это действительно команда интерактивного редактирования. Оставшуюся часть нашего обсуждения мы посвятим программам генерации пакетных файлов для графических объектов, исходные тексты которых приведены на листингах.

Все представленные на листингах программы включают проверку слоя. Предположим, что имеется глобальная переменная со значением текущего слоя. Если передаваемое какой-либо из подпрограмм генерации пакетного файла имя слоя не совпадает с именем текущего слоя, то подпрограмма выполняет установку нового слоя и делает новый слой текущим.

Аргументами для этих подпрограмм генерации пакетного файла являются данные простых типов. Таким образом, координаты представляются как отдельные числа с плавающей точкой. Программы отслеживают, чтобы для правильной генерации пакетных файлов аргументы использовались в соответствующем порядке.

Существует другой способ, который вы можете захотеть изучить. В начале главы, посвященной DXF-файлам, программы помещали данные в хорошо определенные структуры данных. В процессе выполнения программы осуществлялась передача этих структур данных другим процедурам в программе. Вместо передачи нескольких значений как отдельных координат и углов единственным аргументом подпрограммы был указатель на структуру. Я хотел бы предложить передавать подпрограммам генерации пакетного файла вместо нескольких числовых аргументов ссылку на структуру. Если вы воплотили идеи, рассматривавшиеся в предыдущих главах, в работающие программы, то это обеспечит некоторую преемственность логики и структур данных в ваших программах. Поместить единственное значение указателя в программный стек быстрее, чем поместить в стек несколько чисел с плавающей точкой. С другой стороны, ранее я продемонстрировал несколько различных методов, с тем чтобы вы увидели разнообразие доступных для вас подходов.

В табл. 7.1 приведен список графических команд, рассмотренных в данной главе, и номеров соответствующих им листингов подпрограмм, применяемых для вывода этих команд в пакетные файлы. Если нужная вам команда отсутствует в данном списке, проверьте

требования, предъявляемые к команде по справочной документации для системы AutoCAD. Следуя стилю написания представленных в книге подпрограмм, разработайте свою собственную подпрограмму. Комментарии, сопровождающие тексты приведенных программ, подскажут вам последовательность действий. В листингах этих программ вы не найдете такие команды, как **DIMENSION**, **DONUT**, **POLYGON** или **ELLIPSE**.

Таблица 7.1. Список подпрограмм для генерации команд

LINE	Лист. 7.45
POINT	Лист. 7.46
CIRCLE	Лист. 7.47
ARC	Лист. 7.48
SOLID	Лист. 7.50
TEXT	Лист. 7.51
SHAPE	Лист. 7.52
INSERT	Лист. 7.53
ATTDEF	Лист. 7.54
POLYLINE	Лист. 7.55
BLOCK	Лист. 7.56
WBLOCK	Лист. 7.57

Почему пакетные файлы?

Действительно ли необходимо проделывать всю работу по генерации пакетных файлов, запоминая структуру команд, корректно подготавливая данные и переключаясь от языка программирования к языку системы AutoCAD? Не было бы легче для тех деталей, которые вы хотите вычертить, просто сгенерировать DXF-файл? Я вступал в такую дискуссию неоднократно и хочу привести некоторые доводы за и против использования пакетных файлов и DXF-файлов. Оба формата файлов имеют свое назначение и преимущество. Понимание достоинств и недостатков этих двух форматов позволит вам выбрать наилучший вариант для конкретного случая.

В данной главе был рассмотрен формат пакетных файлов. Следующая глава будет посвящена формату DXF-файлов, рассматриваемого в качестве входного формата. Здесь я раскрою преимущества пакетных файлов. Не всегда существуют конфликты между этими форматами, но я постараюсь выразить противоположные точки зрения, если они есть.

Пакетные файлы легче генерировать. Пакетные файлы легче читать и модифицировать. С помощью пакетного файла можно вставлять неизвестные блоки. Пакетный файл, написанный на естественном для человека языке, легко воспринимается. Используя пакетные файлы, можно просто организовать непрерывный показ слайдов или различные демонстрации. Пакетный файл системы AutoCAD в совокупности с пакетным файлом MS-DOS можно использовать для создания, модификации и сохранения чертежа без вмешательства человека.

```
void write_line (SCR_file, from_x, from_y, to_x, to_y, layer)
FILE          *SCR_file;
```



```

double      from_x,
double      from_y;
double      to_x;
double      to_y;
char        *layer;
{
    /* Проверка текущего слоя */

    if (strcmp(layer,current_layer))
    {
        set_layer(SCR_file,layer);
        current_layer = layer;
    }

    /* В пакетном файле для команды LINE допускается любое */
    /* число строк с координатами -- список координат */
    /* завершает дополнительная пустая строка. Нет отличий */
    /* по сравнению с DWG-файлом -- объекты LINE все имеют */
    /* только две конечные точки. */
    /* Потери эффективности по сравнению с DWG-файлами нет. */

    fprintf(SCR_file,"%s %f,%f %f,%f\n\n",LINE,
            from_x, from_y, to_x, to_y);
}

```

Лист. 7.45. Формирование команды **LINE** для пакетного файла

```

void write_point (SCR_file,x,y,layer)
FILE          *SCR_file;
double        x;
double        y;
char          *layer;
{
    /* Проверка текущего слоя */

    if (strcmp(layer,current_layer))
    {
        set_layer(SCR_file,layer);
        current_layer = layer;
    }

    fprintf(SCR_file,"%s %f,%f\n",POINT,x,y);
}

```

Лист. 7.46. Формирование команды **POINT** для пакетного файла

```

#define CENTER_RADIUS      1
#define CENTER_DIAMETER    2
#define CIRCLE_3_POINTS    3
#define CIRCLE_2_POINTS    4

void write_circle (SCR_file,code,a,b,c,d,e,f,layer)
FILE             *SCR_file;
int              code;
double           a;
double           b;
double           c;
double           d;
double           e;
double           f;
char             *layer;
{
    /* Проверка текущего слоя */

    if (strcmp(layer,current_layer))
    {
        set_layer(SCR_file,layer);
        current_layer = layer;
    }

    /* Имеются четыре различных способа задания окружности */
    /* в пакетном файле -- вызывающая программа должна */
    /* передавать код, обозначающий используемый способ. */
    /* Предшествующие операторы #define задают числовые */
    /* значения, определяющие способ построения */
    /* окружности. */
    /* В зависимости от применяемого способа аргументы */
    /* подпрограммы имеют различное назначение. */

    switch(code)
    {
        case CENTER_RADIUS : fprintf(SCR_file,"%s %f,%f\n",CIRCLE,a,b,c); break;
        case CENTER_DIAMETER :
            fprintf(SCR_file,"%s %s %f,%f %f\n",
                CIRCLE,DIAMETER,a,b,c);
            break;
        case CIRCLE_3_POINTS :
            fprintf(SCR_file,"%s %s %f,%f %f,%f %f,%f\n",
                CIRCLE,3P,a,b,c,d,e,f);
            break;
    }
}

```

```

        case CIRCLE_2_POINTS :
            fprintf(SCR_file, "%s %s %f %f %f %f\n",
                    CIRCLE_2P, a, b, c, d);
            break;
    }
}

```

Лист. 7.47. Формирование команды **CIRCLE** для пакетного файла

```

#define ARC_3_POINTS      1
#define ARC_SCE           2
#define ARC_SCA           3
#define ARC_SCL           4
#define ARC_SER           5
#define ARC_SEA           6
#define ARC_SED           7
#define ARC_CONTINUE     8

void write_arc (SCR_file, code, a, b, c, d, e, f, layer)
FILE          *SCR_file;
int           code;
double        a;
double        b;
double        c;
double        d;
double        e;
double        f;
char          *layer;
{
    /* Проверка текущего слоя */

    if (strcmp(layer, current_layer))
    {
        set_layer(SCR_file, layer);
        current_layer = layer;
    }

    /* Имеются восемь различных способов задания дуги */
    /* в пакетном файле -- вызывающая программа должна */
    /* передавать код, обозначающий используемый способ. */
    /* Предшествующие операторы #define задают числовые */
    /* значения, определяющие способ */
    /* построения дуги. */
    /* В зависимости от применяемого способа аргументы */
    /* подпрограммы имеют различное назначение. */

    switch(code)
    {

```

```

case ARC_3_POINTS : /* По трем точкам */
    fprintf(SCR_file, "%s %f,%f %f,%f %f,%f\n", ARC,
            a,b,c,d,e,f);
    break;
case ARC_SCE      : /* По точкам начала,
                    центра и конца */
    fprintf(SCR_file, "%s %f,%f %s %f,%f %f,%f\n",
            ARC,a,b,ARC_C,c,d,e,f);
    break;
case ARC_SCA      : /* Начало, центр и угол */
    fprintf(SCR_file, "%s %f,%f %s %f,%f %s %f\n",
            ARC,a,b,ARC_C,c,d,ARC_A,e);
    break;
case ARC_SCL      : /* Начало, центр и длина */
    fprintf(SCR_file, "%s %f,%f %s %f,%f %s %f\n",
            ARC,a,b,ARC_C,c,d,ARC_L,e);
    break;
case ARC_SER      : /* Начало, конец и радиус */
    fprintf(SCR_file, "%s %f,%f %s %f,%f %s %f\n",
            ARC,a,b,ARC_E,c,d,ARC_R,e);
    break;
case ARC_SEA      : /* Начало, конец и угол */
    fprintf(SCR_file, "%s %f,%f %s %f,%f %s %f\n",
            ARC,a,b,ARC_E,c,d,ARC_A,e);
    break;
case ARC_SED      : /* Начало, конец и
                    направление */
    fprintf(SCR_file, "%s %f,%f %s %f,%f %s %f\n",
            ARC,a,b,ARC_E,c,d,ARC_D,e);
    break;
case ARC_CONTINUE : /* Продолжение */
    fprintf(SCR_file, "%s\n%f,%f\n", ARC,a,b);
    break;
}
}

```

Лист. 7.48. Формирование команды ARC для пакетного файла

```

typedef
struct coord
{
    double    x;
    double    y;
}            coord;

/* Для полилиний и полос требуется непрерывная послед- */
/* довательность координат, так чтобы соответствующим */
/* образом выполнялись соединения и срезы. */

```

```

/* В программах для команд TRACE и POLYLINE предпола- */
/* гается, что точки вершин должны находиться в таблице */
/* типа coord. Также предполагается, что ведется подсчет */
/* числа вершин, заданных в таблице. */
/* Такой подход вы также можете применять и для LINE. */

```

```

int          xy_count = 0;
coord        *xy[100]; /* Вы задаете размер массива */

```

```

void write_trace (SCR_file, xy, n, width, layer)
FILE          *SCR_file;
coord         xy[];
int           n;
double        width;
char          *layer;
{
    int        i;

    /* Проверка текущего слоя */

    if (strcmp(layer, current_layer))
    {
        set_layer(SCR_file, layer);
        current_layer = layer;
    }
    fprintf(SCR_file, "%s %f\n", TRACE, width);
    for(i=0; i<n; i++)
        fprintf(SCR_file, "%f,%f\n", xy[i].x, xy[i].y);
    fprintf(SCR_file, "\n"); /* Завершающая строка */
}

```

Лист. 7.49. Формирование команды **TRACE** для пакетного файла

```

void write_solid (SCR_file, x1, y1, x2, y2, x3, y3, x4, y4, layer)
FILE          *SCR_file;
double        x1;
double        y1;
double        x2;
double        y2;
double        x3;
double        y3;
double        x4;
double        y4;
char          *layer;
{

    /* Проверка текущего слоя */

```

```

if (strcmp(layer,current_layer))
{
    set_layer(SCR_file,layer);
    current_layer = layer;
}

fprintf(SCR_file,"%s %f,%f %f,%f\n",SOLID,x1,y1,x2,y2);
fprintf(SCR_file,"%f,%f %f,%f\n\n",x3,y3,x4,y4);
}

```

Лист. 7.50. Формирование команды **SOLID** для пакетного файла

```

#define LEFT_JUSTIFY      1
#define CENTER_JUSTIFY   2
#define MIDDLE_JUSTIFY   3
#define RIGHT_JUSTIFY    4
#define ALIGN_JUSTIFY    5
#define FIT_JUSTIFY      6
#define CONTINUE_TEXT    7

void write_text (SCR_file,justify,x1,y1,x2,y2,ht,angle,
                txt,style,layer)
FILE          *SCR_file;
int           justify;
double        x1;
double        y1;
double        x2;
double        y2;
double        ht;
double        angle;
char          *txt;
char          *style;
char          *layer;
{
    /* Проверка текущего слоя */
    if (strcmp(layer,current_layer))
    {
        set_layer(SCR_file,layer);
        current_layer = layer;
    }

    if (justify == CONTINUE_TEXT)
        fprintf(SCR_file"\n");
    else
    {
        fprintf(SCR_file"%s ",TEXT);
    }
}

```

```

        if (style) fprintf(SCR_file, "%s %s\n", TEXT_S, style);
    }

    switch(justify)
    {
        case LEFT_JUSTIFY :
            fprintf(SCR_file, "%f,%f %f %f\n",
                    x1,y1,ht,angle);
            break;
        case CENTER_JUSTIFY :
            fprintf(SCR_file, "%s %f,%f %f %f\n", TEXT_C,
                    x1,y1,ht,angle);
            break;
        case MIDDLE_JUSTIFY :
            fprintf(SCR_file, "%s %f,%f %f %f\n", TEXT_M,
                    x1,y1,ht,angle);
            break;
        case RIGHT_JUSTIFY :
            fprintf(SCR_file, "%s %f,%f %f %f\n", TEXT_R,
                    x1,y1,ht,angle);
            break;
        case ALIGN_JUSTIFY :
            fprintf(SCR_file, "%s %f,%f %f %f\n", TEXT_A,
                    x1,y1,x2,y2);
            break;
        case FIT_JUSTIFY :
            fprintf(SCR_file, "%s %f,%f %f %f\n", TEXT_F,
                    x1,y1,x2,y2,ht);
            break;
        case CONTINUE_TEXT : fprintf(SCR_file, "\n");
            break;
    }
    fprintf(SCR_file, "%s\n", txt);
}

```

Лист. 7.51. Формирование команды **TEXT** для пакетного файла

```

void write_shape (SCR_file, shape, x1, y1, ht, angle, layer)
FILE             *SCR_file;
char             *shape;
double           x1;
double           y1;
double           ht;
double           angle;
char             *layer;
{

```

```

/* Проверка текущего слоя */
if (strcmp(layer,current_layer))
{
    set_layer(SCR_file,layer);
    current_layer = layer;
}

fprintf(SCR_file"%s %s %f,%f %f %f\n",
        SHAPE.shape,x1,y1,ht,angle).
}

```

Лист. 7.52. Формирование команды SHAPE для пакетного файла

```

void write_insert (SCR_file,name,x1,y1,
                  xscale,yscale,angle,layer)
FILE             *SCR_file;
char             *name;
double           x1;
double           y1;
double           xscale;
double           yscale;
double           angle;
char             *layer;
{
    /* Проверка текущего слоя */
    if (strcmp(layer,current_layer))
    {
        set_layer(SCR_file,layer);
        current_layer = layer;
    }

    fprintf(SCR_file"%s %s %f,%f %f %f %f\n",INSERT,
            name,x1,y1,xscale,yscale,angle);
}

```

Лист. 7.53. Формирование команды INSERT для пакетного файла

```

void write_attrib (SCR_file,txt)
FILE             *SCR_FILE;
char             *txt;
{
    /* Проверка текущего слоя отсутствует. */
    /* информацию о слое дают */
    /* команды ATTDEF или INSERT. */
}

```



```

fprintf(SCR_FILE, "%s\n", txt);
/* Данная подпрограмма должна следовать */
/* непосредственно после write_insert. */
/* чтобы задать значения для определенных */
/* атрибутов, которые зависят от блока. */
}

```

Лист. 7.54. Формирование модификатора ATTRIBUTE для пакетного файла

```

typedef
struct coord
{
    double    x;
    double    y;
    double    sw; /* Начальная ширина */
    double    ew; /* Конечная ширина */
}            coord;

/* Для полилиний и полос требуется непрерывная после- */
/* довательность координат, так чтобы соответствующим */
/* образом выполнялись соединения и срезы. */
/* В программах для команд TRACE и POLYLINE предпола- */
/* гается, что точки вершин должны находиться в таблице */
/* типа coord. Также предполагается, что ведется подсчет */
/* числа вершин, заданных в таблице. */
/* */
/* Вам требуется ввести дополнительные поля в структуру */
/* coord, чтобы использовать возможности задания в поли- */
/* линиях дуговых сегментов. Дополнительные поля включают */
/* углы, вторые точки дуги или точки центра */
/* */
/* Вам также необходимо поле для указания, какой вариант */
/* задания сегмента полилинии используется (отрезок, */
/* дуга в различных вариациях). */

int            xy_count=0;
coord          *xy[100]; /* Вы задаете размер массива */

void write_pline (SCR_file, xy, n, width, layer)
FILE            *SCR_file;
coord          xy[];
int            n;
char           *layer;
{
    int            i;
    /* Проверка текущего слоя */
    if (strcmp( layer, current_layer))

```

```

    {
        set_layer(SCR_file, layer);
        current_layer = layer;
    }

    fprintf(SCR_file, "%s %f, %f %s %f %f\n", PLINE,
            xy[0].x, xy[0].y, WIDTH, xy[0].sw, xy[0].ew);
    for(i=1; i<n; i++)
    {
        if (xy[i].sw || xy[i].ew)
            fprintf(SCR_file, "%s %f %f\n",
                    WIDTH, xy[i].sw, xy[i].ew);
        fprintf(SCR_file, "%f, %f\n", xy[i].x, xy[i].y);
    }
    fprintf(SCR_file, "\n"); /* Завершающая строка */
}

```

Лист. 7.55. Формирование команды POLYLINE для пакетного файла

```

#define BLK_WINDOW    1
#define BLK_LAST      2

void write_block (SCR_file, name, ix, iy, action, x1, y1, x2, y2)
FILE          *SCR_file;
char          *name;
double        ix;
double        iy;
int           action;
double        x1;
double        y1;
double        x2;
double        y2;
{

    fprintf(SCR_file, "%s %s %f, %f\n", BLOCK, name, ix, iy);

    /* Чтобы определить, какие объекты входят в блок, */
    /* нам надо сформировать группу выбора.          */
    /* Первым способом формирования группы является */
    /* создание окна, в которое попадает блок.      */
    /* Это наиболее подходящий способ.              */
    /* Другой способ состоит в том, чтобы выбрать  */
    /* последний объект.                             */
    /* Этот способ не является универсальным.       */

    switch(action)
    {

```

```

        case BLK_WINDOW : fprintf(SCR_file, "%s %f,%f %f,%f\n\n",
                                WINDOW,x1,y1,x2,y2);
                                break;
        case BLK_LAST    : fprintf(SCR_file, "%s\n\n", LAST);
                                break;
    }
}

```

Лист. 7.56. Формирование команды BLOCK для пакетного файла

```

#define BLK_WINDOW      1
#define BLK_LAST        2

void write_wblock (SCR_file,name,ix,iy,action,x1,y1,x2,y2)
FILE          *SCR_file;
char          *name;
double        ix;
double        iy;
int           action;
double        x1;
double        y1;
double        x2;
double        y2;
{
    fprintf(SCR_file,"%s %s\n\n%f,%f\n",WBLOCK,name,ix,iy);

    /* Чтобы определить какие объекты входят в блок. */
    /* нам надо сформировать набор выбора. */
    /* Первым способом формирования набора */
    /* является создание окна, в которое */
    /* попадает блок. Это наиболее подходящий способ. */
    /* Другой способ состоит в том, чтобы выбрать */
    /* последний объект. */
    /* Этот способ не является универсальным. */

    switch(action)
    {
        case BLK_WINDOW : fprintf(SCR_file, "%s %f,%f %f,%f\n\n",
                                WINDOW,x1,y1,x2,y2);
                                break;

        case BLK_LAST    : fprintf(SCR_file, "%s\n\n", LAST);
                                break;
    }
}

```

Лист. 7.57. Формирование команды WBLOCK для пакетного файла

Глава 8

ВЫВОД ДАННЫХ В ФОРМАТЕ DXF- И DXB-ФАЙЛОВ

Пакетные файлы не единственное средство управления работой системы AutoCAD из внешних программ. В набор команд системы AutoCAD включены команды для ввода информации из двух типов файлов обмена описаниями чертежей. В данной главе рассматривается управление системой с помощью этих файлов.

Неполная генерация DXF-файлов

Система AutoCAD при создании файлов обмена описаниями чертежей типа DXF включает в файл всю относящуюся к чертежу информацию. Это необходимо для полного описания всех деталей чертежа. Наблюдается противоположное, когда система AutoCAD производит считывание DXF-файлов, системе требуется, чтобы DXF-файлы содержали почти минимальную информацию для описания чертежа. Это освобождает нас от обязанности генерировать чрезмерное множество деталей, которые поддерживает сама система AutoCAD. Вернее, мы можем генерировать такой объем информации, какой мы считаем нужным.

Структура DXF-файлов, созданных с помощью прикладных программ, должна подчиняться некоторым правилам, применяемым при создании DXF-файлов системой AutoCAD. Если она не совпадает с некоторой общей структурой, система AutoCAD не будет знать, как ей интерпретировать ваши данные. Мы будем придерживаться структуры DXF-файлов, предназначенных для ввода, так же как мы делали и со структурой DXF-файлов, предназначенных для вывода. Обязательные поля структуры данных будут выделяться особо. Сначала давайте рассмотрим обязательные элементы структуры данных.

Вы должны соответствующим образом завершить данные в DXF-файле. Это похоже на большую подготовку к старту. На лист. 8.1 показаны последние строчки сформированного DXF-файла. Это единственные абсолютно необходимые строчки, которые должны присутствовать в DXF-файле с правильной структурой.

0
EOF

Лист. 8.1. Обозначение конца данных в DXF-файле

На лист. 8.2 представлена функция для записи признака конца DXF-файла. Эта функция настолько мала, что вы можете захотеть использовать только оператор вывода в файл (функция `fprintf`) в том месте вашей программы, где он будет нужен. Все, что находится перед признаком конца файла EOF, делает DXF-файл немного более сложным.

```
void generate_eof(dxf)
FILE      *dxf;
{
    fprintf(dxf, " 0\nEOF\n");
}

void generate_eof(dxf)
FILE      *dxf;
{
    fprintf(dxf, "%3d\nEOF\n", 0);
}
```

Лист. 8.2. Функция для записи признака конца DXF-файла

Если вам нужна последовательность операторов программы, которой вы могли бы руководствоваться при использовании функции записи признака конца DXF-файла и других функций, то на лист. 8.3 вы можете увидеть ряд вызовов различных функций для записи данных в DXF-файл.

```
dxf = fopen(dxf_name, "w");
generate_header(dxf);
generate_tables(dxf);
generate_blocks(dxf);
generate_entities(dxf);
generate_eof(dxf);
fclose(dxf);
```

Лист. 8.3. Общая последовательность действий при создании DXF-файла

Все секции DXF-файла являются необязательными. Если вы не включаете какую-либо секцию, то система AutoCAD прекрасно обходится и без этой секции. Например, если вам не требуется определять какие-либо новые установки параметров для слоев (экзотические типы линий или цвета), вы можете пропустить в секции TABLES таблицу слоев. Однако если вы в секции объектов ссылаетесь на слой, который не был определен, то система AutoCAD создаст слой и установит для него тип линии CONTINUOUS (непрерывный) и цвет WHITE (белый). Тем не менее мы рассмотрим, как создается каждая секция.

Секция заголовка HEADER

В секции заголовка **HEADER** DXF-файла содержится всякого рода информация о параметрах чертежа. Некоторые параметры полезны для использования их в программе, некоторые из этих параметров излишни. Нахождение границ чертежа, с тем чтобы вычертить его или вычислить коэффициенты масштабирования, представляется весьма привлекательным. А изменение значений единиц измерения — менее интересным.

Почему следует задавать все эти значения секции заголовка вне программы, если они совсем не применяются в программе? Приятным является то, что вы не должны генерировать какие-либо значения, которые вы не используете.

Общая структура секции **HEADER** показана на лист. 8.4. Эти данные созданы программой, приведенной, на лист. 8.5. Если в файле присутствует секция **HEADER**, то в DXF-файле обязательно должны быть некоторые записи. Поскольку секция заголовка — это самостоятельная независимая секция, эта секция должна быть открыта (так сказать) и поименована. Если секция была открыта, то она должна быть и корректно закрыта. Закрытие секции служит для системы AutoCAD сигналом приготовиться к приему очередной секции. Вспомним обсуждение конечного автомата для считывания DXF-файлов, теперь мы имеем обратную ситуацию, уже не наша программа, а система AutoCAD представляет собой ранее описанный конечный автомат.

```
0
SECTION
2
HEADER
.....
0
ENDSEC
```

Лист. 8.4. Структура секции заголовка **HEADER**

```
void generate_header(dxf)
FILE          *dxf;
{
    write_DXF(dxf,0,"SECTION");
    write_DXF(dxf,2,"HEADER");
    /* код программы для генерации секции HEADER DXF-файла. */
    write_DXF(dxf,0,"ENDSEC");
}
```

Лист. 8.5. Программа для записи в DXF-файл секции **HEADER**

Вы обратили внимание, что в процедуре генерации секции заголовка используется еще одна функция - `write_DXF`. Это очень полезная функция, которая локализует в себе операции записи в файл. Аргументами функции являются дескриптор файла, один из допустимых маркеров в DXF-файле и связанное с данным маркером значение. Для того чтобы функция могла воспринимать данные различных типов, записываемые в DXF-файл, третьим аргументом функции `write_DXF` является указатель на значение. Такая мера позволяет оставлять неизменным размер аргумента. Как можно увидеть в программе при объявлении аргумента, аргумент имеет тип `void` (пустой тип). Это дает нам некоторую гибкость при использовании данного указателя. Перед использованием этого указателя, невзирая на его универсальность, мы должны выполнить операцию приведения типа указателя к заданному типу данных. А где мы должны получить этот тип? Маркер DXF-файла сообщает нам о том, какого типа должна быть ссылка на заданное в функции значение. Маркер определяет тип данных. Вспомнили обсуждавшийся в гл. 2 вопрос о том, как некоторые маркеры определяют тип связанных с ними данных? В функции `write_DXF` применяется такое же соотношение. Тщательно проанализированное на тип данных с помощью каскада управляющих конструкций `if-then-else` значение маркера определяет тип указателя. Соответствующим образом переадресованное значение оказывается записанным в DXF-файл после сопутствующего ему маркера. На лист. 8.6 приведен исходный текст функции `write_DXF`.

```
void write_DXF(dxf, tag, value)
FILE          *dxf;
int           tag;
void          *value;
{
    fprintf(dxf, "%3d\n", tag);
    if (tag <= 9)
        fprintf(dxf, "%s\n", (char *)value);
    else
        if (tag <= 59)
            fprintf(dxf, "%f\n", *(double *)value);
        else
            fprintf(dxf, "%d\n", *(int *)value);
}
```

Лист. 8.6. Функция записи в DXF-файл `write_DXF`

В программе, спецификации формата вывода, указанном в первом операторе вывода в файл, используется обозначение `%3d`. В DXF-файле маркер появляется в том же виде, как и при выводе целого числа по формату I3 Фортрана. Формат вывода, указанный в тексте программы, соответствует формату Фортрана. Хотя такой формат на самом деле не требуется для системы AutoCAD, другие прикладные программы могут ориентироваться именно на этот формат.

В гл. 5 мы применяли конечный автомат и процедуру распознавания лексем, чтобы определить значения в секции HEADER. Указатели на полученные значения помещают значения в предназначенные для их хранения переменные. В данной главе мы будем использовать еще один метод, связанный с DXF-файлами. Цель нашего обсуждения — помочь при создании DXF-файлов, но рассматриваемые структуры данных также могут быть использованы и при синтаксическом разборе DXF-файлов. Поскольку нам не потребуется полная секция заголовка, мы записываем в файл только имена и значения, найденные в таблице. Таблица, полностью приведенная в приложении Б, содержит следующие поля:

- 1) имя секции HEADER системы AutoCAD;
- 2) тип данных значения;
- 3) указатель на переменную, принимающую (содержащую) значение.

На лист. 8.7 показана структура данных для типов, используемая для этой таблицы, и несколько строк самой таблицы. Я, конечно же, не хотел бы, чтобы для того, чтобы держать эту структуру у себя перед глазами, вы перелистывали страницы книги вперед и назад.

```
#define COORDX      1
#define COORDY      2
#define COORDZ      3
#define ANGLE       4
#define INTEGER     5
#define STRING      6
#define LAYER       7

typedef
struct
{
    char          *name;
    int           type;
    void          *value;
}                header_entry;

double          extminx;
double          extminy;
double          extmaxx;
double          extmaxy;

header_entry     header_table[] = { {"$EXTMIN", COORDX, &extminx},
                                     { NULL,      COORDY, &extminy},
                                     {"$EXTMAX", COORDX, &extmaxx},
                                     { NULL,      COORDY, &extmaxy},
                                     { NULL,      0,  NULL} };
```

Лист. 8.7. Определения для таблицы заголовка

Занесение значений заголовка с помощью этой таблицы в действительности легче, чем синтаксический разбор DXF-файла. Вся нужная нам информация уже содержится в таблице. Необходимо только передать эту информацию функции `write_DXF`, чтобы записать ее на диск. На лист. 8.8 приведен алгоритм процедуры для записи секции **HEADER** в файл. На лист. 8.9 приведена реализация этого алгоритма.

Записать запись **SECTION**

Записать запись **HEADER**

Для каждого элемента таблицы заголовка

Записать имя элемента таблицы с маркером 9

В зависимости от типа данных:

COORD (координаты)	записать координаты X, Y и Z с соответствующими маркерами
INTEGER (целое)	записать значение с маркером 70
ANGLE (угол)	записать значение с маркером 50
STRING (строка)	записать значение с маркером 1
LAYER (слой)	
и т.д.	

Записать запись **ENDSEC**

Лист. 8.8. Алгоритм генерации секции **HEADER**

```

generate_header(dxf)
FILE      *dxf;
{
    header_entry      *h;

    write_DXF(dxf,0,"SECTION");
    write_DXF(dxf,2,"HEADER");
    h = header_table;
    while (h->value)
    {
        if (h->name) write_DXF(dxf,9,h->name);
        switch(h->type)
        {
            case COORDX : write_DXF(dxf,10,h->value);
                          break;
            case COORDY : write_DXF(dxf,20,h->value);
                          break;
            case COORDZ : write_DXF(dxf,30,h->value);
                          break;
            case ANGLE   : write_DXF(dxf,50,h->value);
                          break;
            case INTEGER : write_DXF(dxf,70,h->value);
        }
    }
}

```

```

        break;
    case STRING : write_DXF(dxf,1,h->value);
        break;
    case LAYER  : write_DXF(dxf,8,h->value);
    }
    h++;
}
write_DXF(dxf,0,"ENDSEC");
}

```

Лист. 8.9. Окончательный вариант функции generate_header для записи в DXF-файл секции **HEADER**

Несомненно, в наших возможностях сгенерировать секцию заголовка для DXF-файла более жестким образом. Если вы захотели бы записать только несколько элементов таблицы, то достаточно трудно рационализировать конструкцию таблицы драйвера, вы могли бы разработать функцию, которая методично записывала каждое поле. На лист. 8.10 приведена такая функция. Данная функция осуществляет запись вычисленных границ области изображения и границ чертежа. В этом параграфе были рассмотрены способы генерации секции **HEADER** для DXF-файлов. Чтобы получить больше информации о структуре таблицы заголовка, обратитесь к гл. 5. Полное описание таблицы заголовка можно найти в приложении Б.

```

double      limmin_x;
double      limmin_y;
double      limmax_x;
double      limmax_y;
double      extmin_x;
double      extmin_y;
double      extmax_x;
double      extmax_y;

void generate_header(dxf)
FILE      *dxf;
{
    write_DXF(dxf,0,"SECTION");
    write_DXF(dxf,2,"HEADER");
    write_DXF(dxf,9,"$LIMMIN");
    write_DXF(dxf,10,&limmin_x);
    write_DXF(dxf,20,&limmin_y);
    write_DXF(dxf,9,"$LIMMAX");
    write_DXF(dxf,10,&limmax_x);
    write_DXF(dxf,20,&limmax_y);
    write_DXF(dxf,9,"$EXTMIN");
    write_DXF(dxf,10,&extmin_x);

```

```

write_DXF(dxf, 20, &extmin_y);
write_DXF(dxf, 9, "$EXTMAX");
write_DXF(dxf, 10, &extmax_x);
write_DXF(dxf, 20, &extmax_y);
write_DXF(dxf, 0, "ENDSEC");
}

```

Лист. 8.10. Генерация отдельных частей секции заголовка

Секция таблиц TABLES

Как и в случае с секцией HEADER, секция TABLES может быть сгенерирована, если только она вам нужна. Если вам не требуется что-либо необычное, вы можете пропустить секцию TABLES.

Если вы используете шрифт иной, чем типа STANDARD

Если вы используете тип линии иной, чем CONTINUOUS

Если вы создаете слои с различными цветами,

Если вы конструируете виды,

то вы должны генерировать секцию таблиц TABLES.

Функция верхнего уровня для генерации секции TABLES имеет чисто административное назначение. В приведенном примере (лист. 8.11) показано, как выполняется запись всех таблиц. Итак, вы должны показать запись таблиц в примере, но в вашей прикладной программе они не обязательно должны присутствовать. Если вы записываете в DXF-файл секцию таблиц TABLES, то обязательно в файле должны быть такие записи, как SECTION, TABLES и ENDSEC. Порядок следования различных таблиц в файле несколько критичен. Замечательная иллюстрация к этому: вы должны определить типы линий перед тем, как могут появиться ссылки на эти типы линий. Поэтому функция генерации таблицы типов линий должна выполняться в программе раньше, чем функция генерации таблицы слоев.

```

generate_tables(dxf)
FILE          *dxf;
{
    write_DXF(dxf, 0, "SECTION");
    write_DXF(dxf, 2, "TABLES");
    generate_styles(dxf);
    generate_linetypes(dxf);
    generate_layers(dxf);
    generate_views(dxf);
    write_DXF(dxf, 0, "ENDSEC");
}

```

Лист. 8.11. Центральное управление генерацией таблиц

Все функции генерации таблиц должны выполнять специальную последовательность операций записи для идентификации конкретной таблицы. Так же, как каждая секция в DXF-файле должна иметь соответствующее оформление начала и конца секции, в таблицах должно быть сделано то же самое. На лист. 8.12 показано, как выглядит подобное оформление таблицы.

```

0
TABLE
2
имя таблицы
70
максимальное число элементов в таблице
.....
0
ENDTAB

```

Лист. 8.12. Обязательные записи в таблице

Обратите внимание, что после маркера 2 следует значение TABLE (таблица — единственное число). Имя секции таблиц — TABLES (таблицы — множественное число). Максимальное число допустимых в таблице элементов указывается после маркера 70, которому предшествует имя таблицы. Указание числа элементов в таблице не является необходимым, но оно должно быть больше или равно числу элементов таблицы в вашем DXF-файле. Однако вы можете системе AutoCAD и сэкономить память, если установите это значение в точном соответствии с числом элементов в вашей таблице.

Подсчитать число записей в связанном списке

Если список не пуст

Записать запись TABLE (маркер 0)

Записать тип таблицы (маркер 2)

Записать общее число записей (маркер 70)

Для каждого элемента таблицы

Записать тип таблицы (маркер 0)

Записать имя таблицы (маркер 2)

Записать специальные флажки (маркер 70)

Записать специфические значения таблицы

Записать запись ENDTAB (маркер 0)

Лист. 8.13. Алгоритм генерации секции HEADER

В примерах, приведенных для данного параграфа, я исходил из того, что таблицы в вашей программе по своей структуре подобны структурам данных, рассмотренным в гл. 5. В этих структурах данных имеются ссылки на каждую таблицу, с помощью которых образуется связный список. На лист. 8.13 приведен рабочий алгоритм. Этот алгоритм не требует каких-либо дополнительных пояснений.

На последующих листингах показана реализация данного алгоритма для таблиц всех типов. В табл. 8.1 перечислены типы таблиц и номера листингов соответствующих функций генерации таблиц.

Таблица 8.1. Список подпрограмм для генерации таблиц DXF-файла

LINETYPE	Лист. 8.14
LAYER	Лист. 8.15
STYLE	Лист. 8.16
VIEW	Лист. 8.17

```
typedef
struct      ln
{
    struct ln      *next;
    char           flags;
    char           name[32];
    char           describe[48];
    unsigned char   align;
    unsigned char   dash_count;
    double          pattern_length;
    double          dashes[16];
}
line_node      *line_anchor;

void generate_ltypes(dxf)
FILE      *dxf;
{
    int      f;
    int      k;
    int      cnt;
    line_node *l;

    if ((l = line_anchor) != NULL)
    {
        cnt = 0;
        while (l)
        {
            cnt++;
            l = l->next;
        }
    }
    write_DXF(dxf, 0, "TABLE");
    write_DXF(dxf, 2, "LTYPE");
    write_DXF(dxf, 70, &cnt);
    l = line_anchor;
```

```

while (l)
{
    write_DXF(dxf,0,"LTYPE");
    write_DXF(dxf,2,l->name);
    l->flags = 64;
    write_DXF(dxf,70,&l->flags);
    write_DXF(dxf,3,&l->describe);
    write_DXF(dxf,72,&l->align);
    write_DXF(dxf,73,&l->dash_count);
    write_DXF(dxf,40,&l->pattern_length);
    for (k=0;k<l->dash_count;k++)
        write_DXF(dxf,49,&l->dashes[k]);
    l = l->next;
}
write_DXF(dxf,0,"ENDTAB");
}

```

Лист. 8.14. Запись таблицы типов линий **LINETYPE** в DXF-файл

```

typedef
struct layer_node
{
    struct layer_node    *next;
    char                 *name;
    int                  flags;
    int                  color;
    char                 *linetype;
    layer_node           layer_node;
}

layer_node              *layer_anchor;

void generate_layers(dxf)
FILE                    *dxf;
{
    layer_node           *l;
    int                  cnt;

    if ((l = layer_anchor) != NULL)
    {
        cnt = 0;
        while (l)
        {
            cnt++;
            l = l->next;
        }
        write_DXF(dxf,0,"TABLE");
    }
}

```

```

write_DXF(dxf, 2, "LAYER");
write_DXF(dxf, 70, &cnt);
l = layer_anchor;
while (l)
{
    write_DXF(dxf, 0, "LAYER");
    write_DXF(dxf, 2, l->name);
    write_DXF(dxf, 70, &l->flags);
    write_DXF(dxf, 62, &l->flags);
    write_DXF(dxf, 6, l->linetype);
    l = l->next;
}
write_DXF(dxf, 0, "ENDTAB");
}
}

```

Лист. 8.15. Запись таблицы слоев **LAYER** в DXF-файл

```

typedef
struct      st
{
    struct st *next;
    shape_entry *shape_index;
    char      flags;
    char      name[32];
    double    txt_height;
    double    txt_scale;
    double    oblique;
    char      gen_flags;
    double    last_height;
    char      font[32];
    char      bigfont[32];
}            style_node;

style_node  *sty_anchor;

void generate_styles(dxf)
FILE        *dxf;
{
    int      i;
    style_node  *sty;

    if ((sty = sty_anchor) != NULL)
    {
        cnt = 0;
        while (sty)

```

```

    {
        cnt++;
        sty = sty->next;
    }
}
write_DXF(dxf,0,"TABLE");
write_DXF(dxf,2,"STYLE");
write_DXF(dxf,70,&cnt);
sty = sty_anchor;
while (cty)
{
    write_DXF(dxf,0,"STYLE");
    write_DXF(dxf,2,sty->name);
    sty->flags = 64;
    write_DXF(dxf,70,&sty->flags);
    write_DXF(dxf,40,&sty->txt_height);
    write_DXF(dxf,41,&sty->txt_scale);
    write_DXF(dxf,50,&sty->oblique);
    write_DXF(dxf,71,&sty->gen_flags);
    write_DXF(dxf,42,&sty->last_height);
    write_DXF(dxf,3,&sty->font);
    write_DXF(dxf,4,sty->bigfont);
    sty = sty->next;
}
write_DXF(dxf,0,"ENDTAB");
}

```

Лист. 8.16. Запись таблицы стилей **STYLE** в DXF-файл

```

typedef
struct          vw
{
    struct vw *next;
    char      flags;
    char      name[32];
    double     vw_height;
    double     vw_ctr_x;
    double     vw_ctr_y;
    double     vw_width;
    double     vw_dir_x;
    double     vw_dir_y;
    double     vw_dir_z;
}              view_node;

view_node      *vw_anchor;

write_views(dxf)
FILE *dxf

```



```

{
    int      i;
    view_node *v;

    if ((v = vw_anchor) != NULL)
    {
        cnt = 0;
        while (v)
        {
            cnt++;
            v = v->next;
        }

        write_DXF(dxf, 0, "TABLE");
        write_DXF(dxf, 2, "VIEW");
        write_DXF(dxf, 70, &cnt);
        v = vw_anchor;
        while (v)
        {
            write_DXF(dxf, 0, "VIEW");
            write_DXF(dxf, 2, v->name);
            write_DXF(dxf, 70, &v->flags);
            write_DXF(dxf, 40, &v->vw_height);
            write_DXF(dxf, 10, &v->vw_ctr_x);
            write_DXF(dxf, 20, &v->vw_ctr_y);
            write_DXF(dxf, 41, &v->vw_width);
            write_DXF(dxf, 11, &v->vw_dir_x);
            write_DXF(dxf, 21, &v->vw_dir_y);
            write_DXF(dxf, 31, &v->vw_dir_z);
            v = v->next;
        }
        write_DXF(dxf, 0, "ENDTAB");
    }
}

```

Лист. 8.17. Запись таблицы видов **VIEW** в DXF-файл

Генерация секции блоков BLOCKS

Описания графических объектов и блоков в DXF-файлах по своей структуре весьма схожи. Единственная разница заключается лишь в объявлении блоков. Структура же всей графической информации абсолютно та же самая. Помня об этом, я ненадолго отложу обсуждение того, как генерируются графические элементы блока. Мы рассмотрим генерацию объектов для DXF-файла при разборе генерации секции объектов ENTITIES. Секция блоков, как и все другие секции, должна иметь начало секции и конец. Внутри секции блоков

каждый блок имеет свою идентификацию и определение. В свою очередь, каждый блок имеет свое начало и конец. На лист. 8.18 показан внешний вид секции блоков BLOCKS. Каждый отдельный блок заключается между инструкциями BLOCK и ENDBLK. Им будет уделено внимание в конце обсуждения генерации объектов.

```

0
SECTION
2
BLOCKS
0
BLOCK
2
Имя блока
....
0
ENDBLK
Другие блоки
0
ENDSEC

```

Лист. 8.18. Структура секции блоков BLOCKS

По общему согласию я отложил рассмотрение вывода информации об объектах до следующего параграфа. В нем мы найдем программы для записи в DXF-файл каждого из разнообразных графических объектов. Структура определения блока включает указатель на цепочку объектов. Предполагается, что такой список был создан ранее, скорее всего, при считывании другого DXF-файла или при синтезировании такой структуры данных в программе.

Секция объектов ENTITIES

Построение объектов для записи в DXF-файл является процедурой, обратной процедуре разбора файла. Имея данные, составляющие описание объекта, задачей подпрограмм вывода в файл является порождение соответствующих пар маркер — значение. В оставшейся части вашей программы, посвященной формированию специфических графических объектов (я не говорю о том, как их создать, — это ваша задача), при рассмотрении секции вывода программы внимание уделяется методам вывода информации.

В данном параграфе речь пойдет о создании объектов в том виде, в котором они записываются в формате DXF-файла. Я назвал программы, приведенные на следующих листингах, теми же именами, как и в главе, посвященной пакетным файлам системы AutoCAD. Это демонстрирует, насколько гибко вы можете построить ваши программы с некоторыми стандартными именами. Просто заменяя ссылку на объектный модуль, содержащуюся в файле, предназначенном

для компоновщика, вы могли бы изменить форму вывода в вашей программе на стадии компоновки.

На самом деле вы могли бы переписать эти подпрограммы, чтобы, используя глобальные переменные, управлять формой вывода. С помощью управляющих конструкций типа switch/case и if-then-else вы могли бы скомбинировать операции вывода для пакетного файла в одной подпрограмме. Устанавливая по-разному значение переменной в зависимости от значения аргумента в командной строке, для вывода данных в том или ином формате можно организовать в конкретных подпрограммах передачу управления на соответствующие операторы вывода.

Как и во всех других секциях DXF-файла, внешнее представление секции объектов имеет вложенную структуру. На лист. 8.19 показано, как выглядит эта секция в DXF-файле. На листинге показаны обязательные строки для отдельных объектов. На лист. 8.20 приведен фрагмент текста программы для вывода этих строк. Обратите внимание, что в этом фрагменте применяется повсеместно используемая процедура write_DXF.

```
0
SECTION
2
ENTITIES
:
:
0
ENDSEC
```

Лист. 8.19. Структура секции объектов ENTITIES

```
write_DXF(dxf.0, "SECTION");
write_DXF(dxf.2, "ENTITIES");
/* в этом месте записывается информация о самих объектах */
write_DXF(dxf.0, "ENDSEC");
```

Лист. 8.20. Генерация структуры секции ENTITIES

Каждый объект в DXF-файле имеет свою отличную от других объектов структуру. Это интуитивно понятно, поскольку такие объекты, как линия и текст, весьма различаются между собой. Но тем не менее в описании объектов в DXF-файле имеются некоторые общие сходные между собой компоненты структуры. Два из них появляются во всех описаниях объектов, но также имеются и несколько необязательных.

Первым и наиболее важным компонентом является идентификация объекта. Идентификация объекта вводится с помощью маркера 0 и следующего за ним имени объекта. На лист. 8.21 показан фраг-

мент DXF-файла, в котором идентифицирован объект линия. Обратитесь к таблицам в гл.5, где перечислены имена других объектов. Используя функцию `write_DXF`, вы можете реализовать вывод, идентифицирующей последовательности для DXF-файла в одной строке вашей программы. На лист. 8.22 показано, как это может быть сделано.

```
0  
LINE
```

Лист. 8.21. Идентификация объекта линия

```
write_DXF(dxf,0,"LINE");
```

Лист. 8.22. Генерация идентифицирующей записи

Другим компонентом структуры секции объектов, который обязательно присутствует в описании каждого объекта, является имя слоя. В формате пакетных файлов и DXB-файлов, необходимо изменять слой всякий раз, когда в процессе выполнения программы наступает подходящий момент сделать это. В этом месте в подпрограммах вывода осуществляется сравнение слоя объекта с активным текущим слоем. Ничего не надо делать, если слои совпадают. Но, если это разные слои, вы должны задать командную последовательность, чтобы изменить слой.

Напротив, в DXF-файлах нет активных слоев. Каждый объект находится на каком-то определенном слое. И этот слой указывается в описании каждого объекта. Слой задается с помощью маркера 0 и следующего за ним имени слоя. На лист. 8.23 показано, как объявляется слой в DXF-файле. На лист. 8.24 приведена строка программы для генерации этого фрагмента DXF-файла. Вспомним, что имя слоя является одним из аргументов нашей стандартной подпрограммы вывода.

```
8  
LAYER_1
```

Лист. 8.23. Определение слоя в DXF-файле

```
write_DXF(dxf,0,layer);
```

Лист. 8.24. Генерация определения слоя

Имеются четыре других, входящих в структуру компонента, которые вы можете добавлять к описанию ваших объектов. Эти компоненты представляют собой модификаторы объектов и могут быть не-

приемлемы для некоторых объектов. Ниже перечислены возможные модификаторы:

COLOR	(цвет);
LINETYPE	(тип линии);
ELEVATION	(уровень возвышения);
THICKNESS	(высота (толщина))

Модификатор цвета позволяет вам изменять цвет, заданный по умолчанию для слоя, на котором находится объект. Модификатору цвета предшествует маркер 62. Значением модификатора может быть любое число от 0 до 255. В случае использования значения 0 могут быть получены непредсказуемые результаты. Кроме того, цвет 0 имеет применение в других случаях. Значение 255 устанавливает цвет в соответствии с заданным цветом для блока или цветом слоя для объекта. На лист. 8.25 показано, как выглядит в DXF-файле модификатор цвета. На лист. 8.26 приведена строка программы, в которой осуществляется генерация указанной последовательности в DXF-файле.

62

1

Лист. 8.25. Модификатор цвета в DXF-файле (1 - красный)

Так как модификаторы объекта не являются аргументами нашей стандартной подпрограммы вывода, вам необходимо будет ввести в программу некоторый аппарат для передачи этих данных подпрограмме вывода. Наиболее просто сделать это — добавить аргументы в список параметров подпрограммы вывода. Значение 0 модификатора сообщает, что цвет не изменяется. В последующих листингах предполагается, что в программе присутствует аргумент для передачи модификатора цвета. Ниже приведен пример использования модификатора цвета 0, означающего, что изменение цвета не происходит (лист. 8.26).

```
if (color) write_DXF(dxf,62,&color);
```

Лист. 8.26. Генерация модификатора цвета

Модификатор типа линии позволяет изменять заданный по умолчанию тип линии для слоя. Модификатору типа линии предшествует маркер 6. Значением модификатора является имя типа линии, установленное в таблице типов линий DXF-файла. На лист. 8.27 показано представление модификатора типа линии в DXF-файле. На лист. 8.28 приведен исходный текст программы для генерации данного модификатора.

Лист. 8.27. Модификатор типа линии в DXF-файле

```
if (linetype) write_DXF(dxf,6,linetype);
```

Лист. 8.28. Генерация модификатора типа линии

Модификатор уровня возвышения позволяет задавать для некоторых объектов координату Z. Данному модификатору соответствует маркер 38. Значением модификатора является число с плавающей точкой, определяющее координату Z объекта. На лист. 8.29 показано представление модификатора уровня возвышения в DXF-файле. На лист. 8.30 приведен исходный текст программы для генерации этого модификатора.

38
2.125

Лист. 8.29. Модификатор уровня возвышения в DXF-файле

```
if (elevation) write_DXF(dxf,38,&elevation);
```

Лист. 8.30. Генерация модификатора уровня возвышения

Модификатор высоты (толщины) позволяет вам задавать координату Z для объемного утолщения поверхности, используя объект в качестве базы. Модификатору высоты предшествует маркер 39. Значением модификатора является число с плавающей точкой, представляющее относительное смещение по оси Z утолщаемой части объекта. На лист. 8.31 показано представление модификатора высоты в DXF-файле. На лист. 8.32 приведен исходный текст программы для генерации данного модификатора.

39
12.345

Лист. 8.31. Модификатор высоты в DXF-файле

```
if (thickness) write_DXF(dxf,39,thickness);
```

Лист. 8.32. Генерация модификатора высоты

Оставшаяся часть генерации DXF-файла осуществляется довольно просто. Обратимся к таблицам гл. 5, чтобы получить маркеры и связанные с ними значения для каждого объекта. Если вы не применяете какие-либо другие значения, отличающиеся от значений, задан-

ных по умолчанию, то нет необходимости вводить эти значения в описания объектов.

В табл. 8.2 приведен список листингов подпрограмм вывода в DXF-файл. Вы можете заменить аргументы функций, описывающие индивидуальные параметры, на единую структуру данных. Вы можете добавить дополнительные аргументы к функциям, чтобы передавать значения модификаторов. Эти подпрограммы представляют отправную точку для дальнейшей разработки.

Таблица 8.2. Список подпрограмм вывода в DXF-файл

LINE	Лист. 8.33
POINT	Лист. 8.34
CIRCLE	Лист. 8.35
ARC	Лист. 8.36
TRACE	Лист. 8.37
SOLID	Лист. 8.38
TEXT	Лист. 8.39
SHAPE	Лист. 8.40
INSERT	Лист. 8.41
ATTRIB	Лист. 8.42
ATTDEF	Лист. 8.43
POLYLINE	Лист. 8.44
VERTEX	Лист. 8.45
SEQEND	Лист. 8.46

```
void write_line (DXF_file, from_x, from_y, to_x, to_y, layer)
FILE             *DXF_file;
double           from_x;
double           from_y;
double           to_x;
double           to_y;
char             *layer;
{
    write_dxf(DXF_file, 0, "LINE");
    write_dxf(DXF_file, 8, layer);
    write_dxf(DXF_file, 10, &from_x);
    write_dxf(DXF_file, 20, &from_y);
    write_dxf(DXF_file, 11, &to_x);
    write_dxf(DXF_file, 21, &to_y);
}
```

Лист. 8.33. Запись в DXF-файл описания объекта **LINE**

```
void write_point (DXF_file, x, y, layer)
FILE             *DXF_file;
double           x;
```

```

double      y;
char        *layer;
{
    write_dxf(DXF_file,0,"POINT");
    write_dxf(DXF_file,8,layer);
    write_dxf(DXF_file,10,&x);
    write_dxf(DXF_file,20,&y);
}

```

Лист. 8.34. Запись в DXF-файл описания объекта POINT

```

void write_circle (DXF_file,x,y,r,layer)
FILE      *DXF_file;
double     x;
double     y;
double     r;
char       *layer;
{
    write_dxf(DXF_file,0,"CIRCLE");
    write_dxf(DXF_file,8,layer);
    write_dxf(DXF_file,10,&x);
    write_dxf(DXF_file,20,&y);
    write_dxf(DXF_file,40,&r);
}

```

Лист. 8.35. Запись в DXF-файл описания объекта CIRCLE

```

void write_arc (DXF_file,cx,cy,r,sa,ea,layer)
FILE      *DXF_file;
double     cx;
double     cy;
double     r;
double     sa;
double     ea;
char       *layer;
{
    write_dxf(DXF_file,0,"ARC");
    write_dxf(DXF_file,8,layer);
    write_dxf(DXF_file,10,&cx);
    write_dxf(DXF_file,20,&cy);
    write_dxf(DXF_file,40,&r);
    write_dxf(DXF_file,50,&sa);
    write_dxf(DXF_file,51,&ea);
}

```

Лист. 8.36. Запись в DXF-файл описания объекта ARC


```

void write_trace (DXF_file,x1,y1,x2,y2,x3,y3,x4,y4,layer)
FILE             *DXF_file;
double           x1;
double           y1;
double           x2;
double           y2;
double           x3;
double           y3;
double           x4;
double           y4;
double           layer;
char             *layer;
{
    /* Во внутреннем представлении системы AutoCAD */
    /* объекты TRACE выглядят подобно объектам SOLID. */
    /* Четыре точки, заданные в качестве аргументов */
    /* функции, определяют четыре угла области. */
    /* занимаемой объектом TRACE. В пакетном файле */
    /* указываются линия центра полосы и ее ширина -- */
    /* эти параметры преобразуются в параметры. */
    /* описывающие получаемую область. */
    /*           x1,y1                               x3,y3 */
    /*           /-----/                          */
    /*           /-----/ линия центра -----/    */
    /*           /-----/                          */
    /*           x2,y2                               x4,y4 */
    /*           /-----/                          */

    write_dxf(DXF_file,0,"TRACE");
    write_dxf(DXF_file,8,layer);
    write_dxf(DXF_file,10,&x1);
    write_dxf(DXF_file,20,&y1);
    write_dxf(DXF_file,11,&x2);
    write_dxf(DXF_file,21,&y2);
    write_dxf(DXF_file,12,&x3);
    write_dxf(DXF_file,22,&y3);
    write_dxf(DXF_file,13,&x4);
    write_dxf(DXF_file,23,&y4);
}

```

Лист. 8.37. Запись в DXF-файл описания объекта TRACE

```

void write_solid (DXF_file,x1,y1,x2,y2,x3,y3,x4,y4,layer)
FILE             *DXF_file;
double           x1;
double           y1;
double           x2;

```

```

double      y2;
double      x3;
double      y3;
double      x4;
double      y4;
char        *layer;
{
    write_dxf(DFX_file,0,"SOLID");
    write_dxf(DFX_file,8,layer);
    write_dxf(DFX_file,10,&x1);
    write_dxf(DFX_file,20,&y1);
    write_dxf(DFX_file,11,&x2);
    write_dxf(DFX_file,21,&y2);
    write_dxf(DFX_file,12,&x3);
    write_dxf(DFX_file,22,&y3);
    write_dxf(DFX_file,13,&x4);
    write_dxf(DFX_file,23,&y4);
}

```

Лист. 8.38. Запись в DXF-файл описания объекта SOLID

```

void write_text (DXF_file,justify,x1,y1,xscale,
                ht,angle,oblique,txt,style,layer)
FILE          *DXF_file;
double        x1;
double        y1;
double        xscale;    /* степень сжатия/растяжения */
double        ht;
double        angle;
double        oblique;   /* угол наклона литер */
char          *txt;
char          *style;
char          *layer;
{
    /* В формате DXF-файла для объекта TEXT требуются */
    /* некоторые параметры, которые отсутствуют в команде */
    /* TEXT в пакетном файле. Это несколько изменяет */
    /* список аргументов функции. */

    write_DXF(DXF_file,0,"TEXT");
    write_DXF(DXF_file,8,layer);
    write_DXF(DXF_file,10,&x1);
    write_DXF(DXF_file,20,&y1);
    write_DXF(DXF_file,40,&ht);
    write_DXF(DXF_file,1,txt);
}

```

```

    if (angle) write_DXF(DXF_file,50,&angle);
    if (x2 != 1.0) write_DXF(DXF_file,50,&xscale);
    if (oblique) write_DXF(DXF_file,51,oblique);
    if (style) write_DXF(DXF_file,7,style);
}

```

Лист. 8.39. Запись в DXF-файл описания объекта TEXT

```

void write_shape (DXF_file,shape,x1,y1,ht,angle,xscale,
                 oblique,layer)
FILE             *DXF_file;
char             *shape;
double          x1;
double          y1;
double          ht;
double          angle;
double          xscale;
double          oblique;
char            *layer;
{
    write_DXF(DXF_file,0,"SHAPE");
    write_DXF(DXF_file,8,layer);
    write_DXF(DXF_file,10,&x1);
    write_DXF(DXF_file,20,&y1);
    write_DXF(DXF_file,40,&ht);
    write_DXF(DXF_file,2,shape);
    if (angle) write_DXF(DXF_file,50,&angle);
    if (xscale != 1.0) write_DXF(DXF_file,41,xscale);
    if (oblique) write_DXF(DXF_file,51,oblique);
}

```

Лист. 8.40. Запись в DXF-файл описания объекта SHAPE

```

void write_insert (DXF_file,name,x1,y1,xscale,yscale,angle,
                  attribs,layer)
FILE             *DXF_file;
char             *name;
double          x1;
double          y1;
double          xscale;
double          yscale;
double          angle;
char            *layer;
{
    ini          i;

```

```

i = 1;
write_DXF(DXF_file,0,"INSERT");
write_DXF(DXF_file,8,layer);
/*
/* Если после инструкции INSERT следует информация об */
/* атрибутах, то вы должны объявить о наличии сложного */
/* объекта, записав маркер 66 в начале и завершив */
/* последовательность инструкцией SEQEND. */
/*
if (attribs) write_DXF(DXF_file,66,&i);
write_DXF(DXF_file,2,name);
write_DXF(DXF_file,10,&x1);
write_DXF(DXF_file,20,&y1);
if (xscale != 1.0) write_DXF(DXF_file,41,&xscale);
if (yscale != 1.0) write_DXF(DXF_file,42,&yscale);
if (angle) write_DXF(DXF_file,50,&angle);
}

```

Лист. 8.41. Запись в DXF-файл инструкции INSERT

```

void write_attrib (DXF_file,justify,x1,y1,xscale,ht,
                  angle,oblique,txt,name,style,layer)
FILE             *DXF_file;
double           x1;
double           y1;
double           xscale;    /* степень сжатия/растяжения */
double           ht;
double           angle;
double           oblique;   /* угол наклона литер */
char             *txt;      /* значение атрибута */
char             *name;     /* имя атрибута */
char             *style;
char             *layer;
{
    /* В формате DXF-файла для инструкции ATTRIB требуются */
    /* некоторые параметры, которые отсутствуют в команде */
    /* ATTRIB в пакетном файле. Это несколько изменяет */
    /* список аргументов функции. */
    write_DXF(DXF_file,0,"ATTRIB");
    write_DXF(DXF_file,8,layer);
    write_DXF(DXF_file,10,&x1);
    write_DXF(DXF_file,20,&y1);
    write_DXF(DXF_file,40,&ht);
    write_DXF(DXF_file,1,txt);
    write_DXF(DXF_file,2,name);
    if (angle) write_DXF(DXF_file,50,&angle);
}

```

```

    if (x2 != 1.0) write_DXF(DXF_file,41,&xscale);
    if (oblique) write_DXF(DXF_file,51,&oblique);
    if (style) write_DXF(DXF_file,7,style);
}

```

Лист. 8.42. Запись в DXF-файл инструкции ATTRIB

```

void write_attrdef (DXF_file,justify,x1,y1,xscale,ht,flags,
                    angle,oblique,tst,name,prompt,style,layer),
FILE                *DXF_file;
double              x1;
double              y1;
double              xscale;    /* степень сжатия/растяжения */
double              ht;
int                 flags;
double              angle;
double              oblique;   /* угол наклона литер */
char                *tst;      /* значение атрибута */
                                /* по умолчанию */
char                *name;     /* имя атрибута */
char                *prompt;   /* приглашение к вводу */
char                *style;
char                *layer;
{
    /* В формате DXF-файла для инструкции ATTDEF требуются */
    /* некоторые параметры, которые отсутствуют в команде */
    /* ATTDEF в пакетном файле. Это несколько изменяет */
    /* список аргументов функции. */
    write_DXF(DXF_file,0,"ATTDEF");
    write_DXF(DXF_file,8,layer);
    write_DXF(DXF_file,10,&x1);
    write_DXF(DXF_file,20,&y1);
    write_DXF(DXF_file,40,&ht);
    write_DXF(DXF_file,1,tst);
    write_DXF(DXF_file,3,prompt);
    write_DXF(DXF_file,2,name);

    /* Флажки атрибутов ус: навливаются значениями битов: */
    /* 1 (бит 0) текст невидим */
    /* 2 (бит 1) постоянное (неизменяемое) значение */
    /* 4 (бит 2) проверка ввода */

    write_DXF(DXF_file,70,&flags);
    if (angle) write_DXF(DXF_file,50,&angle);
}

```

```

    if (x2 != 1.0) write_DXF(DXF_file,41,&xscale);
    if (oblique) write_DXF(DXF_file,51,&oblique);
    if (style) write_DXF(DXF_file,7,style);
}

```

Лист. 8.43. Запись в DXF-файл инструкции ATTDEF

```

void write_pline (DXF_file.flags,width1,width2,layer)
FILE             *DXF_file;
int              flags;
double           width1;
double           width2;
char             *layer;
{
    int           i;

    write_DXF(DXF_file,0,"POLYLINE");
    write_DXF(DXF_file,8,layer);

    /* Флажки полилинии устанавливаются значениями бит: */
    /* 1 (бит 0) полилиния замкнута                        */
    /* 2 (бит 1) включено описание дуговых сегментов      */

    write_DXF(DXF_file,70,&flags);

    /* Полилиния является сложным объектом, описание */
    /* которого должно завершаться инструкцией SEQEND. */

    i = 1;
    write_DXF(DXF_file,66,&i);
    write_DXF(DXF_file,40,&width1);
    write_DXF(DXF_file,41,&width2);
}

```

Лист. 8.44. Запись в DXF-файл описания объекта POLYLINE

```

void write_vertex(DXF_file,x,y,width1,width2,bulge,layer)
FILE             *DXF_file;
double           x;
double           y;
double           width1;
double           width2;
double           bulge;
char             *layer;
{

```

```

write_DXF(DXF_file,0,"VERTEX");
write_DXF(DXF_file,8,layer);
write_DXF(DXF_file,10,&x);
write_DXF(DXF_file,20,&y);
if (width1) write_DXF(DXF_file,40,&width1);
if (width2) write_DXF(DXF_file,41,&width2);
if (bulge) write_DXF(DXF_file,42,&bulge);
}

```

Лист. 8.45. Запись в DXF-файл инструкции VERTEX

```

void write_seqend(DXF_file,layer)
FILE      *DXF_file;
char      *layer;
{
    write_DXF(DXF_file,0,"SEQEND");
    write_DXF(DXF_file,8,layer);
}

```

Лист. 8.46. Запись в DXF-файл инструкции SEQEND

Описания блоков также содержат описания объектов. Описания объектов в блоке по своей структуре совпадают с описаниями объектов, не входящих в блок. Вы можете использовать одни и те же подпрограммы для записи объектов в файл. Различие заключается только в окружении объектов. Объекты, связанные с блоками, появляются в секции BLOCKS DXF-файла. Описания объектов помещаются между описанием блока BLOCK и записью ENDBLK.

Описание блока содержит имя блока, некоторые флажки блока и точку привязки (точку вставки) блока.. На лист. 8.47 показана структура описания блока в DXF-файле. Я убрал из описания блока описания объектов, чтобы показать его основную структуру. Запись ENDBLK завершает список объектов, включенных в структуру блока.

```

0
BLOCK
8
0
2
A_BLOCK
10
0.0
20
0.0
:
Описание объектов

```

```

:
0
ENDBLK
8
0

```

Лист. 8.47 Структура блока

На лист. 8.48 приведен исходный текст программы для генерации описания блока. Мы вернулись к структуре данных, в которой описания блока внутри программы содержат указатель на цепочку описаний объектов. Этот указатель передается программе обработки объектов, предназначенной для генерации соответствующих строк в DXF-файле. Не стремитесь при разработке сразу написать весь текст программы, откладывая программирование тех частей, которые можно реализовать как подпрограммы.

```

typedef
struct block_node
{
    struct block_node    *next;
    char                 *name;
    char                 *layer;
    int                  flags;
    double               base_x;
    double               base_y;
    void                 *entities;
}
block_node              *block_anchor;

generate_blocks(dxf)
FILE      *dxf;
{
    block_node    *b;
    entity_node   *e; /* Родовой заголовок */
    if ((b = block_anchor) != NULL)
    {
        write_DXF(dxf, 0, "SECTION");
        write_DXF(dxf, 2, "BLOCKS");
        while (b)
        {
            write_DXF(dxf, 0, "BLOCK");
            write_DXF(dxf, 8, b->layer);
            write_DXF(dxf, 2, b->name);
            write_DXF(dxf, 70, &b->flags);
            write_DXF(dxf, 10, &b->base_x);
            write_DXF(dxf, 20, &b->base_y);

```



```

    e = (entity_node *)b->entities;
    while (e)
    {
        write_entity(dxf,e);
        e = e->next;
    }
    write_DXF(dxf,0,"ENDBLK");
    b = b->next;
}
write_DXF(dxf,0,"ENOSEC");
}
}

```

Лист. 8.48. Генерация блоков для DXF-файла

Использование DXF-файла для создания модифицированного DXF-файла

Возможно, легче написать программу, чем описывать еще один метод генерации DXF-файла. В рассматриваемых методах существующий DXF-файл используется для создания нового DXF-файла.

Основная стратегия такова: считать DXF-файл и отыскать в нем специфические записи, если искомые записи не обнаружены, записать в файл свою строку. Если искомые записи найдены, модифицировать данные и записать обновленную информацию.

В пакете программ для автоматизированного проектирования печатных плат Auto-Board System фирмы The Great SoftWestern Company, Inc. два чертежа разделяют между собой общую информацию. В частном случае такая информация - имена электронных компонентов. Каждая интегральная микросхема имеет свое имя. Каждый резистор имеет свое имя. Все компоненты имеют свои уникальные идентификаторы. Как обычно случается, на этапе производственного процесса изготовления печатной платы желательно изменить имена компонентов, чтобы они лучше отображали позицию компонентов на печатной плате. Однако изменение маркировки имен на печатной плате делает недействительными имена, созданные и используемые разработчиком на электрической принципиальной схеме.

Необходимо было создать программу, которая приводила бы обозначения на электрической схеме в соответствие с новыми именами. Решение заключалось в том, чтобы добавить невидимый атрибут ко всем графическим изображениям компонентов печатной платы. Инженер-технолог не мог бы изменить этот атрибут (или просто не предполагал бы о его существовании). Значением этого атрибута было имя, заданное для компонента разработчиком схемы. Вторым атрибутом представлял имя заданное конструктором. Однако второй атрибут был бы видимым и мог бы быть изменен.

На модифицированной печатной плате были доступны оба имени, как старое, так и измененное. Есть два способа получить список этих имен. Первый способ — использовать команду ATTEXT при работе с чертежом в системе AutoCAD. Команда ATTEXT позволит получить хороший список имен с помощью шаблона, в котором будут определены только поля имен заданных атрибутов.

Второй способ — написать программу осуществляющую просмотр DXF-файла и выборку необходимой информации об атрибутах. Для решения этой задачи может быть модифицирован алгоритм, приведенный в гл. 5. С помощью того или иного способа создается список старых и новых имен компонентов.

Далее пользователь создавал DXF-файл, содержащий описание электрической принципиальной схемы. Программа для модификации схемы быстро считывала и записывала секции HEADER, TABLES и BLOCKS в DXF-файл. В этих секциях не было относящейся к делу информации. Нужная информация находилась в записи об атрибуте. Когда имя компонента появлялось в соответствующем поле атрибута, оно заменялось новым именем и поле записывалось в файл. Имея обновленную электрическую схему в виде DXF-файла, конструктор создавал новый чертеж на основе информации, содержащейся в этом DXF-файле. На лист. 8.49 показан основной фрагмент программы для обнаружения и замены имен.

```
void duplicate_dxf (fi,fo)
FILE *fi;           /* входной DXF-файл */
FILE *fo;           /* выходной DXF-файл */
{
    char        quit;
    int         state;
    char        s[120];
    char        b[120];
    char        dev_name[120];
    int         dev_here;
    int         t;
    int         stop_output;

    quit = FALSE;
    dev_here = FALSE;
    state = 1;
    stop_output = FALSE;
    while (!quit)
    {
        /* Считать LINE из входного DXF-файла. */
        fgets(b,120,fi);
        /* Поиск первой строки, содержащей маркер. */
        sscanf(b,"%s",s);
        /* Преобразовать маркер в номер лексемы. */
    }
```

```

t = get_token(s);
/* Все выполняется в зависимости от текущего */
/* состояния. */
switch (state)
{
    /* Найти маркер 0, помечающий начало секции. */
    case 1: if (t == ZERO) state = 2;
            break;
    /* Продолжить в поисках секции. */
    /* Это может быть имя секции, */
    /* которому предшествует маркер 2. */
    case 2: state = 1;
            if (t == SECTION) state = 3;
            if (t == EOF) quit = TRUE;
            break;
    case 3: state = 1;
            if (t == TWO) state = 4;
            break;
    /* Выполняется просмотр в поисках секции */
    /* ENTITIES, которая начинается с маркера 0. */
    case 4: state = 1;
            if (t == ENTITIES) state = 5;
            break;
    case 5: if (t == ZERO) state = 6;
    /* Маркер 8 помечает начало слоя. */
            if (t == EIGHT) state = 19;
            break;
    /* Обработка элементов, представляющих интерес. */
    case 6: switch (t)
            {
                case EOF: quit = TRUE;
                        break;
                case INSERT: state = 7;
                            break;
                case ATTRIB: state = 12;
                            break;
                case ENDSEC: state = 1;
                            break;
                case ENDSEQ: state = 5;
                            break;
                case EIGHT: state = 18;
                            break;
                default: state = 5;
                        break;
            }
            break;
    case 7: switch (t)

```

```

/* Эта часть программы предназначена для обработки */
/* инструкции INSERT, в данном примере указанные */
/* состояния отсутствуют, но вы можете при желании */
/* ввести эти состояния в программу и при обнаруже- */
/* нии соответствующего маркера выполнить требуемые */
/* действия по модификации значений в DXF-файле. */
{
    case ZERO      : state = 6;
                    break;
    case TWO       : state = 8;
                    break;
    case EIGHT     : state = 16;
                    break;
    case TEN       : state = 9;
                    break;
    case TWENTY    : state = 10;
                    break;
    case FIFTY     : state = 11;
                    break;
    case SIXTY6    : state = 16;
                    break;
    default        : state = 5;
                    break;
}
break;
case 16:
case 8:
case 9:
case 10:
case 11:
    state = 7;
    break;
/* Это место программы предназначено */
/* для обработки инструкции ATTRIB */
/* и изменения значений атрибутов. */
/* Здесь перечисляются маркеры для */
/* значений атрибута. */
case 12: switch(t)
{
    case ZERO      : state = 6;
                    break;
    case ONE       : state = 14;
                    stop_output = TRUE;
                    break;
    case TWO       : state = 15;
                    break;
    case EIGHT     : state = 13;

```

```

        break;
    case TEN : state = 13;
        break;
    case TWENTY : state = 13;
        break;
    case FORTY : state = 13;
        break;
    default : state = 17;
        break;
    }
    break;
case 13: state = 12;
    break;
case 14: state = 12;
/* По заданному значению в DXF-файле выполняется */
/* поиск заменяющего значения. */
   strupr(s);
    strcpy(dev_name, find_new_name(s));
    break;
case 15: state = 5;
/* Инструкция ATTRIB закончилась - записать новое */
/* значение в DXF-файл. */
    fprintf(fo, "\n%s\n2\n", dev_name);
    stop_output = FALSE;
    break;
case 17: state = 5;
    break;
case 18: state = 6;
    break;
case 19: state = 5;
    break;
}
/* До перехода в состояние 15, копировать */
/* входную строку в выходной DXF-файл. */
if (!stop_output) fprintf(fo, "%s\n", s);
}
}

```

Лист. 8.49. Использование DXF-файла для генерации DXF-файла

Файлы формата PGP и DXB

В DXF-файле в символьном формате (в коде ASCII) содержится вся информация о чертеже. Этот файл со всеми маркерами, полными именами и множеством символов, необходимых для формирования чисел, обычно весьма большой по своему объему. Существует более короткая форма файла, применяемая для подготовки входных

данных для системы AutoCAD. Такой файл представляет собой нечто среднее между DXF-файлами и файлами чертежей типа DWG. Этот файл называется файлом типа DXB или DXB-файлом (файл обмена описаниями чертежей в двоичном формате). Он содержит информацию только о графических объектах и имеет чисто двоичный формат. Вы не сможете просмотреть содержимое этого файла с помощью команды TYPE операционной системы MS-DOS или редактора текста.

Формат DXB-файла предоставляет удобный и быстрый метод для считывания неполной графической информации, подготовленной внешней программой. Программы для генерации DXB-файлов приведены в конце данного обсуждения. Но сейчас я хотел бы рассмотреть, как можно запустить на выполнение внешние программы, которые будут создавать DXB-файлы.

Есть возможность выйти из рабочей среды системы AutoCAD и выполнить внешнюю программу. Вы вероятно знаете команду SHELL системы AutoCAD. Команда SHELL представляет собой пустое имя, которое позволит запустить и выполнить внешние программы. Почему эта команда является пустым именем? Команда SHELL является просто точкой входа в файл ACAD.PGP. В этом файле содержатся имена команд, которые система AutoCAD распознает как запросы для выполнения внешних программ.

Если вы посмотрите PGP-файл, вы обнаружите в нем несколько внешних команд. На лист. 8.50 показано содержимое PGP-файла, поставляемого вместе с системой AutoCAD. Все записи в этом файле применяются для запуска на выполнение указанных внешних программ. Вы можете добавить к этому списку ваши собственные программы, и это именно то, о чем я хотел вам рассказать.

```
DEL,DEL,27000,File to delete: .0
DIR,DIR,27000,File specification: .0
FILECOPY,COPY,27000,*Filenames for COPY: .0
EDLIN,EDLIN,42000,File to edit: .0
SHELL,,127000,*DOS Command: .0
TYPE,TYPE,27000,File to list: .0
QSEE,QSEE,200000,*File to View: .0
```

Лист. 8.50. Типичный PGP-файл

В записи PGP-файла имеется пять полей. Поля разделяются друг от друга запятыми.

1. Имя команды — данное имя является именем команды, по которому вы хотели, чтобы система AutoCAD распознавала правильную команду. Вы увидели, что команда SHELL была просто одной из команд в PGP-файле.

2. Имя программы — это имя представляет собой имя внешней программы, которое система AutoCAD передает операционной системе для запуска этой программы на выполнение. К этому имени мо-

гут быть добавлены любые специальные ключи, указываемые в командной строке. Если вы не хотите присваивать в данный момент какое-либо конкретное имя программе вы можете оставить это поле пустым. Отметим, что в команде SHELL имя программы не указано. В других строках файла имя программы присутствует.

3. Объем памяти — данное поле определяет объем памяти, который система AutoCAD должна освободить для операционной системы (MS-DOS), чтобы ваша программа имела достаточно памяти для выполнения. Кроме вашей программы система AutoCAD загружает копию командного процессора операционной системы (command.com). Так системы AutoCAD и ОС MS-DOS управляют загрузкой и выполнением вашей программы. При оценке требуемого для вашей программы объема памяти, учитывайте дополнительные 24 Кбайта памяти, выделяемые для командного процессора.

4. Приглашение к вводу — для вашей программы могут быть нужны дополнительные параметры, передаваемые в командной строке. Иногда для выполнения программы недостаточно указывать при ее запуске только имя программы. Приглашение к вводу применяется как способ передачи программе дополнительных параметров. Если не используется возможность дополнительного ввода, вы можете оставить это поле пустым. Обычно система AutoCAD будет воспринимать ответы на заданное приглашение к вводу до тех пор, пока не будет встречен пробел или нажата клавиша ENTER. Вы можете проинструктировать систему AutoCAD, чтобы она воспринимала пробелы (при наличии нескольких параметров), вставив символ звездочки (*) как первый символ строки приглашения к вводу.

5. Код возврата — в этом поле указываются действия, которые система AutoCAD должна предпринять при завершении выполнения внешней программы. В поле помещается значение, представляющее собой битовую маску. Следуя приведенному ниже описанию, сложите числа для задания желаемого действия. Помещенное в это поле значение 0 сообщает о том, что система AutoCAD не выполняет дальнейшей обработки. Запрос для выполнения внешней программы переводит систему AutoCAD в текстовый режим (в противоположность графическому режиму). Возвращаемое значение 0 оставляет систему в текстовом режиме. Значения, определяющие возможные действия, приведены ниже:

- 1 Загрузить DXB-файл. Ваша программа должна создать файл с именем \$cmd.dxb, содержащий информацию в DXB-формате. Система AutoCAD удалит этот файл.
- 2 Построить блок из данных, содержащихся в DXB-файле. Ваша программа может образовать детали блока в DXB формате. Используя код 1 и код 2 (2+1=3), вы можете создавать заказной блок. Именем блока является значение, введенное в ответ на приглашение к вводу. Имя блока должно быть новым (не используемым в

текущем чертеже), так как вы не можете переопределять существующий блок.

4 Восстановить текстовый/графический режим. Как отмечено выше, система AutoCAD переходит в текстовый режим при запуске на выполнение внешней программы. В случае сложения этого значения с другими значениями кода возврата система AutoCAD перейдет в графический режим работы.

Кроме ввода информации в формате DXB с использованием внешних программ, определенных в PGP-файле, для считывания DXB-файлов имеется обычная команда системы AutoCAD. Эта команда идентична команде DXFIN. Она называется DXBIN. Формат этой команды также совпадает с форматом первой команды. Теперь, когда мы знаем, как ввести информацию в DXB формате в систему AutoCAD, давайте рассмотрим генерацию DXB-файлов.

Для генерации DXB-файлов надо знать следующее. DXB-файл содержит только двоичную информацию. Это не символьный файл, содержащий информацию в коде ASCII, и вы не можете применять при его создании ориентированные на поток функции ввода-вывода, такие, как (fopen, fprintf, fputs и fclose). Вы должны использовать функции open, write и close. Вы должны иметь имя для этого файла. Для программ, запускаемых на выполнение из системы AutoCAD, файл должен иметь имя \$cmd.dxb. На лист. 8.51 показан фрагмент программы для открытия двоичного файла \$cmd.dxb. При разработке программ применялся компилятор Microsoft C. Если вы работаете с другим компилятором, то в функции open могут быть некоторые отличия.

```
int          DXB_file;  
  
DXB_file = open("$cmd.dxb", O_BINARY|O_WRONLY|O_CREATE.  
                I_WRITE|I_READ);
```

Лист. 8.51. Открытие DXB-файла

Остальная часть вашей программы должна быть посвящена созданию графических объектов. Программа генерации этих объектов будет выглядеть довольно схоже с программами генерации DXF-файлов. Отличаться будут только подпрограммы вывода. Подпрограммы вывода для DXF-файлов записывали несколько пар маркеров и значений в выходной поток символьной информации в коде ASCII. Подпрограммы вывода для DXB-файлов помещают параметры графических объектов в двоичные структуры данных и записывают эти структуры в файл вывода.

Последовательности вызовов функций в следующих далее программах идентичны применяемым в программах вывода для DXF-файлов. Поэтому модификация программы для записи информации в DXB-файл должна быть весьма безболезненной. Существует одна

тонкость. В формате DXF ссылка указателя на файл представляла собой указатель на структуру FILE. В формате DXB, значение ссылки на файл является целым числом, определяющим номер канала ввода-вывода, закрепленного за файлом.

Сначала мы рассмотрим некоторые рабочие детали DXB-формата, а затем предоставим пачку листингов, чтобы показать подробности того, как генерируются отдельные объекты. На лист. 8.47 приведены определения параметрических констант, используемых в остальных листингах. Эти константы представляют собой идентифицирующие значения, применяемые в структурах данных DXB-формата.

DXB-файл имеет специальный заголовок. Вы всегда должны включать этот заголовок в файл. Система AutoCAD не сможет обработать ваш файл, если заголовок файла не будет корректно построен. На лист. 8.52 показана конструкция заголовка DXB-файла. В конце DXB-файла должен присутствовать символ NULL. В листингах, представленных в данном параграфе, я использовал несколько констант, чтобы сделать исходные тексты программ более наглядными при чтении. На лист. 8.53 приведен список определений этих констант.

```
void write_header(DXB_file)
int
    DXB_file;
{
    char    header[40];
    strcpy(header, "AutoCAD DXB 1.0\r\n\x1a");
    write(DXB_file, header, strlen(header)+1);
}
```

Лист. 8.52. Генерация заголовка DXB-файла

```
#define DXB_LINE          1
#define DXB_POINT         2
#define DXB_CIRCLE        3
#define DXB_ARC           8
#define DXB_TRACE         9
#define DXB_SOLID         11
#define DXB_SEQEND        17
#define DXB_POLYLINE      19
#define DXB_VERTEX        20
#define DXB_SCALE         128
#define DXB_LAYER         129
#define DXB_L_EXTEND      130
#define DXB_T_EXTEND      131
#define DXB_BLOCK         132
#define DXB_BULGE         133
#define DXB_WIDTH         134
```

```

#define DXB_MODE          135
#define DXB_INTEGER      0
#define DXB_DOUBLE       1

```

Лист. 8.53. Константы, используемые при записи в DXB-файл

Один из интересных аспектов работы с DXB-файлами состоит в том, что система AutoCAD будет считывать данные из файла либо в режиме представления чисел с плавающей точкой, либо в режиме представления целых чисел. В двух этих режимах записи в файле идентичны, но имеют различные типы данных. Эти режимы влияют на ввод координат и углов. Целые числа имеют более компактный формат, под каждое значение требуется только 16 бит. В режиме представления чисел с плавающей точкой под число отводится 64 бита, но это позволяет получать более высокую точность на чертеже. Вы можете принять решение изменить режим представления чисел при создании вашего файла. На лист. 8.54 показаны две подпрограммы для изменения режимов представления чисел в DXB-файле.

```

void set_double(DXB_file)
int          DXB_file;
{
    unsigned char    id;
    int              mode;

    id = DXB_MODE;
    mode = DXB_DOUBLE;
    write(DXB_file, (char *)&id, sizeof(id));
    write(DXB_file, (char *)&mode, sizeof(mode));
}

void set_integer(DXB_file)
int          DXB_file;
{
    unsigned char    id;
    int              mode;

    id = DXB_MODE;
    mode = DXB_INTEGER;
    write(DXB_file, (char *)&id, sizeof(id));
    write(DXB_file, (char *)&mode, sizeof(mode));
}

```

Лист. 8.54. Две функции для изменения режима представления чисел в DXB-файле

Поскольку имеются два режима представления чисел в DXB-файле и эти режимы отражаются на обработке графических объектов, содержащих информацию о координатах и углах, на некоторых листингах приводятся по две подпрограммы. Одна для работы в режиме представления чисел с плавающей точкой, в названия переменных такой подпрограммы включается окончание `double`. Другая для работы в режиме представления целых чисел, в названия ее переменных включается окончание `int`.

В DXB-файлах вы можете изменять слои. Как и в главе, посвященной генерации пакетных файлов, для указания имени текущего слоя мы используем глобальную переменную `current_layer`. Если объект должен находиться на другом слое изображения, то нам необходимо поменять текущий слой и установить новый текущий слой в DXB-файле. Имя слоя является строкой символов в коде ASCII, оканчивающейся символом `NULL`. Это обычный формат строки, принятый в языке Си. Но вы должны быть осторожны, если программируете на языке Паскаль или других языках. Среди выведенных в файл данных должен присутствовать завершающий символ `NULL`, так чтобы программа с помощью функции `strlen` смогла определить длину имени слоя и зарезервировать место для символа `NULL`. На лист. 8.55 представлена программа для записи информации о слое в DXB-файл.

```
void write_layer(DXB_file, layer)
int          DXB_file;
char         *layer;
{
    unsigned char    id;

    id = DXB_LAYER;
    mode = DXB_INTEGER;
    write(DXB_file, (char *)&id, sizeof(id));
    write(DXB_file, layer, strlen(layer)+1);
}
```

Лист. 8.55. Запись имени слоя в DXB-файл

Вы можете установить масштабный множитель для данных вашего чертежа. Я думаю, что это может сократить вдвое объем данных. Я мог бы даже сам использовать эти способы в будущих разработках.

Первым примером применения масштабного множителя является создание компактной формы файла, который должен содержать информацию, представленную в виде чисел с плавающей точкой. Например, если все значения в файле попадают в интервал `-32.000 — +32.000` и требуется точность представления до третьего знака после точки, то вы могли бы использовать в DXB-файле режим представления целых чисел. Такое представление чисел является относительно компактным. Система AutoCAD сама выполнит преобразование

ваших целых значений к надлежащему виду, умножая их на коэффициент масштабирования 0.001.

Второй пример - это преобразование единиц измерения. Если вы имеете программу, которая обычно выполняет вычисления в футах и дюймах (в основном в дюймах), но вам необходимо удовлетворить спрос рынка, ориентированного на метрическую систему измерения (широко используемую в мире): в начало DXB-файла добавьте коэффициент масштабирования, который поможет вам привести ваши данные в соответствие с метрической системой измерения. Умножив значение, выраженное в дюймах, на 2.54, вы получите значение, выраженное в сантиметрах. Вы можете выполнять свои вычисления в дюймах, записать данные, выраженные в дюймах, в DXB-файл и этим помочь работающим в метрической системе измерения. С помощью аргумента, указанного в PGP-файле, можно сообщить вашей программе, что нужно добавить или проигнорировать масштабный множитель. На лист. 8.56 показан исходный текст функции для добавления масштабного множителя в DXB-файл. Работа этой функции не зависит от используемого режима представления чисел.

```
void write_scale(DXB_file, factor)
int          DXB_file;
double       factor;
{
    struct
    {
        unsigned char    id;
        double            scale_factor;
    }                    scaling;

    scaling.id = DXB_SCALE;
    scaling.scale_factor = factor;
    write(DXB_file, (char *)&scaling, sizeof(scaling));
}
```

Лист. 8.56. Установка масштабного множителя в DXB-файле

Один из параметров в записи PGP-файла указывает системе AutoCAD, что входной DXB-файл следует воспринимать как описание блока. Первым значением в массиве данных DXB-файла, содержащего описание блока, должна быть запись BLOCK. Данная запись устанавливает базовые координаты блока. Это значение влияет на преобразование координат, описанных в блоке объектов, при выполнении операции вставки блока. Значение базовых координат обычно 0.0, 0.0, но в качестве базовых могут использоваться и другие значения. На лист. 5.57 показаны программы для определения базовых координат блока.

```

void write_block(DXB_file,x,y,layer)
int          DXB_file;
double       x;
double       y;
char         *layer;
{
struct
{
    unsigned char    id;
    double           x;
    double           y;
}                block_double;

    if (strcmp(layer,current_layer))
    {
        write_layer(DXB_file,layer);
        current_layer = layer;
    }

    block_double.id = DXB_BLOCK;
    block_double.x = x;
    block_double.y = y;
    write(DXB_file,(char*)&block_double,sizeof(block_double));
}

```

```

void write_block(DXB_file,x,y,layer)
int          DXB_file;
int          x;
int          y;
char         *layer;
{
struct
{
    unsigned char    id;
    int              x;
    int              y;
}                block_int;

    if (strcmp(layer,current_layer))
    {
        write_layer(DXB_file,layer);
        current_layer = layer;
    }

    block_int.id = DXB_BLOCK;
    block_int.x = x;

```

```

        block_int.y = y;
        write(DXB_file, (char*)&block_int, sizeof(block_int));
    }

```

Лист. 8.57. Две программы для записи информации о блоке в DXB-файл

В остальных листингах данной главы приведены подпрограммы для обработки других записей DXB-файла. Большинство этих программ весьма схожи с подобными программами для записи информации в DXF-формате и формате пакетных файлов. Предполагается, что данные имеют тот же самый тип данных. Как отмечалось в параграфе, посвященном структуре пакетного файла, для передачи данных подпрограммам вы можете использовать указатели на структуры данных как аргументы этих подпрограмм. То же решение предложено и здесь. Приведенные примеры специально разработаны так, чтобы показать схожесть вызывающих последовательностей в различных подпрограммах вывода. Форма вывода не отражается на подготовке текста какой-либо из рассматриваемых подпрограмм. Основное различие представляет организация обращения к файлу.

Подпрограммы LINE и TRACE заслуживают особых комментариев. Обычные подпрограммы LINE и TRACE порождают записи для одиночного отрезка или сегмента полосы. Для облегчения создания непрерывных линий и повторения концевой точки последнего отрезка система AutoCAD предоставляет для линий и полос записи продолжения. Эти записи позволяют вам генерировать линии с несколькими вершинами, указывая только следующую концевую точку. Эти подпрограммы в листингах программ имеют названия write_ext_line и write_ext_trace.

Такая структура данных, как полилиния, в DXB-файле полностью разбивается на части. Вы делаете отдельные записи для изменения ширины линии, дуговых сегментов и координат вершин. Фактор кривизны представляет собой отношение расстояния между конечными точками и высотой кривой в ее середине. Приведенная на одном из листингов подпрограмма запрашивает расстояние между концевыми точками и высоту и, перед тем как записать эти данные, вычисляет отношение между ними. Полилинии должны заканчиваться записью SEQEND. В табл. 8.3 приведены названия графических объектов и номера листингов соответствующих подпрограмм для их генерации.

Таблица 8.3. Список подпрограмм вывода в DXB-файл

BLOCK	Лист. 8.57
LINE	Лист. 8.58
LINE EXTENSION	Лист. 8.59
POINT	Лист. 8.60
CIRCLE	Лист. 8.61
ARC	Лист. 8.62

TRACE	Лист. 8.63
TRACE EXTENSION	Лист. 8.64
SOLID	Лист. 8.65
POLYLINE	Лист. 8.66
VERTEX	Лист. 8.67
WIDTH	Лист. 8.68
BULGE	Лист. 8.69
SEQEND	Лист. 8.70

```

void write_line (DXB_file,from_x,from_y,to_x,to_y,layer)
int          DXB_file;
double       from_x;
double       from_y;
double       to_x;
double       to_y;
char         *layer;
{
    struct
    {
        unsigned char    id;
        double            from_x;
        double            from_y;
        double            to_x;
        double            to_y;
    } line_double;

    if (strcmp(layer,current_layer))
    {
        write_layer(DXB_file,layer);
        current_layer = layer;
    }

    line_double.id = DXB_LINE;
    line_double.from_x = from_x;
    line_double.from_y = from_y;
    line_double.to_x = to_x;
    line_double.to_y = to_y;
    write(DXB_file,(char*)&line_double,sizeof(line_double));
}

```

```

void write_line (DXB_file,from_x,from_y,to_x,to_y,layer)
int          DXB_file;
int          from_x;
int          from_y;
int          to_x;

```

```

int          to_y;
char        *layer;
{
struct
{
    unsigned char    id;
    int              from_x;
    int              from_y;
    int              to_x;
    int              to_y;
}                line_int;

if (strcmp(layer,current_layer))
{
    write_layer(DXB_file,layer);
    current_layer = layer;
}

line_int.id = DXB_LINE;
line_int.from_x = from_x;
line_int.from_y = from_y;
line_int.to_x = to_x;
line_int.to_y = to_y;
write(DXB_file,(char*)&line_int,sizeof(line_int));
}

```

Лист. 8.58. Две программы для записи информации о линии в **DXB**-файл

```

void write_ext_line (DXB_file,to_x,to_y.)
int          DXB_file;
double       to_x;
double       to_y;
{
struct
{
    unsigned char    id;
    double           to_x;
    double           to_y;
}                l_extend__double;

l_extend__double.id = DXB_L_EXTEND;
l_extend__double.to_x = to_x;
l_extend__double.to_y = to_y;

write(DXB_file,(char*)&l_extend__double,sizeof(l_extend__double));
}

```



```

void write_ext_line (DXB_file,to_x,to_y.)
int      DXB_file;
int      to_x;
int      to_y;
{
struct
{
    unsigned char    id;
    int              to_x;
    int              to_y;
}
    l_extend__int;

    l_extend__int.id = DXB_L_EXTEND;
    l_extend__int.to_x = to_x;
    l_extend__int.to_y = to_y;

write(DXB_file,(char*)&l_extend__int,sizeof(l_extend__int));
}

```

Лист. 8.59. Две программы для записи информации о продолжении линии в DXB-файл

```

void write_point (DXB_file,x,y,layer)
int      DXB_file;
double   x;
double   y;
char     *layer;
{
struct
{
    unsigned char    id;
    double           point_x;
    double           point_y;
}
    point_double;

    if (strcmp(layer,current_layer))
    {
        write_layer(DXB_file,layer);
        current_layer = layer;
    }

    point_double.id = DXB_POINT;
    point_double.point_x = x;
    point_double.point_y = y;
    point_double.point_y = y;
    write(DXB_file,(char*)&point_double,sizeof(point_double));
}

```

```

}

void write_point (DXB_file,x,y,layer)
int      DXB_file;
int      x;
int      y;
char     *layer;
{
struct
{
    unsigned char    id;
    int              point_x;
    int              point_y;
    int              point_int;

    if (strcmp(layer,current_layer))
    {
        write_layer(DXB_file,layer);
        current_layer = layer;
    }

    point_int.id = DXB_POINT;
    point_int.point_x = x;
    point_int.point_y = y;
    write(DXB_file,(char*)&point_int,sizeof(point_int));
}
}

```

Лист. 8.60. Две программы для записи информации о точке в DXB-файл

```

void write_circle (DXB_file,x,y,r,layer)
int      DXB_file;
double   x;
double   y;
double   r;
char     *layer;
{
struct
{
    unsigned char    id;
    double           circle_x;
    double           circle_y;
    double           radius;
    double           circle_double;

    if (strcmp(layer,current_layer))
    {

```

```

        write_layer(DXB_file, layer);
        current_layer = layer;
    }

    circle_double.id = DXB_CIRCLE;
    circle_double.center_x = x;
    circle_double.center_y = y;
    circle_double.radius = y;

write(DXB_file, (char*)&circle_double, sizeof(circle_double));
}

void write_circle (DXB_file, x, y, layer)
int          DXB_file;
int          x;
int          y;
char         *layer;
{
    struct
    {
        unsigned char    id;
        int              circle_x;
        int              circle_y;
        int              radius;
    } circle_int;

    if (strcmp(layer, current_layer))
    {
        write_layer(DXB_file, layer);
        current_layer = layer;
    }

    circle_int.id = DXB_CIRCLE;
    circle_int.center_x = x;
    circle_int.center_y = y;
    circle_int.radius = r;
    write(DXB_file, (char*)&circle_int, sizeof(circle_int));
}

```

Лист. 8.61. Две программы для записи информации об окружности в **DXB**-файл

```

void write_arc (DXB_file, cx, cy, r, sa, ea, layer)
int          DXB_file;
double       cx;
double       cy;
double       r;

```

```

double      sa;
double      ea;
char        *layer;
{
struct
{
    unsigned char    id;
    double           center_x;
    double           center_y;
    double           radius;
    double           start_angle;
    double           end_angle;
}
    arc_double;

    if (strcmp(layer,current_layer))
    {
        write_layer(DXB_file,layer);
        current_layer = layer;
    }

    arc_double.id = DXB_CIRCLE;
    arc_double.center_x = cx;
    arc_double.center_y = cy;
    arc_double.radius = r;
    arc_double.start_angle = sa;
    arc_double.end_angle = ea;
    write(DXB_file,(char*)&arc_double,sizeof(arc_double));
}

```

```

void write_arc (DXB_file,cx,cy,r,sa,ea,layer)
int          DXB_file;
int          cx;
int          cy;
int          sa;
int          ea;
char        *layer;
{
struct
{
    unsigned char    id;
    int             center_x;
    int             center_y;
    int             radius;
    int             start_angle;
    int             end_angle;
}
    arc_int;

```

```

if (strcmp(layer,current_layer))
{
    write_layer(DXB_file,layer);
    current_layer = layer;
}

arc_int.id = DXB_ARC;
arc_int.center_x = cx;
arc_int.center_y = cy;
arc_int.radius = r;
arc_int.start_angle = sa;
arc_int.end_angle = ea;
write(DXB_file,(char*)&arc_int,sizeof(arc_int));
}

```

Лист. 8.62. Две программы для записи информации о дуге в DXB-файл

```

void write_trace (DXB_file,x1,y1,x2,y2,x3,y3,x4,y4, layer)
int DXB_file;
double x1;
double y1;
double x2;
double y2;
double x3;
double y3;
double x4;
double y4;
char *layer;
{
    struct
    {
        unsigned char id;
        double x1;
        double y1;
        double x2;
        double y2;
        double x3;
        double y3;
        double x4;
        double y4;
    } trace_double;

    if (strcmp(layer,current_layer))
    {
        write_layer(DXB_file,layer);
        current_layer = layer;
    }
}

```

```

    }

    trace_double.id = DXB_TRACE;
    trace_double.x1 = x1;
    trace_double.y1 = y1;
    trace_double.x2 = x2;
    trace_double.y2 = y2;
    trace_double.x3 = x3;
    trace_double.y3 = y3;
    trace_double.x4 = x4;
    trace_double.y4 = y4;
    write(DXB_file, (char *)&trace_double,
          sizeof(trace_double));
}

void write_trace (DXB_file, x1, y1, x2, y2, x3, y3, x4, y4, layer)
int     DXB_file;
int     x1;
int     y1;
int     x2;
int     y2;
int     x3;
int     y3;
int     x4;
int     y4;
char    *layer;
{
    struct
    {
        unsigned char    id;
        int               x1;
        int               y1;
        int               x2;
        int               y2;
        int               x3;
        int               y3;
        int               x4;
        int               y4;
    }
    trace_int;

    if (strcmp(layer, current_layer))
    {
        write_layer(DXB_file, layer);
        current_layer = layer;
    }
}

```

```

    trace_int.id = DXB_TRACE;
    trace_int.x1 = x1;
    trace_int.y1 = y1;
    trace_int.x2 = x2;
    trace_int.y2 = y2;
    trace_int.x3 = x3;
    trace_int.y3 = y3;
    trace_int.x4 = x4;
    trace_int.y4 = y4;
    write(DXB_file, (char *)&trace_int, sizeof(trace_int));
}

```

Лист. 8.63. Две программы для записи информации о полосе в **DXB**-файл

```

void write_ext_trace (DXB_file, x3, y3, x4, y4)
int      DXB_file;
int      x3;
int      y3;
int      x4;
int      y4;
{
    struct
    {
        unsigned char    id;
        double            new_x3;
        double            new_y3;
        double            new_x4;
        double            new_y4;
        t_extend_double;
    }

    t_extend_double.id = DXB_T_EXTEND;
    t_extend_double.new_x3 = x3;
    t_extend_double.new_y3 = y3;
    t_extend_double.new_x4 = x4;
    t_extend_double.new_y4 = y4;
    write(DXB_file, (char *)&t_extend_double,
          sizeof(t_extend_double));
}

void write_ext_trace (DXB_file, x3, y3, x4, y4)
int      DXB_file;
int      x3;
int      y3;
int      x4;
int      y4;
{
    struct

```

```

{
    unsigned char    id;
    int              new_x3;
    int              new_y3;
    int              new_x4;
    int              new_y4;
}
t_extend_int;

t_extend_int.id = DXB_T_EXTEND;
t_extend_int.new_x3 = x3;
t_extend_int.new_y3 = y3;
t_extend_int.new_x4 = x4;
t_extend_int.new_y4 = y4;
write(DXB_file, (char *)&t_extend_int,
      sizeof(t_extend_int));
}

```

Лист. 8.64. Две программы для записи информации о продолжении полосы в DXB-файл

```

void write_solid (DXB_file, x1, y1, x2, y2, x3, y3, x4, y4, layer)
int          DXB_file;
double       x1;
double       y1;
double       x2;
double       y2;
double       x3;
double       y3;
double       x4;
double       y4;
char         *layer;
{
    struct
    {
        unsigned char    id;
        double            x1;
        double            y1;
        double            x2;
        double            y2;
        double            x3;
        double            y3;
        double            x4;
        double            y4;
    }
    solid_double;

    if (strcmp(layer, current_layer))
    {

```



```

    write_layer(DXB_file, layer);
    current_layer = layer;
}

solid_double.id = DXB_SOLID;
solid_double.x1 = x1;
solid_double.y1 = y1;
solid_double.x2 = x2;
solid_double.y2 = y2;
solid_double.x3 = x3;
solid_double.y3 = y3;
solid_double.x4 = x4;
solid_double.y4 = y4;
write(DXB_file, (char *)&solid_double,
      sizeof(solid_double));
}

void write_solid (DXB_file, x1, y1, x2, y2, x3, y3, x4, y4, layer)
int      DXB_file;
int      x1;
int      y1;
int      x2;
int      y2;
int      x3;
int      y3;
int      x4;
int      y4;
char      *layer;
{
    struct
    {
        unsigned char    id;
        int               x1;
        int               y1;
        int               x2;
        int               y2;
        int               x3;
        int               y3;
        int               x4;
        int               y4;
    }
    solid_int;

    if (strcmp(layer, current_layer))
    {
        write_layer(DXB_file, layer);
        current_layer = layer;
    }
}

```

```

solid_int.id = DXB_TRACE;
solid_int.x1 = x1;
solid_int.y1 = y1;
solid_int.x2 = x2;
solid_int.y2 = y2;
solid_int.x3 = x3;
solid_int.y3 = y3;
solid_int.x4 = x4;
solid_int.y4 = y4;
write(DXB_file, (char *)&solid_int, sizeof(solid_int));
}

```

Лист. 8.65. Две программы для записи информации о фигуре в DXB-файл

```

void write_pline (DXB_file, closure, layer)
int          DXB_file;
int          closure;
char         *layer;
{
    unsigned char    id;

    if (strcmp(layer, current_layer))
    {
        write_layer(DXB_file, layer);
        current_layer = layer;
    }

    id = DXB_POLYLINE;
    write(DXB_file, (char *)&id, sizeof(id));
    write(DXB_file, (char *)&closure, sizeof(closure));
}

```

Лист. 8.66. Запись информации о полилинии в DXB-файл

```

void write_vertex (DXB_file, x, y)
int          DXB_file;
double       x;
double       y;
{
    struct
    {
        unsigned char    id;
        double            x;
        double            y;
    }
}

```

```

    }
    vertex_double;
    vertex_double.id = DXB_VERTEX;
    vertex_double.x = x;
    vertex_double.y = y;
    write(DXB_file, (char *)&vertex_double,
        sizeof(vertex_double));
}

void write_vertex (DXB_file, x, y)
int DXB_file;
int x;
int y;
{
    struct
    {
        unsigned char id;
        int x;
        int y;
    } vertex_int;
    vertex_int.id = DXB_VERTEX;
    vertex_int.x = x;
    vertex_int.y = y;
    write(DXB_file, (char *)&vertex_int, sizeof(vertex_int));
}

```

Лист. 8.67. Две программы для записи информации о вершине полилинии в DXB-файл

```

void write_width (DXB_file, sw, ew)
int DXB_file;
double sw;
double ew;
{
    struct
    {
        unsigned char id;
        double sw;
        double ew;
    } width_double;
    width_double.id = DXB_VERTEX;
    width_double.sw = sw;
    width_double.ew = ew;
    write(DXB_file, (char *)&width_double,
        sizeof(width_double));
}

void write_width (DXB_file, sw, ew)

```

```

int          DXB_file;
int          sw;
int          ew;
{
struct
{
    unsigned char    id;
    int              sw;
    int              ew;
}                width_int;
width_int.id = DXB_VERTEX;
width_int.sw = sw;
width_int.ew = ew;
write(DXB_file, (char *)&width_int, sizeof(width_int)).
}

```

Лист. 8.68. Две программы для записи информации о ширине полилинии в DXB-файл

```

void write_bulge (DXB_file, h, d)
int          DXB_file;
double       h;
double       d;
{
struct
{
    unsigned char    id;
    float            two_h_over_d;
}                bulge;
bulge.id = DXB_BULGE;
bulge.two_h_over_d = (h + h) / d;
write(DXB_file, (char *)&bulge, sizeof(bulge));
}

```

Лист. 8.69. Запись информации о дуговом сегменте полилинии в DXB-файл

```

void write_seqend (DXB_file)
int          DXB_file;
{
    unsigned char    id;
    id = DXB_SEQEND;
    write(DXB_file, (char *)&id, sizeof(id));
}

```

Лист. 8.70. Запись в DXB-файл инструкции SEQEND

Приложение А

НАЦИОНАЛЬНЫЕ ВЕРСИИ СИСТЕМЫ AUTOCAD

В приведенных листингах содержатся определения английских команд системы AutoCAD для трансляции на французский, немецкий, итальянский и испанский. Имея возможность включения этих определений в программу, как описано в гл. 7, вы можете написать программу, которая сможет работать с любой из национальных версий системы AutoCAD. Приведенные листинги представляют собой все части файла LANG.H, подключаемого к программе с помощью директивы INCLUDE языка Си.

Лист. А.1 содержит стандартные команды на английском языке. Его определения активизируются, если в вашей программе на языке Си определено слово ENGLISH. Задать это определение вы можете или в командной строке при вызове компилятора или внутри программы с помощью директивы #define.

Лист. А.2 содержит команды на французском языке. Определения становятся активными, если определено слово FRENCH.

Лист. А.3 содержит команды на немецком языке. Определения становятся активными, если определено слово GERMAN.

Лист. А.4 содержит команды на итальянском языке. Определения становятся активными, если определено слово ITALIAN.

Лист. А.5 содержит команды на испанском языке. Определения становятся активными, если определено слово SPANISH.

```
#ifndef ENGLISH
#define ALL "A"
#define APERTURE "APERTURE"
#define ARC "ARC"
#define AREA "AREA"
#define ARRAY "ARRAY"
#define ATTDEF "ATTDEF"
#define ATTDISP "ATTDISP"
#define ATTEDIT "ATTEDIT"
#define ATTEXT "ATTEXT"
#define BASE "BASE"
#define BLIPMODE "BLIPMODE"
#define BLOCK "BLOCK"
#define BLUE "BLUE"
#define BREAK "BREAK"
```

```

#define CHAMFER      "CHAMFER"
#define CHANGE      "CHANGE"
#define CIRCLE      "CIRCLE"
#define COLOR       "COLOR"
#define COPY        "COPY"
#define CYAN        "CYAN"
#define DBLIST      "DBLIST"
#define DELAY       "DELAY"
#define DIM         "DIM"
#define DIST        "DIST"
#define DIVIDE      "DIVIDE"
#define DONUT       "DONUT"
#define DOUGHNUT    "DOUGHNUT"
#define DRAGMODE    "DRAGMODE"
#define DTEXT       "DTEXT"
#define DXBIN       "DXBIN"
#define DXBOUT      "DXBOUT"
#define DXFIN       "DXFIN"
#define ELEV        "ELEV"
#define ELLIPSE     "ELLIPSE"
#define END         "END"
#define ERASE       "ERASE"
#define EXPLODE     "EXPLODE"
#define EXTEND      "EXTEND"
#define EXTENTS     "E"
#define FILES       "FILES"
#define FILL        "FILL"
#define FILLET      "FILLET"
#define GREEN       "GREEN"
#define GRID        "GRID"
#define HIDE        "HIDE"
#define ID          "ID"
#define IGESIN      "IGESIN"
#define IGESOUT     "IGESOUT"
#define INSERT      "INSERT"
#define ISOPLANE    "ISOPLANE"
#define LAST        "L"
#define LAYER       "LAYER"
#define LCHANGE     "L"
#define LIMITS      "LIMITS"
#define LINE        "LINE"
#define LINETYPE    "LINETYPE"
#define LIST        "LIST"
#define LOAD        "LOAD"
#define LTSCALE     "LTSCALE"
#define MAGENTA     "MAGENTA"
#define MEASURE     "MEASURE"

```

```

#define MENU "MENU"
#define MINSERT "MINSERT"
#define MIRROR "MIRROR"
#define MOVE "MOVE"
#define MSLIDE "MSLIDE"
#define NEW "NEW"
#define OFF "OFF"
#define OFFSET "OFFSET"
#define ON "ON"
#define OOPS "OOPS"
#define ORTHO "ORTHO"
#define OSNAP "OSNAP"
#define PAN "PAN"
#define PEDIT "PEDIT"
#define PLINE "PLINE"
#define PLOT "PLOT"
#define POINT "POINT"
#define POLYGON "POLYGON"
#define PRPLOT "PRPLOT"
#define PURGE "PURGE"
#define QTEXT "QTEXT"
#define QUIT "QUIT"
#define RED "RED"
#define REDO "REDO"
#define REDRAW "REDRAW"
#define REGEN "REGEN"
#define REGENAUTO "REGENAUTO"
#define RENAME "RENAME"
#define RESUME "RESUME"
#define ROTATE "ROTATE"
#define RSCRIPT "RSCRIPT"
#define SAVE "SAVE"
#define SCALE "SCALE"
#define SCRIPT "SCRIPT"
#define SELECT "SELECT"
#define SET "SET"
#define SETVAR "SETVAR"
#define SH "SH"
#define SHAPE "SHAPE"
#define SHELL "SHELL"
#define SKETCH "SKETCH"
#define SNAP "SNAP"
#define SOLID "SOLID"
#define STATUS "STATUS"
#define STRETCH "STRETCH"
#define STYLE "STYLE"
#define TABLET "TABLET"

```

```

#define TEXT      "TEXT"
#define TIME      "TIME"
#define TRACE     "TRACE"
#define TRIM      "TRIM"
#define U         "U"
#define UNDO      "UNDO"
#define UNITS     "UNITS"
#define VIEW      "VIEW"
#define VIEWRES   "VIEWRES"
#define VPOINT    "VPOINT"
#define VSLIDE    "VSLIDE"
#define WBLOCK    "WBLOCK"
#define WHITE     "WHITE"
#define WIDTH     "W"
#define WINDOW    "W"
#define YELLOW    "YELLOW"
#define YES       "Y"
#define ZOOM      "ZOOM"
#endif

```

Лист. А.1. Стандартные команды на английском языке

```

#ifdef FRENCH
#define ALL        "T"
#define APERTURE   "OUVERTUR"
#define ARC        "ARC"
#define AREA       "AIRE"
#define ARRAY      "RESEAU"
#define ATTDEF     "ATTDEF"
#define ATTDISP    "ATTECRAN"
#define ATTEDIT    "ATTEDIT"
#define ATTEXT     "ATTEXT"
#define BASE       "BASE"
#define BLIPMODE   "MARQUES"
#define BLOCK      "BLOC"
#define BLUE       "BLEU"
#define BREAK      "COUPURE"
#define CHAMFER    "CHNFREIN"
#define CHANGE     "CHANGER"
#define CIRCLE     "CERCLE"
#define COLOR      "COUL"
#define COPY       "COPIER"
#define CYAN       "CYAN"
#define DBLIST     "DBLISTE"
#define DELAY      "DELAI"
#define DIM        "COTATION"
#define DIST       "DISTANCE"

```



```

#define DRAGMODE      "EVOLDYN"
#define DXBIN         "CHARGDXB"
#define DXFIN         "CHARGDXF"
#define ELEV          "ELEV"
#define END           "FIN"
#define ERASE         "EFFACER"
#define EXTENTS       "E"
#define FILES         "FICHIERS"
#define FILL          "REPLIR"
#define FILLET        "RACCORD"
#define GREEN         "VERT"
#define GRID          "GRILLE"
#define HIDE          "CACHE"
#define ID            "ID"
#define INSERT        "INSERER"
#define ISOPLANE      "ISOMETR"
#define LAST          "D"
#define LAYER         "PLAN"
#define LCHANGE       "P"
#define LIMITS        "LIMITES"
#define LINE          "LIGNE"
#define LINETYPE      "TYPELIGN"
#define LIST          "LISTE"
#define LOAD          "CHARGER"
#define LTSCALE       "ECHLTP"
#define MAGENTA       "MAGENTA"
#define MENU          "MENU"
#define MIRROR        "MIROIR"
#define MOVE          "MOVE"
#define MSLIDE        "MEMCLICH"
#define NEW           "NOU"
#define OFF           "IN"
#define ON            "AC"
#define OOPS          "REPRISE"
#define ORTHO         "ORTHO"
#define OSNAP         "ACCROBJ"
#define PAN           "PAN"
#define PEDIT         "PEDIT"
#define PLINE         "POLYLIGN"
#define PLOT          "TRACEUR"
#define POINT         "POINT"
#define PRPLOT        "IMPGRAPH"
#define PURGE         "PURGER"
#define QTEXT         "PARTEXTE"
#define QUIT          "QUITTER"
#define RED           "ROUGE"
#define REDRAW        "REDESS"

```

```

#define REGEN          "REGEN"
#define REGENAUTO      "REGNAUTO"
#define RENAME         "RENOMMER"
#define RESUME         "RESUMER"
#define RSCRIPT        "RSCRIPT"
#define SAVE           "SAUVEGRD"
#define SCRIPT         "SCRIPT"
#define SET            "MET"
#define SHAPE          "FORME"
#define SKETCH         "MAINLEV"
#define SNAP           "RESOL"
#define SOLID          "SOLIDE"
#define STATUS         "ETAT"
#define STYLE          "STYLE"
#define TABLET        "TABLETTE"
#define TEXT           "TEXTE"
#define TRACE          "TRACE"
#define UNITS          "UNITES"
#define VIEW           "VUES"
#define VPOINT         "POINTVUE"
#define VSLIDE         "AFFLICH"
#define WBLOCK         "WBLOC"
#define WHITE          "BLANC"
#define WIDTH          "E"
#define WINDOW         "F"
#define YELLOW         "JAUNE"
#define YES            "O"
#define ZOOM           "ZOOM"
#endif

```

Лист. А.2. Трансляция команд с английского языка на французский

```

#ifdef GERMAN
#define ALL            "A"
#define APERTURE       "OEFFNUNG"
#define ARC            "BOGEN"
#define AREA           "FLAECH"
#define ARRAY          "REIHE"
#define ATTDEF         "ATTDEF"
#define ATTDISP        "ATTZEIG"
#define ATTEDIT        "ATTEDIT"
#define ATTEXT         "ATTEXT"
#define BASE           "BASIS"
#define BLIPMODE       "KPMODUS"
#define BLOCK          "BLOCK"
#define BLUE           "BLAU"
#define BREAK          "BRUCH"

```

```

#define CHAMFER      "FACETTE"
#define CHANGE       "AENDERN"
#define CIRCLE       "KREIS"
#define COLOR        "FARBE"
#define COPY         "KOPIEREN"
#define CYAN         "CYAN"
#define DBLIST       "DBLISTE"
#define DELAY        "PAUSE"
#define DIM          "BEM"
#define DIST         "ABSTAND"
#define DIVIDE       "TEILEN"
#define DONUT        "RING"
#define DOUGHNUT     "RING"
#define DRAGMODE     "ZUGMODUS"
#define DXBIN        "DXBIN"
#define DXBOUT       "DXBOUT"
#define DXFIN        "DXFIN"
#define DXBOUT       "DXFOUT"
#define ELEV        "ERHEBUNG"
#define ELLIPSE      "ELLIPSE"
#define END          "ENDE"
#define ARASE        "LOESCHEN"
#define EXPLODE      "URSPRUNG"
#define EXTEND       "DEHNEN"
#define EXTENTS      "Q"
#define FILES        "DATEIEN"
#define FILL         "FUELLEN"
#define FILLET       "ABRUNDEN"
#define GREEN        "GRUEN"
#define GRID         "RASTER"
#define HIDE         "VERDECKT"
#define ID          "ID"
#define IGESOUT      "IGESIN"
#define IGESOUT      "IGESOUT"
#define INSERT       "EINFUEGE"
#define ISOPLANE     "ISOBENE"
#define LAST        "L"
#define LAYER        "LAYER"
#define LCHANGE      "L"
#define LIMITS       "LIMITEN"
#define LINE         "LINIE"
#define LINTYPE      "LINIENTYP"
#define LIST         "LISTE"
#define LOAD         "LADEN"
#define LTSCALE      "LTFAKTOR"
#define MAGENTA      "MAGENTA"
#define MEASURE      "MESSEN"

```

#define MENU	"MENUE"
#define MINSERT	"MEINFUEG"
#define MIRROR	"SPIEGELN"
#define MOVE	"SCHIEBEN"
#define MSLIDE	"MACHDIA"
#define NEW	"NEU"
#define OFF	"AUS"
#define OFFSET	"VERSETZ"
#define ON	"EIN"
#define OOPS	"HOPPLA"
#define ORTHO	"ORTHO"
#define OSNAP	"OFANG"
#define PAN	"PAN"
#define PEDIT	"PEDIT"
#define PLINE	"PLINIE"
#define PLOT	"PLOT"
#define POINT	"PUNKT"
#define POLYGON	"POLYGON"
#define PRPLOT	"PRPLOT"
#define PURGE	"BEREINIG"
#define QTEXT	"QTEXT"
#define QUIT	"QUIT"
#define RED	"ROT"
#define REDO	"ZLOESCH"
#define REDRAW	"NEUZEICH"
#define REGEN	"REGEN"
#define REGENAUTO	"REGENAUTO"
#define RENAME	"UMBENENN"
#define RESUME	"RESUME"
#define ROTATE	"DREHEN"
#define RSCRIPT	"RSCRIPT"
#define SAVE	"SICHERN"
#define SCALE	"VARIA"
#define SCRIPT	"SCRIPT"
#define SELECT	"WAHL"
#define SET	"SETZ"
#define SHAPE	"SYMBOL"
#define SHELL	"SHELL"
#define SKETCH	"SKIZZE"
#define SNAP	"FANG"
#define SOLID	"SOLID"
#define STATUS	"STATUS"
#define STRETCH	"SKRECK"
#define STYLE	"STIL"
#define TABLET	"TABLETT"
#define TEXT	"TEXT"
#define TIME	"ZEIT"

```

#define TRACE      "BAND"
#define TRIM       "STUTZEN"
#define UNITS      "EINHEIT"
#define VIEW       "AUSSCHNT"
#define VIEWRES    "AUFLOES"
#define VPOINT     "APUNKT"
#define VSLIDE     "ZEIGDIA"
#define WBLOCK     "WBLOCK"
#define WHITE      "WEISS"
#define WIDTH      "B"
#define WINDOW     "F"
#define YELLOW     "GELB"
#define YES        "J"
#define ZOOM       "ZOOM"
#endif

```

Лист. А.3. Трансляция команд с английского языка на немецкий

```

#ifdef ITALIAN
#define ALL        "T"
#define APERTURE   "APERTURA"
#define ARC        "ARCO"
#define AREA       "AREA"
#define ARRAY      "SERIE"
#define ATTDEF     "DEFATT"
#define ATTDISP    "VISATT"
#define ATTEDIT    "EDITATT"
#define ATTEXT     "ESTRATT"
#define BASE       "BASE"
#define BLIPMODE   "PUNTINI"
#define BLOCK      "BLOCCO"
#define BLUE       "BLU"
#define BREAK      "SPEZZA"
#define CHAMFER    "SIMA"
#define CHANGE     "CAMBIA"
#define CIRCLE     "CERCHIO"
#define COLOR      "COLORE"
#define COPY       "COPIA"
#define CYAN       "AZZURRO"
#define DBLIST     "LISTABD"
#define DELAY      "PAUSA"
#define DIM        "DIM"
#define DIST       "DIST"
#define DRAGMODE   "TRASCINA"
#define DXBIN      "DXBIN"
#define DXFIN      "DXFIN"
#define ELEV       "ELEV"

```

```

#define END "FINE"
#define ERASE "CANCELLA"
#define EXTENTS "E"
#define FILES "FILE"
#define FILL "RIEMPIE"
#define FILLET "RACCORDO"
#define GREEN "VERDE"
#define GRID "GRIGLIA"
#define HIDE "NASCONDE"
#define ID "ID"
#define INSERT "INSER"
#define ISOPLANE "PIANOASS"
#define LAST "P"
#define LAYER "PIANO"
#define LCHANGE "P"
#define LIMITS "LIMITI"
#define LINE "LINEA"
#define LINETYPE "TLINEA"
#define LIST "LISTA"
#define LOAD "CARICA"
#define LISCALE "SCALATL"
#define MAGENTA "MAGENTA"
#define MENU "MENU"
#define MININSERT "MININSERT"
#define MIRROR "SPECCHIO"
#define MOVE "SPOSTA"
#define MSLIDE "GENDIA"
#define NEW "NUOVO"
#define OFF "OFF"
#define ON "ON"
#define OOPS "OOPS"
#define ORTHO "ORTO"
#define OSNAP "OSNAP"
#define PAN "PAN"
#define PEDIT "EDITPL"
#define PLINE "PLINEA"
#define PLOT "PLOT"
#define POINT "PUNTO"
#define PRPLOT "PLOTST"
#define PURGE "ELIMINA"
#define QTEXT "TESTOV"
#define QUIT "USCIRE"
#define RED "ROSSO"
#define REDRAW "RIDIS"
#define REGEN "RIGEN"
#define REGENAUTO "RIGENAUTO"
#define RENAME "RINOMINA"

```

```

#define RESUME      "RIPRENDE"
#define RSCRIPT     "RSCRIPT"
#define SAVE        "SALVA"
#define SCRIPT      "SCRIPT"
#define SET         "PIANOCOR"
#define SHAPE       "FORMA"
#define SHELL       "SHELL"
#define SKETCH      "SCHIZZO"
#define SNAP        "SNAP"
#define SOLID       "POLIGONO"
#define STATUS      "STATO"
#define STYLE       "STILE"
#define TABLET     "TAVOLET"
#define TEXT        "TESTO"
#define TRACE       "TRACCIA"
#define UNITS       "UNITA"
#define VIEW        "VISTA"
#define VPOINT      "PVISTA"
#define VSLIDE      "VISDIA"
#define WBLOCK      "MBLOCCO"
#define WHITE       "BIANCO"
#define WIDTH       "G"
#define WINDOW      "F"
#define YELLOW      "GIALLO"
#define YES         "S"
#define ZOOM        "ZOOM"
#endif

```

Лист. А.4. Трансляция команд с английского языка на итальянский

```

#ifdef SPANISH
#define ALL          "T"
#define APERTURE     "APERTURA"
#define ARC          "ARCO"
#define AREA         "AREA"
#define ARRAY        "MATRIZ"
#define ATTDEF       "DEFATR"
#define ATTDISP      "VISUATR"
#define ATTEDIT      "EDITATR"
#define ATTEXT       "EXTRATR"
#define BASE         "BASE"
#define BLIPMODE     "MARCAAUX"
#define BLOCK        "BLOQUE"
#define BLUE         "AZUL"
#define BREAK        "PARTE"
#define CHAMFER      "CHAFLAN"
#define CHANGE       "CAMBIA"

```

```

#define CIRCLE      "CIRCULO"
#define COLOR       "COLOR"
#define COPY        "COPIA"
#define CYAN        "CIANO"
#define DBLIST      "LISTBD"
#define DELAY       "RETARDA"
#define DIM         "ACOTA"
#define DIST        "DIST"
#define DRAGMODE    "ARRASTRE"
#define DXBIN       "CARGADXB"
#define DXFIN       "CARGADXF"
#define ELEV        "ELEV"
#define END         "FIN"
#define ERASE       "BORRA"
#define EXTENTS     "E"
#define FILES       "FICHEROS"
#define FILL        "RELLENA"
#define FILLET      "EMPALME"
#define GREEN       "VERDE"
#define GRID        "TRAMA"
#define HIDE        "OCULTA"
#define ID          "ID"
#define INSERT      "INSER"
#define ISOPLANE    "ISOPLANO"
#define LAST        "P"
#define LAYER       "CAPA"
#define LCHANGE     "C"
#define LIMITS      "LIMITES"
#define LINE        "LINEA"
#define LINETYPE    "TIPOLIN"
#define LIST        "LIST"
#define LOAD        "CARGA"
#define LTSCALE     "ESCALATL"
#define MAGENTA     "MAGENTA"
#define MENU        "MENU"
#define MIRROR      "SIMETRIA"
#define MOVE        "DESPLAZA"
#define MSLIDE      "SACAFOTO"
#define NEW         "CREA"
#define OFF         "DES"
#define ON          "ACT"
#define OOPS        "RECUPERA"
#define ORTHO       "ORTO"
#define OSNAP       "REFENT"
#define PAN         "ENCUADRE"
#define PEDIT       "EDITPOL"
#define PLINE       "POL"

```



```

#define PLOT      "SALTRAZ"
#define POINT     "PUNTO"
#define PRPLOT    "SALIMPR"
#define PURGE     "LIMPIA"
#define QTEXT     "LOCTEXTO"
#define QUIT      "QUITA"
#define RED       "ROJO"
#define REDRAW    "REDIBUJA"
#define REGEN     "REGEN"
#define REGENAUTO "REGENAUTO"
#define RENAME    "RENOMBRA"
#define RESUME    "REANUDA"
#define RSCRIPT   "RELGUION"
#define SAVE      "SALVA"
#define SCRIPT    "GUION"
#define SET       "DEF"
#define SHAPE     "FORMA"
#define SHELL     "PROGREXT"
#define SKETCH    "BOCETO"
#define SNAP      "FORZCOORD"
#define SOLID     "SOLIDO"
#define STATUS    "ESTADO"
#define STYLE     "ESTILO"
#define TABLET   "TABLERO"
#define TEXT      "TEXTO"
#define TRACE     "TRAZO"
#define UNITS     "UNIDADES"
#define VIEW      "VISTA"
#define VPOINT    "PTOVISTA"
#define VSLIDE    "MIRAFOTO"
#define WBLOCK    "BLADISCO"
#define WHITE     "BLANCO"
#define WIDTH     "G"
#define WINDOW    "I"
#define YELLOW    "AMARILLO"
#define YES       "S"
#define ZOOM      "FOCAL"
#endif

```

Лист. А.5. Трансляция команд с английского языка на испанский

Приложение Б

ПАРАМЕТРЫ СЕКЦИИ ЗАГОЛОВКА DXF-ФАЙЛА

В табл. Б.1 представлены параметры, содержащиеся в типичном DXF-файле. Написание имен в точности совпадает с их написанием в DXF-файле. Появлению имени в файле предшествует маркер 9. После имени следуют маркеры, помечающие значения, соответствующие данному имени. В таблице дано краткое описание каждого значения и указаны соответствующие маркеры. В некоторых местах также приведены возможные значения параметров.

Таблица Б.1. Параметры секции заголовка

\$ACADVER	1	Версия базы данных системы AutoCAD
\$ANGBASE	50	Установка точки отсчета (направления) угла 0
\$ANGDIR	70	Направление поворота при отсчете угла 0 = против часовой стрелки 1 = по часовой стрелке
\$ATTMODE	70	Режим видимости атрибутов 0 = атрибуты не отображаются 1 = нормальный, отображаются только "видимые" атрибуты 2 = отображаются все атрибуты
\$AUNITS	70	Формат единиц измерения для углов 1 = градусы в десятичном виде 2 = градусы/минуты/секунды 3 = градусы 4 = радианы 5 = геодезический формат
\$AUPREC	70	Точность представления угла (число знаков в представлении дробной части значения)
\$AXISMODE	70	Отображение осей координат (0 = OFF, 1 = ON)
\$AXISUNIT	10	Цена деления по оси координат X
	20	Цена деления по оси координат Y
\$BLIPMODE	70	Отображение вспомогательных отметок (0 = OFF, 1 = ON)
\$CECOLOR	70	Текущий цвет объекта (0 = BYBLOCK, 256 = BYLAYER)
\$CELTYPE	70	Текущий тип линии
\$CLAYER	8	Имя слоя
\$COORDS	70	Режим отображения координат 0 = статическое отображение 1 = непрерывное динамическое отображение 2 = при выполнении условия
\$DIMASZ	40	Длина размерных стрелок
\$DIMBLK	1	Имя блока размерных стрелок
\$DIMCEN	40	Размер отметки (маркера) центра

\$DIMDLE	40	Длина концов размерной линии, выходящих за выносные линии
\$DIMDLI	40	Интервал между последовательными размерными линиями
\$DIMEXE	40	Длина концов выносных линий, выходящих за размерные линии
\$DIMEXO	40	Расстояние между границей объекта и началом выносной линии
\$DIMLIM	70	Размерные линии (0 = OFF, 1 = ON)
\$DIMRND	40	Точность задания размеров
\$DIMSCALE	40	Линейный масштаб для размеров
\$DIMSE1	70	Подавление вычерчивания 1-ой выносной линии (0 = OFF, 1 = ON)
\$DIMSE2	70	Подавление вычерчивания 2-ой выносной линии (0 = OFF, 1 = ON)
\$DIMTAD	70	Надпись помещается над размерной линией (0 = OFF, 1 = ON)
\$DIMTIH	70	Надпись, помещенная между выносными линиями, вычерчивается горизонтально (0 = OFF, 1 = ON)
\$DIMTM	40	Значение минусового допуска
\$DIMTON	70	Надпись, помещенная за выносными линиями, вычерчивается горизонтально (0 = OFF, 1 = ON)
\$DIMTOL	70	Размерные надписи включают допуски (0 = OFF, 1 = ON)
\$DIMTP	40	Значение плюсового допуска
\$DIMTSZ	40	Размер рисок (используемых вместо стрелок)
\$DIMTXT	40	Высота текста в размерной надписи
\$DIMZIN	70	Отображение нулевых размеров (0 = OFF, 1 = ON)
\$DRAGMODE	70	Режим отслеживания 0 = OFF 1 = ON 2 = Auto
\$ELEVATION	40	Текущий уровень возвышения
\$EXTMAX	10	Габариты чертежа - координата X верхнего правого угла области изображений
	20	Габариты чертежа - координата Y верхнего правого угла области изображений
	30	Габариты чертежа - координата Z верхнего правого

	угла области изображений (если не равна 0.0)
\$EXTMIN	10 Габариты чертежа - координата X нижнего левого угла области изображений
	20 Габариты чертежа - координата Y нижнего левого угла области изображений
	30 Габариты чертежа - координата Z нижнего левого угла области изображений (если не равна 0.0)
\$FILLETRAD	40 Радиус сопряжения
\$FILLMODE	70 Режим закрашки (0 = OFF, 1 = ON)
\$GRIDMODE	70 Отображение координатной сетки (0 = OFF, 1 = ON)
\$GRIDUNIT	10 Шаг координатной сетки по оси X
	20 Шаг координатной сетки по оси Y
\$INSBASE	10 Базовая координата X точки вставки
	20 Базовая координата Y точки вставки
	30 Базовая координата Z точки вставки (если не равна 0.0)
\$LIMCHECK	70 Контроль границ чертежа (0 = OFF, 1 = ON)
\$LIMMAX	10 Границы чертежа - координата X верхнего правого угла области
	20 Границы чертежа - координата Y верхнего правого угла области
\$LIMMIN	10 Границы чертежа - координата X нижнего левого угла области
	20 Границы чертежа - координата Y нижнего левого угла области
\$LTSCALE	40 Линейный масштаб для типов линии
\$LUNITS	70 Формат представления координат и расстояний 1 = Научный (экспоненциальный) 2 = Десятичный 3 = Технический 4 = Архитектурный
\$LUPREC	70 Точность представления координат и расстояний (число знаков в представлении дробной части значения)
\$MENU	1 Имя файла текущего меню
\$ORTHOMODE	70 Режим "ортогонального" рисования (0 = OFF, 1 = ON)
\$OSMODE	70 Режим нахождения заданной точки объекта

\$PDMODE	70	Режим отображения точки
\$PDSIZE	70	Размер точки на экране
\$PLINEWID	40	Текущая (по умолчанию) ширина полилинии
\$QTEXTMODE	70	Режим быстрого отображения текста (0 = OFF, 1 = ON)
\$REGENMODE	70	Режим автоматической регенерации (0 = OFF, 1 = ON)
\$SKETCHINC	40	Текущее приращение при ручной прорисовке
\$SNAPANG	50	Угол поворота сетки привязки
\$SNAPBASE	10	Базовая координата X сетки привязки
	20	Базовая координата Y сетки привязки
\$SNAPISOPAIR	70	Вид в изометрии 0 = Слева 1 = Сверху 2 = Справа
\$SNAPMODE	70	Режим установки разрешающей способности (0 = OFF, 1 = ON)
\$SNAPSTYLE	70	Тип сетки привязки 0 = стандартная 1 = изометрическая
\$SNAPUNIT	10	Интервал разрешения по оси X
	20	Интервал разрешения по оси Y
\$TEXTSIZE	40	Текущая (по умолчанию) высота текста
\$THICKNESS	40	Текущая высота (толщина) объектов
\$TRACEWID	40	Текущая ширина полосы
\$USERI1	70	Целочисленное значение, заданное пользователем
\$USERI2	70	Целочисленное значение, заданное пользователем
\$USERI3	70	Целочисленное значение, заданное пользователем
\$USERI4	70	Целочисленное значение, заданное пользователем
\$USERI5	70	Целочисленное значение, заданное пользователем
\$USERR1	40	Действительное значение, заданное пользователем
\$USERR2	40	Действительное значение, заданное пользователем
\$USERR3	40	Действительное значение, заданное пользователем
\$USERR4	40	Действительное значение, заданное пользователем
\$USERR5	40	Действительное значение, заданное пользователем
\$VIEWCTR	10	Положение текущего вида (его центр) на экране -

	координата X
10	Положение текущего вида (его центр) на экране - координата Y
\$VIEWDIR	10 Текущая точка наблюдения - координата X
	20 Текущая точка наблюдения - координата Y
	30 Текущая точка наблюдения - координата Z (если не равна 0.0)
\$VIEWSIZE	40 Высота текущего вида на экране

Приложение В

СТРУКТУРА ДАННЫХ ДЛЯ СЕКЦИИ ЗАГОЛОВКА DXF-ФАЙЛА

На указанном ниже листинге представлена расширенная структура данных, используемая для генерации параметров секции заголовка DXF-файла. В гл. 8 рассматриваются использование этих параметров и реализация программ обработки.

```

hdr_block      header[] =
{ {"$ACADVER",   STRING, (void *)acadvver},
  {"$ANGBASE",   ANGLE,  (void *)&angbase},
  {"$ANGDIR",    INTEGER, (void *)&angdir},
  {"$AXISMODE",  INTEGER, (void *)&axismode},
  {"$AXISUNIT",  COORDX,  (void *)&axisunitx},
  { NULL,        COORDY,  (void *)&axisunity},
  {"$AUNITS",    INTEGER, (void *)&aunits},
  {"$AUPREC",    INTEGER, (void *)&auprec},
  {"$ATTMODE",   INTEGER, (void *)&attmode},
  {"$BLIPMODE",  INTEGER, (void *)&blipmode},
  {"$CLAYER",    LAYER,   (void *)&clayer},
  {"$COORDS",    INTEGER, (void *)&coords},
  {"$DIMASZ",    REAL,    (void *)&dimasz},
  {"$DIMBLK",    STRING,  (void *)&dimblk},
  {"$DIMDLI",    REAL,    (void *)&dimdli},
  {"$DIMEXE",    REAL,    (void *)&dimexe},
  {"$DIMEXO",    REAL,    (void *)&dimexo},
  {"$DIMCEN",    REAL,    (void *)&dimcen},
  {"$DIMDLE",    REAL,    (void *)&dimdle},
  {"$DIMLIN",    INTEGER, (void *)&dimlim},
  {"$DIMRND",    REAL,    (void *)&dimrnd},
  {"$DIMSCALE",  REAL,    (void *)&dimscale},

```

```

{"$DIMSE1",    INTEGER, (void *)&dimse1},
{"$DIMSE2",    INTEGER, (void *)&dimse2},
{"$DIMENTAD",  INTEGER, (void *)&dimtad},
{"$DIMENTIH",  INTEGER, (void *)&dimtih},
{"$DIMENTM",   REAL,   (void *)&dimtm},
{"$DIMENTOH",  INTEGER, (void *)&dimtoh},
{"$DIMENTOL",  INTEGER, (void *)&dimtol},
{"$DIMENTP",   REAL,   (void *)&dimtp},
{"$DIMTSZ",    REAL,   (void *)&dimtsz},
{"$DIMENTXT",  REAL,   (void *)&dimtxt},
{"$DIMZIN",    INTEGER, (void *)&dimzin},
{"$DRAGMODE",  INTEGER, (void *)&dragmode},
{"$ELEVATION", REAL,   (void *)&elevation},
{"$EXTMAX",    COORDX, (void *)&extmaxx},
{ NULL ,      COORDY, (void *)&extmaxy},
{ NULL ,      COORDZ, (void *)&extmaxz},
{"$EXTMIN",    COORDX, (void *)&extminx},
{ NULL ,      COORDY, (void *)&extminy},
{ NULL ,      COORDZ, (void *)&extminz},
{"$FILLETRAD", REAL,   (void *)&filletrad},
{"$FILLMODE",  INTEGER, (void *)&fillmode},
{"$GRIDMODE",  INTEGER, (void *)&gridmode},
{"$GRIDUNIT",  COORDX, (void *)&gridunitx},
{ NULL ,      COORDY, (void *)&gridunity},
{"$INSBASE",   COORDX, (void *)&insbasex},
{ NULL ,      COORDY, (void *)&insbasey},
{ NULL ,      COORDZ, (void *)&insbasez},
{"$LIMCHECK",  INTEGER, (void *)&limcheck},
{"$LIMMIN",    COORDX, (void *)&limminx},
{ NULL ,      COORDY, (void *)&limminy},
{"$LIMMAX",    COORDX, (void *)&limmaxx},
{ NULL ,      COORDY, (void *)&limmaxy},
{"$LTSCALE",   REAL,   (void *)&ltscale},
{"$LUNITS",    INTEGER, (void *)&lunits},
{"$LUPREC",    INTEGER, (void *)&luprec},
{"$MENU",      STRING, (void *)&menu},
{"$ORTHOMODE", INTEGER, (void *)&orthomode},
{"$OSMODE",    INTEGER, (void *)&osmode},
{"$PDMODE",    INTEGER, (void *)&pdmode},
{"$PDSIZE",    REAL,   (void *)&pdsizex},
{"$PLINEWID",  REAL,   (void *)&plinewid},
{"$QTEXTMODE", INTEGER, (void *)&qtextmode},
double        extminx;
double        extminy;
double        extminz;
double        fastzoom;
double        filletrad;

```

int	fillmode;
int	gridmode;
double	gridunitx;
double	gridunity;
double	insbasex;
double	insbasey;
double	insbasez;
int	limcheck;
double	limmaxx;
double	limmaxy;
double	limminx;
double	limminy;
double	ltscale;
int	lunits;
int	luprec;
char	menu[12];
int	orthomode;
int	ostmode;
int	pdmode;
double	pdsizex;
double	pdsizey;
int	plinewid;
int	qtextmode;
int	regenmode;
double	sketchinc;
double	skpoly;
double	snapang;
double	snaphasex;
double	snaphasey;
int	snaphaspair;
int	snapmode;
int	snapstyle;
double	snapunitx;
double	snapunty;
double	tdcreate;
double	tdindwg;
double	tdupdate;
double	tdusrtimer;
double	textsize;
double	textstyle;
double	thickness;
double	tracewid;
int	useri1;
int	useri2;
int	useri3;
int	useri4;
int	useri5;
double	userr1;


```

double          userr2;
double          userr3;
double          userr4;
double          userr5;
double          usrtimer;
double          viewctrx;
double          viewctry;
double          viewctrz;
double          viewdirx;
double          viewdiry;
double          viewdirz;
double          viewsizex;
double          viewsizy;
double          viewsizz;
double          zoompct;

#define STRING      1
#define INTEGER     2
#define REAL        3
#define COORDX      4
#define COORDY      5
#define COORDZ      6
#define ANGLE       7
#define LAYER       8

```

```

typedef
struct
{
    char          *name;
    int           type;
    void          *var;
}                hdr_block;

```

```

char            acadver[16];
int             angbase;
double          angdir;
int             attmode;
int             aunits;
int             auprec;
int             axismode;
double          axisunitx;
double          axisunity;
int             blipmode;
int             cecolor;
int             celtype;
double          chamfera;
double          chamferb;
int             clayer;
int             coords;
double          dimasz;

```

```

char                dimblk[40];
double              dimcen;
double              dimdle;
double              dimdli;
double              dimexe;
double              dimexo;
double              dimrnd;
double              dimscl;
double              dimtm;
int                 dimlim;
int                 dimse1;
int                 dimse2;
int                 dimtad;
int                 dimtih;
int                 dimtoh;
int                 dimtol;
int                 dimzin;
double              dimtp;
double              dimtsz;
double              dimtxt;
int                 dragmode;
double              elevation;
double              extmaxx;
double              extmaxy;
double              extmaxz;
{"$REGENMODE",      INTEGER, (void *)&regenmode},
{"$SKETCHINC",      REAL,    (void *)&sketchinc},
{"$SNAPANG",        ANGLE,   (void *)&snapang},
{"$SNAPBASE",       COORDX,  (void *)&snapbasex},
{ NULL,             COORDY,  (void *)&snapbasey},
{"$SNAPISOPAIR",    INTEGER, (void *)&snapisopair},
{"$SNAPMODE",       INTEGER, (void *)&snapmode},
{"$SNAPSTYLE",      INTEGER, (void *)&snapstyle},
{"$SNAPUNIT",       COORDX,  (void *)&snapunitx},
{ NULL,             COORDY,  (void *)&snapunity},
{"$THICKNESS",      REAL,    (void *)&thickness},
{"$TEXTSIZE",       REAL,    (void *)&textsize},
{"$TRACEWID",       REAL,    (void *)&tracewid},
{"$USERI1",         INTEGER, (void *)&useri1},
{"$USERI2",         INTEGER, (void *)&useri2},
{"$USERI3",         INTEGER, (void *)&useri3},
{"$USERI4",         INTEGER, (void *)&useri4},
{"$USERI5",         INTEGER, (void *)&useri5},
{"$USERR1",         REAL,    (void *)&userr1},
{"$USERR2",         REAL,    (void *)&userr2},
{"$USERR3",         REAL,    (void *)&userr3},
{"$USERR4",         REAL,    (void *)&userr4},

```

```

{"$USERR5",    REAL,    (void *)&userr5},
{"$VIEWCTR",   COORDX,  (void *)&viewctrx},
{ NULL,        COORDY,  (void *)&viewctry},
{ NULL,        COORDZ,  (void *)&viewctrz},
{"$VIEWDIR",   COORDX,  (void *)&viewdirx},
{ NULL,        COORDY,  (void *)&viewdiry},
{ NULL,        COORDZ,  (void *)&viewdirz},
{"$VIEWSIZE",  REAL,    (void *)&viewsize},
{"$ZOOMPCT",   INTEGER, (void *)&zoompct},
{ NULL, 0,     NULL}};

```

Лист. В.1. Структура данных секции заголовка DXF-файла

Приложение Г

РАБОЧИЕ ОПРЕДЕЛЕНИЯ ТИПОВ ДАННЫХ

На листингах представлены определения типов данных, которые были взяты автором из исходных текстов реальных работающих программ. Определения представляют собой структуры данных для таблиц, блоков и графических объектов, с которыми работает система AutoCAD. Эти определения типов могут отличаться от приведенных в примерах книги, последние часто приводились в качестве иллюстраций. Здесь же даны более строгие рабочие определения типов данных.

Обратите внимание, что на листинге для типа данных `text_type` в структуре данных есть несколько полей, не связанных с объектом `TEXT`. Эти поля появились из-за наличия таких команд, как `ATTDEF` и `ATTRIB`. Все параметры сведены в одну структуру. Та же структура данных используется и для объектов `TRACE` и `SOLID`.

```

typedef
struct dash_type
{
    struct dash_type    *next;
    double              length;
}                      dash_type;

typedef
struct ltype_type
{
    struct ltype_type    *next;
    char                 *name;
    char                 *descript;
    int                  flags;
}

```

```

int            alignment;
int            dashes;
double         length;
struct dash_type *dash_length;
}              {type_type;

```

Лист. Г.1. Определение типа данных для типа линии

```

typedef
struct layer_type
{
    struct layer_type *next;
    char *name;
    int flags;
    int color;
    int used;
    char *lt;
} layer_type;

```

Лист. Г.2. Определение типа данных для слоя

```

typedef
struct style_type
{
    struct style_type *next;
    char *name;
    int flags;
    int gen_flags;
    double txt_scale;
    double height;
    double oblique;
    double last_height;
    char *font;
    char *big_font;
} style_type;

```

Лист. Г.3. Определение типа данных для стиля

```

typedef
struct view_type
{
    struct view_type *next;
    char *name;
    int flags;
    double v_height;
    double v_width;
}

```

```

double          v_ctr_x;
double          v_ctr_y;
double          v_dir_x;
double          v_dir_y;
double          v_dir_z;
}               view_type;

```

Лист. Г.4. Определение типа данных для вида

```

typedef
struct block_type
{
    struct block_type    *next;
    int                  what;
    char                 *name;
    int                  flags;
    layer_type           *lyr;
    ltype_type           *ltype;
    double               base_x;
    double               base_y;
    void                 *ent;
}                       block_type;

```

Лист. Г.5. Определение типа данных для блока

```

typedef
struct line_type
{
    struct line_type     *next;
    int                  what;
    char                 *lyr;
    char                 *lt;
    double               sx;
    double               sy;
    double               ex;
    double               ey;
    double               w1;
    double               w2;
    double               bulge;
    char                 last_one;
    struct line_type     *vertex;
}                       line_type;

```

Лист. Г.6. Определение типа данных для линии

```

typedef
struct point_type
{
    struct point_type    *next;
    int                  what;
    char                 *lyr;
    char                 *lt;
    double               px;
    double               py;
}                        point_type;

```

Лист. Г.7. Определение типа данных для точки

```

typedef
struct circle_type
{
    struct circle_type    *next;
    int                  what;
    char                 *lyr;
    char                 *lt;
    double               cx;
    double               cy;
    double               r;
}                        circle_type;

```

Лист. Г.8. Определение типа данных для окружности

```

typedef
struct arc_type
{
    struct arc_type        *next;
    int                  what;
    char                 *lyr;
    char                 *lt;
    int                  flags;
    double               acx;
    double               acy;
    double               ar;
    double               sa;
    double               ea;
    double               width;;
}                        arc_type;

```

Лист. Г.9. Определение типа данных для дуги

```

typedef
struct trace_type
{
    struct trace_type    *next;
    int                  what;
    char                  *lyr;
    char                  *lt;
    double                x1;
    double                y1;
    double                x2;
    double                y2;
    double                x3;
    double                y3;
    double                x4;
    double                y4;
}
                        trace_type;

```

Лист. Г.10. Определение типа данных для полосы

```

typedef
struct text_type
{
    struct text_type    *next;
    int                  what;
    char                  *lyr;
    char                  *lt;
    double                ix;
    double                iy;
    double                height;
    char                  *text;
    char                  *prompt;
    char                  *def_value;
    char                  *tag;
    int                   field_length;
    int                   att_flags;
    double                angle;
    double                x_scale;
    double                oblique;
    style_type            *font;
    int                   gen_flags;
    int                   justify;
    double                ax;
    double                ay;
}
                        text_type;

```

Лист. Г.11. Определение типа данных для текста

```

typedef
struct shape_type
{
    struct shape_type    *next;
    int                  what;
    char                 *lyr;
    char                 *lt;
    double               ix;
    double               iy;
    double               size;
    char                 *shape_name;
    double               angle;
    double               x_scale;
    double               oblique;
}
                        shape_type;

```

Лист. Г.12. Определение типа данных для графического символа (формы)

```

typedef
struct
{
    struct insert_type    *next;
    int                  what;
    char                 *lyr;
    char                 *lt;
    int                  att_flags;
    char                 *b;
    double               ix;
    double               iy;
    double               angle;
    double               x_scale;
    double               y_scale;
    double               z_scale;
    int                  rows;
    int                  columns;
    double               r_space;
    double               c_space;
}
                        insert_type;

```

Лист. Г.13. Определение типа данных для вставки

ОГЛАВЛЕНИЕ

Предисловие к русскому изданию.....	5
Введение	7
Глава 1. Структуры данных и методы программирования.....	10
Структуры данных.....	10
Представление линий и точек	11
Двоичные деревья.....	22
Украшение двоичного дерева.....	28
Организация хранения строк.....	34
Таблица имен.....	40
Заключение	42
Глава 2. Другие методы программирования	43
Распознавание слов	44
Трансляция во внутреннее представление	59
Программная реализация конечных автоматов.....	65
Заключение	70
Глава 3. Программное окружение системы AutoCAD	71
Команды без параметров.....	72
Команды с одним параметром, выполняющие роль переключателя.....	73
Команды с одним параметром — именем файла	76
Другие команды с одним параметром.....	78
Команды построения простых объектов	79
Команды построения сложных объектов	83
Команда LAYER и ее модификаторы	85
Соглашения операционной системы MS-DOS.....	87
Глава 4. Структура файлов системы AutoCAD	91
Глава 5. Формат DXF-файлов.....	94
Секция заголовка (обзор)	94
Секция таблиц (обзор)	95
Секция блоков (обзор)	96
Секция объектов (обзор).....	96
Синтаксический разбор DXF-файлов.....	97
Считывание файлов формата DXF	102
Основные подпрограммы и структуры данных для разбора DXF-файлов.....	104
Центральная управляющая программа синтаксического разбора.....	111
Чтение секции заголовка HEADER	113
Чтение секции таблиц TABLES	117
Таблица типов линий LINETYPE.....	120
Таблица слоев LAYER.....	128
Таблица стилей начертания STYLE	133
Таблица видов VIEW	138
Чтение секции объектов ENTITIES.....	143
Чтение секции блоков BLOCKS	158

Таблицы описания компонентов DXF-файлов.....	166
Глава 6. Атрибуты	173
Определения атрибутов	184
Атрибуты и структура данных	186
Глава 7. Управление системой AutoCAD с помощью пакетных файлов.....	199
Пакетные файлы. Введение.....	199
Выполнение пакетного файла.....	200
Генерация пакетных файлов	203
Национальные варианты системы AutoCAD	205
Создание пакетных файлов с помощью редактора текстов	209
Секция заголовка.....	214
Секция таблиц.....	223
Типы линий.....	223
Слои	225
Стиль начертания текста	226
Виды.....	228
Секции объектов и блоков	228
Почему пакетные файлы?.....	232
Глава 8. Ввод данных в формате DXF- и DXB-файлов.....	244
Неполная генерация DXF-файлов.....	244
Секция заголовка HEADER.....	246
Секция таблиц TABLES.....	251
Генерация секции блоков BLOCKS.....	257
Секция объектов ENTITIES.....	258
Использование DXF-файла для создания модифицированного DXF-файла	273
Файлы формата PGP и DXB.....	277
Приложение А. Национальные версии системы AutoCAD.....	301
Приложение Б. Параметры секции заголовка DXF-файла	314
Приложение В. Структура данных для секции заголовка DXF-файла.....	318
Приложение Г. Рабочие определения типов данных.....	323

Производственно-техническое издание

Джамп Деннис

AutoCAD. Программирование

Заведующий редакцией Ю.Г. Ивашов
Редактор Л.Ю. Камочкина
Художественный редактор В.И. Мусиенко
Обложка художника В.Д. Лысова
Технический редактор А.Н. Золотарева
Корректор Т.В. Дземидович

ИБ № 2345

Подписано в печать 27.03.92 Формат 60×88/16 Бумага офсетная №2 Гарнитура Таймс
Печать офсетная Усл.печ.л. 20,58 Усл.кр.-отт. 20,95 Уч.изд.л. 16,33
Тираж 50 000 экз. Изд. N 23398 Зак. № 6496 С - 069

Издательство "Радио и связь". 101000 Москва, Почтамт, а/я 693

Ордена Октябрьской Революции и ордена Трудового Красного Знамени МПО "Первая
Образцовая типография" Министерства печати и информации РФ. 113054, Москва,
Валовая, 28.

Вниманию предприятий и организаций!



**Научно-технический центр
"КОНСТРУКТОР"**

Дилер и авторизованный
разработчик фирмы **AUTODESK**

НТЦ "КОНСТРУКТОР" поставляет программно-аппаратные средства для комплексной автоматизации сквозного цикла **"ПРОЕКТИРОВАНИЕ-КОНСТРУИРОВАНИЕ-ИЗГОТОВЛЕНИЕ"**.

НТЦ "КОНСТРУКТОР" предлагает лицензионные версии системы **AutoCAD** и дополнительное программное обеспечение, значительно повышающее эффективность работ конструктора:

ncAD — подготовка документации в соответствии с ГОСТами СССР;
sCHECK — конструкторские расчеты при проектировании в среде **AutoCAD**;
AutoSIZE — расчет размерных цепей;
AutoMASS — расчет массово-инерционных характеристик деталей и сборок;
vPACK — компоновка и расчет нагрузок на опоры транспортных средств;
ERGONOM — система эргономического проектирования места оператора;
AutoSTACK — набор стандартных элементов и нормалей, составление спецификаций;
xBASE — система создания и использования графических баз данных типовых и (или) стандартных элементов;
d-Adjuster — параметризатор в среде **AutoCAD**;
gMENU — создание и ведение графических библиотек;
BEARING — графическая база данных наиболее употребимых видов подшипников;
AutoELECTRO — специализированные графические базы данных в области электротехники, электроники и гидравлики;
AdmitFit — справочная система по допускам и посадкам;
xLOCK — защита LISP-программ от несанкционированного копирования;
CATOK — технологическое расширение **AutoCAD** для токарной обработки.

Для инженеров-технологов поставляется система **Тиграс 3.0**, программно-совместимая с **Perp 2.3** и имеющая собственный интерфейс с **AutoCAD**.

НТЦ "КОНСТРУКТОР" обучает специалистов работе в системах **AutoCAD**, **COSMOS/M**, **Perp**, **Тиграс**. Обучение проводится по двум циклам: для начинающих и для подготовленных пользователей (адаптация и развитие систем).

НТЦ "КОНСТРУКТОР" готов поставить все необходимое оборудование для автоматизации проектно-конструкторских работ.

Наш адрес:

117449, Москва, ул. Шверника, 14/2

НТЦ "КОНСТРУКТОР"

Телефон: 126-11-54, 477-12-68

Факс: 1297381

Надеемся на плодотворное сотрудничество!



**ПРОГРАММНЫЕ ПРОДУКТЫ
ФИРМЫ "ТОР" ПОМОГУТ РЕШИТЬ
ВАШИ ПРОБЛЕМЫ В ОБЛАСТИ
РАСЧЕТОВ, КОНСТРУИРОВАНИЯ,
ТЕХНОЛОГИЧЕСКОЙ ПОДГОТОВКИ
И ОРГАНИЗАЦИИ ПРОИЗВОДСТВА**

**Программно-технический комплекс
конструкторско-технологической подготовки производства
ТЕХТРАН+**

Широкий выбор программных средств для разработки чертежей по ЕСКД, параметризации, расчету зубчатых колес, ременных, реечных, червячных передач, шкивов, пультов электроавтоматики, сварных конструкций.

Разработка управляющих программ для более, чем 200 отечественных и зарубежных моделей станков и систем ЧПУ.

Удобная профессиональная среда для проектирования различных технологий и оформления технологической документации, в том числе: токарная обработка, обработка на свелильно-расточных, фрезерных станках и ОЦ, электроэрозионная обработка, раскрой материала на машинах газовой, плазменной и лазерной резки.

Графическая отладка, текстовое редактирование и ведение базы данных управляющих программ.

Адаптеры связи ПЭВМ и устройств ввода/вывода перфоленты.

Реализована полная совместимость AutoCAD со всеми подсистемами комплекса ТЕХТРАН+ по DXF-интерфейсу.

Сопровождение, адаптация к условиям заказчика.

**Поставляется система AutoCAD -
лицензионный продукт фирмы Autodesk**

**Интерактивная система на основе метода конечных элементов
ELCUT**

Применяется для моделирования электрических, магнитных, тепловых полей, диффузии напряженно-деформированного состояния механических конструкций на IBM PC.

Двумерные задачи с 100 000 неизвестных; Графические пре- и постпроцессоры; Интерфейс с AutoCAD; Автоматическая триангуляция; Скорость решения в 10-20 раз выше, чем у известных FEA-систем (например COSMOS-M, ANSYS-PC).

Локальная вычислительная сеть АЛИСА+ (TORLAN)

Простота, гибкость, эффективность

Соединит в единую гетерогенную сеть ЭВМ типа IBM PC/XT/AT, ЕС-1840/1841, СМ-4/1420, ДБК и ЕС ЭВМ.

*Наш адрес: 198052, Санкт-Петербург, Угловой пер., д.6, НПиКК ТОР
Тел. (812) 292 7210, 292 1965, 295 7671, 295 6998, факс (812) 1101334*

Книга «Введение в AutoCAD»

является учебно-методическим пособием по обучению работе в системе AutoCAD. Цель пособия — дать основные приёмы работы с системой AutoCAD. Пособие в первую очередь предназначено для начинающих пользователей, которые не имеют достаточно большого опыта работы на персональном компьютере. Это могут быть конструкторы, архитекторы, дизайнеры, студенты вузов и др. Читатель, изучивший все примеры и выполнивший все упражнения, не только приобретёт навыки в работе с системой AutoCAD, но и научится настраивать систему на свою область применения. Данное методическое пособие не заменяет документацию по системе AutoCAD, оно лишь расставляет опорные точки для быстрого «вживания» в систему и освоения её возможностей. В книге приводятся примеры создания экранных и графических меню при решении реальных конструкторских задач. В пособии содержатся вопросы, предназначенные для закрепления и практического освоения изученных в документации разделов, приводятся примеры работы в системе AutoCAD.

Методическое пособие ориентировано в основном на 10-ю версию AutoCAD, но может быть использовано и при работе с младшими версиями.

Стоимость книги 25 рублей

Заявки на приобретение высылать по адресу:

129629, Москва, Новоалексеевская, 13, предприятие "Информация и технология"

Наиболее полная и наиболее современная информация о персональных компьютерах IBM и программном обеспечении Microsoft из фирменных технических руководств, справочников и книг собрана в электронном справочнике

"ЭНЦИКЛОПЕДИЯ IBM"

Команды MS DOS, BIOS, Windows, распределение памяти, регистры, перемиčky, соединения - вся необходимая информация для профессиональной работы представлена в удобной форме и снабжена перекрестными ссылками. Возможен резидентный режим работы.

Цена:

"Программно-аппаратная организация IBM" (3 ГМД) - 3000 руб.

"Windows фирмы Microsoft" (2 ГМД) - 2000 руб.

"Энциклопедия IBM" (полный справочник) - 4500 руб.

Заявки (гарантийные письма) направлять по адресу:
111395, Москва, а/я 71, "Интеллект"

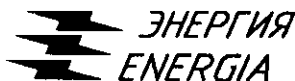


132362, Москва, а/я 393
тел. 4927341, 4927521
факс. (095) 4935330

предлагает пользователям и разработчикам систем автоматизированного проектирования и конструирования услуги по следующим направлениям: **ПРОЧНОСТЬ** – применение метода конечных элементов, суперэлементов, др. численных схем и реализующего их программного комплекса СТАДИО в выполняемых на заказ расчетах стационарных и нестационарных температурных полей, статического и динамического напряженно-деформированного состояния произвольных пространственных комбинированных конструкций и сооружений, учет отечественных и зарубежных норм и нормативов при комплексном обосновании прочности и надежности оборудования, трубопроводов, конструкций и сооружений повышенной опасности.

САПР ТРУБОПРОВОДОВ – поставка по лицензии и авторское сопровождение интегрированной системы автоматизированного проектирования и расчета сложных трубопроводных систем ИРБИС-АСТРА НОВА. Выполнение на заказ особо сложных расчетов трубопроводных систем.

AutoCAD – разработки на заказ прикладных систем с применением версий 10 и 11. Параметрическое проектирование; программирование на AutoLisp, консультации, обучение, поставка по лицензии универсальных утилит и инструментальных программ, используемых совместно с AutoCAD. **СКАН САПР** – наладка технологии "под ключ" и оказание разовых услуг по оптическому считыванию чертежной и картографической информации с бумажных носителей, оцифровке, векторизации, распознаванию графических и текстовых данных.



109396, Москва, ул. Новороссийская, 6
Тел. (095) 2597241, 3504380

изготавливает серийно по конверсии на уровне лучших отечественных технологий и на отечественной элементной базе, поставляет "под ключ", а также обеспечивает гарантийным и послегарантийным обслуживанием аппаратные средства локальных вычислительных сетей и др. интерфейсное оборудование, специально разработанное для отечественных условий, помехозащищенное и ремонтнопригодное.

ЛВС KL-N10.0 – действующая локально-вычислительная сеть для ПЭВМ типа IBM PC магистральной архитектуры с канальной скоростью передачи данных 10 Мбит/сек по коаксиальному кабелю длиной до 500м без ретрансмиттера, поддерживается до 100 станций на сегмент, программно совместима с Novell NetWare.

ЛВС KL-N5.x – действующая гетерогенная локальная вычислительная сеть с возможностью объединения ПЭВМ типа IBM PC и ЭВМ типа "Электроника" магистральной архитектуры с канальной скоростью передачи данных 5 Мбит/сек по коаксиальному кабелю длиной до 4500м без ретрансмиттера, поддерживается до 254 станций на сегмент, программно совместима с Novell NetWare.

Контроллеры интерфейсов RS232C, Centronix, ИРПР, ИРПС для ПЭВМ типа IBM PC, интерфейс Centronix <-> АЦПУ ЕС ЭВМ и др.

По заказам потребителей производится генерация и установка сетевой операционной системы, формирование оптимальной сетевой операционной среды, программная реализация специализированных распределенных процессов, наладка сетевых средств телекоммуникации и компьютерного телефакса.

**Книжные магазины — опорные пункты
издательства "Радио и связь"**

111024, Москва, шоссе Энтузиастов, 24/43, магазин №15

197198, Санкт-Петербург ПС, Большой пр., 34, магазин №55

700070, Ташкент, ул.Шота Руставели, 43, магазин №21

226050, Рига, бул. Падомью, 17, магазин "Гайсма"

634032, Томск, ул.Нахимова, 15/1, магазин №2

503000, Нижний Новгород, пр.Гагарина, 110, магазин №9

630091, Новосибирск, Красный пр., 60, магазин №7
"Техническая книга"

443090, Самара, ул.Советской Армии, 124, магазин №16
"Техническая книга"

173016, Новгород, Ленинградская ул., 13, магазин №2
"Прометей"