

```
HANDLE hDevice,  
DWORD dwIoControlCode,  
LPVOID lpInBuffer,  
DWORD nInBufferSize,  
LPVOID lpOutBuffer,  
DWORD nOutBufferSize,  
LPDWORD lpBytesReturned,  
LPOVERLAPPED lpOverlapped
```

Е.Е. БИЗЯНОВ

***СИСТЕМНОЕ
ПРОГРАММИРОВАНИЕ***

***СИСТЕМНОЕ
ПРОГРАММИРОВАНИЕ***

УЧЕБНОЕ ПОСОБИЕ

АМЧЕВСК, 2018

```
HANDLE hDevice,  
DWORD dwIoControlCode,  
LPVOID lpInBuffer,  
DWORD nInBufferSize,  
LPVOID lpOutBuffer,  
DWORD nOutBufferSize,  
LPDWORD lpBytesReturned,  
LPOVERLAPPED lpOverlapped
```

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
ЛУГАНСКОЙ НАРОДНОЙ РЕСПУБЛИКИ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«ДОНБАССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Е. Е. Бизянов

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Алчевск
2018

УДК 004.45
ББК 32.973-18.02
Б59

Бизянов Евгений Евгеньевич — доктор экономических наук, кандидат технических наук, доцент, профессор кафедры специализированных компьютерных систем ГОУ ВПО ЛНР «ДонГТУ» (г. Алчевск).

Рецензенты:

С. В. Попов — кандидат технических наук, заведующий кафедрой компьютерных систем и сетей ГОУ ВПО ЛНР «Луганский национальный университет им. В. Даля» (г. Луганск);

С. В. Гонтовой — кандидат технических наук, заведующий кафедрой специализированных компьютерных систем ГОУ ВПО ЛНР «ДонГТУ» (г. Алчевск).

*Рекомендовано Ученым советом ГОУ ВПО ЛНР «ДонГТУ»
(Протокол № 10 от 19.06.2018)*

Бизянов Е.Е.

Б59 Системное программирование : учебное пособие. / Е. Е. Бизянов. — Алчевск : ГОУ ВПО ЛНР «ДонГТУ», 2018. — 239 с.

В пособии приведены теоретические положения и представлены практические примеры системного программирования. Рассмотрены системные ресурсы компьютера. Изложены основы языков системного программирования низкого уровня — Си и Ассемблера. Приведены необходимые справочные сведения и примеры для программирования аппаратных средств компьютера, реализации функций ввода-вывода, а также создания служб Windows и демонов Linux.

Пособие предназначено для студентов и аспирантов, обучающихся по напр. подготовки 09.03.01 «Информатика и вычислительная техника» всех форм обучения.

УДК 004.45
ББК 32.973-18.02

© ГОУ ВПО ЛНР «ДонГТУ», 2018
© Е. Е. Бизянов, 2018
© Н. В. Чернышова, художественное
оформление обложки, 2018

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
1 КРАТКИЕ СВЕДЕНИЯ О СИСТЕМНОМ ПРОГРАММИРОВАНИИ..	7
1.1 Предмет и задачи системного программирования	7
1.2 Обзор средств системного программирования	8
1.3 Системные ресурсы персонального компьютера	9
1.4 Прерывания	18
1.4.1 Классификация прерываний	18
1.4.2 Контроллер аппаратных прерываний	20
1.5 Программы	24
1.6 Контрольные вопросы к разделу 1	26
2 ЯЗЫКИ СИСТЕМНОГО ПРОГРАММИРОВАНИЯ	28
2.1 Основы языка программирования Си	28
2.1.1 Переменные, константы, выражения и операции	30
2.1.2 Операторы управления Си	44
2.1.3 Функции	55
2.1.4 Структура программы Си	56
2.2 Основы языка программирования Ассемблер	59
2.2.1 Структура программы на Ассемблере	59
2.2.2 Синтаксис команд Ассемблера	61
2.2.3 Типы данных и переменные	61
2.2.4 Пересылка, адресация и обмен данными	63
2.2.5 Арифметические и логические операции	68
2.2.6 Управление ходом выполнения программы	72
2.3 Интерфейс языков программирования высокого уровня, Си и Ассемблера	78
2.3.1 Ассемблерные вставки	78
2.3.2 Подключение динамических библиотек	88
2.4 Контрольные вопросы к разделу 2	92
3 ПРОГРАММИРОВАНИЕ АППАРАТНЫХ СРЕДСТВ	95
3.1 Ввод с клавиатуры и вывод на экран	95
3.2 Дисковые накопители	105
3.2.1 Работа с дисками	106
3.2.2 Работа с каталогами	112
3.2.3 Работа с файлами	118

3.3 Порты ввода-вывода	122
3.3.1 Общие сведения о портах ввода-вывода.....	122
3.3.2 Параллельный порт	123
3.3.3 Последовательный порт.....	128
3.3.4 Программирование USB [7].....	142
3.4 Контрольные вопросы к разделу 3	182
4 СЛУЖБЫ WINDOWS И ДЕМОНЫ LINUX	185
4.1 Службы Windows [9].....	185
4.1.1 Структура службы.....	182
4.1.2 Регистрация и запуск службы	193
4.1.2 Настройка состояния службы.....	193
4.1.3 Программный код службы.....	187
4.2 Демоны Linux	202
4.2.1 Базовые положения	202
4.2.2 Разработка программного кода демона	205
4.3 Контрольные вопросы к разделу 4	211
ПЕРЕЧЕНЬ ССЫЛОК	212
ПРИЛОЖЕНИЕ А БИБЛИОТЕКИ ЯЗЫКА СИ	213
Б.1 Библиотека ввода-вывода stdio	213
Б.2 Библиотека функций анализа символов ctype	218
Б.3 Библиотека string	219
Б.4 Библиотека математических функций math.....	221
Б.5 Вспомогательные функции	223
Б.6 Макрос для диагностики	227
Б.7 Системно-зависимые константы.....	228
Б.8 Библиотеки даты и времени.....	230
ПРИЛОЖЕНИЕ Б ОПРЕДЕЛЕНИЕ КОНСТАНТ ДЛЯ ПРОГРАММИРОВАНИЯ USB.....	233

ВВЕДЕНИЕ

Системное программирование нацелено, в первую очередь, на создание программных модулей и программ, обеспечивающих надежный и эффективный интерфейс между аппаратной частью вычислительной системы и прикладными программами пользователя.

Программное обеспечение, работающее на компьютере, разделяют на системное и прикладное. Прикладное программирование нацелено на высокоуровневую разработку программ с графическим интерфейсом с использованием языков высокого уровня (например, C++, C#, Object Pascal или Java). Программисту, создающему прикладные программы, необязательно знать во всех подробностях внутреннее устройство компьютера. Особенность системного программирования состоит в том, что программисты должны обладать глубокими знаниями об аппаратных средствах компьютера, а также об операционной системе, с которой они работают.

В пособии рассматриваются базовые принципы создания системных программ и модулей (исключая модули операционных систем) с использованием языков программирования низкого уровня Си и Ассемблер. Для того чтобы студенты могли применить указанные языки программирования, в пособие введен отдельный раздел, в котором приведены необходимые сведения о Си и Ассемблере. Также рассмотрены методы встраивания фрагментов ассемблерного кода в программы на языках высокого уровня и подключения подпрограмм, написанных на Ассемблере к программам, написанных на других языках программирования. Сведения о Си и Ассемблере представлены в объеме, достаточном для создания системных программ.

Значительная часть пособия посвящена программированию аппаратных средств: устройств ввода/вывода, дисков. Подробно рассмотрено программирование портов ввода-вывода, использующих интерфейсы LPT, COM и USB.

Для программирования разнородных устройств необходимо знать аппаратные регистры, настройка которых позволяет прочесть информацию об устройствах, задать их параметры, определить режимы работы и т.п.

В завершающем разделе пособия рассматривается создание специфических системных программ: служб Windows и демонов Linux.

Большая часть примеров приведена как в виде программ на Си, так и на Ассемблере. Кроме того, рассмотрены варианты реализации для наиболее распространенных операционных систем Windows и Linux.

Предполагается, что читатель знаком с существующими системами счисления, используемых в компьютерных программах: двоичной и шестнадцатеричной, а также знает способы задания чисел в этих системах счисления и особенности арифметических и логических операций с ними.

Автор выражает благодарность рецензентам за высказанные рекомендации и пожелания, позволившие улучшить представленный в пособии материал.

1 КРАТКИЕ СВЕДЕНИЯ О СИСТЕМНОМ ПРОГРАММИРОВАНИИ

1.1 Предмет и задачи системного программирования

Системное программное обеспечение представляет собой комплекс управляющих и обрабатывающих программ, описаний и инструкций, обеспечивающих функционирование вычислительной системы, а также разработку и исполнение программ пользователей. Состав системного программного обеспечения мало зависит от характера решаемых задач пользователей [1].

К задачам системного программирования можно отнести:

- разработку модулей операционных систем (включая ядро), обеспечивающих взаимодействие аппаратных средств вычислительной системы и прикладных программ;
- разработку трансляторов, компиляторов;
- разработку загрузчиков и программ связывания;
- разработку драйверов устройств;
- разработку служб (Windows) и демонов (Linux);
- и другие.

На настоящий момент выпускается достаточно широкий ряд модификаций вычислительных машин, поэтому при создании системных программ необходимо учитывать специфические свойства и параметры аппаратной части, оптимизируя при этом программный код с целью увеличения быстродействия и минимизации используемой памяти. Несмотря на то, что вычислительная мощность процессоров и объемы памяти компьютеров непрерывно возрастают, вопросы оптимизации программного кода остаются актуальными.

Постоянно растущий объем программ, увеличение количества системных библиотек, расширение номенклатуры аппаратного обеспечения усложняет работу системного программиста, а также повышает требования к системным программам: они должны не только решать поставленные задачи, но и обрабатывать ошибки, обеспечивая при этом работоспособность оборудования и прикладных программ.

Таким образом, качественные системные программы должны работать на любом оборудовании и под управлением заданной операционной системы.

1.2 Обзор средств системного программирования

К средствам системного программирования относятся:

- языки программирования;
- командные и графические оболочки.

Изначально системные программы записывали в машинном коде (коде команд микропроцессора), что, с одной стороны обеспечивало наивысшую оптимальность программ, но, с другой стороны требовало высочайшей квалификации от системных программистов, которые должны были знать не только систему команд для конкретного процессора, но и особенности настройки и функционирования конкретного оборудования, подключенного к процессору. Программа в машинном коде выглядела примерно так, как показано на рисунке 1.1.

0x55	0x89	0xe5	0xe8	0xfc	0xff	0xff	0xff
0x83	0xf8	0x41	0x75	0x0d	0x68	0x00	0x00
0x00	0x00	0xe8	0xfc	0xff	0xff	0xff	0x83
0xc4	0x04	0xb8	0x00	0x00	0x00	0x00	0x89
0xec	0x5d	0xc3					

Рисунок 1.1 — Фрагмент программы в машинном коде

Появление языков программирования низкого и высокого уровня позволило существенно формализовать процесс разработки системных программ, так как команды Ассемблера или Си стали выглядеть примерно одинаково в программах для Windows, Linux и других операционных систем.

Современные операционные системы включают в свой состав графические средства взаимодействия с пользователем, организованные в виде оконного интерфейса. Не остались в стороне и разработчики графических оболочек для системного программирования. Оболочки для ассемблеров, для компиляторов языков Си и С++, отладчики и другие средства существенно упрощают разработку и наладку программ.

1.3 Системные ресурсы персонального компьютера

Для того, чтобы создавать системное программное обеспечение, необходимо знать сам объект — компьютер: какой микропроцессор используется, сколько оперативной памяти доступно, какое оборудование имеется в системе и пр.

Современные ЭВМ (компьютеры) построены по архитектуре фон Неймана, в которой главным устройством является процессор, реализующий основные математические операции и осуществляющий управление всей системой. В состав ЭВМ входят также устройства для хранения данных — внутренняя и внешняя память, шины для обмена данными и передачи сигналов управления, а также группы вспомогательных устройств для реализации базовых операций ввода-вывода данных, отсчета временных интервалов, хранения данных и т.д.

На рисунке 1.2 изображено устройство персональной ЭВМ (компьютера). Рассмотрим приведенную схему подробнее.

Процессор (в персональных ЭВМ — микропроцессор МП) содержит следующие основные блоки:

- устройство управления (УУ), представляющее собой цифровой автомат с микропрограммным управлением, обеспечивающее управление всеми внутренними узлами микропроцессора;
- арифметико-логическое устройство (АЛУ), реализующее арифметические и логические операции с целочисленными данными;
- внутреннюю память микропроцессора, реализованную в виде набора именованных ячеек (регистров);
- блока математического сопроцессора (МСП), обеспечивающего выполнение операций с вещественными числами.

Микропроцессор извлекает и выполняет команды, хранящиеся в памяти, по очереди. Очередность выполнения команд при этом определяется УУ.

Системная шина включает в себя шину адреса (ША), шину данных (ШД) и шину управления (ШУ). Все шины представляют собой набор проводников, количество которых определяется разрядностью процессора.

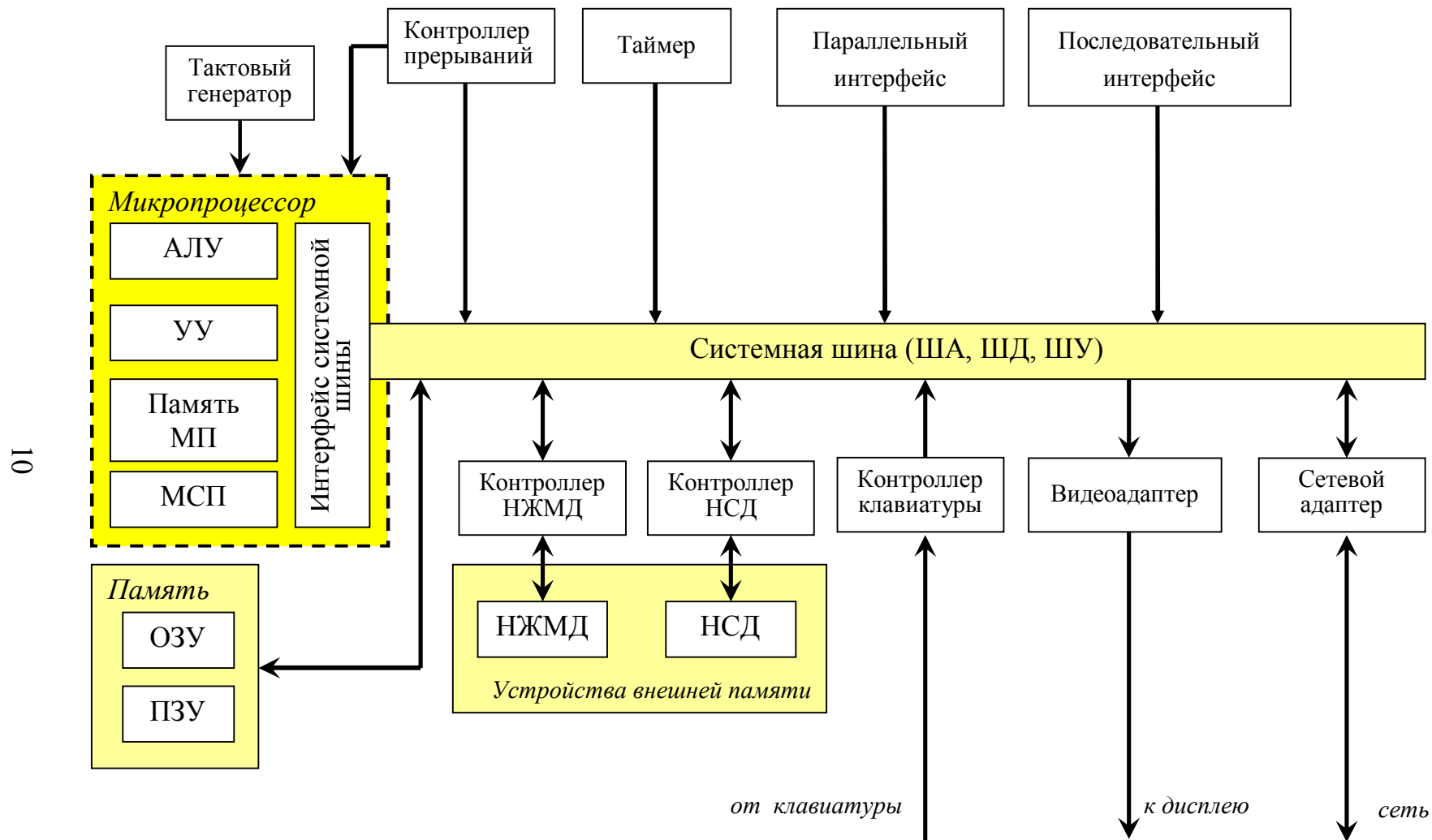


Рисунок 1.2 — Устройство персональной ЭВМ

Шина адреса является однонаправленной (от процессора к другим устройствам) и предназначена для задания адреса устройства, к которому обращается микропроцессор.

Шина данных обеспечивает обмен данными между микропроцессором и другими устройствами. Данные по ШД могут передаваться в двух направлениях.

По *шине управления* между микропроцессором и устройствами осуществляется обмен служебными сигналами: готовности к работе, разрешения операций чтения и записи и т.п.

Память компьютера разделена на память микропроцессора (регистры и кэш-память), постоянное запоминающее устройство (ПЗУ), оперативное запоминающее устройство (ОЗУ), а также дисковую память и видеопамять.

Регистры — это ячейки памяти, размещенные внутри самого МП. Значительная часть операций внутри МП осуществляется именно через регистры. Различают регистры общего назначения (РОН) и специальные регистры. Каждый регистр имеет собственное уникальное имя. Для РОН это имя отображает не только назначение регистра, но и его разрядность. В качестве примера на рисунке 1.3 показан принцип формирования имени наиболее часто используемого РОН — аккумулятора.

		8 бит AH	8 бит AL
		16 бит AX	
	32 бит	EAX	
64 бит	RAX		

Рисунок 1.3 — Принцип формирования имен регистров микропроцессора

Если используются только младшие разряды аккумулятора, этот сегмент имеет имя AL (L — от слова low — нижний), если старшие 8 бит — AH (H — high — верхний). 16-битный аккумулятор имеет имя AX, 32 — EAX (E — exchanged — расширенный), 64-битный аккумулятор называется RAX.

Перечень регистров микропроцессора приведен в таблице 1.1.

Таблица 1.1 — Регистры микропроцессора

Регистр	Разрядность микропроцессора		
	16-битный	32-битный	64-битный
Аккумулятор	AX	EAX	RAX
Базовый	BX	EBX	RBX
Счетчик	CX	ECX	RCX
Регистр данных	DX	EDX	RDY
Указатель стека	SP	ESP	RSP
Указатель базы	BP	EBP	RBP
Индекс источника	SI	ESI	RSI
Индекс приемника	DI	EDI	RDI
Дополнительные регистры данных	R8W — R15W	R8D — R15D	R8 — R15
Адрес инструкции	IP	EIP	RIP
Указатель на сегмент кода	CS	CS	-
Указатель на сегмент данных	DS	DS	-
Указатель на стек	SS	SS	-
Расширенный указатель на сегмент	ES	ES	-

Внимание! Использование регистров RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R_x, R_xD, R_xW, R_xB, SPL, BPL, SIL, DIL возможно только в 64-битном режиме работы микропроцессора.

Регистр флагов FLAGS (16 бит) / EFLAGS (32 бита) / RFLAGS (64 бита) — это массив бит, отражающих текущее состояние процессора, а также результаты выполнения логических или арифметических операций. Состав регистра флагов приведен в таблице 1.2.

Таблица 1.2 — Регистры флагов FLAGS / EFLAGS/ RFLAGS

Бит	Имя	Назначение
Регистр FLAGS / EFLAGS/ RFLAGS		
0	CF	Флаг переноса (Carry Flag)
1	1	Резерв
2	PF	Флаг чётности (Parity Flag)
3	0	Резерв

Продолжение таблицы 1.2

Бит	Имя	Назначение
4	AF	Вспомогательный флаг переноса (Auxiliary Carry Flag)
5	0	Резерв
6	ZF	Флаг нуля (Zero Flag)
7	SF	Флаг знака (Sign Flag)
8	TF	Флаг трассировки (Trap Flag)
9	IF	Флаг разрешения прерываний (Interrupt Enable Flag)
10	DF	Флаг направления (Direction Flag)
11	OF	Флаг переполнения (Overflow Flag)
12	IOPL	Поле приоритета ввода-вывода (I/O Privilege Level)
13		
14	NT	Флаг вложенности задач (Nested Task)
15	0	Резерв
Регистр EFLAGS / RFLAGS		
16	RF	Флаг возобновления (Resume Flag)
17	VM	Режим виртуального процессора 8086
18	AC	Проверка выравнивания (Alignment Check)
19	VIF	Виртуальное прерывание (Virtual Interrupt Flag)
20	VIP	Ожидающее виртуальное прерывание (Virtual Interrupt Pending)
21	ID	Флаг идентификации (ID Flag)
22-31		Резерв
Регистр RFLAGS		
32-63		Резерв

Как видно из таблицы 1.2, младшие биты регистров FLAGS / EFLAGS/ RFLAGS имеют одинаковое назначение, в регистр EFLAGS добавлены 6 флагов, а дополнительные старшие 32 бита регистра RFLAGS зарезервированы для будущего использования.

Рассмотрим доступные программно биты регистра флагов подробнее.

Флаги состояния CF, PF, AF, ZF, SF и OF (биты 0, 2, 4, 6, 7 и 11) используются для идентификации результатов выполнения целочисленных арифметических операций:

– флаг переноса CF устанавливается (т.е. в соответствующий бит записывается логическая 1), если в результате арифметической опера-

ции возник перенос или заем, и, таким образом, показывает переполнение при выполнении арифметических операций. Флаг **CF** часто используют при организации ветвления в программах, осуществляя передачу управления в зависимости от его значения;

- флаг чётности **PF** устанавливается, если младший байт результата содержит чётное число бит, в противном случае — сбрасывается (в соответствующий бит записывается логический 0). Этот флаг часто используется при программировании передачи данных через порты ввода-вывода;

- вспомогательный флаг переноса **AF** устанавливается, если арифметическая операция производит перенос из 3-го бита в 4-й бит в результате сложения, или при заеме из 4-го бита в 3-й в результате вычитания, в остальных случаях сбрасывается. Этот флаг используется вместе с командами, работающими с двоично-десятичными (**BCD**) числами;

- флаг нуля **ZF** устанавливается, если результат операции равен нулю, в противном случае — сбрасывается;

- флаг знака **SF** равен значению старшего бита результата, является знаковым в арифметических операциях (0 — число положительное, 1 — отрицательное);

- флаг переполнения **OF** устанавливается, если возникло переполнение в арифметической операции с числами со знаком.

*Системные флаги и поле **IOPR** управляют операционной средой и не должны использоваться в прикладных программах:*

- флаг разрешения прерываний **IF** разрешает или запрещает реакцию микропроцессора на аппаратные прерывания. Если он установлен, то микропроцессор реагирует на прерывания, если сброшен — не реагирует на них (однако не игнорирует);

- флаг трассировки **TF** устанавливают, чтобы процессор выполнял программу по одной команде с остановом после каждой команды (режим отладки). Если он сброшен — программа выполняется обычным образом;

- поле приоритета ввода/вывода **IOPR** (2 бита) содержит уровень привилегий для операций ввода/вывода для текущей задачи;

– флаг вложенности задач NT устанавливается процессором, при переходе от одной задачи к другой командой CALL или аппаратным прерыванием, и показывает, была ли текущая задача вызвана из предыдущей (флаг NT=1), или нет (NT=0);

– флаг возобновления RF выключает обработку особых ситуаций отладки, чтобы вызвавшая особую ситуацию команда могла быть перезапущена и не вызывала новую особую ситуацию;

– флаг режима виртуального процессора 8086 VM устанавливается, если процессор переводится в режим виртуального 8086, а сбрасывается для возврата в защищенный режим;

– флаг проверки выравнивания AC. Установка этого флага вместе с битом AM в регистре CR0 заставляют процессор проверять выравнивание при доступе к памяти;

– флаг виртуального прерывания VIF является виртуальным образом флага IF, используется совместно с флагом VIP при включённом расширении режима виртуального 8086;

– флаг ожидания виртуального прерывания VIP позволяет отслеживать виртуальные прерывания. Этот флаг обычно используется совместно с флагом VIF;

– флаг идентификации ID предназначен для проверки — поддержки процессором команды CPUID (получение информации о процессоре). Если можно установить и сбросить этот флаг программно, значит, эта команда данным процессором поддерживается..

Дисковая память. Этот вид памяти предназначен для долговременного хранения информации в виде файлов, которые могут быть исполняемыми программными модулями, содержать какие-либо данные (числовые, текстовые, графические и др.). Способ размещения файлов на диске и возможности манипуляций с файлами определяются файловой системой. Каждый файл имеет набор атрибутов, состав которого зависит от используемой файловой системы.

Для каждой операционной системы характерны свои файловые системы (одна или несколько), которые она поддерживает.

Адаптеры и контроллеры. Адаптер является аппаратным средством сопряжения устройства с шинами или интерфейсами компьютера.

Контроллер выполняет также функции сопряжения, но при этом он может выполнять и самостоятельные действия после получения команд от обслуживающей его программы. Сложный контроллер может иметь в своем составе собственный процессор.

Часть периферийных устройств совмещена со своими контроллерами (адаптерами). Примером может служить сетевой адаптер Ethernet. Однако чаще всего периферийные устройства подключаются к своим контроллерам через промежуточные периферийные интерфейсы, находящиеся на нижнем уровне иерархии подключений.

Для взаимодействия с периферийными устройствами процессор обращается к регистрам контроллера (адаптера), передавая через них управляющие команды и данные.

Для этого контроллеры имеют регистры ввода-вывода, управления и состояния, которые могут располагаться либо в адресном пространстве памяти компьютера, либо в специальном пространстве портов ввода-вывода.

Устройства ввода-вывода. Периферийные устройства могут подключаться к интерфейсам системного уровня, или к периферийным интерфейсам. Все устройства ввода-вывода можно условно разделить на следующие группы:

1. Устройства ввода-вывода, осуществляющие непосредственное взаимодействие с пользователем: клавиатура, монитор, мышь.
2. Дополнительные устройства ввода-вывода: принтеры, сканеры, джойстики, проекторы, звуковые колонки.

Взаимодействие системных и прикладных программ с устройствами ввода-вывода может осуществляться через базовую систему ввода-вывода (BIOS), путем записи управляющих данных в соответствующие регистры в пространстве ввода-вывода. При передаче-приеме данных в/из устройств ввода-вывода на системном уровне также широко используется подсистема прерываний (см. далее).

Таймер. Во всех компьютерах используется счетчик-таймер, выполняющий следующие функции:

1. Генерацию прерываний от системных часов, обеспечивающих отсчет системного времени;
2. Формирование запросов на регенерацию памяти;

3. Формирование простых звуковых сигналов.

В качестве счетчиков-таймеров ранее применялись микросхемы i8253 и i8254, на современных системных платах интегрированные в базовый комплект управляющих микросхем компьютера (chipset).

Звук. Стандартный канал управления звуком рассчитан на подключение высокоомного малогабаритного излучателя (динамика), встроенного в корпус ПК. Звук формируется из тонального сигнала от второго канала таймера, работой которого можно управлять программно. Частоту сигнала (тон) можно изменять, программируя коэффициент деления счетчика. Кроме того, можно определять длительность звучания.

Задачей стандартного звукового канала является подача сигналов при загрузке, идентификации ошибок во время теста ПК, когда сообщения на экран вывести еще нельзя, а также сопровождение сообщений о системных ошибках.

Видеосистема современного компьютера состоит из обязательной графической подсистемы (формирующей изображение программно) и дополнительной подсистемы обработки видеоизображений. Обе эти составляющие части обычно используют общий монитор, а соответствующие аппаратные средства системного блока могут располагаться на отдельных платах (картах) различного функционального назначения или объединяться в одном комбинированном *адаптере дисплея*.

Графический адаптер обеспечивает программное формирование графических и текстовых изображений и является промежуточным элементом между монитором и системной шиной компьютера. Изображение строится по программе, исполняемой центральным процессором, которому могут помогать графические акселераторы и сопроцессоры.

Адаптер посылает в монитор сигналы управления яркостью основных цветов палитры RGB (Red, Green, Blue — красный, зеленый и синий) и синхросигналы формирования изображения по вертикали и по горизонтали.

Все компоненты видеоадаптера могут размещаться на отдельной плате расширения, или же интегрироваться прямо на системной плате.

1.4 Прерывания

Прерывание (interrupt) представляет собой передачу управления в ответ на сигналы, асинхронные по отношению к выполнению команд [1]. *Прерывания* также рассматривают, как готовые процедуры, которые компьютер вызывает для выполнения определенной задачи [2].

Кроме того, *прерыванием* называется ситуация, возникающая в результате возникновения какого-либо независимого события, приводящего к временному прекращению выполнения последовательности команд одной программы с целью выполнения последовательности команд другой программы [1].

1.4.1 Классификация прерываний

Прерывания делят на два вида: аппаратные и программные прерывания.

Аппаратные прерывания инициируются аппаратурой, т.е. могут быть вызваны сигналом от таймера, запросом от принтера или сканера, нажатием клавиши на клавиатуре, движением мыши и т.п.

Аппаратные прерывания приостанавливают работу микропроцессора, который обрабатывает прерывание, а затем возвращается на прежнее место в прерванной программе. Адрес места останова запоминается в стеке, вместе с регистром флагов. Затем в регистры CS:IP загружается адрес программы обработки прерывания и уже затем ей передается управление.

Прерывания от схем контроля генерируются в случае возникновения нестандартных ситуаций и системных ошибок, например: прерывание по резервной инструкции, прерывание из-за нарушения защиты памяти, прерывание по нарушению вида доступа к памяти и т. д. Т.к. прерывания от схем контроля в отличие от остальных прерываний приостанавливают выполнение текущей команды программы, их иногда называют *исключениями*.

Вот некоторые важные аппаратные прерывания (в скобках указан шестнадцатеричный номер):

- IRQ0 (INT 8) — прерывания от системного таймера;
- IRQ1 (INT 9) — прерывания от клавиатуры;
- IRQ2 — множитель приоритетных уровней;

- IRQ3 (INT 0Bh) и IRQ4 (INT 0Ch) — прерывания от последовательного порта COM2 и COM1 соответственно;
- IRQ5 (INT 0Dh) и IRQ7 (INT 0Fh) — прерывания от LPT2 и LPT1 соответственно (используются дополнительными устройствами);
- IRQ6 (INT 0Eh) — прерывания от прерывания от магнитного диска;
- IRQ8 (INT 70) — прерывание от часов реального времени;
- IRQ9 (INT 0Ah) — прерывание звуковой карты;
- IRQ10 – IRQ12 — резерв для дополнительных устройств;
- IRQ13 (INT 2) — прерывание по ошибке математического сопроцессора;
- IRQ14 (INT 76h) и IRQ15 (INT 77h) — прерывания от контроллеров жесткого диска IDE1 и IDE2.

Аббревиатура IRQ расшифровывается как Interrupt ReQuest — запрос на прерывание.

Программные прерывания вызываются из системных программ и используются для решения задач ввода-вывода, управления памятью, управления другими программами и пр. Программные прерывания — это подпрограммы, которые вызываются Вашими программами для обработки запросов на обслуживание от аппаратуры или других программ. Как правило, эти подпрограммы содержатся не внутри программы, а в операционной системе, и механизм прерываний дает Вам возможность обратиться к ним. Программные прерывания могут вызываться друг из друга. Например, все прерывания обработки ввода с клавиатуры операционная система DOS используют прерывания обработки ввода с клавиатуры BIOS для получения символа из буфера клавиатуры.

Аппаратное прерывание может получить управление при выполнении программного прерывания. При этом не возникает конфликтов, так как каждая подпрограмма обработки прерывания сохраняет значения всех используемых ею регистров и затем восстанавливает их при выходе.

Адреса программ обработки прерываний называют *векторами*. Каждый вектор имеет длину четыре байта. В первом слове хранится значение для IP, а во втором — CS. Младшие 1024 байт памяти содер-

жат вектора прерываний, таким образом, зарезервировано место под 256 векторов, совокупность которых называется таблицей векторов. Вектор для прерывания 0 начинается с адреса 0000:0000, прерывания 1 — с 0000:0004, 2 — с 0000:0008 и т.д. Если Вы посмотрите на четыре байта, начиная с адреса 0000:0020, в которых содержится вектор прерывания 8H (прерывание времени суток), то обнаружите там значение A5FE00F0. Младший байт слова расположен в начале, а так как порядок следования IP:CS, это 4-байтное значение переводится в F000:FEA5. Это стартовый адрес программы в ПЗУ, выполняющей прерывание 8H.

Внешние прерывания возникают по сигналу от внешних устройств в тех случаях, когда им требуется обслуживание (чтение, запись, контроль состояния).

Исключением из этого правила является *системный таймер*, генерирующий запрос на прерывание через фиксированные интервалы времени (в DOS — приблизительно через 18,2 мс). Таймер используется для отслеживания системного времени, генерации отсчетов времени при работе с внешними устройствами, а также формирования ожиданий (time-out).

Подпрограммы обработки прерываний всегда завершаются инструкцией возврата из прерывания, которая завершает процесс, начатый прерыванием, возвращая старые значения CS:IP и регистра флагов, тем самым давая программе возможность продолжить выполнение из того же состояния, в котором она была приостановлена.

Обработчик прерываний должен:

- сохранять содержимое прерванной программы (значения переменных, адресов, регистров МП);
- запрещать прерывания только в случае крайней необходимости;
- выполняться за минимально возможное время;
- предусматривать возможность повторного входа (reenterable), либо защиту от него;
- обеспечивать срабатывание обработчика, использовавшего данный вектор ранее.

Запросами к операционной системе следует пользоваться только в том случае, если заведомо известно, что данный запрос разрешен к использованию в обработчиках прерываний.

В отличие от программных прерываний, вызываемых по плану самой прикладной программой, аппаратные прерывания всегда происходят асинхронно по отношению к выполняющимся программам. Флаг `IF` в регистре флагов процессора определяет, будет ли процессор воспринимать запросы на прерывания от внешних устройств.

Иногда может возникнуть одновременно сразу несколько запросов на прерывание. Для того, чтобы система знала, какое именно прерывание обслуживать в первую очередь, существует специальная *схема приоритетов*, согласно которой каждому прерыванию назначается свой уникальный приоритет. Поэтому, если происходит одновременно несколько прерываний, то система отдает предпочтение прерыванию с наибольшим приоритетом, откладывая на время обработку остальных прерываний.

Программно поддержка прерываний обеспечивается следующими средствами: векторами прерываний и командами процессора для вызова прерываний.

Так, например, команда `int` Ассемблера вызывает прерывание с вектором, указанным в качестве аргумента команды.

Сам переход к обработчику прерывания процессором всегда выполняется одинаково и может быть проиллюстрирован следующей последовательностью операций в программе на Ассемблере:

```
pushf
push cs
push ip
mov cs, 0:[n*4+2]
mov ip, 0:[n*4]
cli
pop ip
pop cs
popf
```

В приведенном фрагменте команда `push` сохраняет значения в стеке, команда `mov` производит загрузку данных в регистры, команда `cli` сбрасывает флаг разрешения прерываний, а команда `pop` восстанавливает из стека загруженные ранее в него значения.

При управлении группой прерываний часто используют такой прием, как максимирование (наложение маски).

Маскирование прерываний — это запрет для прерывания выполнения критических (по доступу к ресурсам) частей программы или запрет на некоторое время долго обслуживаемых прерываний, например, в системах реального времени.

Применяют следующие способы реализации маскирования: общее маскирование и выборочное.

При *общем маскировании* управление реализуется командами CLI, STI, которые соответственно сбрасывают в ноль или устанавливают в единицу флаг разрешения прерывания IF.

При *выборочном маскировании* запись определенного кода в регистр маски контроллера прерываний 8259, в котором имеется три основных регистра:

IRR — регистр запроса прерывания;

ISR — регистр обслуживания прерывания (порт 20h);

IMR — регистр маскирования прерывания (порт 21h).

К регистру IRR подключены все линии запросов IRQ0..IRQ15.

Регистр ISR хранит приоритет текущего обслуживаемого прерывания. Если запрос прерывания, поступивший в регистр IRR, не замаскирован, то происходит сравнение приоритетов PRIRR и PRISR. Если $PRIRR > PRISR$, то поступивший запрос принимается на обслуживание.

Пример программного запрета прерываний приведен ниже. В нем используется функция вызова прерывания DOS (номер функции — в регистре AH).

Пример 1.1. Запрет прерываний от жесткого диска в Ассемблере

```
mov al, 01000000b; маскируются запросы от  
                ; жесткого диска  
int 21h  
.  
.  
.  
.  
.  
.  
mov al, 0  
int 21h
```

Программирование прерываний мы рассмотрим более подробно в следующих разделах.

1.4.2 Контроллер аппаратных прерываний

Аппаратная часть компьютера обязательно содержит соответствующие средства поддержки прерываний, к которым относятся:

- схема опроса процессором входной линии запроса прерывания перед исполнением очередной команды;
- флаг разрешения прерывания (**IF**) в регистре флагов процессора;
- контроллер прерываний;
- схемы выработки сигналов прерываний во внешних устройствах.

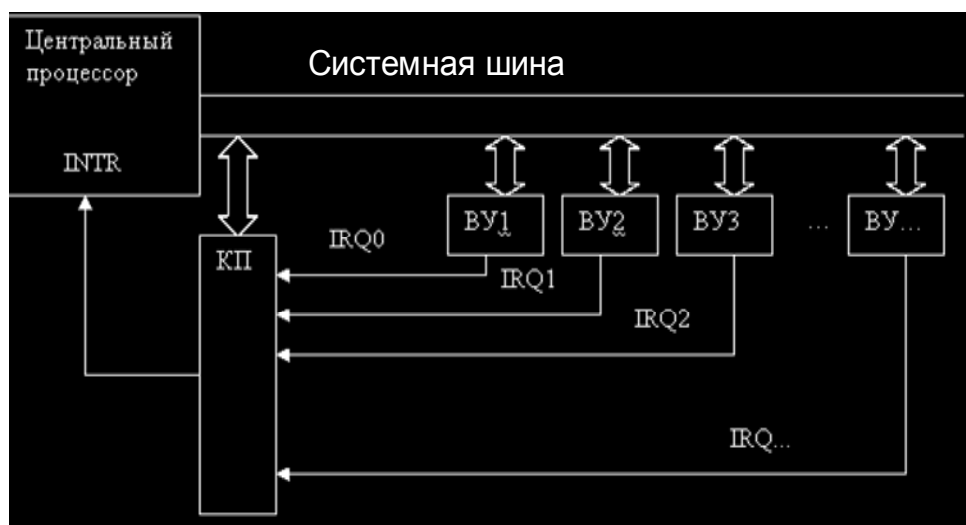
Контроллер прерываний — устройство, логически располагающееся между процессором и внешними устройствами (см. рис. 1.4).

Сигналы запросов на прерывания от внешних устройств поступают по линиям IRQ_x в контроллер прерываний, который отправляет запросы в центральный процессор.

Исторически первым контроллером прерываний была микросхема $i8259A$, имевшая 8 входов IRQ (с IRQ_0 по IRQ_7) [3].

Однако восемью линиями запросов на прерывание недостаточно, поэтому создают двухуровневую схему контроллера прерываний с использованием каскадного соединения, увеличив количество входов IRQ до 15 (рис. 1.5).

Предельное количество входов IRQ составляет 64 и получается при соединении девяти микросхем.



КП — контроллер прерываний; ВУ — внешние устройства

Рисунок 1.4 — Место контроллера прерываний в системе

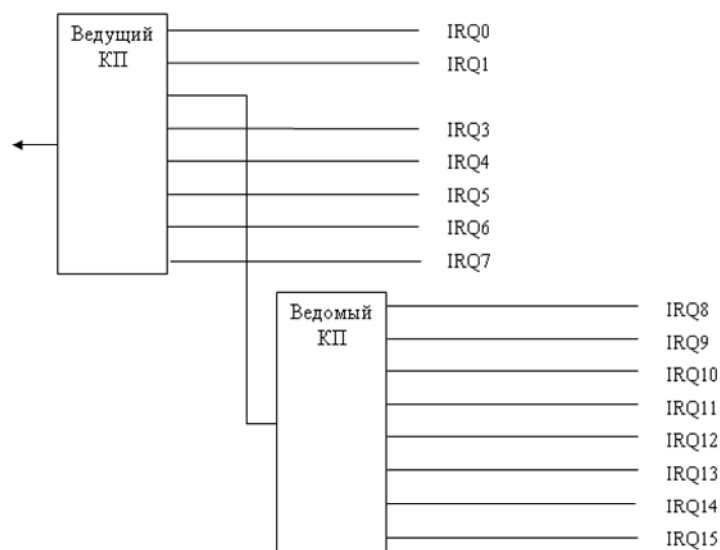


Рисунок 1.5 — Каскадное соединение контроллеров прерываний

Контроллер прерываний выполняет следующие функции [2, 3]:

- фиксирует запросы на прерывания от внешних устройств;
- маскирует поступившие запросы (разрешает или запрещает прохождение запросов с отдельных IRQ входов);
- реализует обслуживание запросов в соответствии с их приоритетами для случая, когда одновременно присутствуют несколько запросов);
- передает процессору номер вектора прерывания, который используется при обработке прерывания.

В контроллере прерываний программируются:

- маска (в регистре памяти),
- арбитр приоритетов,
- начальный номер диапазона номеров векторов прерываний,
- режим работы,
- схема соединения «ведущий — ведомые».

Таким образом, контроллер прерываний обеспечивает идентификацию аппаратных прерываний и реакцию микропроцессора на них.

1.5 Программы

Важнейшей системной программой является *базовая система ввода-вывода* (Basic Input-Output System, или BIOS), которая запускается сразу же при старте компьютера. Программа BIOS записана в ПЗУ и

состоит из нескольких модулей. Первый загружаемый модуль является *программой инициализации* (POST — PowerOn Self Test), которая осуществляет тестирование всего компьютера при включении питания, а также начальную стадию загрузки операционной системы.

В BIOS также содержатся подпрограммы для работы со стандартными устройствами (клавиатура, видеоадаптер, жесткий диск и т.п.), обеспечивающие взаимодействие операционной системы, а также других системных и прикладных программ с аппаратными средствами компьютера.

Следующая по уровню иерархии и важности системная программа — это *операционная система* (ОС), под управлением которой осуществляется работа других системных и прикладных программ. Запуск операционной системы осуществляет *программа-загрузчик*, которая расположена в начальных секторах системного жесткого диска. Взаимодействие ОС с другими программами осуществляется как напрямую, так и через специальные программы — службы.

Каждая системная и прикладная программа использует оперативную память компьютера. Для управления работой программ ОС использует такие их характеристики, как *точка входа* (адрес в памяти, по которому размещена первая команда программы) и требуемый *объем памяти*.

Службы (services) взаимодействуют как с ОС, так и с BIOS, и обеспечивают доступ к аппаратным средствам компьютера и к прикладным программам. Службы — это системные программы Windows, в Linux они называются демонами (daemons).

К системным программам также относятся драйверы устройств, драйверы прерываний, антивирусные программы.

Драйвер (driver) — программа, предназначенная для работы с аппаратными средствами. Драйверы являются посредниками между устройствами и программами, и реализуют возможность использования стандартных процедур при программном доступе к аппаратуре различных марок и производителей. Обычно драйверы поставляются производителями вместе с устройствами, причем для различных версий операционных систем.

Под управлением операционной системы работает множество *прикладных программ*. Большинство прикладных программ загружаются в память, запускаются, а затем удаляются операционной системой при завершении работы.

Прикладные программы могут обращаться к службам ОС, системы BIOS, а также к аппаратным средствам компьютера напрямую. Прикладные программы хранятся на устройствах внешней памяти (как правило, на жестком диске).

1.6 Контрольные вопросы к разделу 1

1. Перечислите задачи системного программирования.
2. Кратко охарактеризуйте средства системного программирования.
3. Структура персональной ЭВМ. Из каких основных узлов она состоит?
4. Из каких основных блоков состоит микропроцессор?
5. Раскройте структуру системной шины. От чего зависит разрядность шины?
6. Структура памяти компьютера. Какие виды памяти Вы знаете?
7. Регистры общего назначения. Правило формирования имен регистров общего назначения.
8. Регистр флагов. Назначение отдельных бит. Как отличаются регистры флагов 16-, 32-и 64-разрядных вычислительных машин?
9. Дисковая память компьютера. Краткая характеристика.
10. Адаптеры и контроллеры. Их функции. Что общего и в чем отличие?
11. Устройства ввода-вывода. Что к ним относится?
12. Системный таймер. Назначение.
13. Звуковые устройства компьютера. Что к ним относится?
14. Видеосистема компьютера. Из чего она состоит?
15. Дайте определение понятию «прерывание». Приведите классификацию прерываний.
16. Аппаратные прерывания. Как возникают, и как их обработать?
17. Программные прерывания. Как возникают, и как их обработать?
18. Особенности реализации обработчиков программных прерываний.

19. Задачи, решаемые обработчиком прерывания.
20. Контроллер аппаратных прерываний. Функции контроллера прерываний. Принцип работы. Каскадирование контроллеров прерываний.
21. Базовая система ввода-вывода. Какие задачи она решает?
22. Операционная система. Какие задачи она решает?
23. Что такое драйвер? Назначение. Какие виды драйверов Вы знаете?
24. Что такое служба Windows? Какие задачи она решает?
25. Что такое демон Linux? Какие задачи он решает?

2 ЯЗЫКИ СИСТЕМНОГО ПРОГРАММИРОВАНИЯ

Несмотря на то, что за все время развития вычислительной техники было разработано несколько сотен языков программирования, большинство из них направлены на решение задач прикладного программирования.

Как было показано в п.1.5, значительная часть системных программ предназначена для реализации взаимодействия с аппаратной частью вычислительной системы, — с т.н. объектами «нижнего уровня»: дисковыми накопителями, клавиатурой, звуковой картой и пр. Важнейшими требованиями, предъявляемыми к системным языкам программирования, являются: простота, эффективность, а также наличие библиотек функций для доступа к аппаратным ресурсам компьютера.

В этом разделе мы рассмотрим основы языков программирования Си и Ассемблер. Подробное изучение указанных языков программирования не является задачей данного пособия, поэтому здесь изложены их основы в объеме, достаточном для написания системных программ.

2.1 Основы языка программирования Си

Язык Си, по утверждению его создателей, относится к языкам «среднего уровня» [5]. В литературе, посвященной этому языку программирования, его называют и языком «среднего уровня», и языком «низкого уровня». Не будем заострять на этом внимания, отметим только, что значительная часть системных программ и современных операционных систем написаны именно на языке Си.

Несмотря на то, что Си был создан несколько десятилетий назад, он все еще популярен и продолжает развиваться. Кроме того, на основе Си разработано много современных языков программирования высокого уровня: C++, Java, C# и др.

Базовый синтаксис Си достаточно прост для изучения и использования, так как содержит небольшое количество типов данных и операторов.

На настоящий момент имеется достаточное количество компиляторов и графических оболочек для создания и отладки программ на Си. В примерах, рассмотренных в пособии, мы будем использовать свобод-

но распространяемый компилятор **gcc**, который позволяет создавать программы как для Windows, так и для Linux.

Рассмотрим простую программу на языке Си.

```
/* helloworld.c */
#include <stdio.h>
int main(int argc, char *argv[]) {
    printf("hello, world\n");
    getch();
    return 0;
}
```

Первая строка программы — это комментарий. Во второй строке подключается стандартная библиотека функций ввода-вывода. В строках с третьей по седьмую написана реализация главной функции `main`. Внутри функции `main` в четвертой строке оператор `printf` выводит строку текста "hello, world" на экран. Функция `getch` в пятой строке ожидает нажатия любой клавиши для завершения программы, а оператор `return` в шестой строке возвращает в операционную систему признак нормального завершения программы.

В нашей первой программе всего три оператора, выполняющих действия. Первый — `printf` — выводит на экран строку текста, второй — `getch` — ожидает нажатия любой клавиши на клавиатуре, и наконец, оператор `return` завершает работу программы.

Для компиляции (создания исполнимого файла) программы для Windows в командной строке выполним команду:

```
gcc helloworld.c -o helloworld.exe
```

После компиляции на диске появится исполнимый файл `helloworld.exe`, который выводит на экран строку `hello, world`.

Для Linux вызов компилятора должен выглядеть следующим образом:

```
gcc -c helloworld.c -o helloworld.o
```

Безусловно, у компилятора `gcc` гораздо больше ключей и настроек, с которыми Вы можете ознакомиться самостоятельно. Пока нам хватит показанных выше.

2.1.1 Переменные, константы, выражения и операции

Идентификаторы языка Си (имена переменных, констант, функций) могут содержать: латинские буквы A..Z, a..z, арабские цифры 0..9, знак подчеркивания “_”.

Важно! Регистр символов в идентификаторах имеет значение: буквы Z и z — разные!

Важные правила:

1. Недопустимо использовать пробелы внутри идентификатора.
2. Идентификатор должен начинаться с буквы.
3. Можно, но нежелательно начинать идентификатор со знака подчеркивания.
4. Нельзя использовать в качестве идентификаторов ключевые слова языка, такие, как `int`, `float`, `if`, `for` и т.п.

При объявлении переменных и констант, а также при задании имен функций придерживайтесь указанных ниже рекомендаций.

Рекомендации:

1. Задавайте переменным, константам и функциям осмысленные имена. Например, пусть переменная хранит температуру процессора. Ее можно назвать `T`, `Temp`, `Temperature` или `TempProc`. Последние два варианта предпочтительнее, как более наглядные и понятные.

2. Не создавайте слишком длинных имен переменных, констант или функций,— это снижает удобочитаемость программ.

Какие именно значения (данные) можно хранить в переменных и константах, определяет их *тип*. Базовые типы данных Си представлены в таблице 2.1.

Таблица 2.1 — Базовые типы данных Си

Тип данных	Описание
<code>char</code>	один байт, содержащий один символ
<code>int</code>	целое число, имеющее типовой размер для целых чисел в данной системе
<code>float</code>	вещественное число одинарной точности
<code>double</code>	вещественное число двойной точности
<code>void</code>	отсутствие типа.

Синтаксис объявления *переменной* в программе на языке Си:
[квалификатор] Тип Имя [= начальное значение];

Здесь и далее мы будем использовать квадратные скобки [] для обозначения необязательных элементов синтаксиса.

Квалификатор расширяет определение типа данных. Перечень квалификаторов Си приведен в табл. 2.2.

Таблица 2.2 — Квалификаторы типов данных Си

Квалификатор	Описание
short	сокращение
long	расширение
signed	со знаком
unsigned	без знака

Важные правила:

1. Квалификаторы short, long, signed и unsigned применимы к целочисленным типам данных.

2. Квалификатор long можно также применять к типу double.

3. Вещественные типы float и double всегда представляют числа со знаком.!

4. В Си нет логического типа данных! Для определения логических переменных и их использования необходимо подключить заголовочный файл stdbool.h, содержащий макросы bool (логический тип), true и false (значения для логических переменных).

Стандартные заголовочные файлы limits.h и float.h содержат символические константы для всех размеров типов данных, а также других свойств системы и компилятора (см. Приложение А).

Константы — это переменные с неизменным значением. Синтаксис объявления констант в Си:

```
const Тип Имя = значение;  
#define имя значение
```

Первый вариант объявления с использованием ключевого слова const определяет константу, как неизменяемую переменную, а второй вариант с использованием ключевого слова define — символическую константу.

Рассмотрим объявление переменных и констант на примерах.

Пример 2.1. Объявления переменных.

```
int a, b, c = 2;  
float d, x1 = 6.6, zz;  
long double Sw;
```

Пример 2.2. Объявления констант.

```
const int a = 2;  
const double z = -0.65;  
#define k 4  
#define s "Hello"
```

Все операции языка Си разделяют на арифметические, логические, операции сравнения (табл. 2.3). Отдельно выделяют операцию присваивания (=). *Операция присваивания* (обозначается знаком равенства =) используется для задания значений переменных и констант, а также при построении выражений.

Таблица 2.3 — Операции языка Си

Группа	Знак	Операция
Арифметические	+	Сложение
	—	Вычитание
	*	Умножение
	/	Деление
	%	Остаток от деления
	++	Инкремент (увеличение на единицу)
Сравнения	--	Декремент (уменьшение на единицу)
	>	Больше
	<	Меньше
	==	Равно
	!=	Не равно
	>=	Больше или равно
	<=	Меньше или равно

Продолжение таблицы 2.3

Группа	Знак	Операция
Логические	&&	Логическое И
	 	Логическое ИЛИ
	!	Логическое НЕ
	&	Побитовое логическое И
	 	Побитовое логическое ИЛИ
	^	Побитовое логическое ИСКЛЮЧАЮЩЕЕ ИЛИ
	<<	Сдвиг влево
	>>	Сдвиг вправо
	~	Побитовое логическое отрицание

Операция присваивания может использоваться в одном выражении несколько раз. Кроме того, операцию присваивания можно совмещать со знаками арифметических и логических операций. Рассмотрим эти особенности на примере.

Пример 2.3. Операции присваивания.

```
int x = 2, s, d = -3, q, w;
const float f = 0.66;
q = w = 9; // и q и w равны 9
s = x + q - w / 3;
q = (w = 1) + x - (x = 2);
s += x; // Эквивалентно s = s + x;
```

В пятом операторе переменным *w* и *x* присваиваются новые значения, а затем с их использованием производится вычисление значения для переменной *q*. Рассмотрим еще один пример использования операции присваивания.

Пример 2.4.. Операции присваивания.

```
int x = 8, y = 5, d;
d = (x + 3) - (++y - x) + x++;
```

Вычисление для переменной *d* производится в следующем порядке:

1. Вычисляется новое значение переменной *y* (инкремент): *y* = 6.
2. Вычисляются выражения в скобках: сначала $(x + 3) = 11$, затем $(++y - x) = 2$.

3. Вычисляется значение $d = (x + 3) - (++y - x) + x = 11 - 2 + 8 = 17$.

4. Вычисляется новое значение переменной x (инкремент): $x = 9$.

Выражения в программе Си — это любая правильная последовательность операторов, констант, переменных и вызовов функций. Большинство выражений являются алгебраическими.

Пример 2.5. Выражения.

```
a = b++;  
cosinus = cos(5*a);  
sinus = sin(b);  
tg = cosinus/sinus;
```

Лишние пробелы в выражении улучшают его удобочитаемость.

Так, приведенные ниже выражения эквивалентны:

```
a=10*b+c/4;  
a = 10*b + c/4;
```

К операторам выражения можно также отнести *пустой оператор* — точку с запятой “;”. Он полезен там, где обязательно должен быть оператор, а делать ничего не нужно.

Внутри каждого оператора вычисление производится в соответствии с приоритетом операций (табл. 2.4). Операции, перечисленные в одной строке, имеют одинаковый приоритет.

Таблица 2.4 — Приоритет операций

Приоритет	Оператор
1	++ --
	()
	[]
2	+ -
	! ~
3	*
	&
	sizeof
4	* / %

Продолжение таблицы 2.4.

Приоритет	Оператор
5	<< >>
6	< <=
	> >=
7	== !=
8	&
9	^
10	
11	&&
12	
13	? :
14	= += -=
	*= /= %=
	<<= >>=
	&= ^= =

Каждая *статическая* переменная имеет свой адрес в памяти компьютера. Допустим, имеется переменная `house` (дом) типа `int` и переменная `AddressOfHouse` (АдресДома), которая может содержать адрес переменной типа `int`.

Оператор `AddressOfHouse = &house;` означает получение адреса (&) переменной `house`.

Переменная, которая содержит адрес другой переменной, такая, как `AddressOfHouse`, называется *переменной-указателем*. Другими словами, если одна переменная содержит адрес другой переменной, то говорят, что первая переменная указывает (ссылается) на вторую. Пусть переменная `house` размещена по адресу 365. Тогда после выполнения указанного выше оператора переменная `AddressOfHouse` будет содержать число 365, т.е. адрес `house`. При этом говорят, что `AddressOfHouse` указывает на `house`.

Теперь есть два способа обратиться к одному и тому же месту в памяти (к одной и той же переменной):

– используя переменную `house`:

`house = 10;`

– используя указатель `AddressOfHouse`:

```
*AddressOfHouse = 10;
```

Здесь знак `*` используется как оператор ссылки. Таким образом, оба описанных выше способа эквивалентны.

Переменную-указатель `AddressOfHouse` следует объявить, как показано на рисунке 2.1.

```
int *AddressOfHouse;
```

тип данных, на который указывает указатель *обозначение указателя* *имя переменной-указателя*

Рисунок 2.1 — Объявление переменной-указателя

Рассмотрим примеры объявления указателей.

Пример 2.6. Объявление указателей.

```
char *char_ptr;
```

```
int *int_ptr;
```

Важно! Переменная-указатель должна указывать только на объявленный тип данных!

Пример 2.7. Ошибки при объявлении указателей.

```
int *int_ptr;
```

```
float ff = 1.35; int ii = 2;
```

```
int_ptr = &ff;            // Неверно !!
```

```
int_ptr = &ii;            // Правильно!
```

Теперь рассмотрим пример использования переменных-указателей — обмен данными через указатели, в котором меняются местами данные, хранящиеся в двух статических переменных.

Пример 2.8. Пример обмена через указатели.

```
int first = 1, second = 2, temp; // Объявление  
// переменных и указателя
```

```
int *iptr;
```

```
iptr = &first;    // Инициализация
```

```
temp = *iptr;    // Считываем данные,
```

```

        // хранящиеся в first
*iptr = second;        // Копируем в second
// данные по адресу, хранящемуся в iptr
second = temp;         // перенос данных в
                        // second

```

Рассмотрим пошагово, как происходит обмен (адрес дан условно).

Переменная	Адрес	Значение
------------	-------	----------

Начальное состояние

first	1111	1
second	2222	2
temp	3333	

После выполнения оператора `iptr = &first;`

first	1111	1
second	2222	2
temp	3333	

После выполнения оператора `temp = *iptr;`

first	1111	1
second	2222	2
temp	3333	1

После выполнения оператора `*iptr = second`

first	1111	2
second	2222	2
temp	3333	1

После выполнения оператора `second = temp;`

first	1111	2
second	2222	1
temp	3333	1

Правила использования указателей приведены ниже.

Правило 1. До того, как указатель будет использоваться, он должен быть инициализирован:

```

char *aptr ;
char ch;
aptr = &ch;

```

Правило 2. Указатель может содержать адрес переменной, а не число:

```
int i, *iptr;
iptr = &i;           // Правильно
iptr = 22;           // Неправильно
```

Правило 3. Указатель нельзя использовать для непосредственного хранения результата выполнения операции:

```
float *fprr;
float f1 = 12.3, f2 = 89.05, f3;
fprr = &(f1/f2);      // Неверно
f3 = f1/f2;
fprr = &f3;           // Верно
```

Особо отметим операторы * (оператор разыменования) и & (оператор получения адреса), которые были использованы нами в приведенных примерах.

Оператор разыменования * перед переменной-указателем позволяет получить значение переменной, на которую указывает указатель. Оператор получения адреса & позволяет присвоить переменной-указателю адрес какой-либо переменной.

Рассмотрим еще один пример, для того, чтобы разобраться в том, как работают указатели.

Пример 2.9. Особенности использования указателей.

```
int v1 = 0, *p1, *p2;
p1 = &v1;      // p1 указывает на v1
*p1 = 42;       // Присвоить переменной v1
                // значение через p1
printf(v1);     // Выведет 42
printf(p1);     // Выведет 42
p2 = p1;        // Теперь p2 указывает на ту
                // же ячейку памяти, что и p1
printf(*p2);    // Выведет 42
v1 = 55;        // Поменяем v1
*p2 = *p1;
printf(*p2);    // Выведет 55
```

Таким образом, если p1 и p2 — переменные-указатели, то инструкция вида p1 = p2 изменяет значение p1 так, что эта переменная указывает на то же место в памяти, что и p2.

С указателями в Си можно выполнять следующие действия:

- присваивание;
- сложение с числом;
- вычитание;
- операция инкремента (приращения);
- операция декремента (уменьшение);
- операция вычисления размера `sizeof ()`;
- сравнение указателей (`<`, `>`, `<=`, `>=`, `==`, `!=`).

Недопустимые операции:

- сложение указателей;
- умножение указателей;
- деление одного указателя на другой.

Рассмотрим сложные типы данных языка Си.

Массив — это набор переменных одного типа, имеющих имя и снабженных индексами. Доступ к конкретному элементу массива осуществляется с помощью индекса. Нумерация элементов массива начинается с нуля.

Синтаксис объявления одномерного массива:

```
тип имя_переменной_массива [размер];
```

Здесь тип обозначает тип элементов массива, размер задает количество элементов массива. Например, следующий оператор объявляет массив из 50 элементов целого типа с именем `array`:

```
int array[50];
```

Доступ к элементу массива осуществляется с помощью имени массива и индекса. Индекс элемента массива помещается в квадратных скобках после имени. Оператор `array[2] = -6;` присваивает 3-му элементу массива `array` значение -6.

Синтаксис объявления многомерного массива:

```
тип имя_массива [Размер1][Размер2]...[РазмерN];
```

Массивы можно инициализировать при объявлении:

```
тип имя_массива [размер1]...[размерN] =  
    { список_значений };
```

или в программе

```
имя_массива [индекс1]...[индексN] = Значение;
```


Список_значений представляет собой набор значений, разделенных запятыми. Типы значений должны быть совместимыми с типом элементов массива. Ниже показан пример инициализации одномерного и двумерного массивов при их объявлении.

Пример 2.10. Инициализация массивов при объявлении.

```
// инициализация одномерного массива
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// инициализация двумерного массива:
int arr[10][2] = {{1, 1}, {2, 2}, {3, 4},
                  {4, 5}, {6, -7}, {6, 0},
                  {7, -11}, {0, 6}, {-22, 11},
                  {10, 20} };
```

Идентификатор массива содержит адрес первого элемента массива, причем это адрес постоянный. Таким образом, можно говорить о том, что идентификатор массива — это указатель.

Объявим две переменные:

```
int E_int[10];
int *ar_ptr;
```

Инициализируем переменную-указатель, присвоив ей значение адреса первого элемента массива:

```
ar_ptr = E_int;
```

Это можно сделать и так:

```
ar_ptr = &E_int[0];
```

Для этого чтобы обратиться к очередному элементу массива через указатель, нужно сделать приращение адреса указателя. Рассмотрим это на примере.

Пример 2.11. Способы обращения к массиву.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int D[5] = {1, 2, 3, 4, 5};
int *ar_ptr;
void main() {
ar_ptr = D; int i;
    for (i=0; i<5;i++)
```

```

printf("D[%d]=%d\n", i, D[i]);
printf("D+2=%d\n", *(D+2));
printf("ar_ptr[2] = %d\n", ar_ptr[2]); // 3
printf("3[ar_ptr] = %d\n", 3[ar_ptr]); // 4
printf("3[D] = %d\n", 3[D]);          // 4
getch(); }

```

Проанализируем этот пример (а именно тело функции `main`).

В первой строке адрес первого элемента массива `D` присваивается указателю `ar_ptr`. Во второй и третьей строке в цикле элементы массива выводятся на экран. Здесь используется обычная адресация — обращение по индексу.

В четвертой строке мы обращаемся к массиву через адрес `(D+2)`. Это означает, что мы выводим на экран содержимое элемента массива, адрес которого равен $D+2 = 0+2 = 2$. Эта ячейка содержит число 3. В пятой строке мы обращаемся к той же ячейке, но через переменную — указатель. В шестой и седьмой строках показаны варианты обращения к элементу массива через указатель и переменную.

Теперь рассмотрим обращение к трехмерному массиву через указатели. Дан трехмерный массив и указатель на него:

```

int arr[L][M][N]; int *ptr;
ptr = &arr[0][0][0];

```

Теперь рассмотрим выражения:

<code>ptr</code>	указывает на элемент с индексом 0, 0, 0 в массиве <code>arr</code>
<code>ptr+i*(M*N)+j*N</code>	адрес j -й строки i -го массива $M \times N$ в массиве <code>arr</code>
<code>*(ptr+i*(M*N)+j*N*K)</code>	k -й элемент j -й строки i -го массива $M \times N$ в массиве <code>arr</code>

Важным элементом любого языка программирования являются строки, позволяющие выводить сообщения, а также реализовывать универсальные процедуры ввода данных.

Строка — это одномерный массив символов, заканчивающийся нулевым символом. В языке Си признаком окончания строки (нулевым символом) служит символ `'\0'`. Таким образом, строка содержит симво-

лы — элементы строки, а также нулевой символ. Функции для работы со строками определены в заголовочном файле `string.h` (см. Приложение А).

Объявляя массив символов, предназначенный для хранения строки, необходимо предусмотреть место для символа `'\0'`, т.е. указать его размер на один символ больше, чем наибольшее предполагаемое количество символов. Например, объявление массива `str`, предназначенного для хранения строки из 10 символов, должно выглядеть так: `char str[11]`. Последний, 11-й байт предназначен для нулевого символа.

Записанная в тексте программы строка символов, заключенных в двойные кавычки, является строковой константой, например:

`"некоторая строка"`.

В конец строковой константы компилятор автоматически добавляет нулевой символ.

Строковая переменная — это не что иное, как массив символов. Пример объявления:

```
char s[10]; // строка из 9 символов
```

Почему из 9 символов, а не из 10? Вспомним, что последний символ всегда `'\0'` — признак окончания строки (нуль-символ).

Если мы занесем в переменную `s` строку «Привет!», символы будут расположены так:

П	р	и	в	е	т	!	\0	?	?
---	---	---	---	---	---	---	----	---	---

Итак, синтаксис объявления строковой переменной:

```
char имя_строковой переменной [Размер+1];
```

Строковую переменную следует инициализировать при объявлении:

```
char hello[20] = "Всем привет!";
```

Операции `=` и `==` в применении к строкам типа `char` не работают.

Нельзя написать так:

```
char my_string[15]; // Объявление
```

```
... ..
```

```
// Попытка инициализации - ошибка!
```

```
my_string = "Пример строки";
```

В рассмотренном ранее фрагменте кода знак `=` используется при инициализации строковой переменной, и там он *допустим*. Прямое же присвоение значения *недопустимо*.

Если строка — это массив, то логично предположить, что и обращаться к ее элементам можно так же, как мы обращаемся к элементам массива, т.е. по индексу:

```
char my_string[15] = "Пример строки";  
.  
.  
.  
printf( my_string[4]); // Выведет 'e'
```

Если мы хотим изменить значение строковой переменной во время исполнения, необходимо использовать функцию `strcpy`:

```
strcpy(my_string, "Пример строки");
```

Синтаксис функции `strcpy`:

```
strcpy(целевая_строка, копируемая_строка);
```

Операцию сравнения `==` применять для непосредственного сравнения двух строк нельзя. Для этого используйте функцию `strcmp`:

```
if (strcmp(str1, str2))  
    printf("Строки не одинаковые");
```

О результате сравнения можно судить по знаку значения, возвращаемого `strcmp`:

Если	Знак результата
<code>str1 < str2</code>	<code>< 0</code>
<code>str1 > str2</code>	<code>> 0</code>
<code>str1 = str2</code>	<code>= 0</code>

Размер строки можно определить с помощью функции `strlen`:

```
int i = strlen(my_string);
```

Функция `strcat` позволяет объединить две строки в одну. Синтаксис функции:

```
strcat(целевая_строка, копируемая_строка);
```

Пример 2.12. Объединение строк.

```
char s1 = "Привет", s2 = " всем!";  
strcat(s1, s2);  
printf(s1); // Выводит «Привет всем!»
```

Для снятия ограничений по длине следует объявлять строку без размера или указатель на строку:

```
char s1[] = "Строка неограниченной длины";  
char *s2 = "Указатель на строку";
```

2.1.2 Операторы управления Си

Условные операторы предназначены для управления ходом выполнения программы. Они позволяют вводить *ветвления* в программе, когда в зависимости от заданного условия выполнение программы производится по той или иной *ветви*. Иногда их называют *операторами принятия решения*.

Условный оператор if позволяет производить вычисления, *если* выполняется определенное *условие*. Формат оператора:

```
if (условие) оператор; [else оператор2;]
```

Эту запись можно расшифровать так: *если* (*if*) условие истинно, *то* выполняется оператор, *иначе* (*else*) выполняется оператор2. Квадратные скобки указывают на необязательный элемент синтаксиса.

Внимание! Условие должно быть заключено в круглые скобки;

Если нужно выполнить не один оператор, а несколько, используйте блок — набор операторов, заключенных в фигурные скобки.

Пример 2.13. Операторы *if*.

```
if (x==0) y++; // Если x=0, то увеличиваем y
              // на единицу
if (y<=0 || x>2) // Если y меньше или равен 0
              // ИЛИ x больше 2, то:
{
    y=x++;      // 1. Присваиваем y значение x
    x*=y;       // 2. Увеличиваем x на единицу
}              // 3. Умножаем x на y и присваиваем
              // результат операции переменной x
if (z>3 && a*4>=3.51) // Если z больше 3 И
{                  // a*4 больше или равно 3.51, то:
    y += 20*x; z = y/4; // 1. y = y + 20x
    k = 21;           // 2. z = y/4
}                    // 3. k = 21
else y = 2;          // Иначе y = 2
```

Во втором и третьем операторах *if* при соблюдении условия выполняется несколько операторов, поэтому мы применили блок.

Оператор варианта switch производит сопоставление заданного значения (*управляющей переменной*) с множеством констант. Формат оператора варианта:

```
switch (управляющая_переменная)
{
    case Значение 1:
        операторы;
        break;
    case Значение 2:
        операторы;
        break;
    . . . . .
    case Значение n:
        операторы;
        break;
    [default: операторы;
        break;]
}
```

Управляющая_переменная может быть целого типа или символом.

*Операторы **break*** применяются для выхода из оператора switch. Если проверяемое значение не совпадает ни с одной из констант, выполняется вариант default (умолчание).

Пример 2.14. Оператор switch.

```
int q;
. . .
switch (q) {
    case 0: x = 2.05*z - 3.6;
        y = 5*x;
        break;
    case 1: x = 3; y = 2; z = 1;
        break;
    case 2: x = 2*y - 6.7*z + 1;
        y = 5*y - x;
        z = x;
```

```

    break;
default: x++; y/= z; z = x+1; break;
}

```

В данном примере в качестве управляющей переменной используется переменная q целого типа. Если $q=0$, то выполняется расчет по формулам: $x=2.05*z - 3.6$ и $y=5*x$; если $q=1$, то переменным x , y и z присваиваются значения: $x=3$, $y=2$ и $z=1$; если $q=2$, то выполняется расчет по формулам: $x=2*y - 6.7*z + 1$, $y=5*y - x$ и $z = x$. Если же q не равно ни 0, ни 1, ни 2, выполняется расчет по формулам: $x++$, $y/= z$ и $z = x+1$.

Если же альтернатив для выбора несколько, их перечисляют подряд, как показано в следующем примере.

Пример 2.15. Оператор switch с несколькими альтернативами.

```

switch (a)
{
case 0: x = 2*a;    // Если a = 0, то:
    y++;           // x = 2a, y = y + 1
    break;
case -2:           // Если a = -2, то:
    x = 2; y = 3;  // x = 2, y = 3
    break;
case 1:           // Если a = 1,
case 2:           // или a = 2,
case 7:           // или a = 7,
case 8:           // или a = 8, то:
    y = 2*a - 1;  // y = 2a - 1
    break;
}

```

Обратите внимание на последние четыре строки во втором примере. Они указывают на то, что если переменная a входит в множество $\{1; 2; 7; 8\}$, выполняется оператор $y = 2*a - 1$.

Оператор безусловного перехода **goto** осуществляет непосредственный переход к указанной строке программы (без условий). Формат оператора:

```
goto имя_метки;
```

Как только оператор `goto` встречается в программе, сразу (без всяких условий) осуществляется переход на строку, которая начинается с *метки*, имеющей имя `имя_метки`. *Метка* — это набор символов (цифр и/или букв). Меткам желательно давать осмысленные имена.

Рассмотрим использование оператора `goto` на примере.

Пример 2.16. Использование `goto`.

```
if (size >= 10) goto label1;
goto label2;
label1: cost = cost*1.05; flag = 2;
label2: bill = cost*flag;
```

Этот фрагмент программы работает следующим образом. Если значение переменной `size` больше или равно 10, осуществляется переход на строку, помеченную меткой `label1` и выполняются операторы `cost = cost*1.05` и `flag = 2`. Если условие не выполняется, осуществляется переход на строку, помеченную меткой `label2` и выполняется оператор `bill = cost*flag`.

Оператор `goto` — это анахронизм, пришедший из более ранних языков программирования, таких, как Fortran и Basic. Если есть возможность не использовать его, то исключайте этот оператор из своих программ. Так, например, текст примера 2.16 можно записать следующим образом:

```
if (size >= 10)
{
    cost = cost*1.05;
    flag = 2;
}
bill = cost*flag;
```

В последнем случае программа стала более понятной и не содержит «подводных камней».

Конечно, из всех правил всегда существуют исключения. Так, например, оператор `goto` удобно применять для выхода из сложного вложенного цикла при обнаружении ошибок.

Цикл — это последовательность повторяющихся действий (операторов). Количество повторений задается либо непосредственно, либо определяется условием.

В Си имеется три разновидности циклов:

- цикл с параметром `for`;
- цикл с предусловием `while`;
- цикл с постусловием `do while`.

Каждый вариант цикла обеспечивает определенные условия выполнения программы, которые мы рассмотрим ниже.

*Оператор цикла **for*** позволяет задать повторение группы операторов заданное количество раз. Такой цикл называют *циклом с параметром*.

Формат записи оператора `for` следующий:

`for` (инициализация; условие; коррекция) оператор;

Здесь:

инициализация — установка начального значения переменной-счетчика;

условие — выражение, определяющее условие, при котором цикл будет выполняться;

коррекция — изменение значения переменной-счетчика.

Алгоритм работы цикла `for`:

1. Вычисляется выражение инициализации, в результате чего переменной-счетчику присваивается начальное значение.

2. Вычисляется условное выражение. Если условие истинно, то выполняется оператор. Если условие ложно, цикл завершается.

3. Вычисляется выражение коррекции, в результате чего переменной-счетчику присваивается новое значение.

4. Осуществляется переход к п. 2.

Рассмотрим простой пример использования оператора `for`.

Пример 2.17. Простой оператор `for`.

```
int n;  
for (n=0;n<10;n++) printf("%d \n",n);
```

В результате выполнения приведенного оператора на экран будут выведены цифры от 0 до 9.

Если нужно выполнить не один оператор, а несколько, включайте их в блок, как показано в следующем примере.

Пример 2.18. Оператор for с блоком.

```
int j=0;
for (j=0;j<5; j++)
{
    float x = 20*sin(j);
    float y = 3*x;
    printf("%d x = %f y = %f \n", j, x, y); }
```

В данном примере 5 раз рассчитываются и выводятся на экран значения функций $x(j)=20*\sin(j)$ и $3*x$. Символы %d и %f в функции printf указывают на позиции вывода целого и вещественного значений соответственно.

Коррекция не обязательно должна быть целочисленной. В следующем примере в разделе коррекции значение управляющей переменной вычисляется по формуле.

Пример 2.19. Оператор for со сложной коррекцией.

```
int x; float y;
for (x=1;y<=75;y=5*x++ + 10)
    printf("x =%d y = %f \n", x, y);
```

Обратите особое внимание на пример 2.19. Если в двух предыдущих примерах для проверки условия и коррекции использовалась переменная-“счетчик цикла”, то в примере 2.19 в для проверки условия и для коррекции используется дополнительная переменная y.

А вот в следующем примере отсутствуют выражения в двух разделах — инициализации и коррекции. Цикл будет выполняться до тех пор, пока Вы не введете число, большее и равное 5.

Пример 2.20. Оператор for без инициализации и коррекции.

```
#include <stdio.h>
int main() {
    int num = 0;
    printf("%s \n", "Enter number");
    for (; num<=5;)
        // Enter number — Введите число
        scanf("%d", num); }
```

```
printf("It is good number!");
return 0;
}
```

Шаг коррекции может быть и дробным.

Пример 2.21. Цикл `for` с дробной коррекцией.

```
for (float i=0; i<2; i+=0.2)
```

В этом примере приращение переменной-счетчика цикла осуществляется с шагом 0.2.

Применение в разделах инициализации и коррекции *операции «запятая»* позволяет существенно увеличить гибкость цикла `for`.

Рассмотрим это на примере программы, которая вычисляет стоимость отправления посылки на почте. Тариф оценки такой: 120 руб. за первый килограмм и по 35,6 руб. за каждые последующие 200 грамм.

Пример 2.22. Использование оператора `for` для вывода таблицы.

```
#include <stdio.h>
#include <locale.h>
#define FIRST 120
#define NEXT 35.6
int main() {
    float weight;
    float cost;
    char* s = "Вес, кг Стоимость, руб.";
    setlocale(LC_ALL, "RUS");
    printf("%s \n", s);
    for (weight=1, cost=FIRST; weight <=10;
weight++, cost+=NEXT)
        printf("%2d %.2f \n", weight, cost);
    return 0;
}
```

Мы использовали запятую в разделах инициализации и коррекции. Это позволило инициализировать сразу две переменные: `weight` (вес) и `cost` (цена). Кроме того, в разделе коррекции мы изменяем значения этих же переменных: переменная `weight` на каждом шаге цикла увеличивается на единицу, а переменная `cost` увеличивается на 35.6.

Условие выполнения цикла определяется с помощью переменной `weight`.

Функция `setlocale` позволяет корректно отображать символы кириллицы на экране. Ее прототип находится в библиотеке `locale.h` (см. Приложение А).

Оператор `while` организует цикл с предусловием. Термин *предусловие* означает, что условие выполнения цикла вычисляется каждый раз в начале цикла. Оператор имеет следующий формат:

```
while (выражение) оператор;
```

Оператор повторяется до тех пор, *пока* выражение не станет ложным или нулем. Выражение может быть произвольного типа.

Рассмотрим примеры.

Пример 2.23. Простой цикл `while`.

```
int df = 2, f = 5;
while (f<100)
{
    f = f+df;
    printf(f);
}
```

Приведенный фрагмент программы выводит числа от 5 до 100.

Пример 2.24. Цикл `while` с условием, определяемым вводом пользователя.

```
#include <stdio.h>
#include <locale.h>
int guess = 1;
char response;
int main () {
    setlocale(LC_ALL, "RUS");
    printf("Загадайте число от 1 до 100 \n");
    printf("А я попробую угадать его \n");
    printf("Отвечайте Y, если догадка верна, \n");
    printf("и N, если я ошибаюсь \n");
    printf("Итак, Ваше число . . .\n");
    scanf("%d", guess);
    while ((response=getch())!='Y')
```

```

    if (response!='\n')    {
printf("Ну, тогда оно равно . . . \n");
printf("%d \n", ++guess);    }
printf ("Число угадано !!! \n");
return 0; }

```

В приведенном примере цикл продолжается до тех пор, пока Вы не ответите Y (Yes). Алгоритм «угадывания» очень простой. Сначала программа предложит вам число 1, потом 2, 3, и т.д. Когда число, подсчитанное программой, будет равно тому, которое Вы загадали, Вы, конечно, ответите Y.

Оба цикла, `for` и `while`, фактически являются *циклами с предусловием*. Проверка истинности условия осуществляется перед началом каждой итерации цикла.

Оператор `do while` позволяет изменить порядок проверки, — он проверяет условие после окончания каждой итерации цикла, то есть является *циклом с постусловием*. Формат оператора:

```
do оператор while (выражение);
```

Оператор повторяется до тех пор, пока выражение не станет ложным или нулем.

Пример 2.25. Цикл `do while`.

```

#include <stdio.h>
#include <math.h>
void main() {
int i =10; double z, y = 5.6, m;
do
{ z = 22 - (y/3.0)*i; m = 33.3*cos(z);
printf("i = %d z = %f \n", i, z);
}
while (--i);
}

```

Подключение заголовочного файла `math.h` в начале программы потребовалось для вызова стандартной функции косинуса.

Определите самостоятельно, что выведет эта программа на экран.

В программе циклы могут *вкладываться* друг в друга.

Вложенным называется цикл, находящийся внутри другого цикла. Количество вложений не ограничено. Однако на практике количество вложений находится в пределах 1–3.

Пример 2.26. Вложенные циклы.

```
int j, k, z;
for (j=0; j<10; j++) {
    k = j-1;
    while (k<5) {
        z = 2*k; k--;
        printf("Z=%d", z);
    }
}
```

В приведенном фрагменте программы во *внутреннем цикле* на экран выводится переменная *z* до тех пор, пока переменная *k* меньше 5. Величина *k* при этом определяется переменной *j* во *внешнем цикле*. Определите самостоятельно, что будет выведено на экран.

Операторы циклов и условные операторы, рассмотренные нами выше, являются важнейшими средствами управления выполнением программы на Си. Однако есть еще два не менее важных оператора — *break* и *continue*, которые позволяют расширить возможности других операторов управления.

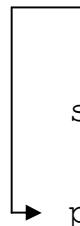
С оператором *break* мы уже познакомились при изучении оператора *switch*. Рассмотрим его подробнее.

Оператор break используется для завершения циклов и блоков в операторе *switch*. Когда в ходе выполнения программы встречается это выражение, его выполнение приводит к выходу из конструкции *switch*, *for*, *while* или *do while*, в которых он содержится, и переходу к следующему оператору программы. Если же выражение *break* находится внутри некоторой совокупности вложенных структур, его действие распространяется только на самую внутреннюю структуру, в которой он содержится.

В приведенной ниже программе показано, как работает выражение *break*, расположенное внутри цикла *while*.

Пример 2.27. Оператор break в цикле while.

```
#include <stdio.h>
#include <locale.h>
void main()
{
    setlocale(LC_ALL, "RUS");
    int num, sum, count;
    printf("Введите 10 положительных чисел \n");
    while (++count <=10)
    {
        scanf("%d", num);
        if (num <0) {
            printf("Ошибка: \n");
            printf("Введено отрицательное число \n");
            printf("на шаге %d", count);
            break;
        }
        sum = sum + number;
    }
    printf("Сумма %d чисел равна %d",
           (count-1), sum); }
```



Выражение continue может использоваться во всех трех видах циклов. Оно завершает текущую итерацию цикла и начинает новую. Перепишем приведенную выше программу указанным образом.

Пример 2.28. Использование оператора continue.

```
#include <stdio.h>
#include <locale.h>
void main() {
    setlocale(LC_ALL, "RUS");
    int num, sum, count;
    printf("Введите 10 положительных чисел \n");
    while (++count <=10)
    {
```

```

→ scanf("%d", num);
   if (num < 0) {
       printf("Ошибка: \n");
       printf("Введено отрицательное число \n");
       printf("на шаге %d", count);
       printf("Повторите ввод \n");
       count--;
       continue;
       break;    }
   sum = sum + number;
   }
   printf("Сумма %d чисел равна %d",
         (count-1), sum);
   }

```

Программа стала более гибкой,— вместо того, чтобы завершать работу при вводе неверного числа, она предложит пользователю повторить ввод и вернет на начало цикла.

2.1.3 Функции

Функция — это полностью завершённый именованный блок программы Си, который выполняет заданный набор действий.

Синтаксис объявления функции в Си следующий:

```

Тип ИмяФункции ( [входные параметры] )
{
    тело функции
}

```

Тип определяет тип результата, возвращаемого функцией. Функция может возвращать результат любого типа данных. Входные параметры — это список внутренних переменных функции, через которые в функцию передаются значения из основной программы. Функция может быть и без параметров, тогда их список будет пустым. Такой пустой список можно указать и в явном виде, поместив для этого внутри скобок ключевое слово `void`.

Все входные параметры функций должны объявляться отдельно, причем для каждого из них надо указывать и тип, и имя. Таким образом, список объявлений параметров должен выглядеть следующим образом:

тип ИП1, тип ИП2, ..., тип ИПN,

где ИП — имя переменной.

Пример 2.29. Объявление переменных в заголовке функции.

```
f(int i, char n, int j) /* правильно */
f(int i, k, float j) /* неправильно, для
                     переменной k должен быть указан
                     собственный спецификатор типа */
```

Переменные, определенные внутри функции, считаются локальными. Единственное исключение из этого правила — переменные, объявленные со спецификатором класса памяти `static`.

В Си существует два способа передачи параметров в функции — по значению и по ссылке. Для передачи по ссылке используются указатели.

Пример 2.30. Передача параметров в функцию.

```
float funct(int x, float y, int *z)
{
    float m = x*2 + y/4;
    float d = z*4;
    &z = 2;
}
```

В данном примере параметры `x` и `y` передаются по значению, а параметр `z` — по ссылке.

Если в качестве аргумента функции используется массив, то функции передается его адрес. Как мы выяснили ранее, имя массива — это указатель на его первый элемент, поэтому при передаче массива мы фактически используем механизм передачи по ссылке. В этом и состоит исключение по отношению к правилу, которое гласит, что при передаче параметров используется вызов по значению.

2.1.4 Структура программы Си

Программа Си — это определение функции `main`, которая для выполнения необходимых действий может вызывать другие функции.

Программный проект может содержать несколько файлов, как показано на рисунок 2.2.

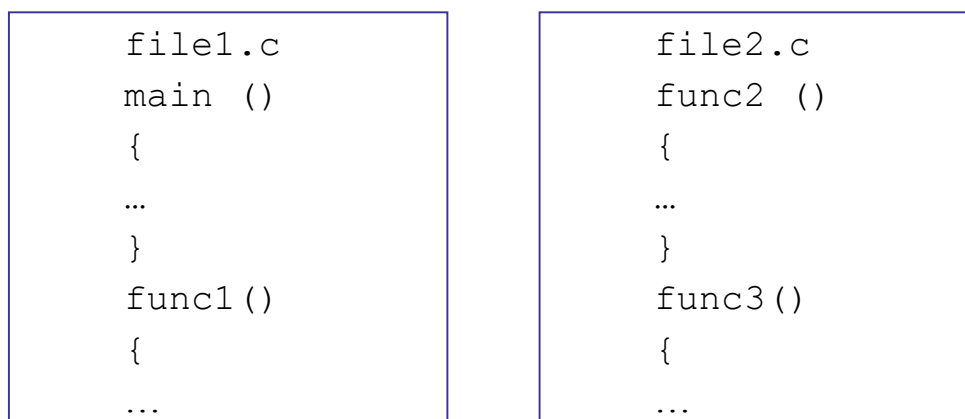


Рисунок 2.2 — Программа Си, состоящая из 2-х файлов

В примере, приведенном на рисунке 2.2, функция `main()` может вызвать любую из 3-х объявленных функций: `func1()`, `func2()` и `func3()`. Любая функция из указанных трех может также вызвать другую функцию, кроме `main()`. Функция `main` может быть в программном проекте только в единственном числе!

Для того, чтобы одна функция могла вызывать другую, она должна делать это через переменные, при этом для корректной работы переменные должны иметь соответствующие классы памяти (рис. 2.3).

В программах Си могут использоваться классы памяти `static` и `extern`.

```
static int i=0; // инициализация
```

или

```
static int i;
```

Переменная, использующая класс памяти `extern`, может быть определена на внешнем уровне только один раз.

Заголовочные файлы подключаются с помощью директивы `#include`. Они обычно имеют расширение `*.h` (первая буква слова `header` — англ. заголовок).

Объявление переменных на внутреннем уровне

```
main ()
{
static int counter=0;
. . .
. . .
}
```

Могут использоваться
любые классы памяти.

```
file1.cpp
main ()
{
extern long counter;
...
}
func1 ()
{
...
}
```

```
file2.cpp
long coun-
ter=0;
func2 ()
{
...
}
func3 ()
```

Объявление переменных на внешнем уровне

Рисунок 2.3 — Классы памяти переменных в программе Си

В заголовочных файлах могут содержаться:

- определения пользовательских типов;
- описания прототипов функций;
- описание данных;
- определение констант;
- перечисления;
- директивы `#include`;
- определения макросов;
- комментарии.

Директива `#include` подключает копию указанного в директиве файла в то место программы, где эта директива находится.

Директива `#include <имя_файла.h>` используется для подключения файлов стандартных библиотек, например:

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

Директива `#include "имя_файла.h"` — для подключения собственных библиотечных модулей, например:

```
#include "mylib.h"
```

В заголовок файла с текстом программы могут также включаться *директивы препроцессора*. *Препроцессор* — это специальная программа, осуществляющая предварительную обработку текста программы.

Директива `#define` служит для создания констант и макросов.

Формат записи:

```
#define Имя_константы замещающий_текст
```

Если замещающий текст слишком длинный, его можно перенести на следующую строку, введя символ “\”.

После записи такой строки все встретившиеся далее в тексте программы имена, совпадающие с элементом `Имя_константы`, будут автоматически заменены на `замещающий_текст` прежде, чем начнется компиляция программы.

Например, директива

```
#define Pi 3.14159
```

определяет константу `Pi`.

Директива `#undef` используется для аннулирования определений символических констант и макросов. Формат:

```
#undef идентификатор
```

Пример 2.31. Директивы `define` и `undef`.

```
#define MyConst 128 // Определение константы
```

```
. . . . .
```

```
Другие операторы программы
```

```
. . . . .
```

```
#undef MyConst // Отмена определения
```

```
. . . . .
```

```
Другие операторы программы
```

```
. . . . .
```

```
#define MyConst 55 // Новое определение
```

2.2 Основы языка программирования Ассемблер

2.2.1 Структура программы на Ассемблере

Программа на Ассемблере состоит из следующих основных разделов, именуемых сегментами:

- сегмент стека (обозначается `stack`);

- сегмент кода (обозначается `code`);
- сегмент данных (обозначается `data`).

Сегмент стека представляет собой область оперативной памяти компьютера, отведенной под стек. Через стек осуществляется передача параметров в функции, в нем же хранятся различные служебные данные.

Сегмент кода содержит команды (операторы) программ.

Сегмент данных содержит объявления данных и переменных.

Запись ключевых слов зависит от диалекта ассемблера.

В различных диалектах Ассемблера объявление разделов могут отличаться. Ниже приведены примеры для программ TASM и NASM.

Пример 2.32. Объявление разделов в Turbo Assembler.

```
.STACK 100h ; сегмент стека
.DATA      ; сегмент данных
Message DB 'Hello World!',13,10,'$'
.CODE      ; сегмент кода
mov ax,@Data
mov ds,ax
mov ah,9
. . . .
END        ; завершение сегмента кода
```

Пример 2.33. Объявление разделов в NASM.

```
segment data
dvar: dw 1234
segment code
function: mov ax,data
inc word [dvar]
ret
. . . .
```

Если объем программного кода значительный, программу разделяют на части (модули), которые записывают в отдельные файлы. При такой организации один модуль (файл) содержит главную программу, а в других модулях размещены подпрограммы.

Модули обычно транслируют и отлаживают отдельно. Такой подход, во-первых, упрощает настройку объемных программ, и, во-вторых, позволяет создавать библиотеки подпрограмм, которые можно использовать в других проектах.

Как видно из примеров 2.32, 2.33, команды Ассемблера отличаются от команд языка Си краткостью и минимальным количеством разделителей (запятых, точек, скобок и т.п.).

2.2.2 Синтаксис команд Ассемблера

Команда (оператор) в программе на Ассемблере имеет следующий синтаксис:

[метка] команда [операнд1 [, операнд 2]]

Для того, чтобы пояснить значения команд, в программу помещают комментарии. Комментарий в Ассемблере начинается от точки с запятой и продолжается до конца строки:

```
; Вывод сообщения на экран
mov ah, 9 ; функция DOS вывода строки
```

Текст, находящийся справа за точкой с запятой, является комментарием, который транслятор игнорирует.

Комментарии в программах Ассемблера только однострочные!

2.2.3 Типы данных и переменные

В программах на Ассемблере возможно размещать данные (числа, строки, адреса) в сегменте данных как в виде набора значений с указанием типа, так и в виде именованных переменных.

Объявление данных и переменных начинается с типа данных, за которым следуют значения.

Тип данных определяет диапазон возможных значений. Так, для DB возможные значения лежат пределах от 0 до 2^8-1 , т.е. от 0 до 255, для DW — от 0 до 65535 ($2^{16}-1$), и т.д. Типы данных Ассемблера перечислены в таблице 2.5.

Таблица 2.5 — Типы данных Ассемблера

Тип данных	Обозначение	Размер, байт
Байт	DB	1
Слово	DW	2
Двойное слово	DD	4
Четверное слово	DQ	8
Целое, 10 байт	DT	10

Формат объявления переменной в Ассемблере следующий:

[Имя_переменной] Тип Значение,

где Имя_переменной — необязательное имя переменной для обращения к ней в программе;

Тип — тип данных, хранящийся в переменной (см. табл. 2.1);

Значение — число или строка.

Если Значение для переменной заранее не известно, можно вместо него записать знак вопроса ?, — в этом случае для переменной будет выделена память соответствующего объема.

Как видно из рассмотренного синтаксиса объявления, имя переменной не является необходимым атрибутом (в отличие от переменных в программах Си). Это объясняется тем, что в программе Ассемблера мы можем обращаться к переменной, указывая ее имя, или же указывая адрес, по которому записано имя этой переменной в памяти компьютера.

Пример 2.34. Объявление переменных в Ассемблере.

```
VAR1 DB 44
msg DB 'Hello, world!', 0dh, 0ah, '$'
VAR22 DW ?
X DW 10h
ARR1 DB 30 DUP(?)
ARR2 DB 10 DUP(11)
MATR DW DUP(2 DUP(1)) ;
```

В приведенном примере переменные принимают следующие значения:

- переменная VAR1 содержит целое число 44;
- переменная msg содержит строку 'Hello, world'. Символы 0dh, 0ah означают переход на новую строку, а спецсимвол \$ завершает строку;
- переменная VAR22 типа «слово» изначально не содержит значения (пустая);
- переменная X содержит шестнадцатеричное значение 10h;
- переменные ARR1, ARR2, MATR - это массивы, содержащие соответственно 30, 10 и 4 элемента. При этом массив ARR1 не содержит

начальных значений, массив ARR2 заполняется числами 11, а массив MATR заполнен единицами.

Обратите особое внимание на особенности объявления строковой переменной msg, и массивов ARR1, ARR2, MATR. Так как эти переменные представляют собой группу байтов, то их имена — это ссылки на первый байт в группе.

2.2.4 Пересылка, адресация и обмен данными

Рассмотрим одну из наиболее часто употребляемых команд языка Ассемблер — команду пересылки данных MOV. Ее назначение — пересылка данных из одного регистра в другой, и из регистра — в память компьютера.

Синтаксис команды следующий:

MOV приемник, источник,

где приемник — регистр или ячейка памяти (переменная), в которую пересылаются данные;

источник — регистр или ячейка памяти (переменная), из которой пересылаются данные.

Пример 2.35. Пересылка данных командой MOV.

MOV AX, BX ; AX ← BX

MOV ES, AX ; ES ← AX

MOV AX, 10 ; AX ← 10

MOV EAX, 0FFFFFFH ; EAX ← 0FFFFFFH

При использовании команды MOV следует учитывать следующие ограничения:

1. Нельзя пересылать данные из одной области памяти в другую непосредственно. Это можно сделать только через регистры общего назначения, например, так:

VAR1 DB 11 ; Объявляем две переменные

VAR2 DB 55 ;

. . . .

MOV AL, VAR1 ; Пересылаем данные из VAR1 в AL

MOV VAR2, AL ; Теперь из AL в VAR2

; После операции: VAR1 = 11 VAR2 = 11

2. Нельзя помещать данные из памяти в сегментные регистры. Как и в предыдущем случае, это можно сделать через регистры общего назначения (РОН):

```
MOV ax, VAR
```

```
MOV ds, ax
```

3. Нельзя производить пересылку данных из одного сегментного регистра в другой напрямую, только через РОН:

```
MOV ax, es
```

```
MOV ds, ax
```

4. Нельзя использовать сегментный регистр CS в качестве приемника, так как в паре регистров CS:IP содержится адрес следующей команды.

К командам пересылки данных относится также команда обмена XCHG (exchange). Она имеет следующий синтаксис:

```
XCHG операнд1, операнд2,
```

где операнд1, операнд2 — регистр или ячейка памяти.

Пример 2.36. Использование команды обмена XCHG.

```
MOV AX, 20 ; AX содержит 20
```

```
MOV CX, 10 ; CX содержит 10
```

```
XCHG AX, CX ; поменять местами
```

```
; Теперь AX содержит 10, а CX содержит 20
```

Как мы уже знаем, для обращения к какому-либо устройству, или ячейке памяти (хранящей, например, значение переменной), необходимо знать их адреса. Для указания адреса в программах Ассемблера используют несколько способов:

- регистровая адресация;
- непосредственная адресация;
- прямая адресация памяти:
 1. базовая и индексная;
 2. базовая и индексная со смещением;
 3. базово-индексная;
 4. базово-индексная со смещением.

При использовании *регистровой адресации* один из операндов (байт или слово) находится в регистре.

Пример 2.37. Регистровая адресация.

```
MOV DS, AX
```

При *непосредственной адресации* операнд размещается в самой команде.

Пример 2.38. Непосредственная адресация.

```
MOV AX, 4A0Ch ; Операнд – 16-ричное число  
MOV DX, offset Var ; Смещение для переменной  
; Var заносится в DX
```

При *прямой адресации* относительный адрес операнда содержится в виде смещения.

Пример 2.39. Прямая адресация.

```
MOV AL, off_db; Off_db – смещение для байта  
MOV AX, off_dw; Off_dw – смещение для слова
```

При *базовой и индексной адресации* памяти относительный адрес ячейки памяти находится в регистре, обозначение которого заключено в квадратные скобки. При использовании регистров BX или BP адресацию называют *базовой*, при использовании регистров SI или DI – *индексной*.

При адресации через регистры BX, SI или DI в качестве сегментного регистра всегда подразумевается DS, при адресации через регистр BP используется регистр SS.

Пример 2.40. Базовая и индексная адресация.

```
MOV AL, [BX]  
; Сегментный адрес в DS, смещение в BX  
MOV DL, ES:[BX]  
; Сегментный адрес ES, смещение в BX  
MOV DX, [BP]  
; Сегментный адрес в SS, смещение в BP  
MOV AL, [DI]  
; Сегментный адрес в DS, смещение в DI
```

Базовая и индексная адресации со смещением предполагает, что относительный адрес операнда определяется суммой содержимого регистра (BX, BP, SI или DI) и указанного в команде числа, которое называют смещением.

Пример 2.41. Базовая и индексная адресация со смещением.

```
; Объявляем массив символов
```

```

mas db 1,2,5,6,7,9,1,3,2
MOV BX,2
; BX — индекс элемента в массиве
MOV DL,mass[BX]
; В DL заносится третий элемент массива
; (адресация с 0), т.е. число 5
; Предполагается, что базовый адрес
; сегмента, в который входит массив mas,
; загружен в DS

```

При *базово-индексной адресации* относительный адрес операнда равен сумме содержимого базового и индексного регистров. Допускается использование следующих пар: [BX][SI], [BX][DI], [BP][SI] и [BP][DI]. Если в качестве базового регистра используется BX, то в качестве сегментного подразумевается DS, при использовании в качестве базового регистра BP сегментным регистром по умолчанию назначается SS. При необходимости можно явно указать непосредственно требуемый сегментный регистр.

Пример 2.42. Базово-индексная адресация.

```

MOV BX,[BP][SI]
; В BX заносится слово из стека
; (сегментный адрес в SS),
; а смещение вычисляется как сумма
; содержимого BP и SI
MOV BX,ES:[BP][SI]
; В BX заносится слово из сегмента,
; адрес которого находится в ES,
; а смещение вычисляется как
; сумма содержимого BP и SI
MOV ES:[BX+DI],AX
; В ячейку памяти, сегментный адрес
; которой хранится в ES, а смещение
; равно сумме содержимого BX и DI,
; пересылается содержимое AX

```

Базово-индексная адресация со смещением предполагает, что относительный адрес операнда определяется суммой содержимого базового и

индексного регистров, а также смещения. Допускается использование тех же пар регистров, что и при базово-индексном способе адресации.

Пример 2.43. Базово-индексная адресация со смещением.

```
MOV mas [BX] [SI] , 20
; Число 20 пересылается в ячейку памяти,
; сегментный адрес которой хранится в DS, а
; смещение равно сумме содержимого
; BX и SI и смещения ячейки mas
MOV AX, [BP+2+DI]
; В AX пересылается из стека слово,
; смещение которого равно сумме BP, DI и 2
```

Особое место в программах Ассемблера занимают операции со стеком. *Стек* — это область оперативной памяти, выделяемая для выполнения операций обмена данными, передачи данных в подпрограммы, а также для временного хранения значений регистров и переменных при обработке прерываний. Стек представляет собой одномерный массив, доступ к содержимому которого реализован по принципу «последним вошел — первый вышел», т.е. данные, помещенные в стек первыми, могут быть извлечены последними.

Для работы со стеком используются следующие регистры:

- RSP (ESP, SP) — регистр указателя стека, содержащий указатель на вершину стека в текущем сегменте стека;
- RBP (EBP, BP) — регистр указателя базы кадра стека, организующий произвольный доступ к данным внутри стека;
- SS обеспечивает доступ к сегменту стека.

Адрес, хранящийся в регистре RSP (ESP, SP), называется *вершиной стека*.

Если необходимо получить доступ к элементам не на вершине, а *внутри* стека, то необходимо использовать регистр RBP (EBP, BP) — это регистр указателя базы кадра стека, где можно записать необходимый адрес стека.

При работе со стеком используются две команды: PUSH — поместить в стек, и POP — извлечь из стека.

Пример 2.44. Занесение данных в стек и извлечение из него.

```
PUSH AX ; заносим в стек содержимое AX
```

PUSH BX ; заносим в стек содержимое BX

. . . .

POP BX ; извлекаем из стека содержимое BX

POP AX ; извлекаем из стека содержимое BX

Есть также команды работы со стеком, позволяющие сохранить и восстановить «пакетом» содержимое регистров AX, DX, CX, DX, SP, BP, SI и DI — это PUSHА и POPА; а также сохранять и восстанавливать регистр флагов — PUSHF и POPF.

2.2.5 Арифметические и логические операции

Современные микропроцессоры обеспечивают реализацию базовых арифметических и логических операций с двоичными и двоично-десятичными числами.

Следует отметить, что значительная часть арифметических операций в Ассемблере выполняются над числами без знака. Однако, так как знак числа в двоичных числах представлен старшим битом, всегда имеется возможность это учесть.

К арифметическим операциям Ассемблера относятся: сложение, вычитание, умножение и деление, а также инкремент (увеличение на единицу) и декремент (уменьшение на единицу).

Для *сложения* предусмотрены две команды: ADD (сложение без учета переноса) и ADC (сложение с учетом переноса). Синтаксис этих команд следующий:

ADD приемник, источник,

ADC приемник, источник,

где приемник, источник — операнды, участвующие в операции сложения.

Результат операции помещается в приемник.

Пример 2.45. Сложение двух байтов.

VAR1 DB 10 ; Объявляем переменные –

VAR2 DB 33 ; операнды

RES DB ? ; Переменная для хранения результата

MOV AX, VAR1 ; Помещаем значение из VAR1 в AX

ADD AX, VAR2 ; Складываем

MOV RES, AX ; Результат – в RES, RES = 43

Пример 2.46. Сложение двух двойных слов.

```
VAR1 DD 02FE540FH
VAR2 DD 0050A4CDH
flagRES DB 0 ; флаг учета переноса из
            ; старшего разряда результата
RES DD 0
MOV AX,VAR1
ADD AX, VAR2 ; сложение младших слов слагаемых
MOV RES, AX ; перенос результата в RES
MOV AX,VAR1+2
ADC ax,VAR2+2 ; сложение старших слов слагаемых
            ; и флага переноса CF
MOV RES+2,AX
ADC flagRES,0 ; учесть возможный перенос
```

Есть еще одна особая команда сложения — сложение с переносом:

XADD приемник, источник,

которая производит сложение содержимого приемника и источника, после чего приемник содержит значение суммы, а источник — начальное значение приемника.

Пример 2.47. Сложение двух байтов с обменом.

```
MOV AL,05H
MOV BL,02H
XADD MOV AL,BL ;AL=07h, BL=05h
```

Операция *вычитания* реализуется двумя командами: SUB и SBB. Первая и вторая производят целочисленное вычитание, но вторая команда учитывает заем. Синтаксис команд:

SUB приемник, источник,
SBB приемник, источник,

где приемник, источник — операнды, участвующие в операции вычитания.

Результат операции помещается в приемник.

Пример 2.48. Вычитание двух байтов.

```
VAR1 DB 20 ; Объявляем переменные -
VAR2 DB 10 ; операнды
RES DB ? ; Переменная для хранения результата
```

```
MOV AX, VAR1 ; Помещаем значение из VAR1 в AX
SUB AX, VAR2 ; Вычитаем
MOV RES, AX ; Результат — в RES, RES = 10
```

Для *умножения* в Ассемблере используют две команды: MUL — целочисленное умножение без знака и IMUL — целочисленное умножение со знаком. Синтаксис этих команд следующий:

```
MUL множитель_1
IMUL множитель_1
IMUL множитель_1, множитель_2
IMUL результат, множитель_1, множитель_2
```

Местоположение второго сомножителя фиксировано и зависит от размера первого сомножителя:

- если операнд, указанный в команде, — байт, то второй сомножитель располагается в AL;
- если операнд, указанный в команде, — слово, то второй сомножитель располагается в AX;
- если операнд, указанный в команде, — двойное слово, то второй сомножитель располагается в EAX.

Результат умножения для команды с одним операндом помещается в строго определенное место, определяемое размером сомножителей:

- при умножении байтов результат помещается в AX;
- при умножении слов результат помещается в пару DX:AX;
- при умножении двойных слов результат помещается в пару EDX:EAX.

Деление производится с помощью команды DIV — беззнаковое деление двух двоичных беззнаковых значений или IDIV — деление со знаком. Синтаксис команд деления:

```
DIV делитель
IDIV делитель
```

Команда выполняет целочисленное деление операндов с выдачей результата в виде частного и остатка от деления. При выполнении операции деления возможно возникновение исключительной ситуации: 0 — ошибка деления. Эта ситуация возникает в одном из двух случаев: делитель равен 0 или частное слишком велико для его размещения в регистре EAX/AX/AL.

Делимое задается в командах `DIV` и `IDIV` задается неявно и размер его зависит от размера делителя:

- если делитель — байт, то делимое должно быть расположено в регистре `AX`. Частное помещается в `AL`, а остаток — в `AH`;
- если делитель — слово, то делимое должно быть расположено в паре регистров `DX:AX`, причем младшая часть делимого находится в `AX`. После операции частное помещается в `AX`, а остаток — в `DX`;
- если делитель — двойное слово, то делимое должно быть расположено в паре регистров `EDX:EAX`, причем младшая часть делимого находится в `EAX`. После операции частное помещается в `EAX`, а остаток — в `EDX`.

Пример 2.49. Целочисленное деление.

```
; деление слова на байт
MOV  AX, 10234; делимое
MOV  BL, 154
DIV  BL      ; AH=остаток, AL=частное
; деление слов
MOV  AX, 1045 ; делимое
MOV  BX, 587  ; делитель
CWD          ; расширение делимого DX:AX
IDIV BX      ; частное в AX, остаток в DX
```

Команды *инкремента* `INC` и *декремента* `DEC` позволяют увеличивать и уменьшать значение числового операнда на единицу. Операнд может иметь размер байта, слова, двойного и учетверенного слова. Синтаксис операций:

```
INC операнд
DEC операнд
```

Пример 2.50. Использование операций инкремента и декремента.

```
MOV AX, 7
INC  AX ; увеличим ax на 1 : AX = 8
MOV  AL, 9
DEC  AL ; AL=8
```

Ассемблер поддерживает следующие логические операции:

- NOT (операция логического отрицания) производит поразрядное инвертирование каждого бита операнда. Результат записывается обратно в тот же операнд;
- AND (операция логического И) выполняет поразрядную логическую операцию И (конъюнкция) Результат записывается в приемник;
- OR (операция логического ИЛИ) выполняет поразрядную логическую операцию ИЛИ (дизъюнкция) Результат записывается в приемник;
- XOR (операция логического исключающего ИЛИ) выполняет поразрядную логическую операцию исключающего сложения;
- TEST (операция проверки) выполняет поразрядную логическую операцию И. Результат никуда не записывается, а идентифицируется через регистр флагов.

Синтаксис логических операций следующий:

NOT операнд

AND источник, приемник

OR источник, приемник

XOR источник, приемник

TEST источник, приемник

Пример 2.51. Использование логических операций.

; Определяем значения операндов

MOV AL, 01h ; AL = 01h

MOV BL, 10h ; BL = 10h

; Операция ИЛИ

OR BL, AL ; AL = 01h BL = 11h

;изменить значение бита 0 регистра AL на

; Исключающее ИЛИ

XOR AL, 01h ; AL = 02h

2.2.6 Управление ходом выполнения программы

К командам Ассемблера, управляющим выполнением программы, относятся команды переходов, команды организации циклов и команды вызова подпрограмм.

Команды переходов позволяют реализовать ветвление в программе в зависимости от заданных условий или результатов выполнения других команд (как правило, арифметических и логических). Среди них особое место занимает команда безусловного перехода JMP, которая

производит переход на указанную строку программы. Синтаксис команды JMP следующий:

JMP [модификатор] адрес_перехода,

где модификатор — служебные слова, указывающие на дальность перехода;

адрес_перехода — адрес строки в программе, на которую нужно произвести переход. В качестве адреса_перехода часто используют текстовую метку.

Модификатор может быть следующим:

- near ptr — прямой переход на метку внутри текущего сегмента кода (в пределах 64 кБайт);
- far ptr — прямой переход на метку в другом сегменте кода;
- word ptr — косвенный переход на метку внутри текущего сегмента кода;
- dword ptr — косвенный переход на метку в другом сегменте кода.

Рассмотрим различные варианты использования команды безусловного перехода на примерах.

Пример 2.52. Безусловные переходы в Ассемблере.

```
;1. Прямой короткий переход
JMP short pt1 ;Переход на метку shpt
;в пределах +127...-128 байт
JMP pt2 ;То же самое, если shpt
;находится выше по тексту программы
; 2. Прямой ближний переход
JMP near ptr pt3 ; Аналогично MOV
; 3. Косвенные ближние переходы
mov BX,offset pt4
; BX=адрес точки перехода
JMP BX ;Переход в точку pt
addrpt dw pt4 ;Ячейка с адресом точки перехода
JMP DS:addr ;Переход в точку pt
JMP word ptr addrpt ;То же самое
addr dw pt5 ;Ячейка с адресом точки перехода
MOV DI,offset addr ; получаем адрес ячейки
```

```

; с адресом точки перехода
JMP [DI] ; Переход в точку pt
; 4. Прямой дальний переход
JMP far ptr farpt ;Переход на метку farpt в
;другом программном сегменте
; 5. Косвенный дальний переход
addr dd pt ;Поле с двухсловным
;адресом точки перехода
JMP DS:addr ;Переход в точку pt
JMP dword ptr addr ;То же самое

```

Команды условного перехода имеют следующий синтаксис:

JCC метка_перехода,

где символы CC определяют конкретное условие, анализируемое командой (см. табл. 2.6);

метка_перехода может находиться только в *пределах текущего сегмента кода*.

Таблица 2.6 — Команды условных переходов

Команда	Переход, если	Флаги
JZ/JE	нуль или равно	ZF=1
JNZ/JNE	не нуль или не равно	ZF=0
JC/JNAE/JB	есть переполнение/не выше и не равно/ниже	CF=1
JNC/JAE/JNB	нет переполнения/выше или равно/не ниже	CF=0
JP / JNP	число единичных бит четное / нечетное	PF=1 / PF=0
JS / JNS	знак равен 1 / 0	SF=1 /SF=0
JO/JNO	Переполнение есть /нет	OF=1 / OF=0
JA/JNBE	выше/не ниже и не равно	CF=0 и ZF=0
JNA/JBE	не выше/ниже или равно	CF=1 или ZF=1
JG/JNLE	больше/не меньше и не равно	ZF=0 и SF=OF
JGE/JNL	больше или равно/не меньше	SF=OF
JL/JNGE	меньше/не больше и не равно	SF≠OF
JLE/JNG	меньше или равно/не больше	ZF=1 или SF≠OF
JCXZ	содержимое CX равно нулю	

Примечание: выше/ниже — для беззнаковых операндов; больше/меньше — для операндов со знаком.

Источниками условия передачи управления могут быть:

- 1) результат выполнения любой команды, изменяющей состояние арифметических флагов;
- 2) команда сравнения CMP, сравнивающая значения двух операндов;
- 3) содержимое регистра RCX/ECX/CX.

Команда сравнения CMP выполняет вычитание операндов и устанавливает соответствующие флаги. Синтаксис команды сравнения:

CMP операнд_1, операнд_2

Флаги, устанавливаемые командой CMP, затем анализируют соответствующими командами условного перехода (табл. 2.7). Если сравнить команды условного перехода, приведенные в таблице 2.6, с командами в табл. 2.7, можно увидеть, что они одинаковы по написанию, но различаются по смыслу.

Таблица 2.7 — Команды условного перехода для анализа результатов сравнения командой CMP

Типы операндов	Команда условного перехода	Критерий условного перехода	Флаги
Любые	JE	операнд_1 = операнд_2	ZF = 1
Любые	JNE	операнд_1 \neq операнд_2	ZF = 0
Со знаком	JL/JNGE	операнд_1 < операнд_2	SF \neq OF
Со знаком	JLE/JNG	операнд_1 \leq операнд_2	SF \neq OF или ZF = 1
Со знаком	JG/JNLE	операнд_1 > операнд_2	SF = OF и ZF = 0
Со знаком	JGE/JNL	операнд_1 \geq операнд_2	SF = of
Без знака	JB/JNAE	операнд_1 < операнд_2	CF = 1
Без знака	JBE/JNA	операнд_1 \leq операнд_2	CF = 1 или ZF = 1
Без знака	JA/JNBE	операнд_1 > операнд_2	CF = 0 и ZF = 0
Без знака	JAЕ/JNB	операнд_1 \geq операнд_2	CF = 0

Команды циклов используют регистр RCX/ECX/CX как счетчик цикла. Синтаксис базовой команды цикла:

LOOP метка_перехода

Команда LOOP позволяет организовать циклы, подобные циклам `for` в Си с автоматическим уменьшением счетчика цикла на каждом шаге на единицу. Последовательность работы цикла LOOP:

- декремент регистра ECX/CX;
- сравнение регистра ECX/CX с нулем: если $(ECX/CX) = 0$, то управление передается на следующую после `loop` команду.

Кроме LOOP, в программах ассемблера используются также такие команды циклов:

LOOPE/LOOPZ/LOOPNE/LOOPNZ метка_перехода

Работа команд LOOPE и LOOPZ заключается в выполнении следующих действий:

- декремент регистра ECX/CX;
- сравнение регистра ECX/CX с нулем;
- анализ состояния: флага нуля ZF если $(RCX/ECX/CX) = 0$ или $ZF = 0$, управление передается на следующую после LOOPE (LOOPZ) команду.

Работа команд LOOPNE и LOOPNZ заключается в выполнении следующих действий:

- декремент регистра RCX/ECX/CX;
- сравнение регистра RCX/ECX/CX с нулем;
- анализ состояния флага нуля: ZF: если $(RCX/ECX/CX) = 0$ или $ZF = 1$, управление передается на следующую после LOOP команду.

Команды LOOPE/LOOPZ и LOOPNE/LOOPNZ расширяют возможности команды LOOP тем, что дополнительно анализируют флаг ZF, что позволяет организовать досрочный выход из цикла, используя этот флаг в качестве индикатора.

Недостаток команд организации цикла LOOP, LOOPE/LOOPZ и LOOPNE/LOOPNZ состоит в том, что они реализуют только короткие переходы (от -128 до $+127$ байт). Для работы с «длинными» переходами придется использовать команды условного перехода и команду JMP.

Рассмотрим фрагмент программы, в котором организовано три вложенных цикла.

Пример 2.53. Организация нескольких вложенных циклов.

```
.DATA
STRING DB '<> $' ;
.CODE
; Пример цикла из 10 шагов, которые будут
; выполняться с задержкой порядка нескольких
; секунд
MOV CX,10 ; Число шагов во внешнем цикле
; Сначала сохраняем счетчик в стеке
cycle: PUSH CX ;
; Выведем на экран строку из трех символов
MOV AH,09h
MOV DX,offset
INT 21h
; Организуем программную задержку
MOV CX, 100 ; Счетчик вложенного цикла
outer:
; Сначала сохраняем счетчик в стеке
PUSH CX
MOV CX,65535 ; Счетчик внутреннего цикла
inner: LOOP inner ; Повторим команду 65535 раз
POP CX ; Восстановим внешний счетчик
LOOP outer ; Повторим 100 раз
POP CX ; Восстановим счетчик внешнего цикла
LOOP cycle ; Повторим внешний цикл 10 раз
```

Поскольку команда LOOP выполняется всегда CX раз, этот регистр приходится использовать в каждом цикле заново. Поэтому перед входом во внутренний (вложенный) цикл текущее значение счетчика внешнего цикла (содержимое CX) сохраняется в стеке командой PUSH, а перед командой LOOP внешнего цикла восстанавливается командой POP.

2.3 Интерфейс языков программирования высокого уровня, Си и Ассемблера

Современные языки программирования высокого уровня (Object Pascal, C, C++, Java и пр.) обеспечивают полноценную поддержку аппаратных средств компьютера и позволяют создавать компактные и быстродействующие программы. Однако иногда возникает необходимость использования Ассемблера:

- когда нужно минимизировать операции обращения к аппаратуре по размеру кода и времени исполнения;
- если уже имеется готовый и проверенный программный код фрагментов программ или подпрограмм на Ассемблере;
- если необходимо вызывать функции, не поддерживаемые другими языками.

Внедрение кода Ассемблера возможно двумя способами: используя ассемблерные вставки в программы Си или других языков, а также через динамические библиотеки (dll — Dinamic Link Library).

2.3.1 Ассемблерные вставки

Синтаксис вставки ассемблерного кода в программу, написанную на Си (для версии компилятора GCC), следующий:

```
asm [volatile] ( AssemblerTemplate
                : OutputOperands
                [ : InputOperands
                [ : Clobbers ] ] ),
```

где *AssemblerTemplate* — строка-шаблон ассемблерного кода;
OutputOperands — список переменных Си, разделенных запятыми, изменяемых инструкциями в *AssemblerTemplate*;

InputOperands — список выражений Си, разделенных запятыми, прочитанных инструкциями в *AssemblerTemplate*. Допускается пустой список;

Clobbers — список регистров, разделенных запятыми, или других значений, изменяемых в *AssemblerTemplate*, исключая указанные как выходы. Допускается пустой список.

Оптимизаторы `gcc` иногда отбрасывают операторы `asm`, если определяют отсутствие выходных переменных. Кроме того, оптимизаторы могут перемещать код из циклов, если они считают, что код всегда будет возвращать тот же результат (т. е. ни одно из его входных значений не меняется между вызовами). Использование классификатора `volatile` отключает этот вид оптимизации.

При определенных обстоятельствах `gcc` может дублировать (или удалять дубликаты) Ваш код при оптимизации. Это может привести к неожиданным дублирующимся символьным ошибкам во время компиляции, если код раздела `asm` определяет символы или метки. Использование `'% ='` решает эту проблему.

Строка-шаблон `AssemblerTemplate` может содержать любые инструкции, распознаваемые Ассемблером, включая директивы. Компилятор `gcc` не анализирует сами инструкции Ассемблера и не знает, что они означают или даже является ли они действительным кодом Ассемблера. Поэтому будьте внимательны при их записи.

Вы можете разместить несколько инструкций ассемблера в одной строке `asm`, разделенной специальными символами перевода строки, чаще всего это `"\n \t"`.

Поскольку компилятор `gcc` не анализирует шаблон Ассемблера, он не видит каких-либо символов, на которые ссылается шаблон. Это может привести к тому, что `gcc` отменит эти символы как необъявленные, если они также не указаны в качестве операндов ввода, вывода или перехода.

В дополнение к выражениям, описанных операндами ввода, вывода, строки специального формата принимают специальные значения в шаблоне Ассемблера:

- `'%%'` выводит символ `'%'` в код Ассемблера;
- `'%='` выводит число, уникальное для каждого экземпляра оператора `asm` во всей программе. Этот параметр полезен при создании локальных меток и обращении к ним несколько раз в одном шаблоне, который генерирует несколько инструкций ассемблера;
- комбинации `'% { \' , \'% |' и \'% }'` выводят символы `'{'`, `'|'` и `'}'` соответственно, в код ассемблера. Эти символы имеют особое значение для обозначения нескольких ассемблерных диалектов.

Компилятор gcc поддерживает несколько *ассемблерных диалектов*. Так, настройка — `masm` определяет, что GCC использует по умолчанию для встроенного ассемблера диалект ассемблера Microsoft.

Если в Вашем коде необходимо поддерживать несколько ассемблерных диалектов, используйте конструкцию следующего формата:

```
{dialect0 | dialect1 | dialect2 ...}
```

Эта конструкция выводит `dialect0` при использовании диалекта № 0 для компиляции кода, `dialect1` для диалекта №1 и т. д. Если в скобках указано меньше альтернатив, чем количество диалектов, поддерживаемых компилятором, конструкция ничего не выводит.

Например, если компилятор x86 поддерживает два диалекта (“att”, “intel”), шаблон ассемблера может выглядеть так:

```
"bt{1 %[Offset], %[Base] | %[Base], %[Offset]};  
    jc %l2"
```

Это эквивалентно одному из выражений:

```
"btl %[Offset], %[Base] ; jc %l2" /* att dialect */  
"bt %[Base], %[Offset]; jc %l2" /* intel dialect */
```

Оператор `asm` имеет ноль или более *выходных операндов*, обозначающих имена переменных Си, модифицируемых кодом Ассемблера.

Пример 2.54. Ассемблерная вставка в программе на Си

```
#include <stdio.h>  
  
int main(int argc, char *argv[]) {  
    int a = 40;  
    int b;  
    asm("movl %1, %%eax; \  
        shr %%eax; \  
        movl %%eax, %0;"  
        : "=r" (b)  
        : "r" (a)  
        : "%eax");  
    printf ("a=%d b=%d\n" , a, b);  
    return(0);  
}
```

Операнды разделяются запятыми. Каждый операнд имеет следующий формат:

`[[asmSymbolicName]] constraint (cvariablename),`
где `asmSymbolicName` указывает символическое имя для операнда;

`constraint` — это строковая константа, определяющая ограничения на размещение операнда;

`cvariablename` задает выражение Си `lvalue` для вывода, как правило, это имя переменной. Круглые скобки являются обязательной частью синтаксиса.

Ссылка на имя в шаблоне Ассемблера заключается в квадратные скобки (т. е. `"% [Значение]"`). Объем имени — это оператор `asm`, содержащий определение. Любое допустимое имя переменной Си разрешено, включая имена, уже определенные в окружающем коде. Не может быть двух операндов внутри одного и того же оператора `asm`, которые используют одно и то же символическое имя. Если Вы не используете `asmSymbolicName`, используйте нулевое положение операнда в списке операндов в шаблоне ассемблера. Например, если есть три выходных операнда, используйте `"%0"` в шаблоне, чтобы сослаться на первый, `"%1"` для ссылки на второй и `"%2"` для ссылки на третий операнд.

Ограничения вывода должны начинаться с `'='` (переменная, переписывающая существующее значение) или `'+'` (при чтении и записи). При использовании `'='` не допускайте, чтобы местоположение сохранило существующее значение при входе в `asm`, за исключением случаев, когда операнд привязан к входу. После префикса должно быть одно или несколько дополнительных ограничений, описывающих, где находится значение. Общие ограничения включают `'r'` для регистра и `'m'` для памяти. Когда вы указываете несколько возможных местоположений (например, `'= rm'`), компилятор выбирает наиболее эффективный, основанный на текущем контексте.

Входные операнды делают доступными для кода сборки значения переменных из программы Си.

Операнды разделяются запятыми. При описании операндов используйте следующий формат:

`[[asmSymbolicName]] constraint (cexpression),`

где `asmSymbolicName` и `constraint` имеют тот же смысл, что и для выходных операндов (см. выше);

`csexpression` — переменная или выражение `C`, передаваемые в `asm`-оператор в качестве входных данных. Круглые скобки являются обязательной частью синтаксиса.

Когда компилятор выбирает регистры для представления входных операндов, он не использует ни один из регистров, указанных в разделе *Clobbers*.

Если нет выходных операндов, однако есть входные операнды, поместите два последовательных двоеточия, в которые будут поступать выходные операнды:

```
asm__ ("некоторые инструкции"  
      : / * Нет выходов. * /  
      : "R" (смещение / 8) );
```

Предупреждение. Не изменяйте содержимое операндов только для ввода (кроме операндов, привязанных к выходу). Компилятор предполагает, что при выходе из оператора `asm` эти операнды содержат те же значения, что и перед выполнением инструкции. Невозможно использовать *Clobbers*, чтобы сообщить компилятору, что значения в этих входах изменяются. Необходимо связать входную переменную с выходной переменной, которая никогда не используется. Обратите внимание, что если код, который следует за оператором `asm`, не использует ни один из выходных операндов, оптимизаторы GCC могут отменить оператор `asm` как ненужный.

Во вставке `asm` поддерживаются модификаторы в операндах (например, можно использовать `"%k2"` вместо `"%2"`). Ниже показан пример такой замены.

Пример 2.55. Использование именованных параметров.

```
asm ("combine %2,%0"  
     : "= R" (foo)  
     : "0" (foo), "g" (bar));  
; Применение символьных имен  
asm ("cmoveq %1,%2,%[result]"  
     : [Result] "= r" (result)  
     : "R" (тест), "r" (new),
```

```
"[result] " (old));
```

В этом примере ограничение "0" для входного операнда 1 говорит, что оно должно занимать то же место, что и выходной операнд 0. Только входные операнды могут использовать числа в ограничениях, при этом каждый из них должен ссылаться на выходной операнд. Только число (или символьное имя Ассемблера) в ограничении может гарантировать, что один операнд находится в том же месте, что и другой. Того факта, что `foo` — значение обоих операндов, недостаточно, чтобы гарантировать, что они находятся в одном и том же месте в сгенерированном коде ассемблера.

Если для вычисления требуются дополнительные регистры, или процессору нужно перезаписать регистр как побочный эффект конкретной команды ассемблера, нужно сообщить компилятору об этих изменениях, перечислите их в списке `Clobber`, каждый элемент которого является строковой константой, заключенной в двойные кавычки и разделенной запятыми.

Описание `Clobber` не может никоим образом перекрываться с входным или выходным операндом. Например, у Вас может не быть операнда, описывающего класс регистра с одним членом при перечислении этого регистра в списке `Clobber`. Переменные, объявленные для записи в конкретных регистрах, и используемые в качестве входных или выходных операндов `asm`, не должны упоминаться в описании `Clobber`.

Когда компилятор выбирает, какие регистры использовать для представления входных и выходных операндов, он не использует ни одного из изменяемых регистров. В результате, изменяемые регистры доступны для любого использования в коде ассемблера, как показано в следующем примере.

Пример 2.56. Указание изменяемых регистров.

```
asm volatile ("movc3 %0, %1, %2"
: /* No outputs. */
: "g" (from), "g" (to), "g" (count)
: "r0", "r1", "r2", "r3", "r4", "r5");
```

Кроме того, есть два специальных аргумента `Clobber`:

— `"cc"` указывает, что код ассемблера изменяет регистр флагов.

На некоторых машинах `gcc` представляет коды условий как специфиче-

ский аппаратный регистр, а "сс" служит для обозначения этого регистра. На других машинах обработка кода условия отличается, и указание "сс" ни на что не влияет;

— "memory" сообщает компилятору, что код сборки выполняет операции с памятью: считывает или записывает элементы, отличные от тех, которые указаны во входных и выходных операндах (например, для доступа к памяти, на которую указывает один из входных параметров). Чтобы память содержала правильные значения, gcc может потребовать сохранить определенные значения регистров в памяти перед выполнением кода раздела asm. Кроме того, компилятор не предполагает, что любые значения, считанные из памяти перед asm, остаются неизменными после этого asm, он перезагружает их по мере необходимости.

Сохранение регистров в памяти имеет последствия для производительности и может быть проблемой для чувствительного ко времени исполнения кода. Чтобы избежать этого, например, при доступе к десяти байтам строки используйте ввод в память:

```
{ "m" ( ( { struct { char x[10]; } *p = (void *) ptr; *p; } ) ) }.
```

В других языках программирования также предусмотрена возможность использования ассемблерных вставок.

В программах Visual C++ ассемблерная вставка имеет следующий синтаксис:

```
___ asm {  
команда 1  
команда 2  
.  
.  
.  
команда N  
}
```

или

```
___ asm команда 1  
___ asm команда 2  
.  
.  
.  
___ asm команда N
```

В Object Pascal (Delphi) ассемблерная вставка делается так:

```
asm  
команда 1
```

```

команда 2
. . . . .
команда N
end;

```

Инструкции встроенного ассемблера могут ссылаться на переменные С или С++ по имени, и содержать:

- а) символы (например, метки и имена переменных и функций);
- б) константы, включая символьные константы и перечисления;
- в) макросы и директивы препроцессора Си (С++);
- г) комментарии (*/* */* и *//*).

Рекомендация: используйте внутри ассемблерных вставок имена переменных, объявленных в основной программе. Объявление переменных внутри ассемблерной вставки допустимо, но нежелательно.

Код ассемблерных программ, записанных в файлах, удобно оформлять в виде *подпрограмм-процедур*, чтобы их можно было вызывать в программах, написанных на языках высокого уровня.

Синтаксис описания процедуры Ассемблера следующий:

```

имя_процедуры PROC ; начало процедуры
операторы          ; тело процедуры
RET                ; возврат из процедуры
имя_процедуры ENDP ; завершение процедуры

```

Для вызова процедур предназначена команда CALL. При вызове процедуры в стеке сохраняется текущее значение регистра IP, что обеспечивает возврат к месту вызова в основной программе.

Возврат из процедуры выполняется командой RET. Эта команда восстанавливает значение из вершины стека в регистре IP.

В программе на языке высокого уровня, вызывающей процедуры ассемблера, следует описать прототип процедуры.

Перед прототипом следует указать компилятору, что процедура объявлена во внешнем файле и имеет соответствующее правило вызова, например, "C":

```

extern "C"
{
//процедура Ассемблера из внешнего файла

```

```

void AsmProcName();
}

```

Компиляторы языков высокого уровня искажают имена вызываемых функций, поэтому указание правила вызова обязательно.

Ассемблерная процедура должна соответствовать ряду требований:

- использовать правила именования сегментов, принятых в языке высокого уровня;
- явно описывать все глобальные и внешние идентификаторы;
- поддерживать принятую в языке высокого уровня последовательность передачи параметров и возврата значения в точку вызова.

Ассемблерный файл должен быть написан с использованием директив описания модели памяти и сегментов:

```

.386
.MODEL FLAT, C ;модель памяти и правило вызова
.DATA
...
; сегмент данных
.CODE
... ;сегмент кода
END ;завершение модуля

```

Все внешние переменные, описанные на языке высокого уровня, и на которые ссылается ассемблерная процедура, должны описываться в ассемблерном коде с использованием директивы EXTRN:

EXTRN имя_переменной: размер

Параметр размер задает число байтов, выделяемых компилятором языка высокого уровня, и равен (соответствие для типов данных C/C++):

- BYTE для типа char;
- WORD для типов int и unsigned int;
- DWORD для типов long и float;
- QWORD для типа double (8 байт).

Эти же правила применяются для указания аргументов, передаваемых в процедуру ассемблера из внешней программы. Если переменная

описана после ключевого слова PROC, то директиву EXTRN указывать не обязательно.

Вызываемая процедура перед возвращением в точку вызова должна восстановить все изменяемые регистры, — перед использованием их значения нужно сохранять, а при выходе из процедуры восстанавливать, что обычно реализуется при помощи стека.

Ниже представлен пример двух файлов: один на языке C++ с прототипом ассемблерной процедуры и ее вызовом, и второй на Ассемблере с процедурой сложения двух чисел.

Пример 2.57. Подключение процедуры Ассемблера к программе, написанной на C++

```
// Файл main.cpp:
extern "C" {
int sum(); // Прототип функции Ассемблера
}

void main() {
int a, b, c;
a = scanf("%d", a); // Ввод переменной a
b = scanf("%d", b); // Ввод переменной b
c = sum(a,b); // Вызов функции, расчет c
// Вывод на экран
printf("c=%d + %d = %d",a,b,c);
}

; Файл asmproc.asm:
.386
.MODEL FLAT, C
.DATA
.CODE
;процедура с двумя аргументами
SUM PROC a:DWORD, b:DWORD
PUSH EBX ; сохранение содержимого регистра EBX
MOV EAX, a
MOV EBX, b
ADD EAX, EBX
POP EBX ; восстановление значения EBX
```



```
RET ; возврат из подпрограммы
sum ENDP
END
```

2.3.2 Подключение динамических библиотек

При разработке фрагментов программ на Ассемблере, их можно включить в состав библиотек динамической компоновки (dll). Эти библиотеки содержат процедуры ассемблера, а также специальную инициализирующую процедуру.

Инициализирующая процедура принимает три обязательных параметра, и использует регистр процессора EAX для определения текущего состояния: если EAX = -1, то библиотека продолжает работу, если же EAX = 0, то работа завершается.

Шаблон динамической библиотеки имеет следующий вид (синтаксис Turbo Assembler):

```
.386
.MODEL FLAT
PUBLIC test1
.DATA
.CODE
;инициализирующая процедура
DllMain PROC hInstDLL:DWORD,
            rs:DWORD,
            rsvd:DWORD
MOV EAX, -1 ; продолжаем работу
RET
DllMain ENDP
;процедура, экспортируемая из библиотеки
test1 PROC C
;код процедуры
RET
test1 ENDP
END DllMain
```

В приведенном примере используются следующие параметры инициализирующей процедуры:

`hInstDLL` — ссылка на обработчик процесса (описатель);

`rs` — состояние динамической библиотеки;

`rsvd` — резервный параметр.

Параметр `rs` может принимать одно из следующих значений:

- `DLL_PROCESS_ATTACH` — библиотека загружается в адресное пространство процесса впервые. Используется для инициализации DLL;

- `DLL_PROCESS_DETACH` — библиотека выгружается из адресного пространства процесса. Используется для освобождения памяти;

- `DLL_THREAD_ATTACH` — процесс, порожденный библиотекой, создает новую ветвь;

- `DLL_THREAD_DETACH` — ветвь в процессе, порожденном библиотекой, уничтожена.

Имена экспортируемых процедур указывают перед сегментом данных с использованием ключевого слова `PUBLIC`. Необходимо также создать файл установок модуля (с расширением `*.def`), в котором указывается имя и описание самой библиотеки, а также имена и параметры содержащихся в ней экспортируемых процедур.

Рассмотрим пример содержимого `def`-файла, созданного для ассемблерного файла `MyLib.asm` с процедурой `myFun`:

```
LIBRARY MyLib
DESCRIPTION 'MyLib — мои asm-процедуры'
EXPORTS myFun
```

Для каждой процедуры указывают ее порядковый номер, используя как префикс, символ `@`. Этот номер часто используют для вызова процедуры из динамической библиотеки. Если в строке экспорта указать параметр `NONAME`, он запретит компилятору включать имя процедуры в таблицу экспорта динамической библиотеки.

После создания исходного кода библиотеки и файла установок модуля следует скомпилировать объектный файл, при этом в качестве параметра компоновщика следует указать файла установок модуля `*.def` и настройку, указывающую, что необходимо построить динамическую библиотеку. Для этого выполните приведенную ниже последовательность команд.

Для Ассемблера TASM:

```
tasm32 /ml MyLib.asm  
tlink32 -aa -Tpd MyLib.obj,,,MyLib.def
```

Для Ассемблера MASM:

```
ml      MyLib.asm      /link/OUT:"MyLib.dll"      /DLL  
/entry:DllMain /DEF: MyLib.def /SUBSYSTEM:CONSOLE
```

Здесь /link — запуск компоновщика с параметрами:

/OUT: MyLib.dll — имя выходного файла;

/DLL - указывает на то, что должен быть создан DLL-модуль,

/entry:DllMain - точка входа - функция DllMain;

/DEF — имя файла компоновки.

После создания файла библиотеки его следует разместить в папке проекта, созданного на языке высокого уровня.

Далее следует подключить созданную нами библиотеку к программе. Это можно сделать динамически и статически. При динамическом подключении необходимо использовать соответствующие функции системных библиотек Вашей операционной системы:

- в Windows это функции LoadLibrary (загрузить библиотеку), FreeLibrary (отключить библиотеку и выгрузить ее из памяти машины) и GetProcAddress (получить адрес нужной процедуры);

- в Linux это функции dlopen (загрузить библиотеку), dlclose (отключить библиотеку и выгрузить ее из памяти машины) и dlsym (получить адрес нужной процедуры).

При динамическом подключении библиотеки программист должен сам отслеживать количество и тип передаваемых и принимаемых параметров.

Динамическое подключение библиотек обеспечивает гибкость в их использовании. Так, в случае отсутствия нужной библиотеки можно предусмотреть возможность ее поиска или подключения другой библиотеки. Недостатком динамического способа подключения библиотек является избыточность программного кода.

Рассмотрим подключение динамической библиотеки Ассемблера в проектах Си/C++. В состав библиотеки включим процедуру sum, созданную нами ранее.

Пример 2.58. Код динамической библиотеки на Ассемблере.

```
.386
.MODEL FLAT
PUBLIC asmsum
.DATA
.CODE
DllMain proc hInstDLL:DWORD, reason:DWORD, reserved1:DWORD
MOV eax, -1
RET
DllMain ENDP
asmsum PROC C, a:DWORD, b:DWORD
PUSH ebx
MOV eax,a
MOV ebx ,b
ADD eax,ebx
POP ebx
RET
asmsum ENDP
END DllMain
```

Ниже представлен фрагмент программы на языке C++, в которой динамически подключается библиотеку `MyLib.dll` и осуществляется вызов функции `asmsum`.

Пример 2.59. Вызов функции из динамической библиотеки.

```
// Объявляем переменные
int a=1, b=2, c=0;
//определяем указатель на процедуру
typedef int(*ProcFromDll)( int, int );
HINSTANCE hLib =
    LoadLibrary(TEXT( " MyLib.DLL" ));
// Проверяем результат загрузки библиотеки
if (hLib == NULL) {
    // Выводим сообщение о невозможности
    // загрузки библиотеки
    return;
```

```

}
// Получаем указатель на процедуру
// и преобразуем его к нужному типу
functionDll f =
    (ProcFromDll)GetProcAddress(hLib, "asmsum");
// Проверяем полученный указатель
if (!f)
    // Выводим сообщение об отсутствии процедуры
    // в библиотеке
else
{
    c=f(a, b); // Вызываем процедуру
}
FreeLibrary(hLib); // Освобождаем память

```

Для использования функций динамического подключения dll-библиотек и соответствующих типов нужно в проект на С или С++ включить заголовочный файл `windows.h`.

В программе, написанной на языке высокого уровня, следует получить указатель на процедуру из динамической библиотеки. Созданная нами процедура принимает два параметра, поэтому при вызове мы должны это указать:

```
typedef int(*ProcFromDll)( int, int );
```

2.4 Контрольные вопросы к разделу 2

1. Какие символы могут содержать идентификаторы языка Си?
2. Какие символы нельзя использовать в идентификаторах языка Си?
3. Базовые типы данных Си.
4. Расширение базовых типов Си с помощью квалификаторов.
5. Объявление переменных и констант в программах Си.
6. Операции языка Си. Особенности применения операции присваивания Си.
7. Операции инкремента и декремента. Правила и особенности применения.

8. Указатели в Си. Правила объявления и использования.
9. Массивы в Си. Объявление и начальная инициализация.
10. Связь массивов и указателей в Си.
11. Строки в программах Си. Основные операции со строками.
12. Условный оператор if. Синтаксис. Принцип работы. Как указать в условии несколько альтернатив?
13. Оператор варианта switch. Синтаксис. Принцип работы. Как указать в разделе case несколько альтернатив?
14. Оператор безусловного перехода goto. Синтаксис. Правила и особенности применения.
15. Оператор цикла for. Синтаксис. Особенности определения разделов.
16. Операторы цикла while и do while. Синтаксис. Чем эти циклы отличаются друг от друга?
17. Операторы break и continue. Правила и особенности применения.
18. Функции в программах Си. Правила объявления. Передача параметров в функции.
19. Структура программы на языке Си. Глобальные и локальные переменные — особенности объявления и использования.
20. Директивы препроцессора Си. Правила записи.
21. Структура программы на Ассемблере. Какие разделы обязательно должны присутствовать в программе?
22. Синтаксис команд Ассемблера.
23. Какие типы данных Ассемблера Вы знаете? Чем они друг от друга отличаются?
24. Как объявляются переменные в Ассемблере?
25. Команды передачи данных Ассемблера. Из чем состоят особенности их применения?
26. Перечислите, какие способы адресации в Ассемблере Вы знаете? Кратко охарактеризуйте каждый вид адресации.
27. Арифметические операции Ассемблера. Синтаксис. Особенности применения.
28. Логические операции Ассемблера. Синтаксис. Особенности применения.

29. Команда безусловного перехода JMP. Синтаксис. Особенности ближних и дальних переходов.

30. Группа команд условных переходов JCC. Как задать условия перехода?

31. Циклы в программах Ассемблера. Особенности организации вложенных циклов.

32. Перечислите способы интеграции языков программирования высокого уровня с модулями Ассемблера.

33. Ассемблерные вставки в программах Си. Синтаксис. Особенности реализации разделов.

34. Особенности использования регистров микропроцессора в ассемблерных вставках Си.

35. Передача параметров в ассемблерную вставку из программы Си.

36. Динамические библиотеки Ассемблера — назначение, структура.

37. Как скомпилировать и подключить динамическую библиотеку Ассемблера к программе Си?

3 ПРОГРАММИРОВАНИЕ АППАРАТНЫХ СРЕДСТВ

3.1 Ввод с клавиатуры и вывод на экран

Рассмотрим возможности программирования операций ввода с клавиатуры и вывода на экран в Windows и Linux с использованием средств языка Си.

В программах Ассемблера для DOS и Windows чтение кодов символов с клавиатуры осуществляется через функцию 1 прерывания `int 21h` DOS, или функцию 0 прерывания `16h` BIOS. Код нажатой клавиши будет находиться в регистре `AL`. Строку символов можно прочесть с помощью функции `0Ah` прерывания `int 21h`.

В программах Ассемблера для Linux для ввода-вывода используется прерывание `int 80h`.

Отметим, что последние версии Windows не позволяют выполнять вызов и обработку прерываний из программ пользователя. Для этого используются функции библиотек Windows API [7]. Желающие могут ознакомиться с вводом-выводом на Ассемблере в литературе [2, 6, 7].

Клавиатура — стандартное устройство ввода данных, подключенное через контроллер клавиатуры (см. рис. 1.1), которое позволяет передавать в компьютер коды нажатых клавиш (т.н. scan-коды). Так, например, когда Вы нажимаете на клавиатуре клавишу `G`, то передается не сам символ, а его код — `47h`. При считывании программа должна идентифицировать клавиши согласно их кодам (их еще называют scan-кодами).

Задачи, решаемые контроллером клавиатуры:

1. Реагировать на нажатие клавиш или комбинации клавиш.
2. Реагировать на «длительное» нажатие на клавишу, и осуществлять соответствующие действия.
3. По положению клавиш генерировать специальный scan-код клавиши.
4. В соответствии с таблицей символов преобразовывать scan-код клавиши в соответствующий ей символ ASCII.
5. Заносить пару «символ» — «scan-код» в клавиатурный буфер.

Каждая клавиша имеет собственный scan-код. Кроме того, комбинации клавиш имеют отдельные scan-коды. Так, например, если scan-

код клавиши «F10» равен 68, то код той же клавиши с нажатой клавишей «Shift» будет равен 93, в комбинации с клавишей «Ctrl» — 103, а в комбинации с клавишей «Alt» — 113.

Существуют т.н. «алфавитно-цифровые» и «управляющие» клавиши. Алфавитно-цифровые клавиши генерируют scan-код и ASCII символы (цифры и буквы). Управляющие клавиши генерируют только расширенный scan-код, а в поле символа выдают ноль.

Благодаря наличию scan-кодов клавиш программист всегда может понять, как обрабатывать код клавиши: выводить ли символ на экран или выполнять управляющую последовательность действий.

Полученный после преобразований в контроллере клавиатуры двухбайтовый код пары «символ»—«scan-код» посылается в кольцевой буфер ввода, для синхронизации ввода данных с клавиатуры и приёма их выполняемой программой. Объём кольцевого буфера небольшой и обычно составляет 30 байт. Кольцевой буфер организован по принципу: «первым записан — первым считан» (FIFO — First Input—First Output), поэтому при его переполнении новые коды в нем не сохраняются, а нажатие на клавиши вызывает предупреждающие сигналы.

Язык программирования Си предоставляет полный набор функций для ввода с клавиатуры и вывода на экран. Рассмотрим их подробно.

Текст программы на Си, использующей хотя бы одну функцию библиотеки ввода-вывода, должен содержать в начале строку подключения заголовочного файла:

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

Простейший механизм ввода — это чтение одного символа из *стандартного устройства ввода* (обычно это клавиатура) функцией:

```
int getchar(void)
```

После каждого своего вызова функция `getchar()` возвращает следующий символ, введенный с клавиатуры.

В качестве примера рассмотрим программу, которая переводит введенный символ в нижний регистр.

Пример 3.1. Ввод и вывод символа.

```
#include <stdio.h>
#include <ctype.h>
main() {
```

```
int c;
while ((c = getchar()) != EOF)
    putchar(tolower(c));
return 0; }
```

Функция `tolower` определена в `<ctype.h>`. Она переводит буквы верхнего регистра в буквы нижнего регистра, а остальные символы возвращает без изменений. Значение `EOF` — признак окончания ввода (обычно `EOF = -1`).

Однако посимвольный ввод не всегда нужен, и к тому же, не всегда удобен. Часто нужно вводить множество разнородных данных, таких, как строки, символы и числа. В этом случае наиболее удобным и универсальным является *форматированный* ввод, который реализуется с помощью функции `scanf`, синтаксис которой:

```
int scanf(char *format, ...),
```

где `format` — строка формата.

Функция `scanf` читает символы из стандартного входного потока (например, клавиатуры), интерпретирует их согласно спецификациям строки формата, и заносит результаты в свои остальные аргументы, в качестве которых используются переменные программы.

Функция `scanf` прекращает работу тогда, когда исчерпан формат или вводимая величина не соответствует заданной спецификации. Функция `scanf` возвращает количество успешно введенных элементов данных. Каждое очередное обращение к `scanf` продолжает ввод с символа, следующего сразу за последним обработанным.

Формат содержит спецификации, необходимые для управления преобразованиями ввода. В формат могут входить следующие элементы:

- пробелы или знаки табуляции (игнорируются компилятором);
- обычные символы (исключая `%`), которые, как ожидается, совпадут с очередными символами, отличными от символов-разделителей входного потока;
- *спецификации* преобразования, каждая из которых начинается со знака `%` и завершается символом-спецификатором типа преобразования. Между этими двумя символами в любой спецификации могут располагаться, причем в том порядке, как они здесь указаны: знак `*`

(признак подавления присваивания); число, определяющее ширину поля; буква `h` или `l`, указывающая на размер получаемого значения; и символ преобразования (`o`, `d`, `x`). Каждая спецификация преобразования служит для управления преобразованием вводимого поля.

Обычно результат помещается в какую-либо переменную, на которую указывает соответствующий аргумент. Однако если в спецификации преобразования присутствует знак `*`, то соответствующее поле ввода пропускается, и присваивание не выполняется.

Поле ввода определяется как строка без символов-разделителей, оно простирается до следующего символа-разделителя или же ограничено шириной поля, если она задана. Поскольку символ новой строки также относится к символам-разделителям, то `scanf` при чтении будет делать перевод с одной строки на другую.

Символами-разделителями являются символы пробела, табуляции, новой строки, возврата каретки, вертикальной табуляции и перевода страницы.

Символ-спецификатор указывает, каким образом следует интерпретировать очередное поле ввода. Соответствующий аргумент должен быть указателем, как того требует механизм передачи параметров по значению, принятый в Си. Символы-спецификаторы приведены в таблице 3.1.

Таблица 3.1 — Спецификаторы преобразования для функции `scanf`

Символ	Значение
<code>%c</code>	Чтение одиночного символа
<code>%d</code>	Чтение десятичного целого числа
<code>%i</code>	Чтение целого числа в десятичном, восьмеричном или шестнадцатеричном формате
<code>%e</code>	Чтение числа с плавающей точкой
<code>%f</code>	Чтение числа с плавающей точкой
<code>%g</code>	Чтение числа с плавающей точкой
<code>%o</code>	Чтение восьмеричного числа
<code>%s</code>	Чтение строки
<code>%x</code>	Чтение шестнадцатеричного числа

Продолжение таблицы 3.1

Символ	Значение
%p	Чтение указателя
%n	Принимает целое значение, равное количеству считанных символов
%u	Чтение десятичного целого числа без знака
%[]	Чтение набора сканируемых символов
%%	Чтение знака процента

Перед символами-спецификаторами d, l, o, u и x может стоять буква h, указывающая на то, что соответствующий аргумент должен иметь тип short* (а не int*), или l (латинская ell), указывающая на тип long*. Перед символами-спецификаторами e, f и g может стоять буква l, указывающая, что тип аргумента — double * (а не float *).

Рассмотрим программу–«калькулятор», в которой организуем ввод с помощью функции scanf и производится суммирование.

Пример 3.2. Форматированный ввод с клавиатуры.

```
#include <stdio.h>
main() /* программа-калькулятор */
{
    double sum = 0.0, v = 0.0;
    while (scanf ("%lf", &v) == 1)
        printf("\t%.2f\n",
            sum += v);
    return 0; }
```

Теперь допустим, что нам нужно прочесть строки ввода, содержащие данные вида 25 дек 2017, т.е. дату. Обращение к scanf для ввода даты выглядит следующим образом:

```
int day, year; /* день, год */
char monthname[20]; /* название месяца */
scanf ("%d %s %d", &day, monthname, &year);
```

Знак & перед monthname не нужен, так как имя массива уже является указателем.

В строке формата могут также присутствовать символы, не участвующие ни в одной из спецификаций; это значит, что эти символы должны появиться на вводе в таком виде, как они записаны.

Прочитать даты можно и в формате mm/dd/yy:

```
int day, month, year; /* день, месяц, год */
scanf("%d/%d/%d", &day, &month, &year);
```

Обращения к функции `scanf` могут перемежаться с вызовами других функций ввода.

Любая функция ввода, вызванная после `scanf`, продолжит чтение с первого еще непрочитанного символа.

Внимание!!! Аргументы функции `scanf` должны быть указателями! Одна из самых распространенных ошибок при использовании функции `scanf` состоит в том, что вместо того, чтобы написать `scanf("%d", &n)` иногда пишут `scanf("%d", n)`, а компилятор не видит в этом ошибки!

Функция `scanf()` поддерживает спецификатор формата общего назначения, называемый *набором сканируемых символов*, который представляет собой множество символов. Когда `scanf()` обрабатывает это множество, то в программу вводятся только те символы, которые входят в заданный набор сканируемых символов. Такой прием называют *фильтрацией ввода*.

Читаемые символы будут помещаться в массив символов, который указан аргументом, соответствующим набору сканируемых символов: все те символы, которые предстоит сканировать, помещают в квадратные скобки. Непосредственно перед открывающей квадратной скобкой должен находиться знак `%`.

Например, следующий набор сканируемых символов дает указание `scanf()` сканировать только символы X, Y и Z: `%[XYZ]`.

Таким образом, при использовании набора сканируемых символов функция `scanf()` продолжает читать символы, помещая их в соответствующий массив символов, пока не встретится символ, не входящий в этот набор.

Пример 3.3. Форматированный ввод с клавиатуры с фильтрацией.

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

```

int main(void) {
int i; char str[80], str2[80];
scanf("%d%[abcdefg]%s", &i, str, str2);
printf("%d %s %s", i, str, str2);
return 0;
}

```

Введите строку “123abcdtye”, а затем нажмите клавишу <ENTER>. После этого программа выведет 123 abed tye. Так как 't' не входит в набор сканируемых символов, то `scanf()` прекратила чтение символов в переменную `str` сразу после того, как встретился символ 't', а оставшиеся символы были помещены в переменную `str2`.

Используя рассмотренный механизм, можно указать набор сканируемых символов, работающий с точностью до наоборот, — тогда первым символом в таком наборе должен быть '^', который дает указание `scanf()` принимать любой символ, который не входит в набор сканируемых символов.

В большинстве реализаций Си для указания диапазона можно использовать знак дефиса. Например, набор сканируемых символов `%[A-Z]` дает функции **`scanf()`** указание принимать символы от 'A' до 'Z'.

Набор сканируемых символов чувствителен к регистру букв. Если нужно сканировать буквы и на верхнем, и на нижнем регистре, то их следует указать отдельно для каждого регистра.

Теперь рассмотрим функции Си, которые можно использовать для вывода на экран.

Функция `int putchar(int c)` отправляет символ `c` в *стандартный вывод* (по умолчанию это экран монитора).

В качестве примера рассмотрим программу, которая осуществляет ввод символов и переводит введенные символы в нижний регистр:

Пример 3.4. Вывод символа на экран.

```

#include <stdio.h>
#include <ctype.h>
int main() {
int c;
while ((c = getchar()) != EOF)

```

```

/* tolower: переводит ввод на нижний регистр */
/* tolower определена в <ctype.h> */
putchar(tolower(c));
return 0; }

```

В Си *форматированный вывод* осуществляет функция `printf`, которая преобразует, форматирует и печатает переданные аргументы в стандартном выводе под управлением формата. Возвращает функция `printf` количество напечатанных символов. Синтаксис функции `printf` следующий:

```
int printf(char *format, arg1 arg2, ...)
```

Строка формата `format` содержит два вида объектов: обычные символы, которые напрямую передаются в выходной поток, и спецификации преобразования, каждая из которых вызывает преобразование и печать очередного аргумента `printf`.

Любая спецификация преобразования начинается символом ‘%’ и заканчивается символом-спецификатором.

Между ‘%’ и символом-спецификатором могут быть расположены (в указанном ниже порядке) следующие элементы:

- знак минуса, обеспечивающий выравнивание аргумента по левому краю поля;
- число, задающее минимальную ширину поля. Лишние позиции слева (или справа при левостороннем расположении) будут заполнены пробелами;
- точка, отделяющая ширину поля от величины, устанавливающей точность;
- число (точность), указывающее максимальное количество печатаемых символов в строке, или количество цифр после десятичной точки для чисел с плавающей запятой, или минимальное количество цифр для целого;
- буква `h`, если печатаемое целое должно рассматриваться как `short`, или `l` (латинская буква `ell`), если целое должно рассматриваться как `long`.

Символы-спецификаторы перечислены в таблице 3.2.

Таблица 3.2 — Спецификаторы преобразования для функции printf

Символ	Значение
%c	Символ
%d	Десятичное целое со знаком
%i	Десятичное целое со знаком
%e	Экспоненциальное представление ('e' в нижнем регистре)
%E	Экспоненциальное представление ('E' в верхнем регистре)
%f	Десятичное с плавающей точкой
%g	В зависимости от того, какой вывод будет короче, используется %e или %f
%G	В зависимости от того, какой вывод будет короче, используется %E или %F
%o	Восьмеричное без знака
%s	Строка символов
%u	Десятичное целое без знака
%x	Шестнадцатеричное без знака (буквы в нижнем регистре)
%X	Шестнадцатеричное без знака (буквы в верхнем регистре)
%p	Вывод указателя
%n	Аргумент, соответствующий этому спецификатору, должен быть указателем на целочисленную переменную, что позволяет сохранить в ней количество символов, записанных до той позиции, в которой находится код %n
%%	Выводит знак %

Ширину и точность можно задать с помощью символа *, значение ширины (или точности) в этом случае берется из следующего аргумента (который должен быть типа int). Например, чтобы напечатать не более max символов из строки s, используйте оператор:

```
printf("%. *s", max, s);
```

Пусть имеется строка "hello, world", содержащая 12 символов. Ниже показано использование различных спецификаций и показывается их влияние на печать этой строки. Поле специально обрамлено двоеточиями, чтобы была видна его длина.

```
:%s: :hello, world:    :%10s: :hello, world:
: %.10s: :hello, wor:  :%-10s: :hello, world:
```



```
:%.15s: :hello, world: :%-15s: :hello, world :  
:%15.10s: : hello, wor: :%-15.10s: :hello, wor :
```

Числа в десятичном формате со знаком отображаются с помощью спецификатора преобразования `%d` или `%i`. Для вывода целого значения без знака используйте `%u`.

Спецификатор преобразования `%f` используется для вывода чисел в формате с плавающей точкой. Соответствующий аргумент должен иметь тип `double`.

Применение спецификатора `%g` показано в следующем примере.

Пример 3.5. Форматированный вывод вещественных чисел.

```
#include <stdio.h>  
int main(void) {  
    double f;  
    for(f=1.0; f<1.0e+8; f=f*10)  
        printf("%g ", f);  
    return 0;  
}  
  
// В результате выполнения программы получим:  
// 1 10 100 1000 10000 100000 1e+06 1e+07
```

У функции `printf()` имеется возможность использовать два дополнительных модификатора: `'*'` и `'#'`.

Размещение `'#'` перед спецификаторами `g`, `G`, `f`, `E` или `e` означает, что при выводе обязательно будет показана десятичная запятая — даже если десятичных цифр нет. Если разместить `'#'` непосредственно перед спецификатором `x` или `X`, то шестнадцатеричное число будет выведено с префиксом `0x`. Если `'#'` непосредственно записан перед спецификатором `o`, число будет выведено с ведущим нулем. К любым другим спецификаторам преобразования модификатор `'#'` применять нельзя.

Модификаторы минимальной ширины поля и точности можно передавать функции `printf()` не как константы, а как аргументы, используя в качестве заполнителя символ `'*'`. При сканировании строки формата функция `printf()` каждой звездочке из этой строки будет ставить в соответствие очередной аргумент, причем в том же порядке, в котором расположены аргументы.

Пример 3.6. Расширение вывода.

```
// Использование модификатора *
#include <stdio.h>
int main(void) {
printf("%*.*f", 10, 4, 1234.56);
return 0;
}
// Результат работы программы
// 1234.5600
```

Несмотря на то, что значительная часть спецификаторов для функций `scanf` и `printf` совпадают, не забывайте свериться с таблицами 3.1 и 3.2 перед выбором формата ввод–вывода.

3.2 Дисковые накопители

В настоящее время такие накопители, как гибкие диски (дискеты), практически не используются. Поэтому вся информация, приведенная в данном подразделе, касается т.н. «жестких» дисков.

Конструктивно жесткий диск представляет собой набор круглых алюминиевых пластин, покрытых магнитным материалом, которые вращаются на шпинделе, приводимом во вращение электродвигателем.

Минимальной единицей хранения информации на диске является *сектор* (sector), имеющий размер, равный 2^n , где n — целое число. Различают *физические* и *логические* секторы. Записать фрагмент информации размером меньше, чем сектор, нельзя. Если, например, размер сектора равен 512 байт, а размер файла, который нужно сохранить на диске, составляет 514 байт, то на диске этот файл будет занимать 2 сектора, т.е. $512+512=1024$ байт.

Секторы объединены в *дорожки*, группа дорожек называется *цилиндром*. Физически цилиндр включает все дорожки с одинаковыми номерами на всех пластинах жесткого диска. При этом группы цилиндров могут относиться к различным операционным системам. Кроме того, допускается разделять фиксированный диск на несколько *разделов* разного размера. Информация с пластин на диске считывается *головками*. Адрес физического сектора задается в формате:

Сектор — Цилиндр.

Логические секторы имеют номера от 0 до N. В программе при непосредственном доступе к диску можно использовать как физическую, так и логическую адресацию. Информация о секторах хранится на самом диске и записывается при его форматировании, и включает идентификационный номер каждого сектора. Базовая система ввода-вывода (BIOS) нумерует секторы диска следующим образом: 1-8, 1-9 или 1-15, в зависимости от емкости диска. Дорожки не нумеруются, т.к. их индекс определяется по смещению головки от внешнего края диска.

Дисковые функции BIOS обращаются к определенному сектору, указывая его номер и номер дорожки. Функции DOS рассматривают все секторы диска, как единый массив, индексы в котором расположены подряд, начиная от 0, и таким образом каждый сектор имеет свой логический номер.

Фиксированные диски имеют главную запись загрузки, которая содержит *таблицу разделов*, позволяющую разделить диск между несколькими операционными системами.

Таблица разделов содержит информацию о том, где на диске начинается раздел соответствующей операционной системы, при этом первый сектор каждого раздела содержит запись начальной загрузки.

Таблица размещения файлов всегда начинается с первого логического сектора (второй сектор дискеты или раздела фиксированного диска), который расположен сразу после блока начальной загрузки. Если таблица занимает более одного сектора, то она продолжается в следующих секторах.

3.2.1 Работа с дисками

Дисковая служба подразделяет физические диски на гибкие диски (дискеты) и жесткие диски. Кроме дисковой службы с дисковыми устройствами связаны также векторы, обслуживающие аппаратные прерывания от контроллера гибких дисков — `int 0Eh` (линия IRQ6) и от контроллера жестких дисков — `int 76h` (линия IRQ 14). При наличии двухканального порта ATA (Advanced Technology Attachment — параллельный интерфейс подключения жёстких дисков и оптических дисководов к компьютеру) второй канал обычно задействует линию IRQ 15 (вектор `int 77h`).

Аппаратные прерывания вырабатываются контроллерами по завершению (нормальному или аварийному) внутренних операций. На эти прерывания BIOS не реагирует, а при инициализации их векторы направляются на программную заглушку (в Ассемблере — инструкцию IRET).

Рассмотрим функции библиотек языка Си для работы с дисками.

Получить номер текущего диска позволяет функция `int _getdrive(void)`, прототип которой находится в `direct.h`.

Все диски в системе нумеруются по порядку: диск А имеет номер 1, диск В — номер 2 и т.д.

Пример 3.7. Чтение номера диска.

```
#include <stdio.h>
#include <direct.h>
#include <locale.h>
int main(void) {
    setlocale(LC_ALL, "RUS");
    printf("Текущий диск: %d.", _getdrive ());
    return 0;
}
```

Функция `int _chdrive(int drivenum)`, прототип которой описан в `direct.h`, изменяет текущий диск на новый, указанный аргументом `drivenum`, где диск А соответствует номеру 1, диск В — номеру 2 и так далее. Функция `_chdrive()` возвращает 0 в случае успешного завершения и -1 в противном случае.

Пример 3.8. Изменение текущего диска.

```
#include <stdio.h>
#include <direct.h>
#include <locale.h>
int main(void) {
    int res;
    setlocale(LC_ALL, "RUS");
    printf("Задаем новый диск ");
    res = _chdrive (3); /* переход на диск С */
    if (res == 0)
        printf("Текущий диск: %d.", _getdrive ());
}
```

```

else printf("Ошибка!!!");
return 0;
}

```

Получить сведения о диске в Windows можно с использованием функции `DeviceIoControl`, прототип которой находится в файле `winioctl.h`.

Синтаксис вызова функции `DeviceIoControl` следующий [7]:

```

BOOL DeviceIoControl(
    HANDLE hDevice,
    DWORD dwIoControlCode,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,
    DWORD nOutBufferSize,
    LPDWORD lpBytesReturned,
    LPOVERLAPPED lpOverlapped
);

```

Формальные параметры функции `DeviceIoControl`:

`hDevice` (входной) — описатель (descriptor) устройства, для которого выполняется операция. Устройством может быть том, каталог, файл или поток. Для получения описателя устройства, используйте функцию `CreateFile`;

`dwIoControlCode` (входной) — управляющий код для операции, который идентифицирует конкретную операцию для выполнения и тип устройства, для которого она будет осуществляться. Документация для каждого управляющего кода рассматривает детали использования параметров `lpInBuffer`, `nInBufferSize`, `lpOutBuffer` и `nOutBufferSize`;

`lpInBuffer` (входной) — указатель на буфер ввода данных, содержащий данные, необходимые для выполнения операции. Формат этих данных зависит от значения параметра `dwIoControlCode`, который может быть пустым (равен `NULL`), если `dwIoControlCode` определяет операцию, не требующей ввода данных;

`nInBufferSize` (входной) — задает размер буфера ввода данных в байтах;

`lpOutBuffer` (выходной) — указатель на буфер вывода данных, который должен получить данные, возвращенные операцией. Формат этих данных зависит от значения параметра `dwIoControlCode`. Этот параметр может быть равен `NULL`, если `dwIoControlCode` определяет операцию, которая не возвращает данные;

`nOutBufferSize` (входной) — размер буфера вывода данных в байтах;

`lpBytesReturned` (выходной) — указатель на переменную, которая получает размер данных, сохраненных в буфере вывода данных, в байтах. Если буфер вывода данных является слишком маленьким, чтобы получить какие-либо данные, вызов завершается ошибкой, `GetLastError` возвращает значение `ERROR_INSUFFICIENT_BUFFER`, а параметр `lpBytesReturned` равен нулю. Если буфер вывода данных имеет недостаточный объем, чтобы вместить все данные, но может вместить некоторые вводимые данные, некоторые драйверы возвратят столько данных, сколько их вместились. В этой ситуации вызов завершается ошибкой, функция `GetLastError` возвращает значение `ERROR_MORE_DATA`, а параметр `lpBytesReturned` указывает объем полученных данных. Ваше приложение должно вызвать функцию `DeviceIoControl` снова с той же самой операцией, определяя новую точку отсчета;

`lpOverlapped` (входной) — указатель на структуру `OVERLAPPED`. Если параметр `lpOverlapped` равен `NULL`, `lpBytesReturned` не может быть равным `NULL`. Даже в случаях, когда операция не возвращает никаких выходных данных, и `lpOutBuffer=NULL`, функция `DeviceIoControl` использует `lpBytesReturned`. После такой операции, значение `lpBytesReturned` не имеет смысла. Если параметр `lpOverlapped` не равен `NULL`, `lpBytesReturned` может быть равен `NULL`. Если этот параметр не равен `NULL`, и операция возвращает данные, параметр `lpBytesReturned` не будет иметь смысла до тех пор, пока асинхронная операция не завершилась. Чтобы извлечь информацию о числе возвраща-

емых данных, вызовите функцию `GetOverlappedResult`. Если `hDevice` связан с портом завершения ввода-вывода данных (I/O), Вы можете извлечь число возвращаемых данных при помощи вызова функции `GetQueuedCompletionStatus`. Если параметр `hDevice` получен без определения `FILE_FLAG_OVERLAPPED`, параметр `lpOverlapped` игнорируется. Если параметр `hDevice` открывался с флагом `FILE_FLAG_OVERLAPPED`, операция выполняется как перекрывающаяся (асинхронная) операция. В этой ситуации, параметр `lpOverlapped` должен указать на допустимую структуру `OVERLAPPED`, которая содержит описатель объекта события. В противном случае, функция завершается ошибкой непредсказуемыми способами.

В асинхронных операциях функция `DeviceIoControl` возвращает значение немедленно, при этом объект события подает сигнал, когда операция завершается. В противном случае, функция не возвращает значение до тех пор, пока операция не завершится или не произойдет ошибка.

Если операция завершена успешно, функция `DeviceIoControl` возвращает ненулевое значение. Если операция завершается ошибкой, `DeviceIoControl` возвращает нуль. Чтобы получить дополнительную информацию об ошибке, вызовите `GetLastError`.

Рассмотрим использование функции `DeviceIoControl` для получения основных параметров жесткого диска.

Пример 3.9. Чтение параметров диска.

```
#include <windows.h>
#include <winioctl.h>
#include <stdio.h>
#include <locale.h>

BOOL GetDriveGeometry(DISK_GEOMETRY *pdg)
{
    HANDLE hDevice; // описатель проверяемого
                    // устройства
    BOOL bResult;   // флаг результата
    DWORD junk;     // сбрасываем результаты
```

```

hDevice = CreateFile("\\\\.\\PhysicalDrive0",
// открываемое устройство
0,          // нет доступа к устройству
FILE_SHARE_READ | // режим совместного доступа
FILE_SHARE_WRITE,
NULL,       // атрибуты безопасности по умолчанию
OPEN_EXISTING, // расположение
0,          // атрибуты файла
NULL);      // не копировать атрибуты файла

setlocale(LC_ALL, "RUS");
if (hDevice == INVALID_HANDLE_VALUE)
// невозможно открыть устройство
return (FALSE);
bResult = DeviceIoControl(hDevice,
// запрошенное устройство
IOCTL_DISK_GET_DRIVE_GEOMETRY, // операция
NULL, 0, // буфера ввода нет
pdg, sizeof(*pdg), // буфер вывода
&junk, // возвращено байтов
// синхронизация ввода/вывода (I/O)
(LPOVERLAPPED) NULL);
CloseHandle(hDevice);
return (bResult);
}

int main(int argc, char *argv[])
{
    DISK_GEOMETRY pdg; // сведения о диске
    BOOL bResult;      // флаг общих результатов
    ULONGLONG DiskSize; // размер диска, в байтах
    bResult = GetDriveGeometry (&pdg);

    if (bResult)
    {

```



```

printf("Параметры диска \n");
printf("Цилиндров = %I64d\n", pdg.Cylinders);
printf("Дорожек в цилиндре = %ld\n",
        (ULONG) pdg.TracksPerCylinder);
printf("Секторов в дорожке = %ld\n",
        (ULONG) pdg.SectorsPerTrack);
printf("Байтов секторе = %ld\n",
        (ULONG) pdg.BytesPerSector);
// Рассчитываем объем диска
DiskSize = pdg.Cylinders.QuadPart *
        (ULONG)pdg.TracksPerCylinder *
        (ULONG)pdg.SectorsPerTrack *
        (ULONG)pdg.BytesPerSector;
printf("Объем диска = %I64d (Байт) = %I64d (Gb)\n",
        DiskSize, DiskSize/(1024*1024*1024));
}
else
printf ("GetDriveGeometry — ошибка: %ld.\n",
        GetLastError ());
return ((int)bResult);
}

```

Выполните этот пример, и получите сведения о дисках Вашего компьютера. Проверьте, верны ли данные, которые выдала программа.

3.2.2 Работа с каталогами

Каталог — это средство организации файлов на логическом уровне.

Каталоги могут иметь различные размеры, в зависимости от объема диска и способа его разбиения на разделы. В зависимости от разбиения на разделы фиксированный диск может иметь различные размеры каталога и номер начального сектора.

Каждый диск имеет один *корневой каталог*, внутри которого «вложены» все остальные каталоги (*подкаталоги*). Корневой каталог может содержать элементы, указывающие на подкаталоги, которые в свою очередь могут содержать ссылки на другие подкаталоги, образуя

древовидную структуру каталогов. Корневой каталог всегда расположен в определенных секторах диска, а подкаталоги хранятся как обычные дисковые файлы, и могут быть расположены в любом месте диска.

Все изменения в корневом каталоге производятся в ходе файловых операций или с помощью специальных функций операционной системы. Однако можно работать с каталогом и напрямую.

Программа может создавать или удалять каталоги и подкаталоги, при выполнении определенных условий. Для создания подкаталога необходимо, чтобы было, по крайней мере, свободное место в корневом каталоге. Для удаления подкаталога необходимо, чтобы он не содержал файлов или ссылок на другие подкаталоги. Кроме того, Вы не можете удалить подкаталог, который является Вашим текущим каталогом. Особо отметим, что корневой каталог удалить невозможно.

Создание каталога (подкаталога) осуществляется функцией `mkdir`, прототип которой находится в заголовочном файле `dir.h`:

```
int mkdir(const char *path);
```

где `path` — полный путь к каталогу.

При успешном завершении `mkdir` возвращает 0. При возникновении ошибки функция `mkdir` возвращает -1 и передает в глобальную переменную `errno` код ошибки: `EACCESS` (доступ запрещен), или `ENOENT` (файл или каталог отсутствует).

Функция `rmdir` удаляет каталог, путь к которому определен параметром `path`. Ее синтаксис:

```
int rmdir(const char *path);
```

где `path` — полный путь к каталогу.

В случае успеха функция `rmdir()` возвращает значение 0. В противном случае возвращается -1, а переменная `errno` устанавливается равной `EACCESS` (доступ запрещен) или `ENOENT` (несуществующий путь).

Рассмотрим пример использования функций `mkdir` и `rmdir`.

Пример 3.10. Работа с каталогами.

```
#include <dir.h>
#include <stdio.h>
#include <locale.h>
```

```

int main(void)    {
int res; // Результат операции
setlocale(LC_ALL, "RUS");
// Создаем каталог
res = mkdir("MyDir");
if (!res) printf("Каталог создан \n");
    else printf("Нельзя создать каталог!");
// Удаляем каталог
res = rmdir("MyDir");
if (!res) printf("Каталог удален \n");
    else printf("Нельзя удалить каталог!");
return 0;
}

```

Прочитать текущий каталог можно с использованием функции `getcwd(char *dir, long len)`, которая копирует полный путь текущего рабочего каталога диска в строку, на которую указывает параметр `dir`. Если полный путь длиннее, чем `len` символов, то возникает ошибка. При ошибке функция `getcwd()` возвращает значение `NULL`, а переменная `errno` устанавливается равной `ENODEV` (устройство не существует), `ENOMEM` (недостаточно памяти) или `ERANGE` (ошибка области).

Функция `chdir(const char* path)` устанавливает в качестве текущего каталог, на который указывает параметр `path`. Путь может включать в себя и спецификацию диска. Каталог должен существовать. В случае успеха функция `chdir()` возвращает 0. При неудаче возвращается значение -1, а переменная `errno` устанавливается в `ENOENT` (недействительный путь).

Пример 3.11. Изменение каталога.

```

#include <stdio.h>
#include <dir.h>
#include <locale.h>
#define MAXDIR 1000
int main(void)    {
setlocale(LC_ALL, "RUS");
char dir[MAXDIR];

```

```

getcwd(dir, MAXDIR);
printf("Текущий каталог: %s \n", dir);
chdir ("C:\\Temp");
getcwd(dir, MAXDIR);
printf("Новый каталог: %s", dir); return 0;
}

```

Функция `int closedir(DIR *ptr)` закрывает текущий каталог, в случае успеха возвращает 0 и -1 в противном случае. При неудаче переменная `errno` устанавливается равной `EBADF` (недействительный каталог).

Функция `DIR *opendir(char *dirname)` открывает поток каталога и возвращает указатель на структуру типа `DIR`, которая содержит информацию о каталоге.

Функция `struct dirent *readdir(DIR *ptr)` открывает поток каталога и возвращает указатель на структуру типа `DIR`, которая содержит информацию о каталоге, а именно название следующего файла в каталоге. Т.о., функция `readdir` читает оглавление каталога по одному файлу за один раз.

Функция `void rewinddir(DIR *ptr)` осуществляет возвращение в начало каталога, на который указывает `ptr` и который был предварительно получен с помощью `opendir`.

Прототипы функций `closedir`, `opendir`, `readdir`, описаны в файле `dirent.h`.

Функция `int getcurdir(int drive, char *dir)` копирует имя текущего каталога, заданного параметром `drive`, в строку, на которую указывает `dir`. Значение 0 для параметра `drive` соответствует диску по умолчанию. Строка, на которую указывает `dir`, должна иметь длину как минимум `MAXDIR` байт. Прототип функции `getcurdir` определен в файле `dir.h`. Особенность функции: имя каталога не содержит имя диска и начальных обратных наклонных черточек. Функция `getcurdir` в случае успеха возвращает 0, а при ошибке, значение -1.

Пример 3.12. Получение сведений о текущем каталоге.

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

```

#include <dir.h>
#include <locale.h>
#define MAXDIR 1000
int main(void) {
char dir[MAXDIR];
setlocale(LC_ALL, "RUS");
getcurdir(0, dir);
printf("Current directory is %s", dir);
return 0;
}

```

Функция `char *_fullpath(char *fpath, const char *rpath, int len)` конструирует полный путь по заданному относительному пути. Относительный путь задается переменной `rpath`. Полный путь помещается в массив, на который указывает параметр `fpath`. Размер массива, на который указывает `fpath`, задается в переменной `len`. Если `fpath` равен `NULL`, то массив будет размещен динамически (В этом случае массив должен быть освобожден с использованием функции `free()`). Функция `_fullpath()` возвращает указатель на `fpath` или `NULL` в случае ошибки. Прототип функции описан в `stdlib.h`.

Пример 3.13. Получение полного пути.

```

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
int main(void)
{
char fpath[80];
setlocale(LC_ALL, "RUS");
// меняем каталог на INCLUDE
_fullpath (fpath, "\\INCLUDE" , 80);
printf("Полный путь: %s\n", fpath);
return 0;
}

```

Функция `void _makepath(char *pname, const char *drive, const char *directory, const char *fname,`

`const char *extension`) конструирует полный путь из элементов, заданных параметрами функции, и помещает результат в массив, на который указывает параметр `pname`. Имя диска задается строкой, на которую указывает `drive`. Каталог (и любой подкаталог) задается с помощью строки, на которую указывает `directory`. На имя файла указывает параметр `fname`, а на расширение — `extension`. Любая из этих строк может быть пустой.

Пример 3.14. Получение составляющих полного пути.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
int main(void)
{
    char fpath[80];
    char fname[9];
    char dir[64];
    char drive[3];
    char ext[5];
    setlocale(LC_ALL, "RUS");
    // конструируем полный путь
    _makepath(fpath, "B", "MYDIR",
              "MYFILE", "TXT");
    printf("%s\n", fpath);
    // разделяем путь на составляющие
    _splitpath(fpath, drive, dir, fname, ext);
    printf ("%s %s %s %s\n",
           drive, dir, fname, ext);
    return 0;
}
```

Использованная в последнем примере функция `splitpath()` разделяет полный путь, заданный в строке `fpath`, на составляющие: символ диска помещается в строку, на которую указывает параметр `drive`, имя каталога и входящих в него подкаталогов помещаются в строку `directory`, имя файла — в строку `fname`, а расширение — в

строку `extension`. Минимальные размеры массивов, на которые указывают эти параметры, представлены в табл. 3.3.

Таблица 3.3 — Параметры функции `splitpath`

Параметр	Размер		Имя константы
	DOS	Windows	
<code>drive</code>	3	3	<code>_MAX_DRIVE</code>
<code>directory</code>	66	256	<code>_MAX_DIR</code>
<code>fname</code>	9	256	<code>_MAX_FILE</code>
<code>extension</code>	5	256	<code>_MAX_EXT</code>

3.2.3 Работа с файлами

На уровне Ассемблера операционные системы MS DOS и Windows поддерживают файловые операции с помощью функций прерывания `21h`. В Ассемблере Linux операции с файлами реализуются через системное прерывание `80h`. Программная реализация в Ассемблере достаточно подробно рассмотрена в литературе [2, 6]. Мы же уделим особое внимание программной реализации операций с файлами в Си.

Типовые операции с файлами следующие:

- создание нового файла;
- открытие существующего файла;
- запись в файл;
- чтение из файла;
- закрытие файла.

Для работы с файлами подключите в начале программы заголовочный файл `<stdio.h>`.

В Си обращение к файлам производится через указатели.

Указатель файла — это указатель на структуру типа `FILE`, которая содержит сведения о файле (имя файла, положение текущей позиции в файле и т.п.). Указатель файла определяет конкретный файл и передается в поток для выполнения функций ввода/вывода.

Объявление переменной–указателя файла производится так:

```
FILE *fp;
```

Рассмотрим, как производятся перечисленные выше операции с файлами в программах Си.

Для *создания и открытия файла* используется функция `fopen`, которая открывает (или при необходимости, создает) файл.

Прототип функции `fopen()`:

```
FILE *fopen(const char *имя_файла,  
            const char *режим);
```

где *имя_файла* — указатель на строку символов, представляющую собой допустимое имя файла, в которое также может входить спецификация пути к этому файлу

режим — определяет, каким образом (для выполнения какой операции) файл будет открыт (см. табл. 3.4).

Таблица 3.4 — Режимы для функции `fopen`

Режим	Пояснение
r	Открыть текстовый файл для чтения
w	Открыть текстовый файл для записи
a	Дополнить текстовый файл
rb	Открыть двоичный файл для чтения
wb	Создать двоичный файл для записи
ab	Дополнить двоичный файл
r+	Открыть текстовый файл для чтения/записи
w+	Создать текстовый файл для чтения/записи
a+	Дополнить текстовый файл или создать текстовый файл для чтения/записи

Пример 3.15. Запись в файл.

```
#include <stdio.h>  
#include <locale.h>  
int main (void)    {  
    setlocale(LC_ALL, "RUS");
```



```

// Переменная — указатель на файловую структуру
FILE *f;
// Открытие файла на запись
f=fopen ("myfile.txt", "w");
// Проверка на ошибки
if (f == NULL) printf ("Возникла ошибка\n");
else printf (" \n");
//Запись данных в файл
fprintf (f, "Произвольный текст");
printf ("Запись в файл выполнена\n");
// Закрытие файла
fclose (f);
return 0;  }

```

Функция `fclose()` закрывает поток, через который производились операции чтения/записи в файл, открытый ранее с помощью функции `fopen()`.

Прототип функции `fclose()` :

```
int fclose(FILE *fptr);
```

где `fptr` — указатель файла, возвращенный после вызова `fopen()`.

Если функция `fclose` вернула ноль, это означает успешную операцию закрытия. В случае ошибки возвращается константа `EOF`. Узнать причину ошибки можно, используя функцию `ferror()`.

Ошибка при выполнении `fclose()` происходит тогда, когда диск был преждевременно удален или на диске не осталось свободного места.

Пример 3.16. Получение кода ошибки.

```
int err = fclose(fp);
```

Обратите внимание, что в предыдущем примере мы вызвали `fclose` без получения кода возврата. Это тоже возможно, но при этом мы теряем сведения о возможной ошибке и не имеем возможности обработать ее.

Запись символов производится функцией `putc()`, которая записывает символы в файл, предварительно открытый с помощью функции `fopen()` в режиме записи. Прототип этой функции:

```
int putc(int ch, FILE * fptr );
```

где `fptr` — это указатель файла, возвращенный функцией `fopen()`,

`ch` — выводимый символ.

Если функция `putc()` завершилась успешно, то возвращается записанный символ, в противном случае возвращается `EOF`.

Функция `getc()` читает символ из файла, который с помощью `fopen()` уже открыт в режиме для чтения. Прототип этой функции:

```
int getc(FILE * fptr);
```

где `fptr` — это указатель файла, имеющий тип `FILE` и возвращенный функцией `fopen()`.

При достижении конца файла функция `getc()` возвращает `EOF`. Поэтому, чтобы прочитать символы до конца текстового файла, можно использовать следующий код;

```
do {  
    ch = getc(fp);  
} while(ch!=EOF);
```

Функция `getc()` возвращает `EOF` и в случае ошибки, которую можно определить с помощью `ferror()`.

Чтение и запись строк производится с помощью функций `fgets()` и `fputs()` соответственно.

Прототипы функций `fgets()` и `fputs()`:

```
int fputs(const char * str, FILE * fptr);  
char *fgets(char *str, int length, FILE * fptr);
```

Функция `fputs()` пишет в заданный поток строку, на которую указывает переменная `str`. В случае ошибки эта функция возвращает `EOF`.

Функция `fgets()` читает из заданного потока строку, и делает это до тех пор, пока не будет прочитан символ новой строки или количество прочитанных символов не станет равным `length-1`. Если был прочитан

разделитель строк, он также записывается в строку, чем функция `fgets()` отличается от функции `gets()`. Полученная в результате строка будет оканчиваться символом конца строки `'\0'`. При успешном завершении работы функция возвращает строку `str`, а в случае ошибки — пустой указатель (`null`).

При рассмотрении файловых функций ввода-вывода мы особо отмечали, что все функции возвращают специальный код в случае возникновения *ошибки*. Идентифицировать ошибку позволяет функция `ferror`. Прототип этой функции следующий:

```
int ferror(FILE * fptr);
```

где `fptr` — указатель файла.

Функция `ferror` возвращает `true`, если при последней операции с файлом произошла ошибка; в противном же случае она возвращает `false`. Так как для каждой конкретной операции с файлом задается собственное условие ошибки, то после каждой такой операции следует сразу вызывать `ferror`, иначе сведения об ошибке будут потеряны.

Пример 3.17. Обработка ошибки ввода.

```
ch = getc(in); if(ferror(in) действие );
```

3.3 Порты ввода-вывода

3.3.1 Общие сведения о портах ввода-вывода

Порты (интерфейсы) используются для подключения внешних устройств к компьютеру. Различают параллельный, последовательный порты, инфракрасный порт и т.д. Перечень разновидностей портов, используемых в компьютерах, постоянно растет по мере появления новых технологий и протоколов передачи данных.

Несмотря на то, что интерфейсы для компьютеров непрерывно развиваются, есть два вида портов — последовательный (COM) и параллельный (LPT), которые используются до сих пор и не теряют своей популярности. Это обусловлено отработанной годами технологией передачи данных, наличием подробной документации, возможность работы с разнообразным оборудованием (включая специфические промышленные установки), а также наличием большого количества оборудования, выпущенного ранее.

Кроме COM и LPT, в последние годы появились такие интерфейсы, как IrDA (Infrared Data Association — интерфейс связи, основанный на открытом оптическом канале в инфракрасном диапазоне), IEEE 1394 (цифровой интерфейс, использующий последовательную шину), Bluetooth (интерфейс, осуществляющий обмен данными посредством беспроводных сетей), Wi-Fi (Wireless Fidelity — высокоскоростной беспроводный интерфейс обмена данными) и другие [3, 7].

Такие интерфейсы передачи данных, как LPT, IrDA, Bluetooth, в настоящее время практически не используются. Последовательный интерфейс COM до сих пор поддерживается как в персональных компьютерах, так и в промышленных устройствах, а также в оборудовании для компьютерных сетей.

Доступ к COM- и LPT-портам ранее производился посредством функций BIOS (например, в Windows ранних версий — до Windows 98 включительно), аппаратных портов, а также функций API (Application Programming Interface — интерфейс прикладного программирования) операционной системы.

3.3.2 Параллельный порт

Изучение параллельного порта LPT, как наиболее простого в программировании, позволяет понять общие принципы программирования портов ввода-вывода. Поэтому рассмотрение мы начнем именно с него.

Передача данных через LPT-порт производится путем отправки байта информации от источника к приемнику по 8 линиям. В процессе передачи данных передается 1 байт за один цикл, при этом приемная и передающая стороны сообщают по отдельной линии о своем состоянии с помощью соответствующих сигналов: когда устройство занято, оно выдает сигнал *busy*, а когда программа подготовила байт данных для передачи, она пересылает устройству сигнал *strobe*. Имеются также отдельные линии связи для контроля передаваемых данных, для сообщения об отсутствии бумаги (при обмене данными с принтером), для передачи сообщений об ошибках и готовности устройства. Максимальная скорость передачи через параллельный интерфейс может составлять 120–200 Кбайт/сек, т.е. сравнительно невысока.

Напрямую обращаться к LPT-портам или же получать из них значения можно из области памяти BIOS. Так, для интерфейса LPT1 ре-

гистр данных имеет адрес 378h, регистр состояния — 379h, а регистр управления — 37Ah. Порт данных содержит данные для передачи между компьютером и устройством и доступен как для записи, так и для чтения. Порт состояния содержит информацию о текущем состоянии подключенного устройства и доступен только для чтения (табл. 3.5). Порт управления позволяет инициализировать устройство (чаще всего это принтер) и установить различные параметры работы (табл. 3.6).

Таблица 3.5 — Формат регистра состояния LPT

№ бита	Назначение
0–2	Не используются (должны быть установлены в 0)
3	Ошибка ввода-вывода (1 — нет, 0 — есть)
4	Выбор устройства (1 — доступно, 0 — недоступно)
5	Закончилась бумага (1 — нет бумаги, 0 — бумага есть)
6	Подтверждение приема данных (0 — устройство готово, 1 — устройство не готово)
7	Готовность устройства (1 — устройство свободно, 0 — занято)

Таблица 3.6 — Формат регистра управления LPT

№ бита	Назначение
0	Состояние линии STROBE (1 — можно передать данные на устройство, 0 — стандартное состояние)
1	Использование автоматического перевода строки LF (Ah) после возврата каретки CR (Dh): 1 — использовать
2	Инициализация порта устройства (0 — выполнить начальную инициализацию)
3	Выбор устройства (1 — устройство доступно, 0 — устройство недоступно)
4	Использовать прерывание от устройства (1 — использовать, 0 — нет)
5–7	Не используются (должны быть установлены в 0)

Перед началом работы с LPT–портом его необходимо инициализировать, как показано в примере ниже. Если произошла ошибка передачи данных, инициализацию следует повторить.

Пример 3.18. Инициализация параллельного порта в Ассемблере

```
; Инициализация параллельного порта
MOV DX, 37Ah ; порт управления LPT1 или LPT2
MOV AL, 0Ch ; подготавливаем порт
OUT DX, AL ; записываем значение в порт
; формируем задержку (не менее 0,5 мкс)
MOV CX, 1500 ; начальное значение счетчика
delay:
LOOP delay ; повтор CX раз
MOV AL, 08h ; инициализация порта
OUT DX, AL ; записываем значение в порт
; управления LPT-порта
```

Пример 3.19. Функция инициализации параллельного порта в Си
(Windows, подключите conio.h и unistd.h)

```
// Функция инициализации
void InitLPTW (unsigned short Port)
{
    __outbyte(Port, 0x0C); // подготавливаем порт
    usleep(1); // задержка 1 секунда
    __outbyte(Port, 0x08); // инициализируем порт
}
```

Пример 3.20. Функция инициализации параллельного порта в Си
(Linux, подключите sys/io.h и stdio.h)

```
int InitLPTL (unsigned int Port)
{
    // Предоставляем доступ к порту
    if(ioperm(Port, 1, 1)) {
        // Если возникла ошибка
        printf("Доступ запрещен, выходим\n");
        return(-1); // Возвращаем код ошибки
    }
    outb(0x0C, Port); // подготавливаем порт
    outb(0x08, Port); // инициализируем порт
}
```

В примерах 3.19, 3.20 функции `__outbyte` (Windows) и `outb` (Linux) осуществляют вывод байта в регистр порта по заданному адре-

су. Функция `ioperm` (Linux) дает доступ к указанному диапазону портов, начиная с `Port`.

После инициализации следует проверить готовность устройства, подключенного через LPT.

Пример 3.21. Проверка готовности порта LPT в Ассемблере

```
XOR AL, AL ; «обнуляем» порт
MOV DX, 379h ; порт состояния LPT1 или LPT2
IN AL, DX ; читаем байт из порта
TEST AL, 10000b; проверяем 4-й бит
JZ NO_ONLINE; устройство не подключено
```

Пример 3.22. Функция проверки готовности порта LPT в Си (Windows)

```
bool TestLPTW (unsigned short Port)
{
    unsigned char dwResult = 0;
    // читаем байт из порта
    __inbyte(Port, &dwResult);
    if ( ( dwResult & 0x10) == 0x01)
        return true; // устройство подключено
    return false; // устройство не подключено
}
```

Пример 3.23. Функция проверки готовности порта LPT в Си (Linux)

```
bool TestLPTW (unsigned short Port)
{
    int dwResult = 0;
    // Предоставляем доступ к порту
    if(ioperm(Port, 1, 1)) {
        // Если возникла ошибка
        printf("Доступ запрещен, выходим\n");
        return(-1); // Возвращаем код ошибки
    }
    // читаем байт из порта
    dwResult = inbyte (Port);
    if ( (dwResult & 0x10) == 0x01)
        return true; // устройство подключено
```

```
return false; // устройство не подключено
}
```

После того, как порт открыт и проведена проверка готовности устройства, можно производить операции ввода-вывода.

Пример 3.24. Функция для передачи байта на принтер (Windows)

```
int WriteByteToLPT (unsigned short Port,
                    unsigned char Data)
{
    unsigned char dwResult = 0;
    int iTimeWait = 5000;
    // проверяем готовность порта
    while (-- iTimeWait > 0)
    {
        // читаем порт состояния
        dwResult = __inbyte(Port + 1);
        if ( (dwResult & 0x80 ) == 0x00 ) break;
        // если закончилось время ожидания
        if (iTimeWait < 1 ) return 1;
    }
    // записываем значение байта данных в порт
    __outbyte(Port, Data);
    // устанавливаем сигнал STROBE
    __outbyte(Port + 2, 0x0D);
    // сбрасываем сигнал STROBE
    __outbyte(Port + 2, 0x0C);
    // проверяем готовность принтера к приему
    // следующего символа
    iTimeWait = 10000;
    while (-- iTimeWait > 0)
    {
        // читаем порт состояния
        dwResult = __inbyte(BasePort + 1);
        if ((dwResult & 0x08) == 0x00)
            return 2; // произошла ошибка
        if ((dwResult & 0x80) == 0x01) break;
        // закончилось время ожидания
    }
```



```

if (iTimeWait < 1) return 1;
}
return 0; // если все в порядке
}

```

Доступ к LPT-порту в Windows можно произвести с использованием функций API.

Пример 3.25. Использование Windows API для доступа к LPT

```

#include <windows.h>
// #include <iostream.h>
#include <stdio.h>
int main() {
HANDLE hLPT = CreateFile("LPT1",
    GENERIC_WRITE, 0, NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL, NULL);
if (hLPT != INVALID_HANDLE_VALUE) {
printf("LPT1 открыт\n");
char buffer[25];
memset(buffer, 0, sizeof(buffer));
strcpy(buffer, "Тестирование LPT ");
DWORD dw; OVERLAPPED ov; int i;
for (i = 0; i < 3; i++)
WriteFile(hLPT, buffer, sizeof(buffer), &dw, &ov);
CloseHandle(hLPT);
}
else printf("Произошла ошибка \n");
return 0;
}

```

Как видно из последнего примера, доступ к LPT-порту через API-функции в Windows осуществляется, как к файлу.

3.3.3 Последовательный порт

Последовательный порт (интерфейс стандарта RS-232) производит передачу данных побитно в двух направлениях с заданной частотой. Передаваемый блок данных в асинхронном режиме состоит из 10 бит: стартового бита, 8 битов данных и стоп-бита.

Скорость передачи данных через COM-порт принято измерять в *бодах* (биты в секунду). Скорость передачи может иметь следующие

значения: 180, 300, 1 200, 2400, 4800, 9600, 19200, 38400, 57600 и 115200 бод. Если два устройства (называемых часто модемами) имеют различные скорости передачи, то для их синхронизации и снижения количества ошибок будет использована наименьшая из заданных скоростей. Стартовый бит и стоп-бит определяют начало и окончание блока данных, а для снижения количества ошибок передачи передается также специальный бит четности.

Распределение адресов для последовательных портов компьютера показано в таблице 3.7.

Таблица 3.7 — Адресация COM-портов ввода-вывода

Номер порта	Регистры ввода-вывода
COM1	3F8h—3FFh
COM2	2F8h—2FFh
COM3	3E8h—3EFh
COM4	2E8h—2EFh

Отметим, что не во всех компьютерах доступны именно четыре COM-порта, но COM1 есть всегда, поэтому далее мы будем ссылаться именно на его адреса. Работа с остальными COM-портами осуществляется аналогично, необходимо только скорректировать адреса в программе.

Регистр 3F8h выполняет несколько задач и доступен как для чтения, так и для записи. Если бит 7 в регистре управления (3FBh) установлен в 0, то через 3F8h можно читать и записывать данные. Если бит 7 в регистре управления (3FBh) установлен в 1, то регистр 3F8h обрабатывает младший байт делителя частоты для скорости передачи данных.

Регистр 3F9h также выполняет несколько задач и доступен для чтения и записи. Если бит 7 в регистре управления (3FBh) равен 0, то регистр 3F9h управляет прерыванием (табл. 3.8). Если бит 7 в регистре управления (3FBh) равен 1, то 3F9h обрабатывает старший байт делителя частоты для скорости передачи данных.

Регистр 3FAh доступен для чтения и записи. Он позволяет определить причину прерывания в режиме чтения (табл. 3.9), а в режиме записи регистр управляет режимом FIFO (табл. 3.10).

Таблица 3.8 — Формат регистра 3F9h в режиме управления прерыванием

Бит	Описание
0	Состояние сигнала прерывания при наличии входных данных (1 — включен, 0 — выключен)
1	Состояние сигнала прерывания при опустошении выходного буфера (1 — включен, 0 — выключен)
2	Состояние сигнала прерывания при появлении ошибки или перерыва в передаче данных (1 — включен, 0 — выключен)
3	Состояние сигнала прерывания при изменении состояния модема (1 — включен, 0 — выключен)
4–7	Не используются и должны быть установлены в 0

Таблица 3.9 — Формат регистра 3F9h в режиме чтения

Бит	Описание
0	Состояние прерывания (1 — нет прерывания, 0 — есть прерывание)
1–2	Причина прерывания (00b — изменение состояния модема, 01b — опустошен буфер передачи, 10b — получены данные, 11b — произошла ошибка или перерыв в передаче данных)
3	Закончилось время ожидания для приемника FIFO
4–5	Не используются и должны быть установлены в 0
6–7	Наличие FIFO (11b — присутствует, 00b — отсутствует)

Таблица 3.10 Формат регистра 3F9h в режиме записи

Бит	Описание
0	Управление работой режима FIFO (1 — включен, 0 — выключен)
1	Очистка приемника FIFO (1 — очистить)
2	Очистка передатчика FIFO (1 — очистить)
3–5	Не используются и должны быть установлены в 0
6–7	Порог срабатывания для чтения данных (00b — 1 байт, 01b — 4 байта, 10b — 8 байт, 11b — 14 байт)

Регистр 3FBh доступен для чтения и записи. Он предназначен для управления линией связи. Формат регистра показан в таблице 3.11.

Таблица 3.11 — Формат регистра управления 3FBh

Бит	Описание
0–1	Длина данных (00b — 5 бит, 01b — 6 бит, 10b — 7 бит, 11b — 8 бит)
2	Количество стоповых битов (0 — 1 бит, 1 — 2 бита)
3-5	Бит четности (000b — нет, 001b — нечетный, 011b — четный)
6	Состояние перерыва передачи, при котором порт выдает нули (1 — включить, 0 — выключить)
7	Управление делителем частоты для выбора скорости передачи данных (1 — установить значение делителя частоты через регистры 3F8h и 3F9h)

Регистр 3FCh доступен для чтения и записи, и управляет работой модема. Формат регистра показан в таблице 3.12.

Таблица 3.12 — Формат регистра управления модемом 3FCh

Бит	Описание
0	Управление линией DTR
1	Управление линией RTS
2	Управление линией OUT1 (0)
3	Управление линией OUT2 (1)
4	Самотестирование модема (1 — включить), при котором выход замыкается на вход
5–7	Не используются и должны быть установлены в 0

Регистр 3FDh доступен только для чтения, и позволяет текущее состояние линии связи. Формат регистра показан в табл. 3.13.

После выполнения функции чтения в регистр AH будет записано состояние порта (табл. 3.13), а в регистр AL — состояние модема (табл. 3.14).

Регистр 3FEh доступен только для чтения. Прочитав его содержимое, можно получить текущее состояние модема. Формат регистра показан в таблице 3.14.

Таблица 3.13 — Формат байта состояния порта 3FDh

Бит	Описание
0	Готовность данных (1 — готовы, 0 — не готовы)
1	Ошибка переполнения данных (1 — есть, 0 — нет)
2	Ошибка четности (1 — есть, 0 — нет)
3	Ошибка синхронизации данных (1 — есть, 0 — нет)
4	Обнаружен перерыв в передаче данных (1 — да, 0 — нет)
5	Регистр хранения данных пуст (1 — нет данных, 0 — есть данные)
6	Регистр сдвига пуст (1 — нет данных, 0 — есть данные)
7	Тайм-аут или ошибка по времени (1 — ошибка, 0 — нет ошибки)

Таблица 3.14 — Формат байта состояния модема

Бит	Описание
0	Изменилось состояние на входной линии CTS: 1 — да
1	Изменилось состояние на входной линии DSR: 1 — да
2	Изменилось состояние на входной линии RI: 1 — да
3	Изменилось состояние на входной линии DCD: 1 — да
4	Произошел сброс для передачи на входной линии CTS: 1 — сброс
5	Готовность модема для чтения данных на входной линии DSR: 1 — готов, 0 — не готов
6	Состояние индикатора звонка на входной линии RI: 1 — есть сигнал
7	Сигнал обнаружения несущей на входной линии DCD: 1 — обнаружен

Регистр 3FCh доступен для чтения и записи. Он является резервным и не используется.

Перед началом работы последовательный порт следует инициализировать. Для этого нужно установить бит 7 регистра 3FBh в 1, затем задать скорость передачи (старший и младший байты) в регистрах 3F8h и 3F9h. После этого в регистр 3FBh следует записать режим работы линии, обнулив бит 7. Затем в регистр 3F9h записываем 0, если прерывания не будут использованы, или же устанавливаем биты 0–3 в соответствии с требуемыми прерываниями. Возможные значения для скорости передачи данных представлены в таблице 3.15.

Таблица 3.15 — Коды значений для установки скорости передачи данных

Код значения		Скорость, бод
3F9h	3F8h	
04h	17h	110
01h	80h	300
00h	C0h	600
00h	60h	1 200
00h	30h	2 400
00h	20h	3 600
00h	18h	4 800
00h	10h	7 200
00h	0Ch	9 600
00h	06h	19 200
00h	03h	38 400
00h	02h	57 600
00h	0lh	115 200

Рассмотрим на примерах программирование типовых операций с последовательным портом в следующем порядке:

1. Инициализация порта.
2. Проверка состояния порта.
3. Настройка порта
3. Передача и прием данных

Пример 3.26. Функция инициализации последовательного порта (Windows)

```
void InitCOMPort(unsigned int Port) {
    // устанавливаем бит 7 в 1 в регистре
    // управления линией
    __outbyte(Port + 3, 0x80);
    // скорость передачи данных 9 600 бод
    __outbyte(Port, 0x00);
    // младший байт делителя
    __outbyte(Port + 1, 0x0C);
    // старший байт делителя
    // настраиваем регистр управления линией
```

```

// 8 бит, 1 стоповый бит, без четности
__outbyte(Port + 3, 0x03);
// настраиваем регистр управления модемом
// сигналы DTR, RTS, OUT1, OUT2
__outbyte(Port + 4, 0x0B);
// настраиваем регистр управления прерываниями
__outbyte(Port + 1, 0x00);
// запрещаем все прерывания
}

```

Пример 3.27. Функция проверки состояния последовательного порта (Windows)

```

bool IsRead_Write_COM(unsigned short Port) {
    DWORD dwResult = 0;
    dwResult = __inword(Port + 5);
    if ((dwResult & 0x10) == 0x01)
        return false; // линия занята
    return true; }

```

Пример 3.28. Функция проверки состояния линии перед чтением из порта (Windows)

```

bool IsRead_Read_COM (unsigned short Port) {
    DWORD dwResult = 0;
    dwResult = __inword(Port + 5);
    if ((dwResult & 0x02) == 0x01)
        return false;
    // в линии остались непрочитанные данные
    return true; }

```

Произведем настройку COM-порта, обратившись к нему, как к файлу.

Пример 3.29. Настройка последовательного порта (Windows)

```

#include <windows.h>
// объявляем структуру для конфигурации
// последовательного порта
DCB deb;
ZeroMemory(&deb, sizeof(DCB));
// описатель порта HANDLE hCom 1 = NULL;

```

```

// Функция инициализации порта COM1
bool Init_COM() {
// открываем порт COM1
hCom_1 = CreateFile("COM1",
    GENERIC_READ | GENERIC_WRITE, 0, NULL,
    OPEN_EXISTING, FILE_FLAG_OVERLAPPED,
    NULL);
if (hCom_1 == INVALID_HANDLE_VALUE)
// если порт не удалось открыть
return false; // выходим из функции
// если порт успешно открыт,
// получаем его состояние
if (!GetCommState (hCom_1, &dcb)) {
// если не удалось получить статус порта,
// выходим из функции
CloseHandle(hCom_1);
return false; // выходим из функции
}
// иначе - настраиваем параметры порта
dcb.BaudRate = CBR_19200;
// скорость передачи 19 200 бод
dcb.ByteSize = 8; // размер байта данных
dcb.StopBits = ONESTOPBIT; // один стоп-бит
dcb.Parity = NOPARITY; // контроля четности нет
// сохраняем новую конфигурацию для порта
if (!SetCommState(hCom_1, &dcb)) {
// если не удалось настроить порт, выходим из
// функции
CloseHandle(hCom_1);
return false; // выходим из функции
}
return true; }

```

Теперь, когда порт открыт и сконфигурирован для работы, необходимо настроить обработку сигналов DSR и CTS.

Пример 3.30. Настройка событий для сигналов DSR и CTS (Windows)

```
// объявляем структуру для асинхронного ввода-
// вывода данных и помещаем ее в глобальную
// область видимости
OVERLAPPED over;
ZeroMemory(&over, sizeof ( OVERLAPPED));
// Функция настройки событий
bool SetEventCOM () {
// настраиваем мониторинг событий порта
if (!SetCommMask(hCom_1, EV_DSR | EV_CTS))
{
// если не удалось настроить порт, выходим
CloseHandle(hCom_1);
return false; // выходим из функции
}
// иначе - создаем объект - событие
over.hEvent=CreateEvent(NULL,FALSE,FALSE,NULL);
if (!over.hEvent) {
// если не удалось создать событие, выходим
CloseHandle (hCom_1);
return false; // выходим из функции
}
return true; }
```

На этом начальная подготовка для работы с портом закончена. Теперь создадим функции чтения и записи данных, а также управления для последовательного порта.

Пример 3.31. Функции чтения и записи для последовательного порта

```
BYTE ReadByteCOM() {
// функция чтения одного байта данных
BYTE read = 0;
DWORD dwByteRead = 0;
do { // читаем байт из порта
if (ReadFile(hCom_1, &read, sizeof(BYTE),
```

```

        sdwByteRead, NULL))
return 0;
} while (!dwByteRead);
return read; // возвращаем прочитанный байт )

DWORD ReadData_COM(void* Data,
                    unsigned int uNumBytes)
{
// функция чтения массива данных
DWORD dwBytesRead = 0;
if (IReadFile(hCom_1, Data, uNumBytes,
              SdwBytesRead, NULL))
// сколько удалось прочитать
return dwBytesRead;
// возвращаем полное число прочитанных байтов
return dwBytesRead;
)
bool WriteByteCOM ( BYTE bByte)
{
// функция для записи одного байта
BYTE write = 0;
DWORD dwByteWrite = 0;
if (IWriteFile(hCom_1, Swrite, sizeof(BYTE),
               SdwByteWrite, NULL))
return false;
return true;
}
// функция для записи массива данных
DWORD WriteDataCOM(void* Data,
                   unsigned int uNumBytes) {
DWORD dwBytesWrite = 0;
if (IWriteFile(hCom_1, Data, uNumBytes,
               SdwByteWrite, NULL))
// сколько удалось записать
return dwBytesWrite;
}

```

```
// возвращаем полное число записанных байтов
return dwBytesWrite; }
```

Пример 3.32. Общая функция для работы с последовательным портом в Windows

```
// добавляем в глобальную область данных значение для выделения сигнала
```

```
DWORD dwSignal;
```

```
void GeneralCOM () {
```

```
// проверяем сигнал в линии
```

```
if (WaitCommEvent(hCom_1, &dwSignal, Sover))
```

```
{
```

```
if (dwSignal & EV_DSR) {
```

```
// если данные готовы для чтения
```

```
// читаем байт из порта
```

```
BYTE data = ReadByteCOM();
```

```
// здесь - сохраняем полученный байт куда-либо
```

```
}
```

```
if (dwSignal & EV_CTS)
```

```
// можно писать данные в порт
```

```
// передаем извне байт и пишем его в порт
```

```
WriteByteCOM(myByte);
```

```
// после завершения работы следует вызвать
```

```
// функцию очистки ресурсов
```

```
void CloseCOM () {
```

```
if (over.hEvent) {
```

```
CloseHandle(over.hEvent);
```

```
// закрываем объект события
```

```
over.hEvent = NULL; }
```

```
if (hCom_1) {
```

```
CloseHandle(hCom_1); // закрываем порт COM1
```

```
hCom = NULL;
```

```
}
```

Программирование последовательного порта для Linux рассмотрим на примерах из документов серии Linux HOWTO (адрес ресурса: <http://stuff.mit.edu>). Материалы из этой серии согласно авторской лицен-

зии, могут быть воспроизведены и распространены полностью или частично на любом физическом или электронном носителе до тех пор, пока это уведомление об авторских правах будет сохранено на всех копиях.

Последовательные порты относятся к устройствам `/dev/ttyS*` Linux и конфигурируются для этого во время запуска системы, о чем необходимо помнить при программировании связи непосредственно через последовательные порты. Например, порты могут быть сконфигурированы во время запуска системы таким образом, что они возвращают "эхо" вводимого символа обратно, что обычно должно быть изменено при передаче данных.

Все параметры могут быть сконфигурированы из программы. Конфигурация сохраняется в структуре `termios`, которая объявлена в заголовочном файле `<asm/termbits.h>`:

```
#define NCCS 19
struct termios {
    /* флаги режима ввода (input) */
    tcflag_t c_iflag;
    /* флаги режима вывода (output) */
    tcflag_t c_oflag;
    /* флаги режима управления */
    tcflag_t c_cflag;
    /* флаги локального режима */
    tcflag_t c_lflag;
    /* дисциплина линии */
    cc_t c_line;
    /* управляющие символы */
    cc_t c_cc[NCCS];    };
```

Этот файл также содержит все объявления флагов. Флаги режима ввода в `c_iflag` управляют всей обработкой ввода, которая подразумевает, что символы, посылаемые от устройства могут быть обработаны перед тем, как они будут прочитаны вызовом функции `read`. Аналогично, флаг `c_oflag` управляет обработкой вывода. Флаг `c_cflag` содержит следующие установки для порта: скорость передачи данных, число битов в передаваемом символе, число стоп-битов, и т.д. Флаги локального режима хранятся в `c_lflag` и определяют, будет ли произ-

водиться "эхо" для вводимых символов, какие сигналы будут посылаться программе, и т.д. И наконец, массив `c_cc` описывает управляющие символы для конца файла, остановки передачи, и т.д.

Значения по умолчанию для управляющих символов описаны в заголовочном файле `asm/termios.h`. Флаги описаны на странице руководства Linux (man page) `termios(3)`.

Структура `termios` содержит также элемент `c_line` (дисциплина линии), который не используется в POSIX-совместимых системах.

Возможно использование трех различных концепций ввода: канонический ввод, неканонический ввод и асинхронный ввод. Для определенного приложения должна быть выбрана подходящая концепция.

Канонический ввод — это нормальный режим обработки для терминалов, однако он также может быть полезен для коммуникации с другим вводом, обрабатываемым построчно, что подразумевает возврат полной строки функцией `read`. Строка по умолчанию заканчивается символом NL (ASCII LF), символом конца строки или символом конца файла. Символ CR (по умолчанию — символ конца строки для DOS/Windows) не будет завершать строку при установках по умолчанию.

При каноническом вводе можно обрабатывать очистку, удаление слова, перепечатку символов, транслировать CR в NL, и т.д.

Обработка неканонического ввода манипулирует фиксированным количеством символов в одной операции чтения. Этот режим должен быть использован, если Ваше приложение всегда должно читать фиксированное количество символов, или если подключенное устройство посылает пакеты символов.

Две описанных выше концепции могут быть использованы при синхронном и асинхронном режиме. Синхронный режим устанавливается по умолчанию, при этом вызов функции `read` будет заблокирован до тех пор, пока операция чтения не будет завершена. В *асинхронном режиме* вызов `read` возвращает управление немедленно и посылает сигнал вызвавшей программе после завершения операции чтения. Этот сигнал может быть принят соответствующим обработчиком.

При обработке канонического ввода размер приемного буфера ограничен 255 символами, подобно максимальной длине строки (см. `linux/limits.h` или `posix1_lim.h`).

Рассмотрим пример реализации канонического ввода. Комментарии в коде объясняют использование различных режимов ввода. Не забудьте предоставить привилегии доступа к соответствующему последовательному порту (например: `chmod a+rw /dev/ttyS1`)!

Пример 3.33. Канонический ввод

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>

/* установки скорости передачи данных объявлены
в <asm/termbits.h>, включенном в <termios.h> */
#define BAUDRATE B38400

/* измените эти объявления для корректного указания
порта */
#define MODEMDEVICE "/dev/ttyS1"
/* POSIX-совместимый источник */
#define _POSIX_SOURCE 1
#define FALSE 0
#define TRUE 1
volatile int STOP=FALSE;
void main()      {
    int fd,c, res;
    struct termios oldtio,newtio;
    char buf[255];

/* Открываем устройство модема для чтения и записи
как неуправляющий терминал (tty), поскольку мы не
хотим завершать процесс когда помехи в линии посы-
лают сигнал CTRL-C. */
    fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY );
    if (fd <0) {perror(MODEMDEVICE); exit(-1); }
    /* сохранить текущие настройки порта */
```

```

tcgetattr(fd, &oldtio);
/* очищаем структуру для новых настроек */
bzero(&newtio, sizeof(newtio));
/* BAUDRATE: устанавливает скорость передачи
данных в bps (бод в секунду).
    CRTSCTS : аппаратное управление выводным пото-
ком данных (используется, только если кабель обла-
дает всеми необходимыми сигнальными линиями).
    CS8 : 8n1 (8 бит, без четности, 1 стоп-бит)
    CLOCAL : локальное подключение, нет управления
модемом
    CREAD : разрешает прием символов */
newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 |
    CLOCAL | CREAD;
/* IGNPAR : игнорировать байты с ошибками чет-
ности
    ICRNL : отобразить символ CR на NL (иначе
ввод CR на другом компьютере не будет завершать
ввод). Иначе установить устройство как raw (нет об-
работки ввода) */
newtio.c_iflag = IGNPAR | ICRNL;
/* Вывод без обработки */
newtio.c_oflag = 0;
/* ICANON : разрешить канонический ввод, забло-
кировать любое "эхо", и не посылать сигналы вызыва-
ющей программе */
newtio.c_lflag = ICANON;
/* инициализация всех управляющих символов
значения по умолчанию могут быть найдены в in
/usr/include/termios.h */
newtio.c_cc[VINTR] = 0; /* Ctrl-c */
newtio.c_cc[VQUIT] = 0; /* Ctrl-\ */
newtio.c_cc[VERASE] = 0; /* del */
newtio.c_cc[VKILL] = 0; /* @ */
newtio.c_cc[VEOF] = 4; /* Ctrl-d */

```

```

    newtio.c_cc[VTIME]    = 0;    /* символный таймер
не используется */
    newtio.c_cc[VMIN]     = 1;    /* блокирование чте-
ния пока поступает один символ until 1 */
    newtio.c_cc[VSWTC]    = 0;    /* '\0' */
    newtio.c_cc[VSTART]   = 0;    /* Ctrl-q */
    newtio.c_cc[VSTOP]    = 0;    /* Ctrl-s */
    newtio.c_cc[VSUSP]    = 0;    /* Ctrl-z */
    newtio.c_cc[VEOL]     = 0;    /* '\0' */
    newtio.c_cc[VREPRINT] = 0;    /* Ctrl-r */
    newtio.c_cc[VDISCARD] = 0;    /* Ctrl-u */
    newtio.c_cc[VWERASE]   = 0;    /* Ctrl-w */
    newtio.c_cc[VLNEXT]   = 0;    /* Ctrl-v */
    newtio.c_cc[VEOL2]    = 0;    /* '\0' */

    /* теперь очищаем модемную линию и активируем
наши установки порта */
    tcflush(fd, TCIFLUSH);
    tcsetattr(fd, TCSANOW, &newtio);

    /* терминальные установки выполнены, теперь
приступаем к обработке ввода. В этом примере, ввод
символа 'z' в начале строки будет завершать
работу программы. */
    while (STOP==FALSE) {
        /* цикл до тех пор, пока не получим условие за-
вершения */
        res = read(fd, buf, 255);
        buf[res]=0; /* установив завершитель строки
таким образом, мы можем выполнять
printf */
        printf(":%s:%d\n", buf, res);
        if (buf[0]=='z') STOP=TRUE;
    }

    /* восстановление старых установок порта */
    tcsetattr(fd, TCSANOW, &oldtio);
}

```


Программа выполняет чтение блока до тех пор, пока не будет получен символ завершения строки, даже если на ввод получено больше чем 255 символов. Если число прочитанных символов меньше чем число доступных символов, то последующее чтение возвратит оставшиеся символы. В переменной `res` будет установлено число реально прочитанных символов.

В режиме обработки *неканонического ввода* построчная сборка ввода, очистка, удаление слов, и прочие операции не производятся. Поведением этого режима управляют два параметра: `c_cc[VTIME]` устанавливает символьный таймер, и `c_cc[VMIN]` устанавливает минимальное количество символов, которые необходимо принять для удовлетворения операции чтения.

Если `MIN > 0` и `TIME = 0`, то `MIN` устанавливает количество символов, которые необходимо принять для выполнения операции чтения. Так как `TIME` установлено в нуль, то таймер не используется.

Если `MIN = 0` и `TIME > 0`, то `TIME` трактуется, как значение времени ожидания. Операция чтения будет завершена успешно, если прочитан одиночный символ, или значение `TIME` будет превышено ($t = \text{TIME} * 0.1$ сек.). Если значение `TIME` превышено, то не будет возвращено ни одного символа.

Если `MIN > 0` и `TIME > 0`, то `TIME` служит как межсимвольный таймер. Операция чтения считается завершенной, если будет принято `MIN` символов, или время между передачей двух символов достигло `TIME`. Таймер перезапускается каждый раз при приеме нового символа и активируется после приема первого символа.

Если `MIN = 0` и `TIME = 0`, то чтение будет завершено немедленно. При этом операция чтения вернет число фактически доступных символов или число запрошенных символов.

Рассмотрим реализацию неканонического ввода на примере.

Пример 3.34. Неканонический ввод через COM-порт

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
```

```

#include <stdio.h>
#define BAUDRATE B38400
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* POSIX-совместимый ис-
точник */
#define FALSE 0
#define TRUE 1
volatile int STOP=FALSE;
void main()    {
    int fd,c, res;
    struct termios oldtio,newtio;
    char buf[255];
    fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY );
    if (fd <0) {perror(MODEMDEVICE); exit(-1); }
    tcgetattr(fd,&oldtio); /* сохранение текущих
установок порта */
    bzero(&newtio, sizeof(newtio));
    newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 |
CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR;
    newtio.c_oflag = 0;
    /* режим ввода: неканонический, без «эха» */
    newtio.c_lflag = 0;
    newtio.c_cc[VTIME] = 0; /* посимвольный таймер
не используется */
    newtio.c_cc[VMIN] = 5; /* блокировка чтения до
тех пор, пока не будут приняты 5 символов */
    tcflush(fd, TCIFLUSH);
    tcsetattr(fd,TCSANOW,&newtio);
    while (STOP==FALSE) { /* цикл ввода */
        res = read(fd,buf,255); /* выход после приема
5-ти символов */
        buf[res]=0; /* для того, чтобы мы могли ис-
пользовать функцию printf... */
        printf(":%s:%d\n", buf, res);
    }
}

```

```

    if (buf[0]=='z') STOP=TRUE;        }
    tcsetattr(fd,TCSANOW,&oldtio);
}

```

Ниже рассмотрен пример асинхронного ввода.

Пример 3.35. Асинхронный ввод через СОМ-порт

```

#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/signal.h>
#include <sys/types.h>
#define BAUDRATE B38400
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* POSIX-совместимый ис-
точник */
#define FALSE 0
#define TRUE 1
volatile int STOP=FALSE;
void signal_handler_IO(int status);
/* объявление обработчика сигнала */
int wait_flag=TRUE; /* флаг ожидания - TRUE, пока
не получен сигнал */
void main()      {
    int fd,c, res;
    struct termios oldtio,newtio;
    struct sigaction saio;
    /* объявление действия сигнала (signal action) */
    char buf[255];
    /* открываем устройство, как не блокируемое (вы-
зов read возвратит управление немедленно) */
    fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY |
              O_NONBLOCK);
    if (fd <0) {perror(MODEMDEVICE); exit(-1); }
    /* устанавливаем обработчик сигнала перед уста-
новкой устройства как асинхронного */

```

```

saio.sa_handler = signal_handler_IO;
saio.sa_mask = 0;
saio.sa_flags = 0;
saio.sa_restorer = NULL;
sigaction(SIGIO, &saio, NULL);
/* разрешаем процессу получать SIGIO */
fcntl(fd, F_SETOWN, getpid());
/* делаем файловый дескриптор асинхронным */
fcntl(fd, F_SETFL, FASYNC);
/* сохраняем текущие настройки порта */
tcgetattr(fd, &oldtio);
/* задаем новые установки порта для обработки канонического ввода */
newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 |
                CLOCAL | CREAD;
newtio.c_iflag = IGNPAR | ICRNL;
newtio.c_oflag = 0;
newtio.c_lflag = ICANON;
newtio.c_cc[VMIN]=1;
newtio.c_cc[VTIME]=0;
tcflush(fd, TCIFLUSH);
tcsetattr(fd, TCSANOW, &newtio);
/* цикл ожидания ввода. Здесь можно производить
другие операции */
while (STOP==FALSE) {
    printf(".\n");usleep(100000);
    /* после получения SIGIO, wait_flag = FALSE, ввод
доступен и может быть прочитан */
    if (wait_flag==FALSE) {
        res = read(fd, buf, 255);
        buf[res]=0;
        printf(":%s:%d\n", buf, res);
        if (res==1) STOP=TRUE; /* останавливаем цикл при
вводе символа CR */
        wait_flag = TRUE; /* ждем нового ввода */
    }
}

```

```

    }
}
/* восстанавливаем старые настройки порта */
tcsetattr(fd, TCSANOW, &oldtio); }
/* обработчик сигнала. устанавливает
wait_flag в FALSE для индикации
наличия принятого символа */
void signal_handler_IO(int status) {
    printf("received SIGIO signal.\n");
    wait_flag = FALSE; }

```

3.3.4 Программирование USB [7]

Универсальная последовательная шина (Universal Serial Bus — USB) обеспечивает расширяемый последовательный интерфейс Plug and Play (автоматическая идентификация подключенного устройства), обеспечивающий стандартное подключение для большинства периферийных устройств, таких как клавиатуры, мыши, джойстики, принтеры, сканеры, устройства хранения данных, модемы и видеокамеры. Переход на USB рекомендован для всех периферийных устройств, которые используют устаревшие порты, такие как PS/2, последовательные и параллельные порты.

Операционные системы Windows включают встроенную поддержку контроллеров, концентраторов и устройств USB, которые соответствуют официальной спецификации. Windows предоставляет программные интерфейсы, которые можно использовать для разработки драйверов устройств и приложений, реализующих обмен данными с USB-устройством.

На момент написания пособия известны следующие версии стандарта USB: 1.0, 1.1, 2.0, 3.0, 3.1, 3.2. Главным отличием между версиями стандарта является пропускная способность шины. Для версии 1.1 максимальная скорость передачи данных не превышает 12 Мбит/с, а для версии 3.2 максимальная скорость заявлена на уровне 20 Гбит/с. Устройства, поддерживающие USB 3.2, планируется выпускать, начиная с 2018 года. На настоящий момент наиболее распространенной является версия USB 2.0.

В первую очередь, для работы с USB необходимо представлять структуру этого интерфейса. Основой интерфейса служит базовое устройство (host), которое связывает компьютер и другие USB-устройства посредством последовательной шины. Топология интерфейса реализована по схеме «звезда», в которую входят разветвители (hub), через которые и подключаются непосредственно сами устройства USB.

Стандарт USB позволяет подключать до 5 базовых устройств и до 127 конечных устройств.

Корневой базовый узел включает в себя контроллер базового узла, выполняющий основные задачи:

1. Автоматическое определение подключения или удаления устройств USB;
2. Управление обменом данными между базовым узлом и устройствами USB;
3. Сбор статистических сведений о текущих состояниях подключенных устройств;
4. Обеспечение и управление питанием подключенных USB-устройств.

Кроме аппаратной, обеспечивается программная поддержка USB, которая реализует следующие возможности:

1. Просмотр устройств USB на шине и получение их конфигурации.
2. Изохронную и асинхронную передачу данных.
3. Управление питанием.
4. Управление информацией для шины и устройств USB.

Стандартная связь между базовым узлом и USB-устройством формируется на основе программно-аппаратного комплекса, в который входят:

1. Физическое устройство, подключенное с одной стороны кабеля USB.
2. Клиентское программное обеспечение, разработанное для конкретного устройства USB и взаимодействующее с базовым узлом.
3. Системное программное обеспечение, предназначенное для поддержки интерфейса USB в конкретной операционной системе.
4. Базовый узел-контроллер, содержащий как аппаратные, так и программные средства для подключения устройства USB.

Обмен данными по шине выполняется посредством пакетов данных. Каждый пакет содержит:

1. Поле синхронизации размером 8 бит для стандарта 1.1 и 32 бита для стандарта 2.0.
2. Поле идентификатора пакета размером 8 бит, в котором младшие 4 бита определяют тип пакета, а старшие — содержат контрольную сумму для правильной расшифровки данных.
3. Поле адреса, служащее для определения конечной точки расположения устройства на шине.
4. Дополнительное поле конечной точки, которое представляет собой дополнительное 4-битовое поле адреса, если устройство требует более одной конечной точки.
5. Поле нумерации счетчика базового узла, имеющее размер 11 бит, может принимать максимальное значение 7FFh.
6. Поле данных, которое может иметь размер от 0 до 1024 байтов и содержит передаваемые данные.

Для управления пакетами данных базовый узел формирует транзакции (последовательности различных действий, объединенных в общую группу), которые состоят из следующих пакетов: пакет описания устройства USB (тип и адрес), направления передачи (от базового узла к устройству или наоборот), пакет данных и пакет подтверждения (результат приема или передачи данных).

В начале работы базовый узел посылает устройству запрос установки для идентификации и определения основных свойств. Размер установочного запроса составляет 8 байт. Его структура приведена в таблице 3.16.

Таблица 3.16 — Структура запроса установки

Смещение	Наименование поля	Размер поля, байт	Описание
0	bmRequestType	1	Определяет тип запроса (см. табл. 3.17)
1	bRequest	1	Стандартный тип запроса (табл. 3.18), который должен поддерживаться всеми устройствами USB

Продолжение таблицы 3.16

Смещение	Наименование поля	Размер поля, байт	Описание
2	wValue	2	Дополнительное описание параметра запроса, как правило, тип описателя (табл. 3.19)
4	wIndex	2	Дополнительное описание параметра запроса
6	wLength	2	Размер передаваемых данных. Зависит от бита направления передачи данных поля bmRequestType

Таблица 3.17 — Тип запроса bmRequestType

Биты	Описание
0–4	Получатель: 0 — устройство, 1 — интерфейс, 2 — конечная точка, 3 — другой получатель
5, 6	Тип запроса: 0 — обычный, 1 — особый для определенного класса устройств, 2 — определенный производителем устройства, 3 — резерв
7	Направление передачи: 0 — от базового узла к устройству, 1 — от устройства к базовому узлу

Таблица 3.18 — Стандартные типы запросов

Код запроса	Тип запроса	Описание
00h	GET_STATUS	Получает состояние определенного устройства (получателя)
01h	CLEAR_FEATURE	Используется для очистки или отключения определенной настройки устройства
03h	SET_FEATURE	Используется для установки или включения определенной настройки устройства
05h	SET_ADDRESS	Устанавливает адрес устройства для дальнейшего доступа
06h	GET_DESCRIPTOR	Получает описатель устройства, если он существует
07h	SET_DESCRIPTOR	Дополнительный запрос для обновления информации о существующем или вновь добавленном описателе устройства

Продолжение таблицы 3.18

Код запроса	Тип запроса	Описание
08h	GET_CONFIGURATION	Получает значение текущей конфигурации устройства
09h	SET_CONFIGURATION	Устанавливает конфигурацию устройства
0Ah	GET_INTERFACE	Получает выбранную настройку для определенного интерфейса
0Bh	SET_INTERFACE	Позволяет базовому узлу выбирать дополнительные настройки для определенного интерфейса
0Ch	SYNCH_FRAME	Предназначен для установки и сообщения структуры объекта синхронизации для конечной точки

Таблица 3.19 — Стандартные типы описателей

Значение	Тип описателя
01h	Описатель устройства
02h	Описатель конфигурации
03h	Строковый описатель
04h	Описатель интерфейса
05h	Описатель конечной точки
06h	Дополнительный описатель устройства
07h	Описатель конфигурации управления питанием
08h	Описатель интерфейса управления питанием

Теперь рассмотрим стандартные запросы, поддерживаемые всеми устройствами USB.

Запрос `CLEAR_FEATURE` позволяет очистить или отключить определенную настройку (свойство) устройства USB, интерфейса или конечной точки. Формат запроса представлен в таблице 3.20.

Таблица 3.20 — Формат запроса CLEAR_FEATURE

Поле запроса	Значение
bmRequestType	0h — устройство, 1h — интерфейс, 2h — конечная точка
bRequest	01h
wValue	Значение свойства получателя (табл. 3.21)
wIndex	0 для конечной точки или номер устройства
wLength	0
Данные	Нет

Таблица 3.21 — Значение свойства получателя

Свойство	Получатель	Значение
ENDPOINT_HALT	Конечная точка	0
DEVICE_REMOTE_WAKEUP	Устройство USB	1
TEST_MODE	Устройство USB	2

Запрос GET_CONFIGURATION получает значение текущей конфигурации устройства. Формат запроса представлен в таблице 3.22.

Таблица 3.22 — Формат запроса GET_CONFIGURATION

Поле запроса	Значение
bmRequestType	80h
bRequest	08h
wValue	0
wIndex	0
wLength	1
Данные	Значение конфигурации

Если поле данных, прочитанное запросом GET_CONFIGURATION, равно 0, значит, устройство не сконфигурировано. В противном случае будет возвращен 1 байт данных конфигурации.

Запрос GET_DESCRIPTOR позволяет получить определенный описатель, если он существует. Формат запроса представлен в таблице 3.23.

Таблица 3.23 — Формат запроса GET_DESCRIPTOR

Поле запроса	Значение
bmRequestType	80h
bRequest	06h
wValue	Тип и индекс описателя
wIndex	0 или Language ID
wLength	Длина описателя
Данные	Описатель

В поле wValue заносят тип описателя (старший байт) и индекс описателя (младший байт). Доступные типы описателя представлены в таблице 3.19.

Описатель индекса используется только для конфигурации или строки, иначе он должен быть равен 0. В поле wIndex следует передать идентификатор языка (только для описателя строки) или 0. В поле wLength нужно записать размер описателя в байтах. Если в результате выполнения запроса реальная длина описателя окажется больше, то базовый узел возвратит только данные, соответствующие заданному в поле wLength значению.

Запрос GET_INTERFACE позволяет получить выбранную настройку (свойство) для определенного интерфейса. Формат запроса представлен в табл. 3.24.

Таблица 3.24 — Формат запроса GET_INTERFACE

Поле запроса	Значение
bmRequestType	81h
bRequest	0Ah
wValue	0
wIndex	Интерфейс
wLength	1
Данные	Текущая настройка

После выполнения запроса GET_INTERFACE базовый узел возвратит 1 байт данных с текущей настройкой интерфейса. Если интерфейс не существует, будет сгенерирован запрос ошибки.

Запрос GET_STATUS позволяет получить текущее состояние получателя, в качестве которого может быть указано устройство USB, интерфейс или конечная точка. Формат запроса представлен в таблице 3.25.

Таблица 3.25 — Формат запроса GET_STATUS

Поле запроса	Значение
bmRequestType	81h
bRequest	80h — устройство, 81h — интерфейс, 82h — конечная точка
wValue	0
wIndex	0, интерфейс, конечная точка
wLength	2
Данные	Текущее состояние

В поле wIndex для получения статуса устройства USB нужно записать 0, а для интерфейса и конечной точки — его номер. После выполнения запроса базовый узел возвратит 2 байта, содержащие информацию о статусе.

Для устройства USB информационными являются следующие биты: бит 0 — определяет источник питания устройства (0 — питание поступает с шины, 1 — устройство имеет собственный источник питания), бит 1 — определяет, доступен ли устройству внешний сигнал на пробуждение (0 — данная возможность выключена, 1 — включена). Остальные биты (с 2 по 15) зарезервированы и не используются.

Для интерфейса все биты зарезервированы и равны 0. Для конечной точки бит 0 определяет блокировку и принимает следующие значения: 0 — конечная точка доступна, 1 — конечная точка выключена. Остальные биты (с 1 по 15) зарезервированы и равны 0.

Запрос SET_ADDRESS позволяет установить адрес устройства для последующего доступа. Формат запроса представлен в таблице 3.26.

Таблица 3.26 — Формат запроса SET_ADDRESS

Поле запроса	Значение
bmRequestType	00h
bRequest	05h
wValue	Адрес
wIndex	00, интерфейс, конечная точка
wLength	0
Данные	Не используется

В поле wValue следует записать значение адреса, которое будет использовано в дальнейшем для доступа к устройству на шине. При установке адреса следует помнить, что максимальное значение не может превышать 127, поскольку так определено стандартом USB. Передача значения 0 также приведет к неопределенному результату.

Запрос SET_CONFIGURATION позволяет задать конфигурацию для устройства USB. Формат запроса представлен в таблице 3.27.

В младший байт поля wValue следует записать значение конфигурации. Оно может быть равно 0 или значению описателя конфигурации.

Таблица 3.27 - Формат запроса SET_CONFIGURATION

Поле запроса	Значение
bmRequestType	00h
bRequest	09h
wValue	Значение конфигурации
wIndex	0
wLength	0
Данные	Не используется

Запрос SET_DESCRIPTOR является дополнительным и позволяет обновить существующий описатель или добавить новый. Формат запроса представлен в таблице 3.28.

Таблица 3.28 — Формат запроса SET_DESCRIPTOR

Поле запроса	Значение
bmRequestType	00h
bRequest	07h
wValue	Тип или индекс описателя
wIndex	0 или Language ID
wLength	Длина описателя
Данные	Описатель

В поле `wValue` следует занести тип описателя (старший байт) и его индекс (младший байт). Доступные типы описателя представлены в таблице 3.19. При этом описатель индекса используется только для конфигурации или строки, иначе он должен быть равен 0. В поле `wIndex` передают идентификатор языка (только для описателя строки) или 0. В поле `wLength` нужно записать размер описателя в байтах.

После успешного выполнения запроса будет возвращена информация о запрошенном описателе или сгенерирован запрос ошибки.

Запрос SET_FEATURE обеспечивает установку или включение определенной настройки (свойства устройства, интерфейса или конечной точки). Формат запроса представлен в таблице 3.29.

Таблица 3.29 — Формат запроса SET_FEATURE

Поле запроса	Значение
bmRequestType	00h — устройство, 01h — интерфейс, 02h — конечная точка
bRequest	03h
wValue	Значение настройки
wIndex	0 или настройка режима тестирования
wLength	0
Данные	Не используется

В поле `wValue` следует записать значение настройки, соответствующее коду, заданному в поле `bmRequestType`. Поле `wIndex` для интерфейса и конечной точки должно быть равно 0, а для устройства USB одному из стандартных значений из таблицы 3.30.

Таблица 3.30 — Настройки режима тестирования

Поле запроса	Значение
00h	Резерв
01h	Test_J (тестирование быстродействия)
02h	Test_K (тестирование быстродействия)
03h	Test_SE0_NAK (тестирование быстродействия)
04h	Test_Packet (тестирование волновых характеристик сигнала)
05h	Test_Force_Enable (тестирование подключений на линии)
06h — BFh	Резерв
C0h — FFh	Зарезервировано для производителей

Запрос SET_INTERFACE позволяет базовому узлу выбирать настройки для определенного интерфейса. Формат запроса представлен в таблице 3.31.

Таблица 3.31 — Формат запроса SET_INTERFACE

Поле запроса	Значение
bmRequestType	01h
bRequest	0Bh
wValue	Значение настройки
wIndex	Интерфейс
wLength	0
Данные	Не используется

В поле wValue следует записать значение настройки, а в поле wIndex — номер интерфейса. Если интерфейс не существует или отсутствует настройка, то будет сгенерирован запрос ошибки.

Запрос SYNCH_FRAME предназначен для установки структуры объекта синхронизации для конечной точки. Формат запроса представлен в таблице 3.32.

Таблица 3.32 — Формат запроса SYNCH_FRAME

Поле запроса	Значение
bmRequestType	82h
bRequest	82h
wValue	0
wIndex	Конечная точка
wLength	2
Данные	Номер кадра

Данный запрос выполняется для конечной точки, которая поддерживает изохронный обмен данными и позволяет синхронизировать ее работу относительно базового узла. В поле `wIndex` следует записать номер конечной точки. В результате выполнения запроса будет возвращено 2 байта с текущим номером кадра.

Далее рассмотрим подробнее, что такое описатель (descriptor). Описатель — это структура данных, которая содержит всю информацию об устройстве USB (интерфейсе или конечной точке), необходимую базовому узлу для взаимодействия с ним. Имеются основной и дополнительный описатель устройства, а также описатели конфигурации, интерфейса и конечной точки.

Формат основного описателя устройства представлен в таблице 3.33.

Таблица 3.33 — Формат основного описателя устройства

Смещение	Размер, байт	Наименование	Описание
00h	1	bLength	Размер описателя в байтах
01h	1	bDescriptorType	Тип описателя. Некоторые из существующих стандартных типов представлены в табл. 3.34
02h	2	BcdUSB	Номер версии текущего стандарта USB (для стандарта 2.0 это 200h), закодированный в двоично-десятичном формате
04h	1	bDeviceClass	Код класса устройства. При использовании значения 00h каждый интерфейс в пределах конфигурации имеет собственный класс, а различные интерфейсы работают независимо

Продолжение таблицы 3.33

05h	1	bDeviceSubClass	Код подкласса устройства. Если поле кода класса равно 00h , то в это поле тоже должен передаваться 0
06h	1	bDeviceProtocol	Код протокола. Определяет протокол, используемый переданным классом устройства. Если значение равно 00h , то устройство не работает с протоколом, заданным для данного класса. Если значение равно FFh, то устройство работает по протоколу производителя
07h	1	bMaxPacketSize0	Максимальное значение пакета данных для нулевой конечной точки. Доступны только следующие значения: 8, 16, 32 и 64
08h	2	idVendor	Код производителя устройства — уникальное значение для каждого производителя
0Ah	2	idProduct	Идентификатор изделия
0Ch	2	bcdDevice	Номер версии устройства в двоично-десятичном формате
0Eh	1	iManufacturer	Индекс смещения для строкового описания производителя
0Fh	1	iProduct	Индекс смещения для строкового описания изделия
10h	1	iSerialNumber	Индекс смещения для строкового описания серийного номера устройства
11h	1	bNumConfigurations	Количество доступных конфигураций устройства

Таблица 3.34 — Типы описателей

Тип	Значение	Описание
USB_DEVICE_DESCRIPTOR_TYPE	01h	Описатель устройства USB
USB_CONFIGURATION_DESCRIPTOR_TYPE	02h	Описатель конфигурации
USB_STRING_DESCRIPTOR_TYPE	03h	Описатель строки

Продолжение таблицы 3.34

Тип	Значение	Описание
USB_INTERFACE_DESCRIPTOR_TYPE	04h	Описатель интерфейса
USB_ENDPOINT_DESCRIPTOR_TYPE	05h	Описатель конечной точки

Таблица 3.35 — Классы устройств

Класс	Значение	Описание
USB_DEVICE_CLASS_RESERVED	00h	Зарезервирован
USB_DEVICE_CLASS_AUDIO	01h	Устройство, работающее со звуком
USB_DEVICE_CLASS_COMMUNICATIONS	02h	Устройство связи
USB_DEVICE_CLASS_HUMAN_INTERFACE	03h	Устройство для взаимодействия с человеком
USB_DEVICE_CLASS_MONITOR	04h	Устройство для вывода изображения
USB_DEVICE_CLASS_PHYSICAL_INTERFACE	05h	Устройство с определенным физическим интерфейсом
USB_DEVICE_CLASS_POWER	06h	Устройство управления питанием
USB_DEVICE_CLASS_PRINTER	07h	Устройство печати
USB_DEVICE_CLASS_STORAGE	08h	Устройство хранения данных
USB_DEVICE_CLASS_HUB	09h	Разветвитель шины (hub)
USB_DEVICE_CLASS_VENDOR_SPECIFIC	FFh	Класс устройства определяется изготовителем

Кроме основного описателя, существует дополнительный, который содержит информацию о быстродействии устройства. Например, если устройство работает на максимальной скорости, то описатель возвратит информацию о том, как устройство могло бы работать на пониженной скорости и наоборот. Формат дополнительного описателя устройства представлен в таблице 3.36.

Таблица 3.36 — Формат дополнительного описателя устройства

Смещение	Размер, байт	Именованние	Описание
00h	1	bLength	Размер структуры описателя в байтах
01h	1	bDescriptorType	Тип дополнительного описателя
02h	2	bcdUSB	Номер версии текущего стандарта USB (для 2.0 это 200h), закодированный в двоично-десятичном формате
04h	1	bDeviceClass	Код класса
05h	1	bDeviceSubClass	Код подкласса
06h	1	bDeviceProtocol	Код протокола
07h	1	bMaxPacketSize0	Максимальный размер пакета данных для другой скорости
08h	1	bNumConfigurations	Число конфигураций для другой скорости
09h	1	bReserved	Значение поля зарезервировано и должно быть равно 0

Следующий тип описателя, который мы рассмотрим, содержит конфигурацию устройства USB. Он может представлять более одного описателя конфигурации для определенного устройства USB. В свою очередь, каждая конфигурация может состоять более чем из одного интерфейса, а интерфейс иметь несколько или ни одной конечной точки. Формат описателя конфигурации представлен в таблице 3.37.

Таблица 3.37 — Формат описателя конфигурации

Смещение	Размер, байт	Наименование	Описание
00h	1	bLength	Размер структуры описателя в байтах
01h	1	bDescriptorType	Тип описателя
02h	2	wTotalLength	Длина описателя в байтах
04h	1	bNumInterfaces	Количество интерфейсов, поддерживаемых конфигурацией

Продолжение таблицы 3.37

Смещение	Размер, байт	Наименование	Описание
00h	1	bLength	Размер структуры описателя в байтах
05h	1	bConfigurationValue	Код конфигурации, используемый в стандартном запросе для ее установки
06h	1	iConfiguration	Смещение для строкового описания конфигурации
07h	1	bmAttributes	Дополнительные свойства конфигурации. Биты 0—4 и 7 зарезервированы и равны 0. Бит 5 определяет, доступен ли устройству внешний сигнал на «пробуждение»: 0 — нет, 1 — доступен. Бит 6 определяет управление питанием: 0 — по шине USB, 1 — свой источник питания
08h	1	bMaxPower	Максимальное значение тока, потребляемое устройством USB в мА с шагом в 2 мА (например, значение 20 соответствует току 40 мА)

Стандартный описатель содержит сведения об интерфейсе для заданной конфигурации, которая поддерживает один или несколько интерфейсов, каждый из которых включает в себя описатель конечной точки.

Запрос GET_CONFIGURATION возвращает описатель интерфейса, а также описатель конечной точки. Следует помнить, что описатель интерфейса нельзя получить непосредственно запросом GET_DESCRIPTOR, а только через описатель конфигурации. Формат описателя интерфейса представлен в таблице 3.38.

Таблица 3.38 — Формат описателя интерфейса

Смещение	Размер, байт	Именованение	Описание
00h	1	bLength	Размер структуры описателя в байтах
01h	1	bDescriptorType	Тип описателя
02h	1	bInterfaceNumber	Номер интерфейса в пределах конфигурации. Нумерация начинается с 0
03h	1	bAlternateSetting	Дополнительные настройки для интерфейса, номер которого задан в поле bInterfaceNumber
04h	1	bNumEndpoints	Количество поддерживаемых интерфейсом конечных точек, исключая нулевую. При использовании интерфейса нулевой конечной точки значение этого поля равно 0
05h	1	bInterfaceClass	Код класса интерфейса. Значение FFh определяет код класса, заданный производителем устройства
06h	1	bInterfaceSubClass	Код подкласса интерфейса. Если значение кода класса равно 0, то в данном поле также должен быть 0
07h	1	bInterfaceProtocol	Код протокола. Зависит от кодов класса и подкласса интерфейса. Значение 0 информирует о том, что интерфейс не поддерживает протокол данного класса устройства, а значение FFh определяет работу интерфейса по протоколу, заданному производителем
08h	1	iInterface	Индекс смещения для строкового описания интерфейса.

Интерфейс поддерживает от одной до нескольких конечных точек, каждая из которых имеет свой описатель. Этот описатель содержит информацию о пропускной способности конечной точки и является частью данных конфигурации, возвращаемых с помощью запроса GET_CONFIGURATION. Получить описатель конечной точки непосредственно через запрос GET_DESCRIPTOR нельзя. Кроме того, для нулевой конечной точки описателя не существует. Формат описателя конечной точки представлен в таблице 3.39.

Таблица 3.39 — Формат описателя конечной точки

Смещение	Размер, байт	Именованное	Описание
00h	1	bLength	Размер структуры описателя в байтах
01h	1	bDescriptorType	Тип описателя
02h	1	bEndpointAddress	Адрес конечной точки для устройства USB
03h	1	bmAttributes	Свойства конечной точки.
04h	2	wMaxPacketSize	Максимальный размер пакета данных, поддерживаемый конечной точкой.
06h	1	bInterval	Интервал опроса конечной точки (в миллисекундах) при передаче данных

Адрес конечной точки для устройства USB (параметр bEndpointAddress) задается следующим образом: биты 3–0 — номер конечной точки, биты 6–4 зарезервированы и должны быть равны 0, бит 7 — направление передачи данных (0 — от базового узла к конечной точке, 1 — от конечной точки к базовому узлу). Значение бита 7 не используется.

Свойства конечной точки (параметр bmAttributes) определены соотношением бит: биты 7 и 6 не используются и должны быть равны 0, биты 5–4 определяют вариант использования конечной точки (00h — конечная точка данных, 01h — конечная точка обратной связи, 02h — конечная точка для неявной обратной связи, 03h — резерв), биты 3–2 определяют тип синхронизации (00h — без синхронизации, 01h —

асинхронный, 02h — адаптивный, 03h — синхронный), биты 1–0 задают свойства конечной точки (00h — контрольная, 01h — изохронная, 02h — для передачи данных, 03h — для прерываний).

Максимальный размер пакета данных, поддерживаемый конечной точкой (параметр `wMaxPacketSize`), задается следующим образом: биты 15–13 зарезервированы и должны быть установлены в 0, биты 12–11 определяют количество транзакций в одном кадре (00h — одна транзакция, 01h — две, 02h — три, 03h — резерв), а биты 10–0 содержат максимальный размер пакета данных в байтах

Описатель параметров строки не является обязательным и в этом случае возвращаемые данные равны 0. Данные описателя строки заданы в формате Unicode и поддерживают все языки, определяемые этим форматом. Структура описателя строки представлена в таблице 3.40.

Таблица 3.40 — Формат описателя строки

Смещение	Размер, байт	Наименование	Описание
00h	1	bLength	Размер структуры описателя в байтах
01h	1	bDescriptorType	Тип описателя — USB_STRING_DESCRIPTOR_TYPE
02h	N	bString	строка в формате Unicode

Так как устройство USB может поддерживать несколько языков, для получения описателя строки следует передавать в запрос идентификатор языка (Language ID). Формат идентификатора языка показан в таблице 3.41.

Таблица 3.41 — Формат идентификатора языка

Смещение	Размер, байт	Наименование	Описание
00h	1	bLength	Размер структуры описателя в байтах
01h	1	bDescriptorType	Тип описателя — USB_STRING_DESCRIPTOR_TYPE

Продолжение таблицы 3.41

Смещение	Размер, байт	Наименование	Описание
02h	2	wLANGID[0]	идентификатор языка
N	2	wLANGID[x]	идентификатор языка

Нумерация идентификаторов языка начинается с 0. Структура идентификатора языка состоит из следующих данных: биты 9–0 определяют первичный идентификатор языка, а биты 15–10 — дополнительный идентификатор диалекта. Примеры некоторых идентификаторов языков представлены в таблице 3.42.

Таблица 3.42 — Идентификаторы языка

Идентификатор	Язык
0x042B	Армянский
0x0423	Белорусский
0x0409	Английский (США)
0x0419	Русский
0x0422	Украинский

Для программирования запросов Вам понадобятся файлы определений (в частности `hidsdi.h`), входящие в состав пакета для разработки драйверов (DDK — Driver Developers Kits). Целесообразно разместить в отдельном файле необходимые для поддержки USB константы и структуры. Полное содержание этого файла приведено в Приложении Б.

Теперь, когда все необходимые структуры данных определены, следует создать новый класс для работы с устройствами USB. Чтобы продемонстрировать основные принципы программирования данного типа оборудования, рассмотрим два базовых метода, совмещенных в одном классе. Пусть новый класс называется `USB`, и для него создадим два файла `USB.h` (файл определений) и `USB.cpp` (файл реализации методов и функций класса).

Реализация поддержки устройств USB в представленном классе осуществляется двумя способами: посредством набора функций мене-

джера установки устройств и с помощью универсальной функции ввода-вывода `DeviceIoControl`. Рассмотрим каждый из них более подробно.

В первом случае вызывается функция `_getDevicesParams`. Она находит все устройства определенного класса и выделяет их основные параметры (идентификаторы производителя и изделия) для дальнейшего доступа. Чтобы явно указать класс USB-устройств, используем функцию `HidD_GetHidGuid`, которая предназначена для определения уникального глобального идентификатора (GUID), поддерживающего все устройства с интерфейсом USB.

Затем передадим GUID в функцию `SetupDiGetClassDevs` для получения информации об устройствах определенного класса (в нашем случае — USB). После этого, периодически вызывая функцию `SetupDiEnumDeviceInterfaces`, определяем параметры всех имеющихся в системе USB-устройств. Для этого дополнительно применим функцию `SetupDiGetDeviceInterfaceDetail`. Главная ее задача состоит в возвращении пути к найденному устройству. Используя путь, открываем устройство функцией `CreateFile` и получаем основные параметры посредством функции `HidD_GetAttributes`. Получаем информацию о производителе и серийном номере устройства (`HidD_GetManufacturerString`, `HidD_GetProductString` и `HidD_GetSerialNumberString`). И в завершении закрываем текущее устройство и переходим к поиску следующего.

Когда все устройства найдены, можем приступить к непосредственной работе с ними. Сначала следует проанализировать возможности устройства, особенно если планируется выполнять обмен данными. Эта задача решается в функции `GetDeviceCaps`. Вызываем функцию `HidD_GetPreparsedData` (предварительная подготовка данных об устройстве) и в случае успешного завершения определяем непосредственно поддерживаемые устройством возможности (`HidP_GetCaps`).

Пусть нас интересуют только три параметра: размер данных для управления свойствами устройства (функция `FeatureReportByteLength`), минимальные размеры в байтах входного (функция `InputReportByteLength`) и выходного (функция

OutputReportByteLength) буферов для обмена данными. Для получения и установки свойств используются соответственно функции HidD_GetFeature и HidD_SetFeature. Обмен данными с устройствами USB осуществляется с помощью стандартных файловых функций Win32 — ReadFile и WriteFile.

Второй способ основан на использовании уже известной нам функции DeviceIoControl. Принцип работы этой функции базируется на прямом взаимодействии программы и устройства (драйвера) посредством специальных управляющих кодов. Так, например, для сброса параметров USB-порта служит код IOCTL_INTERNAL_USB_RESET_PORT.

Полный список всех поддерживаемых кодов можно посмотреть в файле usbioctl.h, входящем в состав пакета DDK. Кроме того, посредством функции DeviceIoControl можно выполнять стандартные запросы для устройств USB. Рассмотрим это на примере.

Пример 3.36. Использование стандартных запросов

```
#define CTL_CODE(DeviceType, Function, Method,
Access) ( ((DeviceType) << 16) | ((Access) << 14) |
          ((Function) << 2) | (Method) )

// Определяем константы
#define METHOD_BUFFERED 0
#define FILE_ANY_ACCESS 0
#define FILE_DEVICE_UNKNOWN 0x00000022
#define USBIO_IOCTL_VENDOR 0x0800
#define IOCTL_USBIO_WRITE_PACKET CTL_CODE( \
FILE_DEVICE_UNKNOWN, \
USBIO_IOCTL_VENDOR + 10, \
METHOD_BUFFERED, \ FILE_ANY_ACCESS)

// структура для описания пакетной команды
struct ioPacket {
unsigned char Recipient;
unsigned char DeviceModel;
unsigned char MajorCmd;
unsigned char MinorCmd;
unsigned char DataMSB;
```

```

unsigned char DataLSB;
unsigned short Length;
} VENDORPACKET, *PVENDORPACKET;
// функция для обработки команды
int UsbCmd(PVENDORPACKET pPacket) {
    ULONG nBytes;
    BOOLEAN bSuccess;
    // выполняем команду
    bSuccess = DeviceIoControl(hUsb,
        IOCTL_USBIO_WRITE_PACKET,
        pPacket, sizeof( VENDORPACKET ), pPacket,
        sizeof( VENDORPACKET ), &nBytes, NULL) ;
    if(!bSuccess ) return NULL;
    else          return nBytes;
}

```

Все перечисленные способы программирования устройств USB используют специальный системный драйвер. Чтобы получить прямой доступ к устройствам посредством портов ввода-вывода, понадобится предварительно собрать информацию об адресах, номерах прерываний и других выделенных системой ресурсах.

В завершении мы рассмотрим универсальный контроллер базового узла- USB фирмы Intel (UHCI — Universal Host Controller Interface). Работать с контроллером можно как напрямую посредством портов, так и через управляющие программные модули (драйверы). Второй вариант лучше, поскольку отвечает концепции работы с оборудованием, принятым в операционных системах Windows. Контроллер доступен через базовый набор регистров. Они делятся на две группы: регистры ввода-вывода и регистры конфигурации для устройств на шине PCI. Первая группа отвечает за отображение адресов ввода-вывода и статуса в адресном пространстве шины PCI. Адреса регистров даны в виде смещения относительно базового адреса. Список регистров ввода-вывода представлен в таблице 3.43.

Таблица 3.43 — Регистры ввода-вывода

Адрес	Регистр
Базовый адрес + (00h—01h)	USB Command (регистр команд)
Базовый адрес + (02h—03h)	USB Status (регистр состояния)
Базовый адрес + (04h—05h)	USB Interrupt Enable (регистр управления прерываниями)
Базовый адрес + (06h—07h)	Frame Number (регистр счетчика кадров)
Базовый адрес + (08h—0Bh)	Frame List Base Address (регистр адресов списка кадров)
Базовый адрес + 0Ch	Start Of Frame Modify (регистр стартового кадра)
Базовый адрес + (10h— 11h)	Port 1 Status/Control (регистр управления и статуса для порта 1)
Базовый адрес + (12h— 13h)	Port 2 Status/Control (регистр управления и статуса для порта 2)

Вторая группа регистров необходима для представления в пространстве шины PCI адресных регистров для поддержки устройств USB. Следует учитывать, что контроллер не взаимодействует с шиной PCI, а лишь предоставляет ей адреса для обработки их счетчиком PCI. Счетчик идентифицирует все доступные устройства в пределах пространства шины PCI и распределяет системные ресурсы для них. Список регистров конфигурации показан в таблице 3.44.

Таблица 3.44 — Регистры конфигурации

Адрес смещения PCI	Регистр
00h—08h	Регистр обрабатывается, как определено в спецификации для регистра устройства PCI
09h—0Bh	Регистр кода класса
0Ch—1Fh	Регистр обрабатывается, как определено в спецификации для регистра устройства PCI

Продолжение таблицы 3.44

Адрес смещения PCI	Регистр
20h—23h	Регистр адресов
24h—5Fh	Регистр обрабатывается, как определено в спецификации для регистра устройства PCI
60h	Регистр номера версии последовательной шины
61h—FFh	Регистр обрабатывается, как определено в спецификации для регистра устройства PCI

Основная часть адресов регистров ввода-вывода выбирается в регистр конфигурации шины PCI. Часть регистров позволяет читать текущие состояния портов, но не обязательно записывают значения в порт. Программа должна проверить состояние регистра и определить момент, когда порт освободится, и можно будет передавать данные. Это сделано для того, чтобы не нарушить функционирование шины PCI и подключенных к ней устройств.

Регистр команд содержит сведения о том, что команда будет выполнена базовым контроллером узла. Запись в регистр значения инициирует выполнение команды. Регистр поддерживает дополнительные биты для выполнения, остановки команды, а также для отладки. Регистр доступен для чтения и записи. Значение по умолчанию равно 0000h . Формат регистра представлен в табл. 3.45.

Таблица 3.45 — Формат регистра команд

Бит	Описание	Назначение
15–8	Резерв	Не используются
7	Размер пакета данных	Определяет максимальный размер пакета данных в байтах (0 — 32 байта, 1 — 64 байта)
6	Флаг конфигурации	Задаёт флаг конфигурации. Данный флаг не влияет на контроллер, а лишь помогает программному обеспечению сообщить о завершении процесса конфигурации, установив бит в 1

Продолжение таблицы 3.45

Бит	Описание	Назначение
5	Режим отладки	Установка бита в 1 включает режим отладки, иначе (значение 0) контроллер переходит в нормальный режим работы
4	Общее управление	Осуществляет общее управление устройствами USB. Установка бита в 1 вынуждает контроллер передать общий сигнал готовности для работы. Программное обеспечение должно по истечении 20 мс установить бит в 0, тем самым сообщив контроллеру, что устройства USB готовы к работе. Контроллер может устанавливать данный бит в различных ситуациях: подключение или отключение устройства, переход в режим ожидания (Suspend).
3	Режим ожидания (Suspend mode)	Управляет общим сигналом режима Suspend mode. Установка бита в 1 включает режим Suspend mode для всех устройств USB. Для выхода из этого режима следует установить бит в 0.
2	Общий сброс	Управляет общим сбросом всех регистров и устройств. Установка бита в 1 инициирует сброс устройств USB и перезагружает логику, включая внутренние регистры разветвителя. Программное обеспечение устанавливает бит в 0 не ранее, чем через 10 мс
1	Сброс контроллера	Управляет сбросом контроллера базового узла. Установка бита в 1 выполняет сброс внутренних счетчиков, таймеров и другой логики контроллера. При этом обмен данными с устройствами USB немедленно прекращается
0	Управление выполнением команды	Установка бита в 1 указывает на выполнение команды. Установка бита в 0 заставляет базовый узел завершить выполнение текущей команды и остановиться

Регистр состояния позволяет отслеживать текущие прерывания и состояния контроллера базового узла. Он доступен для чтения и записи. Запись значения 1 очищает (устанавливает в 0) соответствующий бит. Значение по умолчанию равно 0000h . Формат регистра представлен в таблице 3.46.

Таблица 3.46 — Формат регистра состояния

Бит	Описание	Назначение
15–6	Резерв	Не используются
5	Состояние выполнения команды	Базовый узел записывает сюда 1 после того, как выполнение команды было остановлено (записано значение 0 в бит 0 регистра команд) программным обеспечением или самим контроллером базового узла, например, для отладки
4	Признак ошибки в процессе обработки базовый узлом описателя	Устанавливается базовым узлом в 1, когда возникает неустранимая ошибка. При этом базовый узел сбрасывает бит 0 командного регистра, чтобы прекратить дальнейшую обработку команды. Генерируется аппаратное прерывание
3	Признак системной ошибки	Устанавливается базовым узлом в 1, если происходит ошибка, связанная с системным доступом к базовому контроллеру узла. При этом базовый узел очищает бит 0 командного регистра. Генерируется аппаратное прерывание
2	Признак состояния ожидания	Устанавливается базовым узлом в 1, при получении сигнала Suspend mode от устройства USB. Значение бита корректно только в том случае, если базовый узел находится в режиме ожидания (бит 3 в регистре команд установлен в 1).
1	Признак ошибки прерывания на устройстве USB	Устанавливается базовым узлом в 1, когда выполнение транзакции на устройстве USB приводит в ошибку (например, переполнение счетчика)
0	Состояние прерывания устройства USB	Устанавливается базовым узлом в 1, когда прерывание вызвано завершением транзакции на устройстве USB или обнаружен неполный пакет данных

Регистр управления прерываниями позволяет включать или выключать запросы на прерывания для программного обеспечения. Когда прерывания разрешены, генерируется прерывание для базового узла. Регистр доступен для чтения и записи. Значение по умолчанию равно 0000h . Формат регистра представлен в таблице 3.47.

Таблица 3.47 — Формат регистра управления прерываниями

Бит	Описание	Назначение
15–4	Резерв	Не используются
3	Доступность прерывания для пакета данных	Определяет работу прерывания при передаче короткого пакета данных. Установка бита в 1 включает, а установка в 0 выключает прерывание
2	Доступность завершения прерывания	Определяет выдачу прерывания на завершение передачи (1 — включено, 0 — выключено)
1	Доступность состояния прерывания	Управляет прерыванием для режима Suspend mode (1 — включено, 0 — выключено)
0	Доступность прерывания по времени и контрольной сумме	Управляет прерыванием для ситуаций, когда истекает время выполнения операции, и для результата проверки контрольной суммы пакета данных

Регистр счетчика кадров служит для анализа числа кадров, передаваемых в пакете данных. Обновляется после каждого завершения передачи списка кадров. Если базовый узел остановлен, в этот регистр писать нельзя. Регистр доступен для чтения и записи. В режиме записи следует передавать только 16-разрядное значение, иначе возникнет неопределенная ошибка. Значение по умолчанию равно 0000h. Формат регистра представлен в таблице 3.48.

Таблица 3.48 — Формат регистра счетчика кадров

Бит	Описание	Примечание
15–11	Резерв	Не используются
10–0	Список текущих индексов или номеров кадров. Содержат число кадров в списке	Значение увеличивается примерно через каждую 1 мс

Регистр адресов для списка кадров содержит базовый адрес списка кадров в системной памяти. Регистр загружается перед началом выполнения команды базовым контроллером узла. При этом используются только старшие 20 битов, а оставшиеся 12 устанавливаются в 0 для выравнивания до 2 байтов. Содержание регистра объединено со счетчиком количества кадров для получения возможности последовательной обработки списка кадров. Поддерживаются до 1024 списков. Регистр досту-

пен для чтения и записи. В биты 12–31 следует записать базовый адрес в памяти. Биты 0–11 зарезервированы.

Регистр стартового кадра имеет размер 1 байт и служит для генерации стартового кадра пакета данных. Каждое новое значение, записанное в регистр, позволяет синхронизировать следующий кадр. Таким образом, регистр может применяться для корректировки счетчика кадров относительно системных часов. Регистр доступен для чтения и записи. В биты 0–6 заносится значение для синхронизации стартового кадра, бит 7 зарезервирован.

Регистры управления и статуса — это два однотипных 16-разрядных регистра для 1 и 2 портов базового контроллера узла. Служат для отражения состояния и управления базовым контроллером узла. Регистры доступны для чтения и записи. Запись следует производить только в виде слова. По умолчанию значение регистра равно 0080h.

Регистры недоступны после выключения питания, общего сброса или сброса базового контроллера узла. Формат регистра представлен в таблице 3.49.

Таблица 3.49 — Формат регистра управления и статуса

Бит	Описание	Примечание
15–13	Резерв	Установить в 0 при записи
12	Режим ожидания	Доступен для чтения и записи и определяет состояние режима Suspend mode (1 — работа порта приостановлена, 0 — порт работает)
11–10	Зарезервированы	Должны быть установлены в 0 при записи
9	Сброс порта	Управляет сбросом порта (1 — порт был сброшен, 0 — порт не был сброшен)
8	Тип подключенного устройства (по скорости)	Определяет тип подключенного устройства USB (1 — подключено низкоскоростное устройство, 0 — подключено высокоскоростное устройство). Бит используется только для чтения
7	Зарезервирован	Только для чтения. Всегда возвращает значение 1.
6	Режим ожидания	Определяет состояние порта для режима приостановки (1 — есть, 0 — нет). Запись программой значения 1 в этот бит позволяет получать состояние режима приостановки для порта

Продолжение таблицы 3.49

Бит	Описание	Примечание
5–4	Статус линии	Доступны только для чтения и определяют логический уровень сигнала на линии
3	Контроль изменений порта	Позволяет определить состояние порта (1 — порт был включен, выключен и состояние изменилось, 0 — состояние не изменилось). Бит доступен для чтения и для записи значения сброса текущего состояния
2	Контроль включения порта	Управляет работой порта (1 — порт включен, 0 — порт выключен). Этот бит доступен для чтения и записи
1	Состояние соединения	Определяет состояние соединения. Доступен для чтения и записи значения очистки текущего состояния (1 — есть изменение, 0 — нет изменения).
0	Текущее состояние соединения	Определяет текущее состояние соединения и доступен только для чтения (1 — устройство подключено, 0 — устройство не подключено).

Регистры конфигурации в первую очередь необходимы для связи с пространством адресов шины PCI. Каждое устройство на шине PCI должно принадлежать к определенному классу и иметь свои адреса памяти. К регистрам конфигурации относятся: регистр кода класса, регистр базового адреса и регистр номера версии.

Регистр кода класса доступен только для чтения и предназначен для идентификации кода класса и подкласса устройства, требуемых шиной PCI для выделения системных ресурсов. Регистр 24-разрядный. Значение по умолчанию равно 010180h. Формат регистра представлен в таблице 3.50.

Таблица 3.50 — Формат регистра кода класса

Бит	Описание
23–16	Базовый код класса. Для контроллера последовательной шины равен 0Ch
15–8	Код подкласса. Для контроллера базового узла USB равен 03h
7–0	Программируемый интерфейс. Всегда равен 00h

Регистр базового адреса позволяет устанавливать или получать базовый адрес ввода-вывода для обмена данными с устройством USB. Регистр является 32-разрядным и доступен для чтения и записи. По умолчанию значение регистра равно 00000001h. Формат регистра представлен в таблице 3.51.

Таблица 3.51 — Формат регистра базового адреса

Бит	Описание
31–16	Зарезервированы и должны быть установлены в 0
15–5	Значение базового адреса
4–1	Зарезервированы и должны быть установлены в 0
0	Доступен только для чтения и всегда равен 1

Регистр номера версии является 8-разрядным и определяет номер текущей версии USB, которую поддерживает контроллер базового узла. Регистр доступен только для чтения. Биты 7—0 содержат номер версии для последовательной шины (00h — предварительная версия 1.0, 01h — версия 1.0, 02h — версия 2.0).

Теперь рассмотрим работу с USB-портом в Linux [8].

Разработчик может написать полноценный драйвер для работы со своим USB-устройством, применяя уже имеющийся в системе заголовочный файл `linux/usb.h`. Драйвер, написанный для поддержки USB-устройства, должен решать следующие задачи:

- регистрация и удаление драйвера;
- регистрация и удаление устройства;
- обмен данными с устройством.

Если нет времени писать драйвер, но надо создать программу, работающую с USB-устройством, которая сама будет решать, как, когда и с каким из таких устройств ей работать? Для этого используем библиотеку `libusb` (<http://www.libusb.org/>). Если в дистрибутиве Linux ее нет, то можно самостоятельно установить этот пакет.

Первое действие — получение сведений об установленных USB-устройствах.

Пример 3.37. Чтение информации об устройствах USB в Linux

```
// Пример на базе библиотеки libusb
// (http://libusb.sourceforge.net)
// Версия пакета libusb: libusb-0.1.12
// Используемые при сборке библиотеки:
// libusb.so
// Компилятор примера: g++ (GCC) 4.4.0
#include <stdio.h>
#include "/usr/local/include/usb.h"
#include <sys/types.h>

void                                print_endpoint(struct
usb_endpoint_descriptor
                                *endpoint) {
    // Печать информации о конечной точке
}

void print_altsetting(struct
                                usb_interface_descriptor *interface)
{
    int i;
    // Печать информации об интерфейсе
    for (i = 0; i < interface->bNumEndpoints; i++)
{
    printf("\n Точка #%d\n", i);
    print_endpoint(&interface->endpoint[i]); }
}

void print_interface(struct usb_interface
                                *interface) {
    int i;
    for (i = 0; i < interface->num_altsetting; i++)
print_altsetting(&interface->altsetting[i]);
}

void                                print_configuration(struct
usb_config_descriptor *config)
{
    int i;
```

```

        //Массив интерфейсов, поддерживаемый для данной
        // конфигурации
        for (i = 0; i < config->bNumInterfaces; i++) {
printf("\n                Интерфейс                %#d\n",i);
print_interface(&config->interface[i]); }
        }
        int main(int argc, char «argv[]) {
        // Структура описания USB-шины
        struct usb_bus *bus;
        // Структура описания USB-устройства
        struct usb_device *dev;
        usb_init(); // Инициализация структур, необхо-
// димых
        // для дальнейшей работы
        usb_find_busses(); // Находим все USB-шины в
        // системе. Учитываются все изменения с момента
        // прошлого вызова данной Функции
        usb_find_devices(); // Определяем все устрой-
// ства
        // для всех шин, учитываются все изменения с
        // момента проилого вызова Функции
        for (bus = usb_busses; bus; bus = bus->next) {
            for (dev = bus->devices; dev;
dev = dev->next) {
                int ret, i;
                char string[256];
                usb_dev_handle *udev;
                //Указатель                                USB-устройства
printf("=====\n");
                //Основные параметры текущего устройства
                printf("Шина:  %s, Устройство:  %s, Поставщик:
%04X,    Продукт:    %04X\n",    bus->dirname,    dev-
>filename,
                    dev->descriptor.idVendor,
                    dev->descriptor.idProduct);

```

```

    //Получаем доступ к устройству для работы с ним
    udev = usb_open(dev);
    //Пытаемся извлечь строковые данные о произво-
// дителе
    // устройства
    if (udev) {
        if (dev->descriptor.iManufacturer) {
            ret = usb_get_string_simple(udev,
                dev->descriptor.iManufacturer, string,
                sizeof(string));
            if (ret > 0)
                printf(" Производитель : %s\n", string);
            else
                printf(" Нельзя получить строку производителя\n");
        }
        // Пытаемся извлечь строковые данные об устрой-
// стве
        if (dev->descriptor.iProduct) {
            ret = usb_get_string_simple(udev,
                dev->descriptor.iProduct, string,
                sizeof (string));
            if (ret > 0)
                printf(" Устройство : %s\n", string);
            else
                printf(" Нельзя получить строку устройства\n");
        }
        //Пытаемся извлечь строку серийного номера
        // устройства
        if (dev->descriptor.iSerialNumber) {
            ret = usb_get_string_simple(udev,
                dev->descriptor.iSerialNumber, string,
                sizeof (string));
            if (ret > 0)
                printf(" Серийный номер: %s\n", string);
            else

```

```

printf(" Нельзя получить серийный номер\n"):
}
//Закрываем доступ к текущему устройству
usb.close(udev);
}
//Печатаем детальную информацию о конфигурации
// устройства
if (!dev->config) {
printf(" Нельзя получить описатели \n");
continue; }
for (i = 0;
i < dev->descriptor.bNumConfigurations; i++)
print_configuration(idev->config[i]);
}
}
return 0;
}

```

В данном примере идет простой перебор всех устройств, что содержатся в системе и что имеют отношение к USB. Но это делать необязательно. Для работы с конкретным устройством нас может интересовать определенный код изготовителя продукта (`idVendor`) и код продукта (`idProduct`).

Наборы функций `libusb-win32` (для Windows) и `libusb` (для Linux) совпадают, а платформенные различия указаны в документации. Это и есть основное достоинство библиотеки `libusb` — кроссплатформенная переносимость исходного кода.

3.4 Контрольные вопросы к разделу 3

1. В каком формате поступает в компьютер информация о клавишах, нажатых на клавиатуре?
2. Перечислите задачи, решаемые контроллером клавиатуры.
3. Чем отличаются коды алфавитно-цифровых и функциональных клавиш?
4. Каким образом осуществляется ввод символов с клавиатуры в программах Си?

5. Каким образом осуществляется ввод строк с клавиатуры в программах Си? Как отформатировать ввод с клавиатуры?
6. Как ограничить диапазон вводимых с клавиатуры символов в программах Си?
7. Каким образом осуществляется вывод символов на экран в программах Си?
8. Каким образом осуществляется вывод строк на экран в программах Си? Как отформатировать вывод на экран?
9. На какие элементы разделено дисковое пространство?
10. Как прочитать сведения о диске в программе Си?
11. Функция DeviceIoControl Windows — назначение, особенности задания ее параметров.
12. Дайте определение понятий «каталог» и «файл». Чем каталог отличается от файла?
13. Какие функции для работы с каталогами в программах Си Вы знаете?
14. Какие функции для работы с файлами в программах Си Вы знаете?
15. Особенности чтения из файла и записи в файл в программах Си. Как идентифицировать возможную ошибку ввода–вывода?
16. Перечислите интерфейсы, используемые для подключения внешних устройств к компьютеру.
17. Параллельный порт. Принцип работы. Основные настройки.
18. Как инициализировать параллельный порт? Как проверить его готовность к работе?
19. Как передать данные через параллельный порт?
20. Последовательный порт СОМ. Принцип работы. Основные настройки.
21. Как инициализировать последовательный порт? Как проверить его готовность к работе?
22. Как задать настройки последовательного порта?
23. Как передать данные через параллельный порт?
24. Особенности работы с последовательным портом в Linux. Концепции ввода через последовательный порт.
25. Программный доступ к последовательному порту в Linux.

26. Порт USB. Принцип работы. Функции корневого базового узла USB.

27. Какие элементы входят в аппаратно-программный комплекс USB?

28. Как выполняется обмен данными через USB?

29. Какие типы запросов можно передавать через USB?

30. Какие регистры используются при работе с USB?

31. Как прочитать сведения об устройствах USB?

4 СЛУЖБЫ WINDOWS И ДЕМОНЫ LINUX

Службы Windows (Windows Services) — это системные программы, работающие в фоновом режиме. Запуск службы осуществляется либо при загрузке Windows, либо по требованию пользователя (вручную). Большинство служб, как правило, не имеют графического интерфейса для взаимодействия с пользователем. Службы могут реагировать на определенные события, либо выполнять свои функции по графику, в определенные моменты времени. Службы Windows выполняются под управлением диспетчера управления службами (Service Control Manager, сокращенно SCM).

В виде служб обычно реализуют системы управления базами данных, почтовые серверы, систему управления печатью и т.п. В Windows постоянно работает несколько десятков служб.

Демон Linux (аналог службы Windows) — это фоновый процесс, работающий автономно с минимальным вмешательством пользователя или вообще без него. Примером может служить HTTP-демон веб-сервера Apache, который работает в фоновом режиме, сканируя заданные порты и обслуживая страницы или выполняя определенные сценарии в зависимости от вида запроса.

4.1 Службы Windows [9]

4.1.1 Структура служб

Консольное приложение Windows можно преобразовать в службу, выполнив следующие действия:

- создать новую точку входа `main()`, регистрирующую службу в SCM, и предоставляющую точку входа и имена логических служб;
- преобразовать функцию `main()` в функцию `ServiceMain()`, регистрирующую обработчик команд службы и информирующую SCM о своем состоянии. Имя `ServiceMain()` является заменителем имени логической службы, причем логических служб может быть несколько;

– создать функцию обработчика управляющих команд службы, которая выполняет заданные действия в ответ на команды, поступающие от SCM.

Рассмотрим функции, входящие в состав службы.

Функция `main()` выполняет регистрацию службы в SCM и запускает диспетчер службы, для чего служит функция `StartServiceCtrlDispatcher`, которой передаются имена и точки входа для одной или нескольких логических служб.

Функция `StartServiceCtrlDispatcher` принимает адрес `lpServiceStartTable` массива элементов типа `SERVICE_TABLE_ENTRY`, каждый из которых содержит имя и точку входа для логической службы. Признак конца массива — два последовательных значения `NULL`. Функция возвращает `TRUE`, если регистрация службы прошла успешно. Если служба уже выполняется или возникают проблемы с обновлением записей реестра (`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services`), функция завершается с ошибками, обработка которых может осуществляться обычным путем.

Основной поток процесса службы, вызывающей функцию `StartServiceCtrlDispatcher`, связывает поток с SCM, который регистрирует службу с вызывающим потоком в качестве потока диспетчера службы. SCM не осуществляет возврата в вызывающий поток до тех пор, пока не завершат выполнение все службы. Заметьте, однако, что фактического запуска логических служб в этот момент не происходит; запуск службы требует вызова функции `StartService`.

Рассмотрим типовой шаблон для функции `main()`.

Пример 4.1. Шаблон функции `main` для службы Windows

```
// Глобальные переменные
DWORD dwErrCode;    // код ошибки
SERVICE_STATUS ss; // состояние службы
// описатель статуса службы
SERVICE_STATUS_HANDLE ssHandle;
// имя службы
LPTSTR SomeServiceName = "SomeService";
```

```

// Функция main()
void main(int argc, char* argv[]) {
    // таблица точек входа
    SERVICE_TABLE_ENTRY DispatcherTable[] =
    { { SomeServiceName,
      (LPSERVICE_MAIN_FUNCTION)ServiceMain },
      { NULL, NULL } };
    // SomeServiceName - имя службы
    // ServiceMain - главная функция службы
    /запуск диспетчера
    if(!StartServiceCtrlDispatcher(DispatcherTable))
    {
        printf("StartServiceCtrlDispatcher:          Error
        %ld\n",
            GetLastError());
        getch();
        return;
    }
}

```

Функция `StartServiceCtrlDispatcher()` связывает главный поток службы с SCM. Когда SCM запускает службу, он ждет выполнения ею функции `StartServiceCtrlDispatcher()`. Главный поток службы должен вызвать эту функцию как можно быстрее, и, если функция выполнена успешно, происходит связывание вызывающего ее потока с SCM. Работа функции не завершается, пока не будет остановлена служба. SCM использует это соединение, чтобы посылать службе управляющие запросы.

Если служба запускается как отдельный процесс, она должна вызывать функцию `StartServiceCtrlDispatcher()` немедленно. Всю необходимую инициализацию следует делать позже, в функции `ServiceMain()`. Если в рамках одного процесса запускается несколько служб, главный поток должен вызвать функцию `StartServiceCtrlDispatcher()` не позднее, чем через 30 секунд.

Функция `ServiceMain()` определяется процессом, как точка входа для службы. Аргументы функции `ServiceMain()` следующие:

argc - количество аргументов, argv - указатель на массив, который содержит указатели на строковые аргументы функции.

Пример 4.2. Функция регистрации службы

```
void WINAPI ServiceMain(DWORD argc, LPSTR* argv) {
    //регистрация управляющей функции службы
    ssHandle =
        RegisterServiceCtrlHandler(SomeServiceName,
        ServiceControl);
    if(!ssHandle) // Если возникла ошибка
    {
        printf("Ошибка регистрации ServiceControl\n");
        getch();
        return; }
    }
```

Функция RegisterServiceCtrlHandler() регистрирует функцию, которая будет обрабатывать управляющие запросы к службе (например, останов службы). В случае успешного завершения функция возвращает описатель статуса службы. При неудачном завершении функция возвращает ноль.

Задание состояния службы рассмотрим на примере.

Пример 4.3. Задание состояния службы Windows

```
// структура, определяющая состояние службы:
//(служба выполняется как отдельный процесс)
ss.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
// состояние ожидания запуска службы
SetSomeServiceStatus(SERVICE_START_PENDING,
                     NO_ERROR, 4000);
//инициализация службы SomeService
InitSomeServiceData(argc, argv);
//устанавливаем состояние работающей службы
SetSomeServiceStatus(SERVICE_RUNNING, NO_ERROR, 0);
//основной код программы
```

```
... ..
```

```
return;
```

Разберем последний пример.

Поле `dwServiceType` структуры `ss` установлено как `SERVICE_WIN32_OWN_PROCESS`, поэтому служба будет выполняться как отдельный процесс.

Затем устанавливаем состояние ожидания запуска службы с помощью вспомогательной функции `SetSomeServiceStatus()`, которая изменяет содержимое структуры `ss`, и использует SCM для получения информации о службе.

Далее выполняется инициализация службы с помощью пользовательской функции `InitSomeServiceData()`.

Когда инициализация выполнена, вызывается функция `SetSomeServiceStatus()` с параметром `SERVICE_RUNNING`, чтобы установить состояние работающей службы, после чего выполняется основной код программы.

Функция `ServiceControl()` является управляющей функцией службы и может носить любое имя, определенное приложением.

Пример 4.4. Функция для управления службой Windows

```
void WINAPI ServiceControl(DWORD dwControlCode) {  
    // анализируем код команды и выполняем ее  
    switch(dwControlCode) {  
        // команда остановка службы  
        case SERVICE_CONTROL_STOP:  
            // устанавливаем состояние - ожидания остановка  
            SetSomeServiceStatus(SERVICE_STOP_PENDING,  
                                NO_ERROR, 0);  
            // выполняем останов службы  
            StopSomeService();  
            // устанавливаем состояние - останов службы  
            SetSomeServiceStatus(SERVICE_STOPPED,
```

```

        NO_ERROR, 0);
break;
// определение текущего состояния службы
case SERVICE_CONTROL_INTERROGATE:
    SetSomeServiceStatus(ss.dwCurrentState,
NO_ERROR, 0);
    break;
default:
    SetSomeServiceStatus(ss.dwCurrentState,
NO_ERROR, 0);
    break; }
}

```

В приведенном примере с помощью функции `SetSomeServiceStatus()` мы устанавливаем состояние ожидания остановки службы, после чего можно вызвать определяемую пользователем функцию `StopSomeService()`, которая выполнит все необходимые действия перед остановкой службы. Содержимое этой функции различается для каждой конкретной службы.

Далее с помощью функции `SetSomeServiceStatus()` сообщаем SCM, что служба остановлена.

Когда служба получает команду `SERVICE_CONTROL_INTERROGATE`, она должна немедленно обновить информацию о статусе службы, используемой SCM.

Функция `SetSomeServiceStatus()` изменяет содержимое структуры `ss`, которое использует SCM для получения информации о статусе службы.

Рассмотрим реализацию функции получения статуса службы.

Пример 4.5. Функция для получения статуса службы Windows

```

void SetSomeServiceStatus(DWORD dwCurrentState,
        DWORD dwWin32ExitCode,
        DWORD dwWaitHint) {
    // счетчик шагов длительных операций

```

```

    static DWORD dwCheckPoint = 1;
    // если служба не находится в процессе запуска,
    // её можно остановить
    if (dwCurrentState == SERVICE_START_PENDING)
        ss.dwControlsAccepted = 0;
    else
        ss.dwControlsAccepted = SERVICE_ACCEPT_STOP;
    ss.dwCurrentState = dwCurrentState;
    ss.dwWin32ExitCode = dwWin32ExitCode;
    // задаем время ожидания запуска службы
    ss.dwWaitHint = dwWaitHint;
    // если служба не работает и не остановлена,
    // увеличиваем значение счетчика
    // шагов длительных операций
    if ((dwCurrentState == SERVICE_RUNNING( ||
        (dwCurrentState == SERVICE_STOPPED))
        ss.dwCheckPoint = 0;
    else
        ss.dwCheckPoint = dwCheckPoint++;
    // обновляем информацию о службе
    SetServiceStatus(ssHandle, &ss);
}

```

Значение поля `dwCheckPoint` следует периодически изменять во время длительного запуска или останова. Например, если при запуске службы инициализация выполняется в несколько этапов, то нужно увеличивать значение `dwCheckPoint` после каждого этапа. При этом программа, которая запросила услуги службы, может узнать, какой объем работы уже выполнен службой. Поле `dwCheckPoint` должно быть равно нулю, если служба не находится в состоянии запуска, останова или выполнения длительной операции.

Поле `dwControlsAccepted` определяет управляющий код, который служба может принимать и обрабатывать. По умолчанию все службы могут принимать команду `SERVICE_CONTROL_INTERROGATE`. Если это поле имеет значение `SERVICE_ACCEPT_STOP`, то служба может быть остановлена с помощью команды `SERVICE_CONTROL_STOP`.

Функции `InitSomeServiceData()` и `StopSomeService()` производят инициализацию службы, а также действия, которые необходимо выполнить перед остановом службы. Содержимое этих функций зависит от конкретной службы.

Пример 4.6. Шаблоны функций инициализации службы Windows

```
BOOL InitSomeServiceData() {
    // программный код
    ... ..
    return TRUE;
}

BOOL StopSomeService() {
    // программный код
    ... ..
    return TRUE;
}
```

Функции `ServiceMain()` являются усовершенствованными версиями основной программы (логическими службами), преобразуемыми в службу, причем каждая логическая служба будет активизироваться в ее собственном потоке SCM. В свою очередь, логическая служба может запускать дополнительные потоки, например, рабочие потоки сервера. Часто внутри службы существует только одна логическая служба. В то же время, логические службы на основе соединений (sockets) и именованных каналов могут выполняться в рамках одной и той же службы Windows, что потребует предоставления основных функций обеих служб.

Несмотря на то, что функция `ServiceMain()` является адаптированным вариантом функции `main()` с такими же параметрами, между ними имеется одно незначительное отличие: главная функция службы должна быть объявлена с типом `void`, а не `int`, как в случае функции `main()`.

Для регистрации обработчика управляющих команд службы, который представляет собой функцию, вызываемую SCM для осуществления управления службой, требуется дополнительный программный код.

4.1.2 Регистрация и запуск службы

Обработчик управляющих команд службы, вызываемый SCM, должен обеспечивать управление соответствующей логической службой, которая должна зарегистрировать свой обработчик с помощью функции `RegisterServiceCtrlHandlerEx`. Вызов этой функции помещают в начало функции `ServiceMain()` и впоследствии нигде не повторяют. Сам обработчик вызывается SCM после получения запроса службы.

Синтаксис функции регистрации службы следующий:

```
RegisterServiceCtrlHandlerEx(LPCTSTR lpServiceName,  
                             LPHANDLER_FUNCTION_EX lpHandlerProc,  
                             LPVOID lpContext)
```

Функция принимает следующие формальные параметры:

`lpServiceName` — определяемое пользователем имя службы, которое подставляется в соответствующем поле таблицы диспетчеризации, отведенного для данной логической функции;

`lpHandlerProc` — адрес функции расширенного обработчика;

`lpContext` — определяемые пользователем данные, передаваемые обработчику, позволяющие обработчику различать ассоциированные с ним службы, которых может быть несколько.

В случае возникновения ошибки возвращаемое функцией значение — объект `SEPARARE_STATUS_HANDLE`, равно 0.

4.1.3 Настройка состояния службы

Когда управляющая программа зарегистрирована, необходимо перевести службу в состояние `SERVICE_START_PENDING`, воспользо-

вавшись для этого функцией `SetServiceStatus`. Функция `SetServiceStatus` информирует SCM о текущем состоянии службы. Описания других возможных состояний службы, характеризующихся значениями параметра состояния, отличными от `SERVICE_STATUS_PENDING`, приведены ниже.

Обработчик службы должен устанавливать состояние службы при каждом вызове, даже если ее состояние не менялось.

Любой из потоков службы может в любой момент вызвать функцию `SetServiceStatus`, чтобы сообщить сведения о степени выполнения задачи, а также предоставить информацию об ошибках или иную служебную информацию, причем для периодического обновления состояния многие службы часто выделяют отдельный поток. Длительность промежутка времени между вызовами обновления состояния указывается в одном из полей структуры данных, выступающей в качестве параметра. Если в пределах указанного промежутка времени состояние не обновлялось, то SCM может предположить, что произошла ошибка.

Синтаксис функции настройки службы:

```
BOOL SetServiceStatus(  
    SERVICE_STATUS_HANDLE hServiceStatus,  
    LPSERVICE_STATUS lpServiceStatus)
```

Функция принимает следующие формальные параметры:

`hServiceStatus` — описатель, возвращенный функцией `RegisterCtrlHandlerEx`, которая была вызвана ранее;

`lpServiceStatus` — указатель на структуру `SERVICE_STATUS`, содержащую описание свойств, состояния и возможностей службы.

Структура `SERVICE_STATUS` объявлена следующим образом:

```
typedef struct _SERVICE_STATUS {  
    DWORD dwServiceType;  
    DWORD dwCurrentState;  
    DWORD dwControlsAccepted;  
    DWORD dwWin32ExitCode;  
    DWORD dwServiceSpecificExitCode;  
    DWORD dwCheckPoint;  
    DWORD dwWaitHint;
```

```
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

Поля структуры имеют следующее назначение:

`dwWin32ExitCode` — код завершения потока, используемый логической службой. Служба должна установить этот код равным `NO_ERROR` в процессе выполнения и при нормальном завершении;

`dwServiceSpecificExitCode` — может использоваться в качестве кода завершения, если ошибка возникает при запуске или остановке службы. Это значение игнорируется, если значение параметра `dwWin32ExitCode` не было установлено равным `ERROR_SERVICE_SPECIFIC_ERROR`;

`dwCheckPoint` — служба должна периодически увеличивать значение этого параметра для индикации выполнения на всех стадиях, включая стадии инициализации и останова. Этот параметр не действует и должен устанавливаться равным 0, если служба не находится в состоянии запуска, останова, паузы и не выполняет никаких длительных операций;

`dwWaitHint` — ожидаемая длительность интервалов времени (в миллисекундах) между последовательными вызовами функции `SetServiceStatus`, осуществляемыми с увеличенным значением параметра `dwCheckPoint` или измененным значением параметра `dwCurrentState`. Как отмечалось ранее, если на протяжении этого промежутка времени вызова функции `SetServiceStatus` не происходит, то SCM предполагает, что это вызвано возникновением ошибки;

`dwServiceType` указывает тип службы, и должен иметь одно из значений, описанных в таблице 4.1;

`dwCurrentState` указывает на текущее состояние службы. Возможные значения этого параметра перечислены в таблице 4.2;

`dwControlsAccepted` определяет управляющие коды, которые служба будет воспринимать и обрабатывать с помощью своего обработчика (см. следующий раздел). В таблице 4.3 указаны четыре возможных значения, которые могут объединяться посредством операции поразрядного "ИЛИ".

Таблица 4.1 — Типы служб

Значение	Описание
SERVICE_WIN32_OWN_PROCESS	Указывает на службу Windows, выполняющуюся в собственном процессе с собственными ресурсами.
SERVICE_WIN32_SHARE_PROCESS	Указывает на службу Windows, разделяющую процесс с другими службами, в результате чего несколько служб могут совместно использовать одни и те же ресурсы, переменные окружения и так далее.
SERVICE_KERNEL_DRIVER	Указывает на драйвер устройства Windows.
SERVICE_FILE_SYSTEM_DRIVER	Определяет драйвер файловой системы Windows.
SERVICE_INTERACTIVE_PROCESS	Указывает на процесс службы Windows, который может взаимодействовать с пользователем через рабочий стол.

Таблица 4.2 — Значения параметра состояния службы

Значение	Описание
SERVICE_STOPPED	Служба не выполняется.
SERVICE_START_PENDING	Служба находится на стадии запуска, но пока не готова отвечать на запросы. Например, могут еще не быть запущены рабочие потоки.
SERVICE_STOP_PENDING	Служба находится на стадии останова, но еще не завершила своего выполнения. Например, мог быть установлен глобальный флаг завершения, но рабочие потоки еще не успели на это отреагировать.

Продолжение таблицы 4.2

Значение	Описание
SERVICE_RUNNING	Служба выполняется.
SERVICE_CONTINUE_PENDING	Служба переходит в состояние выполнения после нахождения в состоянии паузы.
SERVICE_PAUSE_PENDING	Служба переходит в состояние паузы, но ее безопасное нахождение в этом состоянии пока не обеспечено.
SERVICE_PAUSED	Служба находится в состоянии паузы.

Таблица 4.3 — Некоторые коды, воспринимаемые службой

Значение	Описание
SERVICE_ACCEPT_STOP	Разрешает посылку команды останова
SERVICE_ACCEPT_PAUSE_CONTINUE	Разрешает посылку команд паузы и продолжения работы
SERVICE_ACCEPT_SHUTDOWN	Уведомляет службу о прекращении работы системы. Это дает системе возможность послать службе команду выключения.
SERVICE_ACCEPT_PARAMCHANGE	Требуется системам NT5. Обеспечивает изменение параметров запуска без выполнения самого перезапуска.

4.1.4 Программный код службы

После того как обработчик зарегистрирован и служба находится в состоянии SERVICE_START_PENDING, она может инициализировать себя и вновь установить свое состояние.

Обработчик управляющих команд службы, определяемый с помощью функции RegisterServiceCtrlHandlerEx, имеет следующий прототип:

```
DWORD WINAPI HandlerEx(DWORD dwControl,  
    DWORD dwEventType, LPVOID lpEventData,  
    LPVOID lpContext),
```

где `dwControl` — управляющая команда, поступившая в обработчик от SCM (см. табл. 4.4);

`dwEventType` — определяет дополнительную информацию, которая требуется соответствующим событиям (обычно принимает значение 0);

`lpContext` — пользовательские данные, передаваемые в функцию RegisterServiceCtrlHandlerEx во время регистрации обработчика.

Таблица 4.4 — Константы для параметра `dwControl`

Константа	Назначение
SERVICE_CONTROL_CONTINUE	Временно остановленная служба должна возобновить работу.
SERVICE_CONTROL_INTERROGATE	Служба должна сообщить информацию о своем текущем состоянии диспетчеру управления службами (SCM).
SERVICE_CONTROL_NETBINDADD	Уведомляет сетевую службу, что есть новый компонент для соединения и служба должна соединиться с ним. Приложение вместо него должно использовать функциональные возможности Plug and Play.

Продолжение таблицы 4.4

Константа	Назначение
SERVICE_CONTROL_NETBINDDISABLE	Уведомляет сетевую службу, что одна из его связей заблокирована. Служба обновляет информацию о связывании и удаляет эту связь. Приложение вместо него должно использовать функциональные возможности Plug and Play.
SERVICE_CONTROL_NETBINDENABLE	Уведомляет сетевую службу, что отключенное связывание включилось в работу. Служба должна обновить информацию о связывании и добавить новое связывание. Вместо него приложение должно использовать функциональные возможности Plug and Play.
SERVICE_CONTROL_NETBINDREMOVE	Уведомляет сетевую службу, что компонент для связывания был удален. Служба должна обновить информацию связывания и удалить связывание с удаленным компонентом. Вместо этого приложения должны использовать функциональные возможности Plug and Play.

Продолжение таблицы 4.4

SERVICE_CONTROL_PARAMCHANGE	Уведомляет службу, что специальные для нее параметры запуска изменились. Служба должна обновить параметры запуска.
SERVICE_CONTROL_PAUSE	Уведомляет службу, что она должна сделать паузу.
SERVICE_CONTROL_SHUTDOWN	Уведомляет службу о том, что система закрывается таким образом, что служба может выполнить задачи очистки.
SERVICE_CONTROL_STOP	Уведомляет службу о том, что она должна остановиться.

Возвращаемое значение для функции HandlerEx зависит от полученного управляющего кода:

- если служба не обрабатывает управление, функция возвратит значение ERROR_CALL_NOT_IMPLEMENTED;
- если служба обрабатывает SERVICE_CONTROL_DEVICEEVENT, функция возвратит значение NO_ERROR, чтобы разрешить запрос и код ошибки, которая отвергает запрос;
- если служба обрабатывает SERVICE_CONTROL_HARDWAREPROFILECHANGE, функция возвратит значение NO_ERROR, чтобы разрешить запрос и код ошибки, которая отвергает запрос;
- если служба обрабатывает SERVICE_CONTROL_POWEREVENT, функция возвратит значение NO_ERROR, чтобы разрешить запрос и код ошибки, которая отвергает запрос.

Для всех других управляющих кодов, которые обрабатывают службы, возвращаются значения NO_ERROR.

Когда служба запускается, ее функция `ServiceMain` должна немедленно вызвать функцию `RegisterServiceCtrlHandlerEx`, чтобы определить функцию `HandlerEx`, которая обрабатывает запросы на управление. Чтобы определить управляющие коды, которые будут приняты, используйте функции `SetServiceStatus` и `RegisterDeviceNotification`.

Диспетчер управления в пределах главного потока процесса службы вызывает функцию обработки управления для указанной службы всякий раз, когда он получает запрос на управление от диспетчера управления службами. После обработки запроса на управление, обрабатывающая программа управления должна вызвать функцию `SetServiceStatus`, чтобы сообщить о ее текущем состоянии диспетчеру управления службами.

Когда диспетчер управления службами отправляет управляющий код службе, он ждет обработчика функции, чтобы вернуть значение перед отправкой дополнительных управляющих кодов другим службам. Если обработчик функции не возвращает значение быстро, диспетчер может заблокировать другие службы, чтобы они не получили управляющих кодов.

Управляющий код `SERVICE_CONTROL_SHUTDOWN` должен обрабатываться только службами, которые должны очищаться в ходе закрытия, потому что имеется только ограниченное время (около 20 секунд) доступное для закрытия службы. После того, как это время истекает, система возобновляет процесс закрытия независимо от того, завершается ли закрытие службы полностью.

Внимание! Если систему оставить в состоянии закрытия (не перезапустить или не выключить питание), служба продолжит запускаться.

Если служба требует большего времени для очистки памяти, она должна отправить сообщения о состоянии `STOP_PENDING`, наряду с указанием ожидания. Таким образом, мы оповещаем диспетчер службы о том, как долго ждать перед оповещением в системе, что закрытие службы завершено полностью. Чтобы воспрепятствовать службе останавливать закрытие, задайте лимит времени диспетчеру службы. Чтобы

изменять этот интервал, модифицируйте значение `WaitToKillServiceTimeout` в следующем ключе реестра:

`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control`

Убедитесь, что обработка событий устройства "Plug and Play" происходит насколько можно быстрее, иначе система может стать невосприимчивой. Если Ваш обработчик событий должен выполнить операцию, которая может блокировать выполнение (типа операции ввода-вывода), лучше запустить другой поток, чтобы выполнить нужную операцию асинхронно.

Обработчик активизируется SCM в том же потоке, что и основная программа, и обычно содержит ряд операторов `switch`.

4.2 Демоны Linux [10]

4.2.1 Базовые положения

После запуска демон выполняет подготовительные действия в следующей последовательности:

- отделение от родительского процесса — создание подпроцесса;
- изменение файловой маски;
- открытие журналов на запись;
- создание уникального идентификатора сеанса (sid);
- изменение текущего рабочего каталога на безопасный;
- закрытие стандартных файловых дескрипторов;
- переход к коду собственно демона.

Демон запускается самой системой, пользователем в терминале или через сценарий. Во время запуска его процесс ничем не отличается от любого другого процесса в системе. Чтобы сделать его по-настоящему автономным, нужно создать дочерний процесс, в котором будет выполняться код демона, для чего используется функция `fork()`.

Функция `fork()` возвращает либо идентификатор дочернего процесса `PID`, не равный нулю, либо `-1` в случае ошибки. Если получение `PID` от `fork()` прошло успешно, то родительский процесс должен завершиться.

Пример 4.7. Получение идентификатора процесса Linux и отделение от родительского процесса

```
pid_t pid;
/* отделение от родительского процесса */
pid = fork();
/* Если идентификатор процесса не получен,
   то выход */
if (pid < 0) exit(EXIT_FAILURE);
/* Если идентификатор процесса получен, то
   родительский процесс можно завершать. */
else      exit(EXIT_SUCCESS);
```

Тип `pid_t` — это целый тип, размерность которого зависит от конкретной системы. Значениями этого типа можно оперировать как обычными целыми числами типа `int`.

Чтобы иметь возможность вести запись в любые файлы (включая журналы), созданные демоном, файловая маска должна быть изменена таким образом, чтобы они могли быть, как записаны, так и прочитаны. Делается это с помощью функции `umask()`, как показано в следующем примере.

Пример 4.8. Изменение файловой маски

```
/* отделяемся от родительского процесса —
   см. Пример 4.7 */
umask(0); // Изменяем файловую маску
```

После установки `umask` в 0 мы получили полный доступ к файлам, созданным демоном. Даже если Вы не планируете использовать какие-либо файлы вообще, это следует сделать.

Далее можно открыть где-нибудь в системе файл журнала на запись (но не обязательно). Это действие выполняют для контроля над работой демона.

Затем созданный нами дочерний процесс должен получить уникальный идентификатор сеанса `SID` от ядра операционной системы. Делается это с помощью функции `setsid()`, как показано ниже.

Пример 4.9. Получение идентификатора сеанса

```
pid_t pid, sid;
/* отделяемся от родительского процесса —
```

```

см. Пример 4.7*/
/* Изменяем файловую маску — см. Пример 4.8*/
umask(0);
/* Создаем новый SID для дочернего процесса */
sid = setsid();
/* Фиксируем возможный сбой */
if (sid < 0) exit(EXIT_FAILURE);

```

Функция `setsid()` возвращает данные того же типа что и `fork()`. Чтобы проверить, что функция создала SID для дочернего процесса, Вы можете использовать аналогичную процедуру проверки на ошибки.

Далее текущий рабочий каталог нужно изменить, указав на некоторое место, гарантированно присутствующее в системе. Поскольку многие дистрибутивы Linux не полностью следуют стандарту иерархии файловой системы Linux, то в системе гарантированно присутствует только корень файловой системы. Сменить каталог можно при помощи функции `chdir()`.

Пример 4.10. Смена каталога для демона

```

pid_t pid, sid;
/* отделяемся от родительского процесса —
см. Пример 4.7*/
/* Изменяем файловую маску — см. Пример 4.8*/
umask(0);
/* Создаем новый SID для дочернего процесса —
см. Пример 4.9*/
/* Изменяем текущий рабочий каталог */
/* Фиксируем возможный сбой */
if ((chdir("/")) < 0) exit(EXIT_FAILURE);

```

Функция `chdir()` при возникновении ошибки возвращает -1, что позволяет проконтролировать смену каталога.

Одним из последних шагов в стартовой настройке демона является закрытие стандартных файловых дескрипторов (таких как `STDIN`, `STDOUT`, `STDERR`). Поскольку демон не может использовать терминал, эти файловые дескрипторы не нужны и только создают угрозу безопасности. Закрывают их при помощи функции `close()`.

Пример 4.11. Закрытие стандартных файловых дескрипторов

```
pid_t pid, sid;  
/* отделяемся от родительского процесса —  
см. Пример 4.7 */  
/* Изменяем файловую маску — см. Пример 4.8 */  
umask(0);  
/* Создаем новый SID для дочернего процесса —  
см. Пример 4.9 */  
/* Изменяем текущий рабочий каталог  
— см. Пример 4.10 */  
/* Закрываем стандартные файловые дескрипторы */  
close(STDIN_FILENO);  
close(STDOUT_FILENO);  
close(STDERR_FILENO);
```

Улучшение переносимости кода достигается использованием определенных для файловых дескрипторов констант.

4.2.2 Разработка программного кода демона

Сначала нужно выполнить *инициализацию*. При инициализации в демоне необходимо применять методы обнаружения и обработки ошибок. На этапе отладки демона рекомендуется писать в собственные журналы как можно больше информации об его работе.

Как правило, основной код демона находится внутри бесконечного цикла. Чаще всего используется цикл `while`, имеющий бесконечное условие завершения с вызовом функции останова `sleep` при необходимости выполнения через фиксированные интервалы. Таким образом, демон выполняет требуемые действия, затем «засыпает» на какое-то время, потом опять выполняет запрограммированные действия и т.д.

Пример 4.12. Основной цикл демона Linux

```
pid_t pid, sid;  
pid_t pid, sid;  
/* отделяемся от родительского процесса —  
см. Пример 4.7 */  
/* Изменяем файловую маску — см. Пример 4.8 */  
umask(0);
```

```

/* Создаем новый SID для дочернего процесса —
см. Пример 4.9 */
/* Изменяем текущий рабочий каталог — см. При-
мер 4.10 */
/* Закрываем стандартные файловые дескрипторы —
см. Пример 4.11 */
/* Инициализация демона */
/* «Бесконечный» цикл */
while (1) {
    /* Выполняем действия ... */
    sleep(30); /* ждем 30 секунд */
}

```

Рассмотрим законченный пример демона, который иллюстрирует все этапы, необходимые для его запуска и дальнейшей работы. Для его выполнения скомпилируйте код при помощи `gcc` и запустите на выполнение из командной строки. Для завершения работы демона выясните его PID и воспользуйтесь командой `kill` в терминале Linux.

В рассматриваемом примере демона также задействованы системные журналы, использование которых рекомендуется фиксации информации о запуске/останове/паузе/завершении демона. Также можно использовать Ваши собственные журналы, вызывая функции `fopen()`, `fwrite()` и `fclose()`.

Пример 4.12. Программный код демона Linux.

```

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <syslog.h>
#include <string.h>
int main(void) {
    /* Идентификаторы процесса и сеанса */
    pid_t pid, sid;

```

```

/* Ответвляемся от родительского процесса */
pid = fork();
if (pid < 0) exit(EXIT_FAILURE);
else exit(EXIT_SUCCESS);
/* Изменяем файловую маску */
umask(0);
/* Создание нового SID для дочернего процесса */
sid = setsid();
if (sid < 0) exit(EXIT_FAILURE);
/* Изменяем текущий рабочий каталог */
if ((chdir("/")) < 0) exit(EXIT_FAILURE);
/* Закрываем стандартные файловые дескрипторы */
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);
/* Инициализация демона */
/* «Большой» цикл */
while (1) {
    /* Выполняем действия ... */
    sleep(30); /* ждем 30 секунд */
}
exit(EXIT_SUCCESS);
}

```

Ниже приведен пример реализации демона, который выполняет каждые 10 минут команду `who` — получение списка подключенных пользователей, и записывает результат в журнал [10].

Пример 4.13. Пример демона Linux, выводящего список подключенных к системе пользователей

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>

```



```

#include <errno.h>
#include <fcntl.h>
#include <syslog.h>
#include <time.h>
int Daemon(void);
char* getTime();
int writeLog(char msg[256]);
char* getCommand(char command[128]);
/*функция возвращает форматированную дату и
время*/
char* getTime() {
    time_t now;
    struct tm *ptr;
    static char tbuf[64];
    bzero(tbuf, 64);
    time(&now);
    ptr = localtime(&now);
    strftime(tbuf, 64, "%Y-%m-%e %H:%M:%S", ptr);
    return tbuf;
}
// функция для возврата результата выполнения
// команды Linux
char* getCommand(char command[128]) {
    FILE *pCom;
    static char comText[256];
    bzero(comText, 256);
    char buf[64];
    pCom = popen(command, "r"); //выполняем
    if(pCom == NULL) {
        writeLog("Error Command");
        return "";
    }
    strcpy(comText, "");
    //читаем результат

```

```

        while(fgets(buf, 64, pCom) != NULL) {
strcat(comText, buf);
        }
        pclose(pCom);
        return comText;
    }
    // функция записи строки в журнал
    int writeLog(char msg[256]) {
        FILE * pLog;
        pLog =
fopen("/home/daemon.log",
        "a");
        if(pLog == NULL) return 1;
        char str[312];
        bzero(str, 312);
        strcpy(str, getTime());
        strcat(str, " =====\n");
        strcat(str, msg);
        strcat(str, "\n");
        fputs(str, pLog);
        fclose(pLog);
        return 0;
    }
    int main(int argc, char* argv[]) {
        writeLog("Daemon Start");
        pid_t parpid, sid;
        parpid = fork(); //создаем дочерний процесс
        if(parpid < 0) {
            exit(1);
        } else if(parpid != 0) {
            exit(0);
        }
        /* Изменяем файловую маску */
        umask(0);
        //получаем уникальный индекс процесса

```

```

sid = setsid();
    if(sid < 0) {
        exit(1);
    }
    if((chdir("/")) < 0) {
// выходим в корень файловой системы
        exit(1);
    }

// закрываем доступ к
// стандартным потокам ввода-вывода
close(STDIN_FILENO); close(STDOUT_FILENO);
close(STDERR_FILENO);
    return Daemon(); }
// «бесконечный» цикл демона
int Daemon(void) {
    char *log;
    while(1) {
        log = getCommand("who");
// если в системе кто-то работает,
// то пишем в журнал
if(strlen(log) > 5) writeLog(log);
        }
sleep(600); //ждем 10 минут }
    return 0; }

```

После создания программного кода программы-демона ее следует скомпилировать, выполнив команду:

```
gcc daemon.c -g -o daemon
```

В этой команде используются следующие ключи компилятора: `-g` включает режим отладки, `-o` указывает имя для выходного файла.

После проверки работы демона остановите его командой `killall daemon` в терминале Linux.

4.3 Контрольные вопросы к разделу 4

1. Что такое служба Windows? Какие задачи выполняет эта программа?
2. Что такое демон Linux? Какие задачи выполняет эта программа?
3. В чем состоит отличие демона Linux от службы Windows?
3. Перечислите функции, которые обязательно должны входить в состав службы Windows.
4. Какие вызовы должна включать главная функция службы Windows?
5. Каким образом осуществляется управление состоянием службы Windows?
6. Как инициализировать службу Windows?
7. Порядок инициализации службы Windows.
8. Какие типы служб Windows Вы знаете?
9. Порядок запуска службы Windows.
10. Порядок останова службы Windows.
11. Какие действия должен выполнять демон Linux в процессе запуска?
12. Какие функции языка Си используются для задания параметров демона Linux?
13. Почему действия демона Linux рекомендуется заносить в файл журнала?
14. Какие действия выполняются в основном цикле демона Linux?

ПЕРЕЧЕНЬ ССЫЛОК

1. Дьяконов В.Ю. Системное программирование : учеб. пособие / В.Ю. Дьяконов, В.А. Китов, И.А. Калинин ; под ред. А.Л. Горелика. — М. : Высшая школа, 1990. — 222 с.; ил.
2. Джордейн Р. Справочник программиста персональных компьютеров типа IBM PC, XT, AT : Пер с англ. — М. : «Финансы и статистика», 1992. — 544 с.
3. Гук М. Аппаратные средства IBM PC : Энциклопедия. — 3-е изд. — СПб. : Питер, 2006. — 1072 с.; ил.
4. Лав Р. Linux. Системное программирование. — СПб. : Питер, 2008. — 416 с.; ил. — (Серия «Бестселлеры O'Reilly»).
5. Керниган Б. Язык программирования Си : пер. с англ. / Б. Керниган, Д. Ритчи ; под ред. Вс.С. Штаркмана. — М. : Финансы и статистика, 1992. — 272 с.
6. Юров В. Assembler : учебник. — СПб. : «Питер», 2000. — 624 с.
7. Несвижский Всеволод. Программирование аппаратных средств в Windows / В. Несвижский. — СПб. : БХВ-Петербург, 2004. — 880 с.; ил.
8. Перов Михаил. Программирование USB в Linux [Электронный ресурс] / Михаил Перов. — Мир ПК. — 2010. — № 02. — Режим доступа: <https://www.osp.ru/pcworld/2010/02/13001015/>.
9. Разработка классических приложений Windows [Электронный ресурс]. — Режим доступа: <https://developer.microsoft.com/ru-ru/windows/desktop/develop>.
10. Шевелев Д. Пишем "ДЕМОНА" своими руками. [Электронный ресурс] / Д.Шевелев. — Режим доступа: <http://www.opennet.ru/base/dev/daemonexample.txt.html>.

ПРИЛОЖЕНИЕ А

БИБЛИОТЕКИ ЯЗЫКА СИ

Стандартная библиотека Си не является частью собственно языка, однако всякая среда, поддерживающая язык Си, обязана предоставить программисту объявления функций, типов и макросов, содержащиеся в этой библиотеке.

Функции, типы и макросы стандартной библиотеки объявлены в стандартных заголовочных файлах [5]:

```
<assert.h> <float.h> <math.h> <stdarg.h>
<stdlib.h> <ctype.h> <limits.h> <stddef.h>
<string.h> <errno.h> <locale.h> <signal.h>
<stdio.h> <time.h>
```

Б.1 Библиотека ввода–вывода **stdio**

Функции ввода-вывода, типы и макросы, определенные в файле `<stdio.h>`, составляют почти треть этой библиотеки.

Функция `fopen` открывает именованный файл и возвращает указатель потока или `NULL`, если попытка открытия оказалась неудачной. Синтаксис функции:

```
FILE *fopen(const char *filename, const char
*mode)
```

где `filename` – имя файла;

`mode` — режим открытия.

Параметр `mode` может принимать следующие значения:

"r" — открытие текстового файла для чтения

"w" — создание текстового файла для записи; старое содержимое, если оно было, стирается;

"a" — открытие или создание текстового файла для записи в конец (дописывания);

"r+" — открытие текстового файла для модифицирования (чтения и записи);

"w+" создание текстового файла для модифицирования; старое содержимое, если оно было, стирается;

"a+" открытие или создание текстового файла для модифицирования, записи в конец.

Режим модифицирования позволяет выполнять чтение и запись в один и тот же файл, перед переходом от чтения к записи и обратно необходимо вызвать функцию `fflush`, или функцию позиционирования в файле. Если включить в параметр режима букву `b` после первой буквы (например, `"rb"` или `"w+b"`), это будет означать, что файл — двоичный. Имя файла должно быть не длиннее `FILENAME_MAX` символов. Одновременно могут оставаться открытыми не более `FOPEN_MAX` файлов.

Функция `freopen` открывает файл в заданном режиме и ассоциирует с ним поток. Она возвращает указатель на поток или `NULL` в случае ошибки. Как правило, `freopen` используется для замены файлов, ассоциированных с потоками `stdin`, `stdout` и `stderr`. Синтаксис функции:

```
FILE *freopen(const char *filename,
              const char *mode, FILE *stream)
```

где `stream` — указатель на поток.

Применительно к потоку вывода функция `fflush` выполняет запись всех буферизованных, но еще не записанных данных; результат применения к потоку ввода не определен.

```
int fflush(FILE *stream)
```

В случае ошибки записи функция возвращает `EOF`, в противном случае — нуль. Вызов `fflush (NULL)` выполняет указанные операции для всех потоков вывода. Синтаксис функции:

```
int fclose(FILE *stream)
```

Функция `fclose` выполняет запись буферизованных, но еще не записанных данных, уничтожает непрочитанные буферизованные входные данные, освобождает все автоматически выделенные буфера, после чего закрывает поток. Возвращает `EOF` в случае ошибки и нуль в противном случае.

Функция `remove` удаляет файл с указанным именем; последующая попытка открыть файл с этим именем приведет к ошибке. Возвращает ненулевое значение в случае неудачной попытки. Синтаксис функции:

```
int remove(const char *filename)
```

Функция `rename` заменяет старое имя файла (`oldname`) на новое (`newname`). Она возвращает ненулевое значение, если попытка изменить имя оказалась неудачной.

Синтаксис функции:

```
int rename(const char *oldname,  
           const char *newname)
```

Функция `tmpfile` создает временный файл с режимом доступа `"wb+"`, который автоматически удаляется при его закрытии или нормальном завершении программы. Эта функция возвращает указатель потока или `NULL`, если не смогла создать файл.

```
FILE *tmpfile(void)
```

Вызов `tmpnam (NULL)` создает строку, не совпадающую ни с одним из имен существующих файлов, и возвращает указатель на внутренний статический массив. Синтаксис функции:

```
char *tmpnam(char s[L_tmpnam])
```

Вызов `tmpnam (s)` помещает строку в `s` и возвращает ее в качестве значения функции; длина `s` должна быть не менее `L_tmpnam` символов. При каждом вызове `tmpnam` генерируется новое имя; при этом гарантируется не более `TMP_MAX` различных имен за один сеанс работы программы. Отметим, что `tmpnam` создает имя, а не сам файл.

```
int setvbuf(FILE *stream, char *buf# int mode,  
            size_t size)
```

Функция `setvbuf` управляет буферизацией потока; ее нужно вызывать до того, как будет выполняться чтение, запись или какая-либо другая операция. Параметр `mode` со значением `_IOFBF` задает полную буферизацию, `_IOLBF` — построчную буферизацию текстового файла, а `_IONBF` отменяет буферизацию вообще. Если параметр `buf` не равен `NULL`, то его значение будет использоваться как указатель на буфер; в противном случае для буфера будет выделена память. Параметр `size` задает размер буфера.

Функция `setbuf` возвращает ненулевое значение в случае ошибки. Синтаксис функции:

```
void setbuf(FILE *stream, char *buf)
```


Если параметр `buf` равен `NULL`, то для потока `stream` буферизация отменяется. В противном случае вызов `setbuf` эквивалентен вызову `(void) setvbuf (stream, buf, _IOFBF, BUFSIZ)`.

Функция `fprintf` преобразует и выводит данные в поток `stream` под управлением строки формата `format`. Синтаксис функции:

```
int fprintf(FILE *stream, const char *format,
            ...)
```

Функция возвращается количество записанных символов или, в случае ошибки, отрицательное число.

Строка формата содержит специальные символы, определяющие вид выводимой информации (см. табл. 3.2).

Функция `sprintf` работает так же, как и `printf`, только вывод выполняет в строку `s`, завершая ее символом `' \0 '`. Строка `s` должна быть достаточно большой, чтобы вмещать результат вывода. Возвращает количество записанных символов без учета `' \0 '`. Синтаксис функции:

```
int sprintf(char *s, const char *format, ...)
```

Функции `vprintf`, `fprintf` и `vsprintf` эквивалентны соответствующим функциям `printf` с той разницей, что переменный список аргументов в них представлен параметром `arg`, инициализированным с помощью макроса `va_start` и, возможно, вызовами `va_arg` (см. описание файла `<stdarg.h>`). Синтаксис функций:

```
int vprintf (const char *format, va list arg)
int fprintf (FILE *stream, const char *format,
            va_list arg)
int vsprintf (char *s, const char *format,
            va_list arg)
```

Функция `fread` считывает из потока `stream` в массив `ptr` не больше `nobj` объектов размера `size`. Она возвращает количество считанных объектов, которое может оказаться меньше запрашиваемого. `size_t`. Синтаксис функции:

```
fread(void *ptr, size_t size, size_t nobj,
      FILE *stream)
```

Для определения успешного завершения или ошибки при чтении следует использовать `feof` и `ferror`.

Функция `fwrite` записывает `nobj` объектов размера `size` из массива `ptr` в поток `stream`; она возвращает число записанных объектов, которое в случае ошибки будет меньше `nobj`. Синтаксис функции:

```
size_t fwrite(const void *ptr, size_t size,
              size_t nobj, FILE *stream)
```

Функция `fseek` устанавливает файловый указатель в заданную позицию в потоке `stream`; последующее чтение или запись будет производиться с этой позиции. Синтаксис функции:

```
int fseek(FILE *stream, long offset,
           int origin)
```

Для двоичного файла позиция устанавливается со смещением `offset` символов относительно точки отсчета, задаваемой параметром `origin`: начала, если `origin` равен `SEEK_SET`, текущей позиции, если `origin` равен `SEEK_CUR`; и конца файла, если `origin` равен `SEEK_END`. Для текстового файла параметр `offset` должен быть нулем или значением, полученным с помощью вызова функции `ftell` (соответственно `origin` должен быть равен `SEEK_SET`). Функция возвращает ненулевое число в случае ошибки.

Функция `ftell` возвращает текущую позицию указателя в потоке `stream`, или `-1L`, если обнаружена ошибка. Синтаксис функции:

```
long ftell(FILE *stream)
int fgetpos(FILE *stream, fpos_t *ptr)
```

Функция `fgetpos` записывает текущую позицию в потоке `stream` в `*ptr` для последующего использования ее в функции `fsetpos`. Тип `fpos_t` позволяет хранить значения такого рода. В случае ошибки функция возвращает ненулевое число.

```
int fsetpos(FILE *stream, const fpos_t *ptr)
```

Функция `fsetpos` устанавливает позицию в потоке `stream`, читая ее из аргумента `*ptr`, куда она была записана ранее с помощью функции `fgetpos`. В случае ошибки функция возвращает ненулевое число.

Многие функции библиотеки в случае обнаружения *ошибки* или достижения конца файла устанавливают индикаторы состояния, которые можно изменять и проверять.

Кроме того, целочисленное выражение `errno` (объявленное в `<errno.h>`) может содержать номер ошибки, который дает дополнительную информацию о последней из случившихся ошибок.

Функция `clearerr` сбрасывает индикаторы конца файла и ошибок потока `stream`:

```
void clearerr(FILE *stream)
```

Функция `feof` возвращает ненулевое значение, если для потока `stream` установлен (включен) индикатор конца файла:

```
int feof(FILE *stream)
```

Функция `ferror` возвращает ненулевое значение, если для потока `stream` установлен (включен) индикатор ошибки.

```
int ferror(FILE *stream)
```

Б.2 Библиотека функций анализа символов `ctype`

В заголовочном файле `<ctype.h>` объявлены прототипы функций, предназначенных для анализа символов. Аргумент каждой из них имеет тип `int` и должен представлять собой либо `EOF`, либо `unsigned char`, приведенный к `int`, возвращаемое значение тоже имеет тип `int`. Функции возвращают ненулевое значение (истина), если аргумент `c` удовлетворяет некоторому условию или принадлежит к указанному классу символов, и нуль (ложь) в противном случае:

`isalnum(c)` — `isalpha (c)` или `isdigit (c)` есть истина

`isalpha(c)` — `isupper (c)` или `islower (c)` есть истина

`iscntrl(c)` — управляющий символ

`isdigit(c)` — десятичная цифра

`isgraph(c)` — отображаемый символ, за исключением пробела

`islower(c)` — буква в нижнем регистре

`isprint(c)` — отображаемый символ, в том числе пробел

`ispunct(c)` — отображаемый символ, за исключением пробела, буквы или цифры

`isspace(c)` — пробел, прогон страницы, конец строки, возврат каретки, табуляция, вертикальная табуляция

`isupper(c)` — буква в верхнем регистре

`isxdigit(c)` — шестнадцатеричная цифра

В семибитовом символьном наборе ASCII отображаемые символы занимают диапазон от `0x20` (' ') до `0x7E` ('-'); управляющие символы — от `0` (NUL) до `0x1F` (US), а также `0x7F` (DEL).

Помимо перечисленных, есть еще две функции, изменяющие регистр букв:

`int tolower(int c)` — переводит `c` в нижний регистр;

`int toupper(int c)` — переводит `c` в верхний регистр.

Б.3 Библиотека `string`

Функции для работы со строками описаны в заголовочном файле `string.h`. Имеются две группы функций для работы со строками, определенных в заголовочном файле `<string.h>` — имена функций первой группы начинаются с префикса `str`, а второй — с префикса `mem`. За исключением `memmove`, результат работы функций не определен, если выполняется копирование объектов, перекрывающихся в памяти. Функции сравнения воспринимают аргументы как массивы элементов типа `unsigned char`.

В таблице Б.1 переменные `s` и `t` принадлежат к типу `char *`, `cs` и `ct` — к типу `const char *`, `n` — к типу `size_t`, а `c` является числом типа `int`, приведенным к типу `char`.

Таблица Б.1 — Функции для работы со строками

Функция	Пояснение
<code>char *strcpy(s,ct)</code>	Копирует строку <code>ct</code> в строку <code>s</code> , включая <code>'\0'</code> ; возвращает <code>s</code>
<code>char *strncpy(s,ct, n)</code>	Копирует не более <code>n</code> символов строки <code>ct</code> в <code>s</code> ; возвращает <code>s</code> . Дополняет результат символами <code>'\0'</code> , если в <code>ct</code> меньше <code>n</code> символов

Продолжение таблицы Б.1

Функция	Пояснение
char *strcat(s,ct)	Присоединяет ct в конец строки s; возвращает s
char *strncat(s,ct,n)	Присоединяет не более n символов строки ct к s, завершая s символом '\0'; возвращает s
int strcmp(cs,ct)	Сравнивает строки cs и ct; возвращает < 0, если cs<ct; 0, если cs==ct; и >0, если cs>ct
int strncmp(cs,ct,n)	Сравнивает не более n символов строк cs и ct; возвращает < 0, если cs<ct; 0, если cs==ct; и > 0, если cs>ct
char *strchr(cs,c)	Возвращает указатель на первое вхождение c в cs или null при отсутствии такового
char *strrchr(cs,c)	Возвращает указатель на последнее вхождение c в cs или null при отсутствии такового
size_t strspn(cs,ct)	Возвращает длину начального участка cs, состоящего из символов, входящих в строку ct
size_t strcspn(cs,ct)	Возвращает длину начального участка cs, состоящего из символов, не входящих в ct
char *strpbrk(cs,ct)	Возвращает указатель на первый символ в строке cs, который совпадает с одним из символов, входящих в строку ct, или null, если такого не оказалось
char *strstr(cs,ct)	Возвращает указатель на первое вхождение строки ct в строку cs или null, если такого не оказалось
char *strerror (n)	Возвращает указатель на строку, соответствующую номеру ошибки n (зависит от реализации)
size_t strlen(cs)	Возвращает длину cs
char *strtok(s,ct)	Ищет в строке s лексемы, ограниченные символами из строки ct; более подробное описание см. ниже

Последовательность вызовов `strtok(s, ct)` разбивает строку `s` на лексемы, ограничителем которых служит любой символ из строки `ct`. При первом вызове в этой последовательности указатель `s` не равен `NULL`. Функция находит в строке `s` первую лексему, состоящую из символов, которые не входят в `ct`; она заканчивает работу тем, что записывает поверх следующего символа `s` символ '\0' и возвращает указатель на лексему. Каждый последующий вызов, в котором указатель `s` равен

NULL, возвращает указатель на следующую такую лексему, которую функция будет искать сразу после конца предыдущей. Функция `strtok` возвращает NULL, если далее не обнаруживает никаких лексем. Параметр `ct` от вызова к вызову может меняться.

Функции семейства `mem.` . . . предназначены для манипулирования объектами как массивами символов; они образуют интерфейс к быстродействующим системным функциям. В приведенной ниже табл. Б.2 параметры `s` и `t` относятся к типу `void *`; `cs` и `ct` — к типу `const void *`; `n` — к типу `size_t`; а параметр `c` представляет собой число типа `int`, приведенное к типу `char`.

Таблица Б.2 – Функции группы `mem`

Функция	Пояснение
<code>void *memcpy (s, ct, n)</code>	Копирует <code>n</code> символов из <code>ct</code> в <code>s</code> , возвращает <code>s</code>
<code>void *memmove (s, ct, n)</code>	Делает то же самое, что и <code>memcpy</code> , но работает корректно также в случае перекрывающихся в памяти объектов
<code>int memcmp(cs/ct/n)</code>	Сравнивает первые <code>n</code> символов <code>cs</code> и <code>ct</code> ; возвращает те же результаты, что и функция <code>strcmp</code>
<code>void *memchr (cs, c, n)</code>	Возвращает указатель на первое вхождение символа <code>c</code> в <code>cs</code> или <code>null</code> , если он отсутствует среди первых <code>n</code> символов
<code>void *memset (s, c, n)</code>	Помещает символ <code>c</code> в первые <code>n</code> позиций массива <code>s</code> и возвращает <code>s</code>

Б.4 Библиотека математических функций `math`

В заголовочном файле `<math.h>` объявляются математические функции и макросы. Макросы `EDOM` и `ERANGE` (определенные в `<errno.h>`) — это отличные от нуля целочисленные константы, используемые для индикации ошибки области определения и ошибки выхода за диапазон; `HUGE_VAL` представляет собой положительное число типа `double`. Ошибка области определения возникает, если аргумент выходит за пределы числовой области, в которой определена функция. При возникновении такой ошибки переменной `errno` присваивается

значение `EDOM`; возвращаемое значение зависит от реализации. Ошибка выхода за диапазон возникает тогда, когда результат функции нельзя представить в виде `double`. В случае переполнения функция возвращает `HUGE_VAL` с правильным знаком, и в `errno` помещается значение `ERANGE`. Если происходит потеря значимости (результат оказывается меньше, чем можно представить данным типом), функция возвращает нуль; получает ли в этом случае переменная `errno` значение `ERANGE` — зависит от реализации.

В таблице Б.3 параметры x и y имеют тип `double`, n — тип `int`, и все функции возвращают значения типа `double`. Углы для тригонометрических функций задаются в радианах.

Таблица Б.3 — Математические функции Си

Функция	Пояснение
<code>sin(x)</code>	Синус x
<code>cos(x)</code>	Косинус x
<code>tan(x)</code>	Тангенс x
<code>asin(x)</code>	Арксинус x в диапазоне $[-\pi/2, \pi/2]$, $x \in [-1, 1]$
<code>acos(x)</code>	Арккосинус x в диапазоне $[0, \pi]$, $x \in [-1, 1]$
<code>atan(x)</code>	Арктангенс x в диапазоне $[-\pi/2, \pi/2]$
<code>atan2(y, x)</code>	Арктангенс y/x в диапазоне $[-\pi, \pi]$
<code>sinh(x)</code>	Гиперболический синус x
<code>cosh(x)</code>	Гиперболический косинус x
<code>tanh(x)</code>	Гиперболический тангенс x
<code>exp(x)</code>	Экспоненциальная функция e^x
<code>log(x)</code>	Натуральный логарифм $\ln(x)$, $x > 0$
<code>log10(x)</code>	Десятичный логарифм $\lg(x)$, $x > 0$
<code>pow(x, y)</code>	Возведение в степень x^y . Возникает ошибка области определения, если $x = 0$ и $y < 0$ или если $x < 0$ и y — не целое
<code>sqrt(x)</code>	Квадратный корень x , $x > 0$
<code>ceil(x)</code>	Наименьшее целое число в формате <code>double</code> , не меньшее x

Продолжение таблицы Б.3

Функция	Пояснение
<code>floor(x)</code>	Наибольшее целое число в формате <code>double</code> , не превосходящее <code>x</code>
<code>fabs(x)</code>	Абсолютное значение $ x $
<code>ldexp(x, n)</code>	Рассчитывает $x \cdot 2^n$
<code>frexp(x, int *exp)</code>	Разбивает <code>x</code> на два сомножителя, первый из которых — нормализованная дробь в интервале $[1/2, 1)$, возвращаемая функцией, а второй — степень двойки; это число помещается в <code>*exp</code> . Если <code>x</code> равен нулю, то обе части результата равны нулю
<code>modf(x, double *ip)</code>	Разбивает <code>x</code> на целую и дробную части, обе с тем же знаком, что и <code>x</code> . Целая часть помещается в <code>*ip</code> , а дробная часть возвращается из функции
<code>fmod(x, y)</code>	Остаток от деления <code>x</code> на <code>y</code> в виде вещественного числа. Знак результата совпадает со знаком <code>x</code> . Если <code>y</code> равен нулю, результат зависит от реализации языка

Б.5 Вспомогательные функции

В заголовочном файле `stdlib.h` объявлены функции, предназначенные для преобразования чисел, распределения памяти и других подобных задач.

Функция `atof` преобразует строку `s` в число типа `double`:

```
double atof(const char *s)
```

Функция `atoi` преобразует строку `s` в число типа `int`:

```
int atoi(const char *s)
```

Функция `atol` преобразует строку `s` в число типа `long`:

```
long atol(const char *s)
```

Функция `strtod` преобразует первые символы строки `s` в число типа `double`, игнорируя пустое пространство в начале:

```
double strtod(const char *s, char **endp)
```

Функция `strtod` помещает указатель на не преобразованную часть в `*endp` (если `endp` не `NULL`). При переполнении она возвращает `HUGE_VAL` с соответствующим знаком; в случае потери значимости (ре-

зультат слишком мал для его представления данным типом) возвращается 0; в обоих случаях переменная `errno` устанавливается равной `ERANGE`.

Функция `strtol` преобразует первые символы строки `s` в число типа `long`, игнорируя пустое пространство в начале.

```
long strtol(const char *s, char **endp, int
base)
```

Функция `strtol` помещает указатель на не преобразованную часть в `*endp` (если `endp` не `NULL`). Если `base` находится в диапазоне от 2 до 36, то преобразование выполняется в предположении, что число в строке записано по основанию `base`. Если `base` равно нулю, то основанием числа считается 8, 10 или 16. Число, начинающееся с цифры 0, предполагается восьмеричным, а с `0x` или `0X` — шестнадцатеричным. Цифры от 10 до `base-1` записываются первыми буквами латинского алфавита в любом регистре. При основании 16 в начале числа разрешается помещать `0x` или `0X`.

В случае переполнения функция `strtol` возвращает `LONG_MAX` или `LONG_MIN` в зависимости от знака результата, а в `errno` помещается код ошибки `ERANGE`.

Функция `strtoul` работает так же, как и `strtol`, с той разницей, что она возвращает результат типа `unsigned long`, а в случае ошибки — `ULONG_MAX`:

```
unsigned long strtoul(const char *s,
char **endp, int base)
```

Функция `rand` возвращает псевдослучайное число в диапазоне от 0 до `RAND_MAX` (значение `RAND_MAX` не меньше 32767):

```
int rand(void)
```

Функция `srand` использует параметр `seed` как инициализирующее значение для новой последовательности псевдослучайных чисел (вначале параметр `seed` равен 1):

```
void srand(unsigned int seed)
```

Функция `calloc` возвращает указатель на место в памяти, отведенное для массива `nobj` объектов, каждый из которых имеет размер

size или NULL, если запрос на память выполнить нельзя. Выделенная область памяти обнуляется. Синтаксис:

```
void *calloc(size_t nobj, size_t size)
```

Функция malloc возвращает указатель на место в памяти для объекта размера size, или NULL, если запрос невыполним. Выделенная область памяти не инициализируется. Синтаксис:

```
void *malloc(size_t size)
```

Функция realloc изменяет размер объекта, на который указывает p, на заданный size:

```
void *realloc(void *p, size_t size)
```

Для участка объекта, длина которого равна меньшему из старого и нового размеров, содержимое не изменяется. Если новый размер больше старого, дополнительное пространство не инициализируется. Функция возвращает указатель на новый участок памяти или NULL, если запрос невыполним (в этом случае *p не изменяется).

```
void free(void *p)
```

Функция free освобождает область памяти, на которую указывает p; она не делает ничего, если p равен NULL. Переменная p должна указывать на область памяти, ранее выделенную одной из функций calloc, malloc или realloc.

Функция abort вызывает аварийное завершение программы:

```
void abort(void)
```

Функция exit вызывает нормальное завершение программы:

```
void exit(int status)
```

При этом функции, зарегистрированные с помощью atexit, выполняются в порядке, обратном регистрации. Записываются и очищаются буферы открытых файлов, закрываются открытые потоки, и управление возвращается в операционную среду. Какие значения аргумента status передавать в среду — зависит от реализации, однако ноль общепринят как сигнал успешного завершения программы. Можно также использовать значения EXIT_SUCCESS (успешное завершение) и EXIT_FAILURE (ошибочное завершение).

Функция `atexit` регистрирует функцию `fCn` для вызова при нормальном завершении программы; возвращает ненулевое значение, если регистрация невыполнима:

```
int atexit(void (*fcn) (void))
```

Функция `system` передает строку `s` в операционную среду для выполнения:

```
int system(const char *s)
```

Если `s` равна `NULL` и при этом командный процессор среды существует, то `system` возвращает ненулевое значение. Если `s` — не `NULL`, то возвращаемое значение зависит от реализации.

Функция `getenv` возвращает строку среды, ассоциированную с `name`, или `NULL`, если такой строки не существует:

```
char *getenv(const char *name)
```

Функция `bsearch` ищет среди `base[0]...base[n-1]` элемент, соответствующий ключу поиска `*key`:

```
void *bsearch(const void *key, const void
              *base, size_t n, size_t size,
              int (*cmp)(const void *keyval,
                         const void *datum))
```

Функция сравнения `cmp` должна возвращать отрицательное число, если ее первый аргумент (ключ поиска) меньше второго (записи в таблице), нуль в случае их равенства и положительное число, если ключ поиска больше. Элементы массива `base` должны быть упорядочены по возрастанию. Функция возвращает указатель на элемент с совпавшим ключом или `NULL`, если ключ не найден.

Функция `qsort` сортирует массив `base[0]...base[n-1]` объектов размера `size` в порядке возрастания:

```
void qsort(void *base, size_t n,
            size_t size (int (*cmp)(const void *,
                                     const void *)))
```

Функция сравнения `cmp` должна иметь те же свойства, что и в описании `bsearch`.

Функция `abs` возвращает абсолютное значение своего аргумента типа `int`:

```
int abs(int n)
```

Функция `labs` возвращает абсолютное значение своего аргумента типа `long`:

```
long labs(long n)
```

Функция `div` вычисляет частное и остаток от деления числителя `num` на знаменатель `denom`:

```
div_t div(int num, int denom)
```

Результаты запоминаются в элементах `quot` и `rem` типа `int` структуры типа `div_t`.

Функция `ldiv` вычисляет частное и остаток от деления `num` на `denom`:

```
ldiv_t ldiv(long numf, long denom)
```

Результаты запоминаются в элементах `quot` и `rem` типа `long` структуры `ldiv_t`.

Б.6 Макрос для диагностики

Описан в заголовочном файле `assert.h`.

Макрос `assert` используется для включения в программу средств диагностики.

```
void assert (int выражение)
```

Если заданное выражение имеет значение 0 во время выполнения оператора `assert {выражение}`, то в поток `stderr` будет выведено сообщение примерно следующего вида:

```
Assertion failed: выражение, file имя_файла,
line nnn
```

После этого будет вызвана функция `abort` для завершения работы. Имя исходного файла и номер строки берутся из макросов препроцессора `FILE` и `LINE`.

Если в момент включения файла `<assert.h>` имя `NDEBUG` является определенным, то макрос `assert` игнорируется.

Б.7 Системно-зависимые константы

В заголовочном файле `<limits.h>` определяются константы, описывающие размеры целочисленных типов. В таблице Б.4 приведены минимально допустимые величины, а в конкретных реализациях возможны значения, большие указанных.

Таблица Б.4 — Размеры целочисленных типов данных

Константа	Значение	Описание
CHAR_BIT	8	Битов в char
SCHAR_MAX	uchar_max или SCHAR_MAX	Максимальное значение char
CHAR_MIN	0 или CHAR_MIN	Минимальное значение char
INT_MAX	+32767	Максимальное значение int
INT_MIN	-32767	Минимальное значение int
LONG_MAX	+2147483647	Максимальное значение long
LONG_MIN	-2147483647	Минимальное значение long
SCHAR_MAX	+127	Максимальное значение signed char
SCHAR_MIN	-127	Минимальное значение signed char
SHRT_MAX	+32767	Максимальное значение short
SHRT_MIN	-32767	Минимальное значение short
UCHAR_MAX	255	Максимальное значение unsigned char
UINT_MAX	65535	Максимальное значение unsigned int
ULONG_MAX	4294967295	Максимальное значение unsigned long
USHRT_MAX	65535	Максимальное значение unsigned short

Имена, приведенные в таблице Б.5, взяты из файла `float.h` и являются константами для использования в вещественной арифметике с плавающей точкой. Имена с префиксом `FLT` относятся к величинам типа `float`, а с префиксом `DBL` — к величинам типа `double`.

Различные реализации библиотеки устанавливают свои значения.

Таблица Б.5 — Размеры вещественных типов данных

Константа	Значение	Описание
FLT_RADIX	2	Основание для экспоненциальной формы представления, например 2,16
FLT_ROUNDS		Режим округления при сложении чисел с плавающей точкой
FLT_DIG	6	Точность (количество десятичных цифр)
FLT_EPSILON	1E-5	Наименьшее число x такое, что $1.0 + x \neq 1.0$
FLT_MANT_DIG		Количество цифр по основанию FLT_RADIX в мантиссе
FLT_MAX	1E+37	Наибольшее число с плавающей точкой
FLT_MAX_EXP		Наибольшее n , — такое, что FLT_RADIX ^{n} - 1 представимо
FLT_MIN	1E-37	Наименьшее нормализованное число с плавающей точкой
FLT_MIN_EXP		Наименьшее n , такое, что 10 ^{n} — нормализованное число
DBL_DIG	10	Количество значащих десятичных цифр
DBL_EPSILON	1E-9	Наименьшее x , такое, что $1.0 + x \neq 1.0$, где x — double
DBL_MANT_DIG		Количество цифр по основанию FLT_RADIX в мантиссе для чисел типа double
DBL_MAX	1E+37	Наибольшее число с плавающей точкой
DBL_MAX_EXP		Максимальное n , такое, что FLT_RADIX ^{n} - 1 представимо в виде числа типа double
DBL_MIN	1E-37	Наименьшее нормализованное число с плавающей точкой
DBL_MIN_EXP		Минимальное n , такое, что 10 ^{n} представимо в виде нормализованного числа типа double

Б.8 Библиотеки даты и времени

В заголовочном файле `time.h` объявляются типы и функции для работы с датой и временем. Некоторые функции работают с местным временем, которое может отличаться от календарного времени, например, в связи с часовыми поясами. Определены арифметические типы `clock_t` и `time_t` для представления времени, а структура `struct tm` содержит компоненты календарного времени.

В таблице Б.6 приведены переменные библиотеки даты и времени.

Таблица Б.6 — Переменные даты и времени

Переменная	Что хранит
<code>int tm_sec</code>	Секунды от начала минуты (0,61)
<code>int tm_min</code>	Минуты от начала часа (0,59)
<code>int tm_hour</code>	Часы от полуночи (0,23)
<code>int tm_mday</code>	Число месяца (1,31)
<code>int tm_mon</code>	Месяцы после января (0,11)
<code>int tm_year</code>	Годы с 1900
<code>int tm_wday</code>	Дни с воскресенья (0,6)
<code>int tm_yday</code>	Дни с 1 января (0,365)
<code>int tm_isdst</code>	Признак летнего времени

Поле `tm_isdst` имеет положительное значение, если активен режим летнего времени, нуль в противном случае и отрицательное значение, если информация о сезоне времени недоступна.

Функция `clock` возвращает время, измеряемое процессором в тактах от начала выполнения программы, или -1, если оно неизвестно:

```
clock_t clock(void)
```

Пересчет этого времени в секунды выполняется по формуле `clock () / CLOCKS_PER_SEC`.

Функция `time` возвращает текущее календарное время или -1, если время не известно:

```
time_t time(time_t *tp)
```

Если указатель `tp` не равен `NULL`, возвращаемое значение записывается также и в `*tp`.

Функция `difftime` возвращает разность `time2-time1`, выраженную в секундах:

```
double difftime (time_t time2, time_t time1)
```

Функция `mktime` преобразует местное время, заданное структурой `*tp`, в календарное и возвращает его в том же виде, что и функция `time`:

```
time_t mktime(struct tm *tp)
```

Компоненты структуры будут иметь значения в указанных выше диапазонах. Функция возвращает календарное время или `-1`, если оно не представимо.

Следующие четыре функции возвращают указатели на статические объекты, каждый из которых может быть модифицирован другими вызовами.

Функция `asctime` преобразует время из структуры `*tp` в строку вида `Sun Jan 3 15:14:13 1988\n\0`. Синтаксис функции:

```
char *asctime(const struct tm *tp)
```

Функция `ctime` преобразует календарное время в местное, что эквивалентно вызову функции `asctime(localtime(tp))`. Синтаксис функции:

```
char *ctime(const time_t *tp)
```

Функция `gmtime` преобразует календарное время в так называемое скоординированное универсальное время (Coordinated Universal Time — UTC):

```
struct tm *gmtime(const time_t *tp)
```

Она возвращает `NULL`, если UTC не известно. Имя этой функции сложилось исторически и означает `Greenwich Mean Time` (среднее гринвичское время).

Функция `localtime` преобразует календарное время `*tp` в местное:

```
struct tm *localtime (const time_t *tp)
```

Функция `strftime` форматирует дату и время из `*tp` и помещает ее в строку `s` согласно строке формата `fmt`, аналогичной той, которая используется в функции `printf`. Синтаксис функции:

```
size_t strftime(char *s, size_t smax,
```



```
const char *fmt# const struct tm *tp)
```

Обычные символы (включая завершающий символ '\0') копируются в *s*. Каждая пара, состоящая из % и буквы, заменяется, как описано ниже, с использованием значений по форме, соответствующей конкретной культурной среде. В строку *s* помещается не более *smax* символов. Функция возвращает количество символов без учета '\0' или нуль, если число сгенерированных символов больше *smax*.

Спецификаторы формата для функции `strftime` приведены в таблице Б.7.

Таблица Б.7 — Спецификаторы формата для функции `strftime`

Спецификатор	Что отображает
%a	Сокращенное название дня недели
%A	Полное название дня недели
%b	Сокращенное название месяца
%B	Полное название месяца
%c	Местное представление даты и времени
%d	День месяца (01-31)
%H	Час (по 24-часовому времени) (00-23)
%I	Час (по 12-часовому времени) (01-12)
%j	День от начала года (001-366)
%m	Месяц (01-12)
%M	Минута (00-59)
%p	Местное представление времени до и после полудня
%s	Секунда (00-61)
%U	Неделя от начала года (считая первым днем недели воскресенье) (от 00 до 53)
%w	День недели (0-6, номер воскресенья равен 0)
%W	Номер недели от начала года (считая первым днем недели понедельник) (от 00 до 53)
%x	Местное представление даты
%X	Местное представление времени
%y	Год без указания века (00-99)
%Y	Год с указанием века
%Z	Название часового пояса, если есть
%%	Символ %

ПРИЛОЖЕНИЕ Б

ОПРЕДЕЛЕНИЕ КОНСТАНТ ДЛЯ ПРОГРАММИРОВАНИЯ USB

```
// Файл usbdefs.h [7]
// подключение файла определений из пакета DDK
extern "C" {
#include "hidsdi.h" }
// базовые коды ввода-вывода
#include <winioctl.h>
// максимальная длина строки описания
#define MAXIMUM_USB_STRING_LENGTH 255
// типы определителей (descriptors)
#define USB_CONFIGURATION_DESCRIPTOR_TYPE 0x02
#define USB_STRING_DESCRIPTOR_TYPE 0x03
#define USB_INTERFACE_DESCRIPTOR_TYPE 0x04
// типы классов устройств USB
#define USB_DEVICE_CLASS_RESERVED 0x00
#define USB_DEVICE_CLASS_AUDIO 0x01
#define USB_DEVICE_CLASS_COMMUNICATIONS 0x02
#define USB_DEVICE_CLASS_HUMAN_INTERFACE 0x03
#define USB_DEVICE_CLASS_MONITOR 0x04
#define USB_DEVICE_CLASS_PHYSICAL_INTERFACE 0x05
#define USB_DEVICE_CLASS_POWER 0x06
#define USB_DEVICE_CLASS_PRINTER 0x07
#define USB_DEVICE_CLASS_STORAGE 0x08
#define USB_DEVICE_CLASS_HUB 0x09
#define USB_DEVICE_CLASS_VENDOR_SPECIFIC 0xFF
// коды ввода-вывода для функции DeviceIoControl
#define FILE_DEVICE_USB FILE_DEVICE_UNKNOWN
#define USB_IOCTL_INDEX 0x00ff
#define IOCTL_USB_GET_NODE_INFORMATION CTL_CODE (
    FILE_DEVICE_USB, \
    USB_IOCTL_INDEX + 3,
    METHOD_BUFFERED, FILE_ANY_ACCESS )
#define IOCTL_USB_GET_ROOT_HUB_NAME CTL_CODE (
```

```

        FILE_DEVICE_USB, \
        USB_IOCTL_INDEX + 3,
        METHOD_BUFFERED, FILE_ANY_ACCESS )
#define IOCTL_USB_GET_NODE_CONNECTION_INFORMATION
CTL_CODE ( FILE_DEVICE_USB, USB_IOCTL_INDEX + 4,
        METHOD_BUFFERED, FILE_ANY_ACCESS)
#define
IOCTL_USB_GET_DESCRIPTOR_FROM_NODE_CONNECTION
CTL_CODE(FILE_DEVICE_USB, USB_IOCTL_INDEX + 5,
        METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_USB_GET_NODE_CONNECTION_NAME CTL_CODE
        (FILE_DEVICE_USB, USB_IOCTL_INDEX + 6,
        METHOD_BUFFERED, FILE_ANY_ACCESS)
#define
IOCTL_USB_GET_NODE_CONNECTION_DRIVERKEY_NAME
CTL_CODE(FILE_DEVICE_USB, USB_IOCTL_INDEX + 9,
        METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_GET_HCD_DRIVERKEY_NAME CTL_CODE (
        FILE_DEVICE_USB, \
        USB_IOCTL_INDEX + 10,
        METHOD_BUFFERED, FILE_ANY_ACCESS)
// структуры данных
#pragma pack ( 1 )
// описатель интерфейса устройства
typedef struct _USB_INTERFACE_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bInterfaceNumber;
    UCHAR bAlternateSetting;
    UCHAR bNumEndpoints;
    UCHAR bInterfaceClass;
    UCHAR bInterfaceSubClass;
    UCHAR bInterfaceProtocol;
    UCHAR iInterface;
} USB_INTERFACE_DESCRIPTOR,

```

```

    *PUSB_INTERFACE_DESCRIPTOR;
// дополнительный описатель интерфейса устройства
typedef struct _USB_INTERFACE_DESCRIPTOR2 {
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bInterfaceNumber;
    UCHAR bAlternateSetting;
    UCHAR bNumEndpoints;
    UCHAR bInterfaceClass;
    UCHAR bInterfaceSubClass;
    UCHAR bInterfaceProtocol;
    UCHAR iInterface;
    USHORT wNumClasses;
} USB_INTERFACE_DESCRIPTOR2,
    *PUSB_INTERFACE_DESCRIPTOR2;
// описание конфигурации
typedef struct _USB_CONFIGURATION_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    USHORT wTotalLength;
    UCHAR bNumInterfaces;
    UCHAR bConfigurationValue;
    UCHAR iConfiguration;
    UCHAR bmAttributes;
    UCHAR MaxPower; } USB_CONFIGURATION_DESCRIPTOR,
    *PUSB_CONFIGURATION_DESCRIPTOR;
// основной описатель устройства
typedef struct _USB_COMMON_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
} USB_COMMON_DESCRIPTOR, *PUSB_COMMON_DESCRIPTOR;
// описатель корневого устройства
typedef struct _USB_ROOT_HUB_NAME {
    ULONG ActualLength;
    WCHAR RootHubName[1];

```

```

} USB_ROOT_HUB_NAME, *PUSB_ROOT_HUB_NAME;
// параметры подключенного к базовому узлу
// устройства
typedef struct _USB_NODE_CONNECTION_NAME {
ULONG ConnectionIndex;
ULONG ActualLength;
WCHAR NodeName[1];
} USB_NODE_CONNECTION_NAME,
  *PUSB_NODE_CONNECTION_NAME;
// описание драйвера устройства
typedef struct _USB_NODE_CONNECTION_DRIVERKEY_NAME
{
ULONG ConnectionIndex;
ULONG ActualLength;
WCHAR DriverKeyName[1];
} USB_NODE_CONNECTION_DRIVERKEY_NAME,
  *PUSB_NODE_CONNECTION_DRIVERKEY_NAME;
// строковый описатель
typedef struct _USB_STRING_DESCRIPTOR {
UCHAR bLength;
UCHAR bDescriptorType;
WCHAR bString[1];
} USB_STRING_DESCRIPTOR, *PUSB_STRING_DESCRIPTOR;
// описание подключенного устройства
typedef struct _STRING_DESCRIPTOR_NODE {
struct _STRING_DESCRIPTOR_NODE *Next;
UCHAR DescriptorIndex;
USHORT LanguageID;
USB_STRING_DESCRIPTOR StringDescriptor[0];
} STRING_DESCRIPTOR_NODE, *PSTRING_DESCRIPTOR_NODE;
// описание базового узла
typedef struct _USB_HUB_DESCRIPTOR {
UCHAR bDescriptorLength;
UCHAR bDescriptorType;
UCHAR bNumberOfPorts;

```

```

USHORT wHubCharacteristics;
UCHAR bPowerOnToPowerGood;
UCHAR bHubControlCurrent;
UCHAR bRemoveAndPowerMask[64];
} USB_HUB_DESCRIPTOR, *PUSB_HUB_DESCRIPTOR;
// дополнительная информация о базовом узле
typedef struct _USB_HUB_INFORMATION {
USB_HUB_DESCRIPTOR HubDescriptor;
BOOLEAN HubIsBusPowered;
} USB_HUB_INFORMATION, *PUSB_HUB_INFORMATION;
typedef struct _USB_MI_PARENT_INFORMATION {
ULONG NumberOfInterfaces;
} USB_MI_PARENT_INFORMATION,
    *PUSB_MI_PARENT_INFORMATION;
// параметры драйвера устройства
typedef struct _USB_HCD_DRIVERKEY_NAME {
ULONG ActualLength;
WCHAR DriverKeyName[1];
} USB_HCD_DRIVERKEY_NAME, *PUSB_HCD_DRIVERKEY_NAME;
// описание базового узла
typedef enum _USB_HUB_NODE {
UsbHub, UsbMIParent } USB_HUB_NODE;
typedef struct _USB_NODE_INFORMATION {
USB_HUB_NODE NodeType;
union {
USB_HUB_INFORMATION HubInformation;
USB_MI_PARENT_INFORMATION MiParentInformation;
} u;
} USB_NODE_INFORMATION, *PUSB_NODE_INFORMATION;
// параметры устройства
typedef struct _USB_DEVICE_DESCRIPTOR {
UCHAR bLength;
UCHAR bDescriptorType;
USHORT bcdUSB;
UCHAR bDeviceClass;

```

```

    UCHAR bDeviceSubClass;
    UCHAR bDeviceProtocol;
    UCHAR bMaxPacketSize0;
    USHORT idVendor;
    USHORT idProduct;
    USHORT bcdDevice;
    UCHAR iManufacturer;
    UCHAR iProduct;
    UCHAR iSerialNumber;
    UCHAR bNumConfigurations;
} USB_DEVICE_DESCRIPTOR, *PUSB_DEVICE_DESCRIPTOR;
// состояние соединения
typedef enum _USB_CONNECTION_STATUS {
    NoDeviceConnected,
    DeviceConnected,
    DeviceFailedEnumeration,
    DeviceGeneralFailure,
    DeviceCausedOvercurrent,
    DeviceNotEnoughPower,
    DeviceNotEnoughBandwidth
} USB_CONNECTION_STATUS, *PUSB_CONNECTION_STATUS;
// дополнительная информация о подключенном
// устройстве
typedef struct _USB_ENDPOINT_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bEndpointAddress;
    UCHAR bmAttributes;
    USHORT wMaxPacketSize;
    UCHAR bInterval;
} USB_ENDPOINT_DESCRIPTOR,
    *PUSB_ENDPOINT_DESCRIPTOR;
typedef struct _USB_PIPE_INFO {
    USB_ENDPOINT_DESCRIPTOR EndpointDescriptor;
    ULONG ScheduleOffset;

```

```

} USB_PIPE_INFO, *PUSB_PIPE_INFO;
typedef struct _USB_NODE_CONNECTION_INFORMATION {
ULONG ConnectionIndex;
USB_DEVICE_DESCRIPTOR DeviceDescriptor;
UCHAR CurrentConfigurationValue;
BOOLEAN LowSpeed;
BOOLEAN DeviceIsHub;
USHORT DeviceAddress;
ULONG NumberOfOpenPipes;
USB_CONNECTION_STATUS ConnectionStatus;
USB_PIPE_INFO PipeList[0];
} USB_NODE_CONNECTION_INFORMATION,
*PUSB_NODE_CONNECTION_INFORMATION;
// текущее состояние устройства
typedef struct _USB_DESCRIPTOR_REQUEST {
ULONG ConnectionIndex;
struct {
UCHAR bmRequest;
UCHAR bRequest;
USHORT wValue;
USHORT wIndex;
USHORT wLength;
} SetupPacket;
UCHAR Data[0];
} USB_DESCRIPTOR_REQUEST, *PUSB_DESCRIPTOR_REQUEST;
// общие параметры устройства
typedef struct {
PCHAR HubName;
PUSB_NODE_INFORMATION HubInfo;
PUSB_NODE_CONNECTION_INFORMATION ConnectionInfo;
PUSB_DESCRIPTOR_REQUEST ConfigDesc;
PSTRING_DESCRIPTOR_NODE StringDescs;
} USBDEVICEINFO, *PUSBDEVICEINFO;
#pragma pack ( )

```


УЧЕБНОЕ ИЗДАНИЕ

Евгений Евгеньевич Бизянов

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

В авторской редакции

Художественное оформление обложки

Н. В. Чернышова

Заказ №. 290. Формат 60x84 ¹/₁₆.

Бумага офс. Печать RISO.

Усл. печат. л. 13,9 Уч.-изд. л. 12.

Издательство не несет ответственность за содержание
материала, предоставленного автором к печати.

Издатель и изготовитель:

ГОУВПО ЛНР «Донбасский государственный технический университет»
пр. Ленина, 16, г. Алчевск, ЛНР, 94204

(ИЗДАТЕЛЬСКО-ПОЛИГРАФИЧЕСКИЙ ЦЕНТР, ауд. 2113, т/факс 2-58-59)

Свидетельство о государственной регистрации издателя, изготовителя
и распространителя средства массовой информации

МИ-СГР ИД 000055 от 05.02.2016