

Катупития Я.,
Бентли К.

Управление электронными устройствами

на C++



Управление электронными устройствами на C++

Янта Катупития, Ким Бентли

Управление электронными устройствами на C++

Разработка практических приложений

Jayantha Katupitiya Kim Bentley

Interfacing with C++

Programming Real-World Applications

Янта Катупития Ким Бентли

Управление электронными устройствами на C++

Разработка практических приложений



Москва, 2016

УДК 621.3:004.438C++
ББК 31.26с
К29

Янта Катупития, Ким Бентли.

К29 Управление электронными устройствами на C++. Разработка практических приложений. / Перевод с англ. Бакомчев И. В. – М.: ДМК Пресс, 2016. – 442 с.

ISBN 978-5-97060-175-4

Книга предназначена всем, кому интересно изучение C++ и управление электронными устройствами на реальных и интересных примерах. Читателю представлена возможность научиться писать программы для выполнения конкретных задач, а не просто скучное изложение материала с картинками. Также рассказывается как создавать программы, взаимодействующие с внешними устройствами посредством специально разработанной интерфейсной платы. Книга, интерфейсная плата и предлагающееся программное обеспечение представляют собой набор простых и несложных для понимания устройств, таких как цифро-аналоговый преобразователь, аналого-цифровой преобразователь, устройство управления коллекторными и шаговыми электродвигателями, измерители температуры и напряжения, таймеры на базе компьютера и простое устройство сбора данных. Также материал книги содержит сведения из области автоматического управления, электроники и механотроники.

Издание будет полезно студентам, инженерам и научным работникам, техникам и радиолюбителям.

УДК 621.3:004.438C++
ББК 31.26с

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-3-540-25378-5 (англ.)
ISBN 978-5-97060-175-4 (рус.)

© Springer-Verlag Berlin Heidelberg
© Перевод, оформление, ДМК Пресс, 2016

ОГЛАВЛЕНИЕ

1	Начало.....	11
1.1.	Введение.....	11
1.2.	Среда разработки программ.....	11
1.2.1.	Редактирование	12
1.2.2.	Компилирование	12
1.2.3.	Редактирование связей	14
1.3.	Программа на C++.....	14
1.3.1.	Комментарии в программах	15
1.3.2.	Заголовочные файлы	16
1.3.3.	Синтаксис программы.....	17
1.3.4.	Служебные слова	17
1.3.5.	Тип возвращаемого значения.....	18
1.3.6.	Тело функции main()	18
1.4.	Функции	19
1.4.1.	Программа с вызовом функции	20
1.5.	Базовые типы данных.....	23
1.6.	Функции с параметрами и возвращаемыми значениями	26
1.7.	Заключение	28
1.8.	Литература	29
2	Базовые сведения о параллельном порте и работа с ним	30
2.1.	Введение.....	30
2.2.	Что такое параллельный порт?	30
2.2.1	Цифровые логические схемы.....	31
2.2.2.	Устройство параллельного порта.....	32
2.3.	Представление данных.....	35
2.4.	Программа для отображения шестнадцатеричных и десятичных чисел	37
2.5.	Заключение	38
2.6.	Литература	38
3	Тестирование параллельного порта	39
3.1.	Введение.....	39
3.2.	Источник питания интерфейсной платы	39
3.3.	Интерфейс параллельного порта	42
3.3.1.	Схема драйвера светодиодов.....	44
3.3.2.	Работа светодиода.....	44
3.4.	Элементарный вывод через параллельный порт	46
3.5.	Ввод через параллельный порт	48
3.6.	Коррекция внутренней инверсии.....	52
3.6.1.	Вывод данных.....	52

3.6.2. Работа в качестве входа.....	55
3.7. Заключение	56
3.8. Литература	57

4 Объектно-ориентированное программирование	58
4.1. Введение	58
4.2. Воображаемые и реальные объекты	58
4.3. Реальные объекты	59
4.3.1. Внешний интерфейс объекта	60
4.3.2. Создание и уничтожение объекта	61
4.4. Классы объектов	61
4.5. Инкапсуляция	62
4.5.1. Инстанцирование объектов	63
4.6. Абстрактные классы	63
4.7. Иерархии классов	64
4.8. Наследование	64
4.9. Множественное наследование	65
4.10. Полиморфизм	66
4.11. Пример иерархии объектов	66
4.12. Преимущества объектно-ориентированного программирования	71
4.13. Недостатки объектно-ориентированного программирования	71
4.14. Заключение	72
4.15. Литература	72

5 Объектно-ориентированное программирование	73
5.1. Введение	73
5.2. Правила обозначения	73
5.3. Разработка класса	74
5.3.1. Данные-члены	75
5.3.2. Функции-члены	75
5.3.3. Атрибуты доступа	75
5.3.4. Определение класса	76
5.3.5. Конструктор	77
5.3.6. Автоматический конструктор	78
5.3.7. Перегрузка конструкторов	78
5.3.8. Деструкторы	78
5.4. Класс ParallelPort – этап 1	79
5.4.1. Определение класса	79
5.5. Программирование с классами	83
5.5.1. Примеры с атрибутами доступа	86
5.6. Класс ParallelPort – этап 2	89
5.7. Класс ParallelPort – этап 3	93
5.7.1. Полнофункциональный класс ParallelPort	93
5.8. Заключение	96

5.9. Литература	97
-----------------------	----

6 Цифро-аналоговое преобразование 98

6.1. Введение.....	98
6.2. Цифро-аналоговое преобразование.....	98
6.2.1. Основные сведения об операционном усилителе.....	99
6.2.2. Принципы работы ЦАП.....	101
6.2.3. Работа DAC0800.....	105
6.2.4. Характеристики и параметры ЦАП.....	108
6.3. Работа с цифро-аналоговым преобразователем.....	109
6.4. Производные классы	112
6.5. Добавление членов к производному классу.....	119
6.5.1. Спецификаторы доступа	122
6.5.2. Полиморфные функции	124
6.6. Заключение	134
6.7. Литература	134

7 Управление светодиодами 135

7.1. Введение.....	135
7.2. Циклы	135
7.2.1. Цикл for	135
7.2.2. Циклы while и do-while	138
7.3. Переходы.....	139
7.3.1. Выражение if.....	139
7.3.2. Выражения break и continue.....	141
7.3.3. Выражение switch-case.....	142
7.4. Массивы	143
7.5. Указатели	146
7.5.1. Объявление указателей	147
7.5.2. Указатели на скалярные величины.....	148
7.5.3. Указатели на объекты классов.....	149
7.5.4. Указатели на массивы.....	150
7.5.5. Массивы указателей.....	152
7.5.6. Арифметические операции над указателями.....	152
7.5.7. Указатели на функции.....	155
7.5.8. Указатели на void	158
7.5.9. Указатель this.....	159
7.6. Работа с указателями.....	160
7.6.1. Массивы чисел для светодиодов	160
7.7. Макросы	168
7.8. Динамическое выделение памяти	169
7.9. Обработка исключений.....	172
7.10. Заключение.....	177
7.11. Литература.....	178

8	Управление шаговыми и коллекторными электродвигателями.....	179
8.1.	Введение.....	179
8.2.	Двигатели постоянного тока.....	179
8.2.1.	Конструкция и характеристики двигателей постоянного тока	180
8.2.2.	Управление коллекторным двигателем.....	181
8.3.	Шаговые двигатели	182
8.3.1.	Конструкции шаговых двигателей.....	183
8.3.2.	Устройство шаговых двигателей.....	183
8.3.3.	Управление шаговым двигателем.....	189
8.3.4.	Характеристики шаговых двигателей.....	190
8.4.	Иерархия классов для двигателей	191
8.5.	Введение в виртуальные функции.....	193
8.5.1.	Чистая виртуальная функция	196
8.5.2.	Формирование сигнала с ШИМ	203
8.6.	Виртуальные функции в приложении	211
8.6.1.	Виртуальные деструкторы.....	216
8.7.	Ввод с клавиатуры	232
8.8.	Заключение	245
8.9.	Литература	245
9	Методы программирования	247
9.1.	Введение.....	247
9.2.	Методы эффективного программирования.....	247
9.3.	Модульные программы.....	255
9.3.1.	Разбиение программы на модули.....	255
9.3.2.	Сборка многофайловой программы	256
9.4.	Пример программы управления двигателями.....	261
9.5.	Заключение	273
9.6.	Литература	273
10	Измерение напряжения и температуры	274
10.1.	Введение	274
10.2.	Преобразование напряжения в последовательность импульсов.....	274
10.3.	Измерение температуры	275
10.4.	Класс ГУН.....	276
10.5.	Измерение напряжения с помощью ГУН.....	280
10.6.	Работа с графикой – отображение прямоугольных импульсов.....	287
10.6.1.	Работа с экраном	288
10.7.	Измерение температуры	292
10.7.1.	Калибровка термистора.....	293
10.8.	Заключение.....	297
10.9.	Литература.....	297

11	Аналого-цифровое преобразование	298
11.1.	Введение	298
11.2.	Аналого-цифровое преобразование	298
11.3.	Методы преобразования	301
11.4.	Измерение напряжения при помощи АЦП	307
11.5.	Класс АЦП	313
11.6.	Измерение напряжения при помощи АЦП	320
11.7.	Измерение температуры при помощи АЦП	323
11.8.	Заключение	326
11.9.	Литература	326

12	Сбор данных с использованием перегрузки операторов	327
12.1.	Введение	327
12.2.	Перегрузка операторов	327
12.2.1.	Передача параметров функции по значению	329
12.2.2.	Передача параметров функции по ссылке	330
12.2.3.	Выбор способа передачи параметров	331
12.2.4.	Конструктор копии	335
12.2.5.	Перегрузка операторов при помощи функций-членов	342
12.2.6.	Перегрузка операторов при помощи обычных функций	344
12.2.7.	Дружественные связи	346
12.2.8.	Потоки ввода/вывода	348
12.2.9.	Транзитные объекты	349
12.2.10.	Оператор присвоения	351
12.3.	Сбор данных	354
12.4.	Заключение	358
12.5.	Литература	358

13	Таймер персонального компьютера	359
13.1.	Введение	359
13.2.	Устройство таймера персонального компьютера	359
13.2.1.	Конфигурирование счётчиков	361
13.2.2.	Регистр управления	362
13.2.3.	Режимы работы таймеров	364
13.2.4.	Чтение данных таймера	366
13.3.	Работа с таймером	367
13.3.1.	Чтение текущего значения таймера 0 и количества тиков	368
13.4.	Класс PCTimer	368
13.5.	Измерение времени	374
13.6.	Измерение скорости реакции человека	376
13.7.	Формирование развёртки во времени	378
13.8.	Сбор данных с метками времени	381

13.8.1. Схема заряда/разряда	381
13.8.2. Сбор данных с метками времени	382
13.9. Заключение	387
13.10. Литература	388

А Приложение. Электронное оборудование 389

Принципиальная схема	389
Печатная плата	389
Сборка	390
Пайка	392
Правила чтения принципиальной схемы	393
Наладка	393
Демонтаж компонентов	394
Соединительные кабели и провода	395
Блок питания	396
Интерфейс параллельного порта	399
Драйвер светодиодов	402
Цифро-аналоговый преобразователь	404
Схема управления двигателями	408
Управляемый напряжением генератор импульсов	412
Аналого-цифровой преобразователь	415
Мультиплексор	418
Управляемый источник тока	421
Повторитель напряжения	423
Схема заряда/разряда	425
Пара светодиод-фотоприёмник	427
Кнопка, потенциометр, диод и стабилитрон	428
Перечень элементов и материалов интерфейсной платы	430

В Приложение 434

Служебные слова C++	434
Приоритет операторов	435
Символы ASCII	436

Предметный указатель 437

1 Начало

Содержание главы:

- Что такое разработка программ?
- Написание и запуск первой программы на C++.
- Синтаксис программы.
- Функции.
- Основные типы данных.

1.1. Введение

В этой главе даны базовые сведения для начала программирования на C++. Будут созданы несколько простых программ на C++, а также освоен синтаксис и правила написания программ. Одной из основных структурных единиц любой программы на C++ является функция. Будут рассмотрены основные понятия и использование функций в C++. В C++ есть встроенные типы данных, на базе которых могут быть созданы пользовательские типы данных. Некоторые из этих типов данных описаны в этой главе.

По ходу главы мы поэтапно рассмотрим весь процесс создания программы: от проектирования небольшой программы до создания исполняемого файла при помощи *среды разработки программ*. При этом не будет задействован объектно-ориентированный подход, поэтому программы будут просты для понимания на начальном уровне. Основы объектно-ориентированного программирования будут даны в главе 4 и далее будут повсеместно использоваться.

1.2. Среда разработки программ

Разработка программы проходит в несколько этапов. Для создания программы понадобятся: *редактор*, *компилятор* и *редактор связей*. В современных средствах разработки программ эти средства интегрированы в один пакет и весь процесс протекает незаметно для пользователя. Такие пакеты называются *интегрированными средами разработки* или *IDE (Integrated Development Environment)*. Большинство современных пакетов C++ (программы для создания программ на C++) представляют собой IDE. В качестве примеров можно привести Turbo C++, Borland C++, C++ Builder и Visual C++, являющиеся коммерческими пакетами. Существуют так называемые версии для *командной строки*. Для запуска редактора в таких версиях необходимо набрать команду (в командной строке DOS). Затем нужно набрать другую команду для запуска компилятора и т. д.

Кроме редактора, компилятора и редактора связей пакеты предоставляют возможность использования библиотек. Иногда эти библиотеки называют библиотеками времени выполнения (*run-time library*, *RTL*). В них содержатся разнообразные функции, которые можно использовать в программах. Важно понимать, что происходит на каждом этапе, независимо от используемого пакета. В следующих разделах будут описаны редактирование, работа препроцессора, компилирование и работа редактора связей.

1.2.1. Редактирование

Первым шагом при создании программы является набор её текста в каком-либо редакторе. Для этого подходит не каждый редактор. В DOS можно пользоваться командой *edit*, а в Windows можно воспользоваться редактором *Notepad*. В интегрированных средах разработки (IDE), входящих в состав пакетов C++, есть встроенные редакторы, так называемые текстовые редакторы. После редактирования нужно записать содержимое редактора в файл. Два упомянутых выше редактора запишут в файл только то, что было набрано с клавиатуры. Они не записывают в файл дополнительные символы (в отличие от некоторых других редакторов). Всё, что мы обычно печатаем, состоит из цифр, букв, знаков пунктуации, пробелов, знаков табуляции, возвратов каретки и переводов строки. Символ перевода строки используется редактором для перемещения курсора на следующую строку. Возврат каретки устанавливает курсор на начало строки. Файл с программой не должен содержать символы, не относящиеся к вышеперечисленным. Файл, в котором находятся все команды программы (текст программы), называется *исходным файлом* (*source file*). Об исходном файле говорят, что он содержит *исходный код* (*source code*), являющийся ничем иным, как командами программы.

1.2.2. Компилирование

Второй этап – это *компилирование* исходного файла. Компилирование осуществляется при помощи специальной программы, которая называется *компилятор*. Сначала выполняется входящая в состав компилятора программа, называемая *препроцессором*. Это происходит перед компилированием исходного кода. Препроцессор обрабатывает в исходном коде все выражения, начинающиеся со знака `"#"`. Далее в листингах программ можно найти строки, начинающиеся с символа `"#"`. Эти выражения называются *директивами препроцессора*. Препроцессор выполняет действия, предписанные этими выражениями, и вносит соответствующие изменения в текст исходного файла. После препроцессора все строки, начинающиеся символом `"#"`, уже обработаны и не обрабатываются компилятором. Этот процесс показан на **Рис. 1.1**. Есть тенденция к объединению препроцессора и компилятора – большинство современных компиляторов имеют встроенные препроцессоры.

Полученный после препроцессора файл затем обрабатывается компилятором, который создаёт так называемый *объектный файл*. Объектный файл содержит объектный код, называемый также *машинным кодом* и «понятный» центральному процессору (ЦП) компьютера. Но всё же компьютер не может выполнять объектный код, так как пока ещё отсутствуют некоторые детали. На этом этапе программа похожа на недостроенное шоссе, где одни участки готовы, а другие ещё нет. Поэтому скомпилированная программа ещё не может выполняться компьютером.

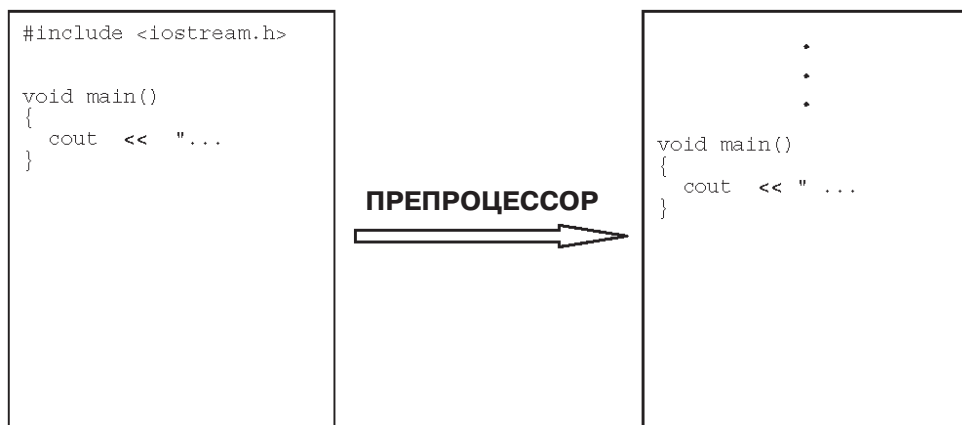


Рис. 1.1. Преппроцессор обрабатывает все строки, начинающиеся символом “#”

На этой незавершённой стадии в объектном коде имеются *неопределённые ссылки*. Неопределённая ссылка указывает на хранящиеся где-то фрагменты объектного кода, которые нужно вставить в программу. В объектном файле нет повсеместно определённого кода, точно так же, как и в случае недостроенного шоссе. Процесс компилирования показан на **Рис. 1.2**.

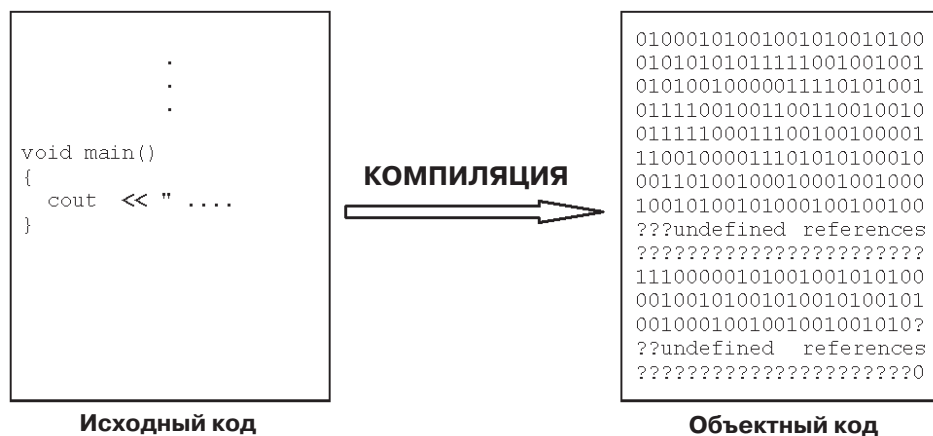


Рис. 1.2. Компилятор преобразовывает исходный код в объектный код

Синтаксис (правила написания) выражений программы крайне важен. Синтаксис определяет правила использования знаков пунктуации в исходном файле. Большинство знаков пунктуации служат *разделителями*. Разделитель отделяет переменные, служебные слова, числа, выражения и т. д. Пробел, запятая, точка с запятой, двоеточие, скобки являются разделителями в различных вариантах использования. Компиляторы обладают ограниченной интеллектуальностью. Если вы пропустите точку с запятой, то это будет обнаружено компилятором и он сообщит об ошибке, хотя сам исправить эту ошибку не сможет.

Выше упоминалось, что объектный код содержит неопределённые области и не может выполняться. Объектный код, например, может содержать вызовы различных подпрограмм. В объектном файле есть вызовы этих функций. Инструкции, на которые ссылаются вызовы, на самом деле отсутствуют. Эти инструкции могут находиться в этом же объектном файле, могут быть в библиотечном файле или другом объектном файле. Заметим, что поиск недостающей информации не является задачей компилятора – в общем случае компилятор можно рассматривать как преобразователь, выполняющий синтаксический анализ содержимого исходного файла.

1.2.3. Редактирование связей

Программа, заполняющая неопределённые участки и осуществляющая сборку программы, называется *редактором связей* (*linker*). Она ищет недостающие инструкции во всех объектных файлах и библиотеках. Иногда редактору связей нужно указать библиотеки и объектные файлы для поиска. Это могут быть как приобретённые библиотеки и объектные файлы сторонних производителей, так и созданные собственными силами. Редактор связей автоматически находит файлы библиотек и объектные файлы, относящиеся к пакету C++, каждый из которых называется библиотекой времени выполнения. Недостающие инструкции редактор связей записывает в соответствующие места, ликвидируя «дыры» в программе. Этот процесс называется редактированием связей. После редактирования связей получается файл, который может быть выполнен компьютером – так называемый *исполняемый файл*. Чтобы программа могла быть выполнена, её нужно загрузить в память компьютера. Это делает загрузчик, являющийся частью исполняемого кода. Большинство редакторов связей помещают загрузчик в начало исполняемого файла. Поэтому когда мы запускаем на выполнение программу, то сначала выполняется загрузчик и загружает программу в память, а затем уже начинается выполнение собственно программы. На **Рис. 1.3** изображён процесс редактирования связей.

1.3. Программа на C++

С точки зрения компьютера программа представляет собой набор инструкций, которые нужно выполнить. Программист упорядочивает эти инструкции различным образом, в зависимости от требуемых от компьютера действий. Простой пример: если вы хотите написать программу сложения двух чисел, числа сначала должны быть введены, а затем выполнено сложение. Значит, сначала должна выполняться команда чтения чисел, а затем выполнено их сложение.

У каждого языка программирования свой уникальный синтаксис. Синтаксис определяет оформление программы и использование знаков препинания. Здесь будет изучаться синтаксис языка программирования C++. Выше упоминалось, что основным структурным элементом программы на C++ можно считать функцию – процедуру, возвращающую результат. Поэтому в программе вместе с набором исполняемых команд обязательно *должна* быть функция. Одна из функций является особой и называется `main()`. Чтобы в тексте различать функции и другие имена, мы будем ставить пару скобок после имени функции. Простые про-

граммы могут иметь только одну функцию `main()`. Наша первая программа будет печатать сообщение на экране. Программа приведена в **Листинге 1.1**.

Листинг 1.1. Программа для вывода сообщения на экран

```
/* Эта программа печатает текстовое сообщение на экране.
   Программа состоит только из одной функции main(). */
#include <iostream.h>

// Функция main()
void main()
{
    cout << "Getting Started" << endl;
}
```

После запуска этой программы на экран будет выведено сообщение:

Getting Started

Смысл текста этой программы будет объяснён далее.

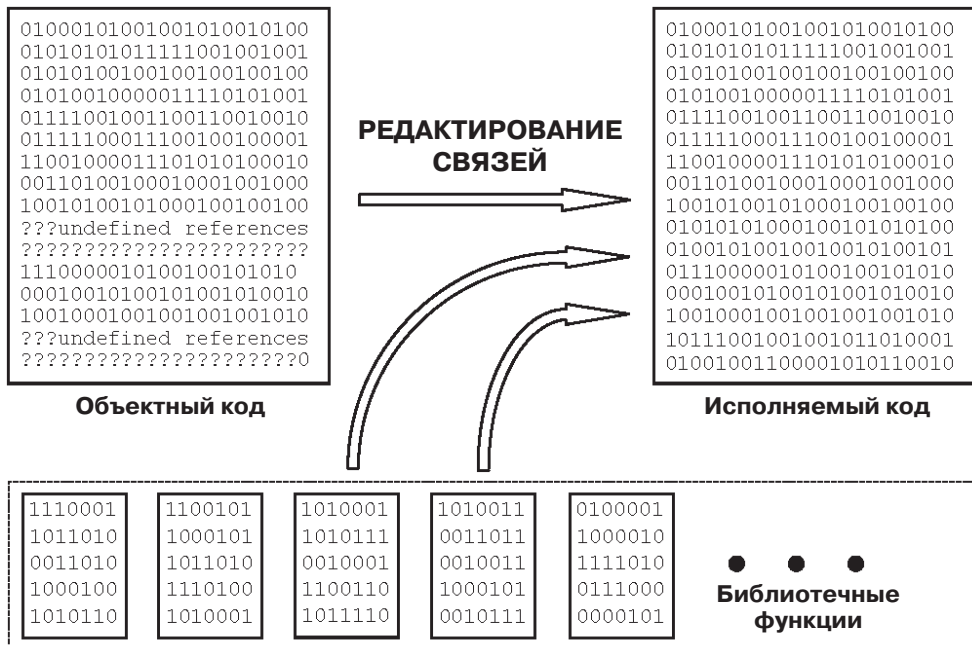


Рис. 1.3. Редактор связей ликвидирует неопределённые участки

1.3.1. Комментарии в программах

Комментарии – это пояснения, включенные в программу, чтобы программист мог описывать её работу. На практике они описывают программу или её отдельные части и не являются выполняемыми компьютером командами программы.

Комментарии должны быть соответствующим образом обозначены в программе, чтобы он не считал их кодом при подготовке исполняемой программы. Есть два разных способа записи комментариев:

- Однострочные и многострочные комментарии могут быть обозначены символами `"/**` в начале комментария и `*/` в конце комментария.
- Однострочный комментарий начинается с символов `"/`.

В **Листинге 1.1** присутствуют однострочный и многострочный комментарии. Многострочный комментарий имеет вид:

```
/* Эта программа печатает текстовое сообщение на экране.  
   Программа состоит только из одной функции main(). */
```

А это однострочный комментарий:

```
// Функция main()
```

Текст между `"/**` и `*/` игнорируется компилятором. То же относится и к тексту после `"/` на той же строке.

1.3.2. Заголовочные файлы

В первой строке после многострочного комментария в **Листинге 1.1** находится директива включения:

```
#include <iostream.h>
```

Она заставляет препроцессор заменить её содержимым файла `iostream.h`. В нашей программе это происходит непосредственно перед функцией `main()`. Файлы с расширением `«.h»` называются *заголовочными файлами* или *включаемыми файлами*. Заголовочный файл может входить в состав пакета C++ или быть написанным программистом самим. Если этот файл из пакета C++, то он размещается в специальном каталоге, называемом *каталогом заголовочных файлов (Include Directory)*, как в случае файла `iostream.h`. В программе может быть несколько директив включения, приводящих к просмотру компилятором нескольких заголовочных файлов.

Заголовочные файлы – это текстовые файлы, содержащие выражения C++, в большинстве своём не являющиеся исполняемыми командами. Не все выражения в рассматриваемой программе являются исполнимыми командами. Тем не менее, выражения из заголовочного файла играют важную роль при формировании программы. Большинство выражений заголовочного файла способствует тщательной проверке компилятором выражений в вашей программе. После написания и тестирования заголовочных файлов они больше не изменяются. Если компилятор сообщает об ошибке или несоответствии, то изменения нужно вносить в программу, а не в заголовочный файл.

Библиотечные функции – это готовые к применению программы. При написании библиотек с функциями нужно создавать для них и заголовочные файлы. В заголовочных файлах устанавливаются правила использования библиотечных функций, строго контролируемые компилятором при написании программы.

В **Листинге 1.1** присутствуют такие элементы, как `cout`, двойной знак «меньше» `<<` и `endl`. В данном случае они не имеют никакого отношения к языку C++. Компилятор не сможет их обработать, пока не будет дано описание этих элементов и правил их использования. В заголовочном файле `iostream.h` содержатся все необходимые указания для компилятора по правильному использованию этих элементов. Эта информация должна быть предоставлена до появления `cout`, `<<` и `endl` в программе. Компилятору не нужно всё содержимое файла `iostream.h` для преобразования программы из **Листинга 1.1** в понятный компьютеру код. В нашем случае нужна только та часть, где описываются `cout`, `endl` и действия оператора `<<`. Но на самом деле очень трудно определить нужные для каждой конкретной программы части заголовочного файла. Поэтому компиляторы вынуждены обрабатывать их полностью. Размер заголовочных файлов никак не влияет на размер исполнимых файлов, только лишь незначительно увеличивает время обработки программы. При необходимости мы можем использовать несколько заголовочных файлов. Также один заголовочный файл может включать другой заголовочный файл.

В заключение отметим, что сначала нужно включить в программу заголовочный файл с объявлениями констант, типов данных и функций и только после этого начинать использовать их в программе.

1.3.3. Синтаксис программы

Синтаксис касается применения знаков пунктуации в программе. В нашей программе использованы символ решетки (`#`), угловые скобки (`<>`), пары круглых скобок (`()`), фигурные скобки (`{}`), точка с запятой (`;`) и двойной знак «меньше» (`<<`). Эти знаки должны быть расставлены в соответствующих местах, чтобы компилятор «понял» программу. Программа в **Листинге 1.1** написана с применением основного синтаксиса. Чем сложнее программа, тем более усложняется её синтаксис.

Все строки, начинающиеся с символа решетки `"#"`, являются директивами препроцессора, обсуждавшегося выше, и имеющего особую роль в пакете разработки программ.

В нашей программе только одна функция — `main()`. Начало *тела функции* `main()` обозначено открывающейся фигурной скобкой `"{"`, а конец — закрывающейся фигурной скобкой `"}"`. Между скобками находятся команды программы. Выполнение программы всегда начинается со строки, содержащей `main()`, а завершается на закрывающейся фигурной скобке тела функции.

Синтаксис функции `main()` можно выразить в компактной форме:

```
void main() {выражение1; выражение2; выражение3;}
```

Это функция с именем `main()`. Пара круглых скобок после имени `main` могут быть как пустыми, так и заполненными — в нашей простой программе они пусты. Как можно было заметить, выражения отделяются друг от друга точкой с запятой. Точка с запятой необходима и в конце последнего выражения, хотя и кажется ненужной.

1.3.4. Служебные слова

Служебные слова являются уже занятыми для нужд языка программирования. Их нельзя использовать в качестве любых элементов программы, а только по

назначению, в соответствии с правилами языка программирования. Служебное слово, например, не может быть *идентификатором*. Идентификаторы – это придуманные нами имена для обозначения таких объектов, как функции, пользовательские типы данных и сами данные. Пока нам встретилось только одно ключевое слово – `void`. Перечень служебных слов приведён в конце книги (Приложение А).

1.3.5. Тип возвращаемого значения

Слово `void` справа от функции `main()` определяет *тип возвращаемого значения* этой функции. Тип возвращаемого значения должен указываться для любой функции, будь то `main()` или какая-либо другая функция. Возвращаемое значение играет роль результата или итога. Если функция должна возвращать значение, то программисту нужно указать тип возвращаемого (выдаваемого) значения. Можно написать функцию, которая ничего не возвращает. Такие функции обычно выполняют какие-либо действия, но не формируют никаких возвращаемых значений. В этом случае тип возвращаемого значения будет `void`. В нашей программе как раз такая функция. Заметим, что если функция не возвращает значение, это должно указываться словом `void`.

Примечание

Если для функции `main()` не указан тип возвращаемого значения, то по умолчанию будет подразумеваться тип `integer`. Это значит, что функция обязана возвращать результат типа `integer`.

1.3.6. Тело функции `main()`

Тело функции `main()` состоит только из одного выражения. Эта строка заключена в фигурные скобки “{” и “}”. Если тело функции `main()` состоит более чем из одного выражения, то все они должны находиться между этими скобками. Это единственное выражение нашей программы выглядит так:

```
cout << "Getting Started" << endl;
```

Слово `cout` заставляет компьютер направить *поток* из того, что следует за этим словом в стандартное устройство вывода, в данном случае на экран. Язык C++ поддерживает работу с потоками. На данном этапе достаточно понимать, что поток – это упорядоченная передача каких-либо элементов (символов, целых чисел и т. д.) куда-либо. Сначала на экране появится `Getting Started`. Затем на экран будет направлено `endl`. В результате этого курсор будет перемещён в начало следующей строки. На этом выполнение программы завершится.

Можно поэкспериментировать, заменив предыдущее выражение на другое:

```
cout << "Getting Started";
```

На экран будет направляться только `Getting Started`, без `endl`. Курсор будет находиться в конце фразы “`Getting Started`”.

1.4. Функции

Выше было сказано, что функции являются важной неотъемлемой частью языка C++. В этом разделе будет рассмотрена работа с функциями. Ранее говорилось, что функция может рассматриваться как процедура, формирующая результат, используя входные данные. Входные данные передаются в функцию при помощи *параметров* или *формальных аргументов*. Формальные аргументы служат для указания типа принимаемых функцией значений при её написании. Когда функция выполняется на компьютере, то формальные аргументы заменяются *фактическими аргументами*. Например, мы можем создать функцию с формальным аргументом *a*. При выполнении функции формальный аргумент нужно заменить фактическим аргументом, например числом 3. Одну и ту же функцию можно **вызывать** (выполнять) много раз, меняя при этом фактические аргументы и получая разные возвращаемые значения. Отметим, что кроме обычного способа данные из функции можно передавать и по-другому.

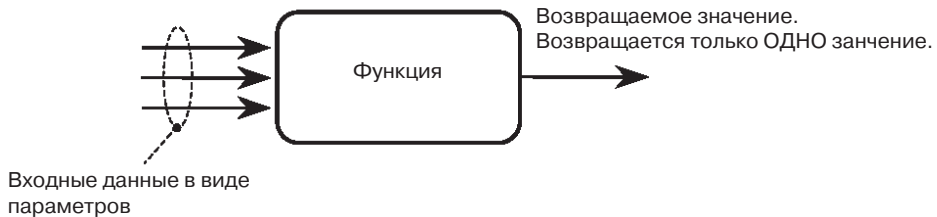


Рис. 1.4. Функция в общем случае

До этого момента функция рассматривалась только в общем виде. Это иллюстрирует **Рис. 1.4**. Существует несколько частных случаев. В этих случаях функция принимает или не принимает аргументы, возвращает или не возвращает результат. Функции могут иметь различное количество аргументов.

Функция всегда возвращает только один результат, и это должна быть *скалярная величина*. Под термином скалярная величина понимается целый и неделимый объект. Другими словами, функция не может возвращать *массивы* (наборы объектов). Функция может возвращать, например, целое число. Она не может вернуть несколько целых чисел. На **Рис. 1.5** и **1.6** показаны некоторые типичные виды функций.

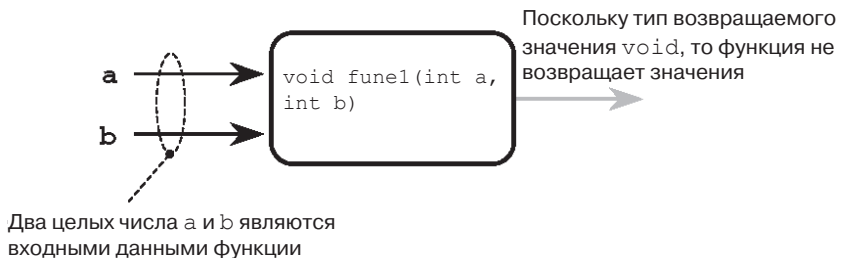


Рис. 1.5. Функция принимает два аргумента и не возвращает результата

Функция, соответствующая **Рис. 1.5**, может, например, выполнять вычисления и выводить результат на экран. Если задачи функции на этом исчерпаны, то нет необходимости возвращать значение.

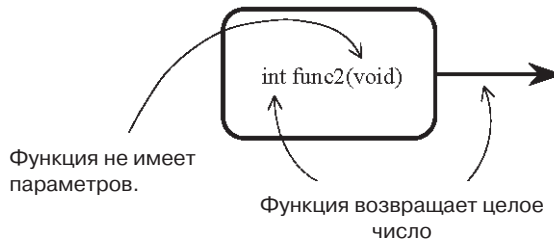


Рис. 1.6. Функция не имеет параметров, но значение возвращает

Согласно **Рис. 1.6** функция может принимать данные от внешнего источника, например, порта принтера, и возвращать целое число. Целое число, например, может указывать на наличие бумаги в принтере: 0 означает отсутствие бумаги, а 1 – её наличие.

1.4.1. Программа с вызовом функции

Программа из **Листинга 1.2** делает то же, что и программа из **Листинга 1.1**. Отличие в использовании функции для вывода данных на экран. Эта функция не принимает значений и не возвращает результат. В этом разделе обсуждается *абстрактность подпрограмм*. Абстрактность подразумевает сокрытие деталей реализации функции внутри её при вызовах для выполнения нужных действий.

Листинг 1.2. Программа с вызовом функции

```
/* Программа печатает текстовое сообщение
   на экране компьютера. Программа состоит
   из двух функций PrintMessage() и main().
```

```
#include <iostream.h>

// Функция PrintMessage()
void PrintMessage()
{
    cout << "Getting Started" << endl;
}

// Функция main()
void main()
{
    PrintMessage(); // вызов функции
}
```

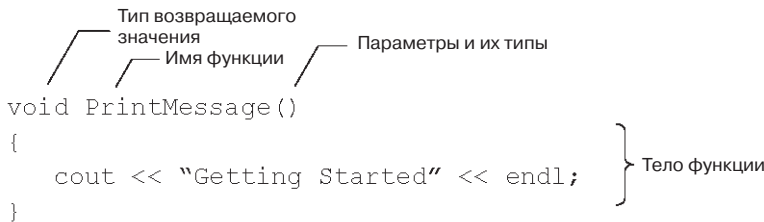
В этой программе есть новая функция `PrintMessage()`. Имя `PrintMessage` мы придумали сами. К имени `PrintMessage` мы добавили пару круглых скобок, чтобы показать, что это функция. Между скобками мы ничего не пишем (что эквивалентно слову `void` между скобок), потому что функция не имеет параметров. Тип возвращаемого значения `void`, так как функция `PrintMessage()` не возвращает результат. Определение функции `PrintMessage()` выглядит следующим образом:

```
void PrintMessage()
{
    cout << "Getting Started" << endl;
}
```

В определении функции должны быть указаны четыре вещи, а именно:

1. Тип возвращаемого значения.
2. Имя функции.
3. Параметры и их типы.
4. Тело функции.

Синтаксис функции изображён на **Рис. 1.7**.



The diagram illustrates the components of the function definition `void PrintMessage()` and its body. Labels with leader lines point to the following parts:

- Тип возвращаемого значения** (Return type): points to `void`.
- Имя функции** (Function name): points to `PrintMessage`.
- Параметры и их типы** (Parameters and their types): points to the empty parentheses `()`.
- Тело функции** (Function body): points to the curly braces `{ ... }` containing the statement `cout << "Getting Started" << endl;`.

Рис. 1.7. Синтаксис определения функции

Определение функции является собственно функцией, компилятору указываются команды, которые нужно выполнять. Другими словами, тело функции `PrintMessage()` начинается с открывающей фигурной скобки, а заканчивается закрывающей фигурной скобкой. Обратите внимание, что после имени функции `PrintMessage()` точка с запятой не ставится. Тем самым следующее далее тело функции ставится в соответствие имени функции.

У функции `PrintMessage()` тип возвращаемого значения `void`. Имя функции `PrintMessage`. Параметров функция не имеет, а тело функции состоит из выражения `cout`.

Объявление (declaration) функции незначительно отличается (как показано на **Рис. 1.8**). Для компилирования вызовов функции компилятору C++ достаточно встретить её объявление до вызова функции. На этом этапе определение функции не нужно. Но для того, чтобы функция могла быть выполнена, нужно откомпилированное определение функции.

В объявлении функции указываются только три элемента:

1. Тип возвращаемого значения.
2. Имя функции.
3. Параметры и их типы.

```
void PrintMessage();
```

Тип возвращаемого значения
Имя функции
Параметры и их типы

Рис. 1.8. Объявление функции, также называемое прототипом функции

Тело функции не является обязательным элементом, но для формирования выполнимой программы оно необходимо. Если тело функции находится в библиотеке, то оно будет позаимствовано оттуда на этапе редактирования связей. Если тела функции нет в библиотеке или другом объектном файле, то его нужно определить где-либо в тексте программы. В нашем случае объявление функции будет выглядеть так:

```
void PrintMessage();
```

Обратите внимание, что в конце строки стоит точка с запятой.

В С++ *прототип* функции – это то же самое, что и её объявление. Но в языке С объявление функции и её прототип – разные вещи. Смотрите пример в разделе 1.6.

Тело функции `main()` немного изменено. Оно состоит из единственного выражения:

```
PrintMessage();
```

Обратите внимание, что в конце стоит точка с запятой, а в начале нет указания типа возвращаемого значения. Такое выражение называется *вызовом функции*. При вызове функции указываются два элемента:

1. Имя функции.
2. Фактические аргументы.

```
PrintMessage();
```

Тип возвращаемого значения не указывается
Имя функции
Формальные аргументы заменяются фактическими аргументами

Рис. 1.9. Пример синтаксиса вызова функции

При выполнении функции формальные аргументы заменяются фактическими аргументами. Обратите внимание на синтаксис, и что в конце стоит точка с запятой. Пример функции с несколькими параметрами приведён в разделе 1.6.

Выполнение программы всегда начинается с функции `main()`. Выполняется тело функции `main()`. Компьютер встречает команду:

```
PrintMessage();
```

Это вызов функции, который приводит к выполнению тела функции `PrintMessage()`. Поэтому на экране будет напечатано `Getting Started`. Как упоминалось в начале этого раздела, функция `PrintMessage()` в теле функции `main()` скрывает свои действия – это и называется абстрактностью подпрограмм (раздел 1.6).

1.5. Базовые типы данных

Большинство команд предназначены для работы с данными. Поэтому данные имеют большое значение в программах. Данные представляются различными *типами данных*, которые иногда комбинируют друг с другом. Очень важно уметь определять принадлежность данных к тому или иному типу данных. В C++ есть несколько встроенных типов данных, называемых *базовыми типами данных*. Типы данных характеризуются тремя признаками:

1. Имя типа.
2. Размер типа в байтах.
3. Диапазон значений, которые могут быть представлены этим типом.

Размер типов данных, а следовательно, и диапазон представляемых данных, зависят от того, пишем 32-битную или 16-битную программу. Речь о битах и байтах пойдёт в следующей главе. Сейчас нам достаточно знать, что 32-битные данные занимают больше памяти и имеют больший диапазон значений.

Типы данных могут быть разделены на три категории:

1. Целый тип.
2. Вещественный тип (с плавающей точкой).
3. Указатели.

Целые типы используются для хранения целых чисел, тогда как в вещественных типах хранятся числа с дробной частью. В указателях хранятся адреса памяти. Что такое адреса памяти, будет рассказано в главе 2, а указатели обсуждаются в главе 7.

Целый тип подразделяется, в свою очередь, на типы со знаком и беззнаковые. Типы со знаком могут хранить положительные и отрицательные числа, тогда как только положительные числа. Вещественным типом представляются и отрицательные, и положительные числа. Указатели всегда имеют положительные значения, так как не существует отрицательных адресов памяти. Программист может создавать свои более сложные типы данных из этих базовых типов. Для начала рассмотрим три базовых типа:

```
char
int
float
```

Первые два типа являются целыми типами, а третий тип вещественный. Три вышеперечисленных типа сведены в **Табл. 1.1** вместе с их размерами в байтах и диапазонами значений.

Таблица 1.1. Некоторые базовые типы данных

Тип данных	Размер в байтах	Диапазон значений
char	1	-128...127
unsigned char	1	0...255
int	2	-32768...32767

Тип данных	Размер в байтах	Диапазон значений
unsigned int	2	0...65535
float	4	$\pm 3.4 \times 10^{-38} \dots \pm 3.4 \times 10^{38}$
double	8	$\pm 1.7 \times 10^{-308} \dots \pm 1.7 \times 10^{308}$

Тип *char*

Этот тип обычно используется для хранения символов. Данные типа *char* занимают один байт памяти. Знаковая форма типа обозначается *char*, а беззнаковая – *unsigned char*. Данные этого типа могут использоваться для хранения небольших целых чисел, которым хватает одного байта памяти.

Тип *int*

Данные этого типа хранят целые числа. Они занимают 2 байта памяти в 16-битных программах. Знаковая форма типа обозначается *int*, а беззнаковая – *unsigned int*. Для 16-битного типа *int* тип *short int* является синонимом.

Тип *float*

Это тип для хранения вещественных чисел. Данные типа *float* занимают 4 байта памяти. Числа могут быть положительными и отрицательными, а беззнаковой формы типа не существует!

Идентификаторы

Идентификаторы – это условные обозначения или имена, служащие для обозначения таких элементов, как целые и вещественные числа, вещественные числа, символы, адреса памяти, функции, объекты, классы и многое другое. Идентификаторы чувствительны к регистру и могут иметь любую длину.

Примечание

Идентификаторы должны начинаться с буквы (строчной или прописной) или знака подчёркивания “ ”. Они могут содержать цифры от 0 до 9, но только не в начале идентификатора.

Перед использованием в программе идентификатор нужно объявить. При объявлении идентификатора нужно указать:

1. Тип данных.
2. Идентификатор.

`int a;`

Рис. 1.10. Пример объявления идентификатора

Вот пример объявления идентификатора:

```
int a;
```

Тип данных `int`, а имя идентификатора `a`. В одном выражении могут быть объявлены несколько идентификаторов, как показано на **Рис. 1.11**:

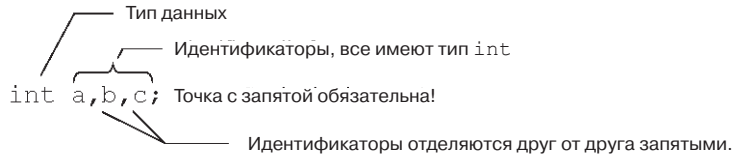


Рис. 1.11. Пример объявления нескольких идентификаторов

Подобные объявления должны предшествовать использованию идентификаторов в программе. Идентификатор может быть объявлен и *инициализирован* одновременно. В этом случае, кроме объявления переменной, ей ещё присваивается начальное значение. Например:

```
int a = 0;
```

Переменная `a` типа `int` инициализирована значением 0, **Рис. 1.12**.

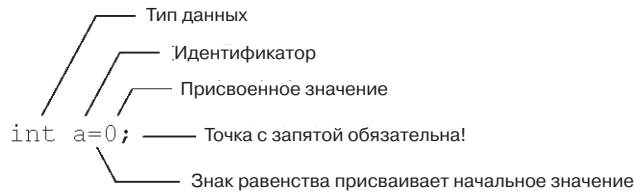


Рис. 1.12. Пример объявления и инициализации идентификатора

Рассмотренные объявления идентификаторов могут комбинироваться друг с другом. Такое объявление показано на **Рис. 1.13**

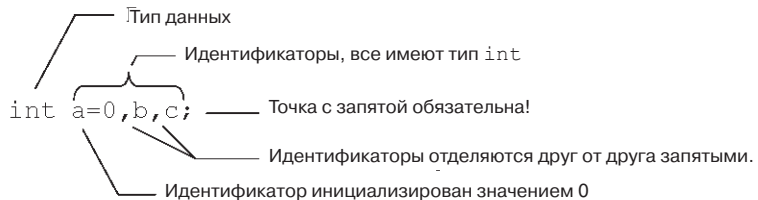


Рис. 1.13. Пример синтаксиса объявления идентификаторов в общем случае

1.6. Функции с параметрами и возвращаемыми значениями

В разделе 1.4.1. мы научились вызывать функцию и дали определение понятию абстрактности подпрограмм. В этом разделе рассмотрим функцию, которая вызывается для сложения двух чисел. Мы напишем функцию, принимающую два числа в качестве аргументов и возвращающую их сумму. Это поможет понять роль параметров функции и её возвращаемого значения.

Листинг 1.3. Функция с параметрами и возвращаемым значением

```

/*****
    В этой программе для сложения двух чисел
    вызывается функция (дважды) из функции main(),
    результат выводится на экран.
*****/

#include <iostream.h>

float Add(float a, float b) // Функция Add()
{
    float sum;

    sum = a + b;
    return sum;
}

void main() // Функция main()
{
    float p = 1, q = 2.3, r = 3, s = 4.5;
    float Sum1, Sum2;

    Sum1 = Add(p, q); // Первый вызов функции Add()
    cout << "First Sum " << Sum1 << endl;
    Sum2 = Add(r, s); // Второй вызов функции Add()
    cout << "Second Sum " << Sum2 << endl;
}

```

В программе на **Листинге. 1.3** мы определили новую функцию `Add()`. Как упоминалось ранее, определение функции состоит из типа возвращаемого значения, имени функции, списка параметров с указанием их типов и тела функции. В отличие от рассматривавшихся ранее функций, в функции `Add()` между круглыми скобками не пусто, что означает наличие *аргументов*. В нашем случае функция `Add()` имеет два аргумента типа `float`. Тип возвращаемого значения у функции `Add()` также `float`. Это значит что функция **должна** возвращать значение типа `float`. В теле функции возвращаемое значение должно указываться в *выражении возврата*. Ниже приведена функция `Add()` и объяснена её работа:

```
float Add(float a, float b)
{
    float sum;

    sum = a + b;
    return sum;
}
```

Примечание

По правилам языка C функция `Add()` объявляется так:

```
float Add();
```

Из этого следует, что объявление в C не несёт информации о параметрах. Прототип функции `Add()` выглядит так:

```
float Add(float, float);
```

Такая запись содержит информацию о параметрах. В C++ объявление и прототип функции это одно и то же. Поэтому прототип (и объявление) функции `Add()` будет таким:

```
float Add(float, float);
```

В теле этой функции мы объявили идентификатор `sum` типа `float`. Затем `sum` присваивается результат сложения `a` и `b`. Наконец, оператор `return` выводит значение `sum` за пределы функции. Обратите внимание, что тип возвращаемого значения, т. е. тип `sum` (которым является `float`), тот же самый, что и тип возвращаемого значения функции `Add()`, указанный в первой строке этой функции.

Функция `main()` нашей программы приведена ниже, жирным шрифтом выделены вызовы функции. В первом вызове функции `Add()` её формальные аргументы `a` и `b` заменяются копиями фактических аргументов `p` и `q`, в которых содержатся истинные значения аргументов. Параметры `a` и `b` могут рассматриваться как поля ввода. Во втором вызове функции `Add()` её параметры заменяются копиями значений `r` и `s`.

```
void main() // Функция main()
{
    float p = 1, q = 2.3, r = 3, s = 4.5;
    float Sum1, Sum2;

    Sum1 = Add(p, q); // Первый вызов функции Add()
    cout << "First Sum " << Sum1 << endl;
    Sum2 = Add(r, s); // Второй вызов функции Add()
    cout << "Second Sum " << Sum2 << endl;
}
```

После первого вызова функции `Add()` возвращённое ей значение присваивается `Sum1`. Поэтому `Sum1` принимает значение суммы `p` и `q`. В нашем случае `Sum1` будет иметь значение 3.3. Аналогично, `Sum2` будет иметь значение 7.5. Поскольку

функция `Add()` вызывается из функции `main()`, то `main()` является *источником* вызова и в то же время *приёмником* любых возвращаемых значений. С этой точки зрения тело функции `main()` будет иметь смысл *контекста вызова* (*calling environment*). Остальные строки в `main()` являются объявлениями и/или инициализацией идентификаторов, а также выражениями вывода `Sum1` и `Sum2` на экран.

На **Рис. 1.14** показаны стадии выполнения программы. Вся программа состоит из функции `main()`, нескольких других функций и данных. Выполнение программы начинается с функции `main()`, из которой, в свою очередь, вызываются остальные функции.

Функция `main()` и все остальные функции хранятся в так называемой *области кода* памяти программы, и при этом подразумевается их неизменность в ходе выполнения программы. Данные хранятся в *области данных* и её содержимое может меняться. Наряду с данными в фиксированных областях, данные могут создаваться во временной памяти, называемой *стеком*, а также в квази-непрерывной области, называемой *кучей* (*heap*) или *свободной памятью*.

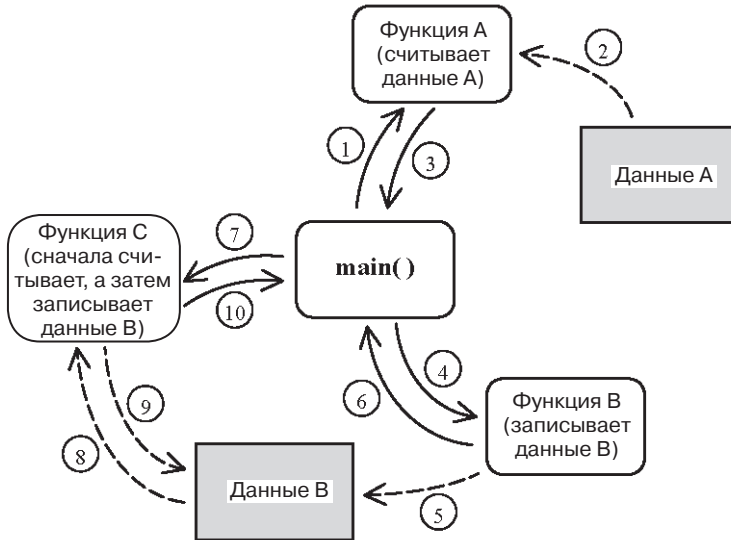


Рис. 1.14. Программа с функцией `main()` и другими функциями и данными

1.7. Заключение

Для написания программ на C++ обычно используются пакеты разработки. В этих пакетах есть интегрированная среда для редактирования, компилирования и редактирования связей в программах. Встроенные библиотеки называются библиотеками времени выполнения и содержат полезные функции. Чтобы воспользоваться этими функциями в начало программы нужно включить заголовочный файл с объявлениями функций. Программа на C++ состоит из исходного кода, написанного без синтаксических ошибок, с комментариями, служебными слова-

ми, идентификаторами, базовыми типами данных, пользовательскими типами данных и заголовочными файлами. Служебные слова являются частью языка C++. Базовые типы данных это встроенные типы данных и на их основе можно создавать пользовательские типы данных. Идентификаторы придумываются программистом и не могут быть служебными словами C++. И функции, и идентификаторы нужно объявлять перед использованием в программе.

Программы на C++ работают с использованием функций, которые обрабатывают некоторые данные. Это упрощает программирование, поскольку для выполнения какой-либо задачи программисту достаточно только вызвать функцию, не зная при этом, как именно реализована функция (это называется абстракцией). Программа начинает и заканчивает своё выполнение на функции `main()`. После выполнения своих действий, функции могут возвращать данные. Тип возвращаемых данных нужно указывать в определении функции, поэтому функция обладает таким атрибутом, как тип возвращаемого значения. Кроме этого, в функциях часто возникает необходимость во входных данных, над которыми нужно выполнять определённые действия. Данные передаются в функцию посредством её аргументов.

Сначала в этой главе мы рассмотрели элементарную программу на C++, состоящую из одной функции `main()`. Затем в эту программу была введена дополнительная функция для выполнения некоторых действий и демонстрации абстрактности функций. Наконец, была обсуждена представленная программа, выполняющая сложение двух чисел при помощи функции с двумя параметрами и возвращаемым значением.

1.8. Литература

1. Kelley, A. and I. Pohl, *A Book on C-programming in C*, Benjamin Cummins, 1995.
2. House, R., *Beginning with C – An Introduction to Professional Programming*, Interantional Thompson Publishing, 1994.
3. Deitel H. M. and P. J. Deitel *C: How to Program*, Prentice Hall, 1994.

2 Базовые сведения о параллельном порте и работа с ним

Содержание главы:

- Устройство и функционирование параллельного порта.
- Основы логики.
- Системы счисления: десятичная, шестнадцатеричная и двоичная.
- Электроника: регистр, байт, синхронный, асинхронный, адреса.

2.1. Введение

Для эффективной работы с параллельным портом необходимо понимание основ логики и умение преобразовывать данные из одной системы счисления в другую. Здесь будут рассмотрены эти вопросы, а также описано устройство параллельного порта. Будут разъяснены такие базовые понятия, как двоичная логика, логические уровни, пространство адресов ввода/вывода и физическое подключение к порту.

Материал данной главы достаточен для использования параллельного порта в программах и подключения к нему. Эти знания понадобятся в дальнейших главах при создании программ для управления и контроля оборудования по параллельному порту.

Понимание основ цифровой схемотехники также полезно при сборке и наладке схем на интерфейсной плате.

2.2. Что такое параллельный порт?

В общих чертах порт представляет собой электронную схему, служащую интерфейсом для соединения с другим электронным устройством, чтобы обеспечить возможность обмена информацией. Такое соединение позволяет вводить информацию в порт, выводить из порта, а также осуществлять двунаправленную передачу данных через порт.

Параллельный порт обладает возможностью обмена данными между компьютером и внешним миром. Обычно он используется для передачи данных на принтер и иногда его называют портом принтера. В старых компьютерах порт принтера выполнен в виде схемы на отдельной печатной плате, которая вставляется в материнскую плату компьютера. В новых компьютерах порт принтера обычно уже встроен в материнскую плату.

Владение такими понятиями, как *семейства логических схем*, *логические уровни* и *запас помехоустойчивости* позволит лучше понимать цифровой обмен между устройствами. Эти знания окажутся полезными и при отладке цифровых схем на интерфейсной плате.

2.2.1 Цифровые логические схемы

Ранее упоминалось, что компьютерные программы выполняются электронными схемами двоичной логики, называемые также *цифровыми логическими схемами*. В двоичной логике есть только два возможных состояния: ВКЛЮЧЕНО и ВЫКЛЮЧЕНО. Обычно эти состояния двоичной логики представляются в *двоичном формате*, где 1 соответствует состоянию ВКЛЮЧЕНО, а 0 состоянию ВЫКЛЮЧЕНО.

Состояния ВКЛЮЧЕНО и ВЫКЛЮЧЕНО в параллельном порту, как и в большом количестве прочих цифровых схем, соответствуют уровням напряжения, называемым *логическими уровнями*, обычно из диапазона 0...5 В. Заметим, что не у всех логических схем такие же логические уровни. Логические схемы могут быть в виде *интегральных схем*, в которых элементы схемы размещены на одном полупроводниковом кристалле, на так называемом *чипе*. Чип помещается внутрь пластмассового или керамического корпуса с металлическими выводами, соединёнными с чипом, которые служат для подключения внешних элементов.

Из логических схем наиболее распространены два типа (или *семейства*): ТТЛ (Транзисторно-Транзисторная Логика) и КМОП (Комплементарный Металло-Оксидный-Полупроводник). Эти семейства логики изготавливаются по разным технологиям, что приводит к различиям в эксплуатационных характеристиках. Основные различия электрических параметров семейств ТТЛ и КМОП представлены на **Рис. 2.1**. На рисунке показаны различия электрических параметров семейств логики. Следует отметить, что схемы некоторых семейств КМОП работают при напряжениях вне диапазона 0...5 В. Кроме того, выходное напряжение этих схем зависит от величины протекающего через выход тока.

Эти различия между семействами по логическим уровням имеют большое значение при взаимодействии логических схем различных семейств. В качестве примера рассмотрим варианты **а** и **б** на **Рис. 2.1**; в нашем случае микросхема ТТЛ передаёт ВЫСОКИЙ логический уровень (+2.4...+5 В) микросхеме КМОП. Микросхема ТТЛ в самом худшем случае может формировать ВЫСОКИЙ уровень не ниже +2.4 В без риска выйти из строя. Чтобы микросхема КМОП могла правильно воспринять ВЫСОКИЙ уровень, он должен составлять, по меньшей мере, +3.2 В и не превышать +5 В. Проблема заключается в том, что микросхема может ТТЛ формировать более низкие напряжения, вплоть до +2.4 В, что является недостаточным для правильного распознавания микросхемой КМОП ВЫСОКОГО уровня. Поэтому ВЫСОКИЙ логический уровень ТТЛ может быть ошибочно воспринят как НИЗКИЙ.

На **Рис. 2.1** проиллюстрирован также *запас помехоустойчивости* при передаче сигнала между логическими схемами одного и того же семейства. Рассмотрим случай, когда схема КМОП передаёт НИЗКИЙ логический уровень на вход другой схемы КМОП, **Рис. 2.1-г**. Передающая схема может формировать выходное напряжение 0...+0.2 В при соблюдении её норм эксплуатации, а принимающая схема, в свою очередь, воспринимает сигнал с уровнем 0...+1.5 В как НИЗКИЙ. Если такой выходной уровень принять равным +0.2 В, как худший случай, то помеха по напряжению для этого сигнала может достигать $1.5 - 0.2 = 1.3$ В, при этом на входе принимающей схемы будет НИЗКИЙ логический уровень. В этом примере *запас помехоустойчивости* составляет 1.3 В.

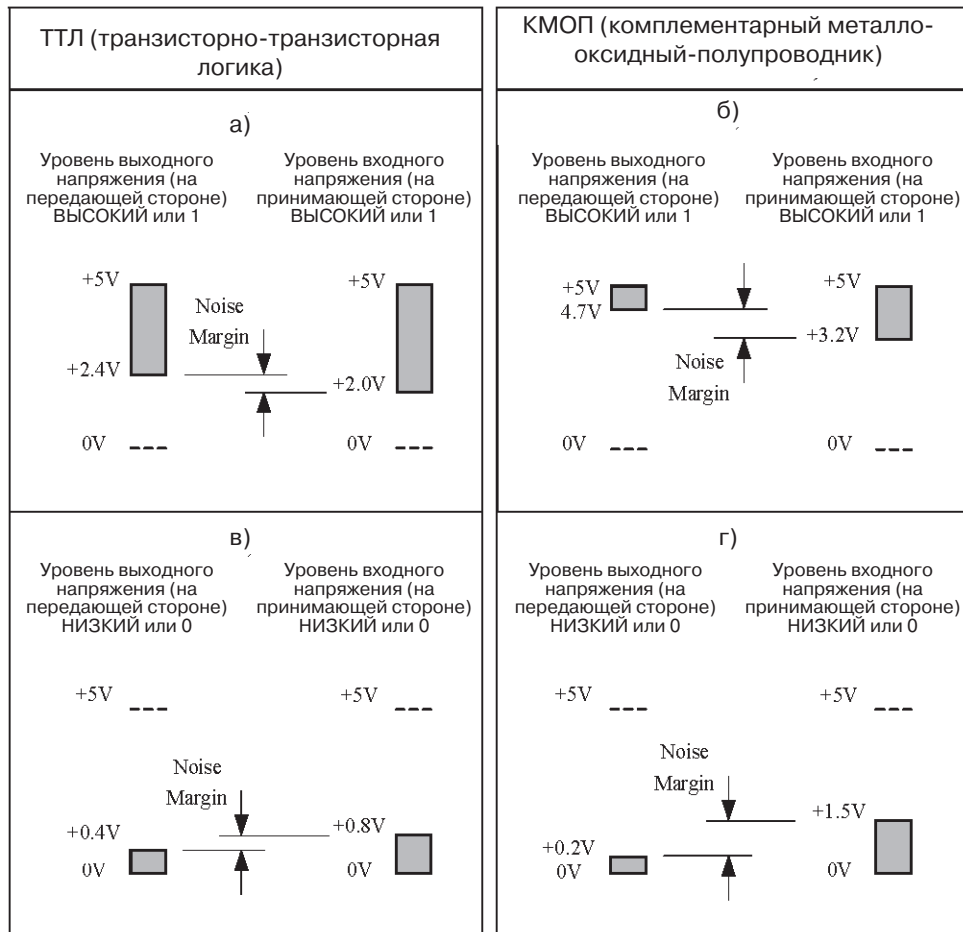


Рис. 2.1. Типичные логические уровни ТТЛ и КМОП (напряжение питания +5 В)

Если рассматривать аналогичный случай для элементов ТТЛ (**Рис. 2.1-в**), когда передаётся **НИЗКИЙ** логический уровень, можно увидеть, что запас помехоустойчивости составляет только 0.4 В. Обычно схемы КМОП обладают большей помехоустойчивостью, чем схемы ТТЛ. Схемы ТТЛ и КМОП также различаются по энергопотреблению, входным токам, нагрузочной способности по току и скорости переключения между состояниями. Дополнительная информация о семействах логических схем содержится в литературе, перечисленной в конце главы.

2.2.2. Устройство параллельного порта

Параллельный порт обеспечивает передачу данных для печати на принтер и приём информации о состоянии принтера. Данные от компьютера передаются по восьми линиям, что позволяет передавать в принтер *байты* информации. Байт это

группа из восьми *bit*, образующих в совокупности «порцию» данных. Каждая линия может передавать один бит данных. Бит может принимать два логических значения: 0 и 1. Остальные девять линий нужны для приёма информации о состоянии принтера и управления потоком данных. Эти девять линий разделяются на две группы: одна из пяти выходных линий, а другая из четырёх двунаправленных линий, **Рис. 2.2**. Физическое подключение к порту осуществляется посредством разъёма, обозначаемого D25F (буква D означает форму разъёма в форме буквы D). Разъём представляет собой 25-контактную розетку (буква F означает «розетка»). Кабель принтера имеет соответствующую 25-контактную вилку.

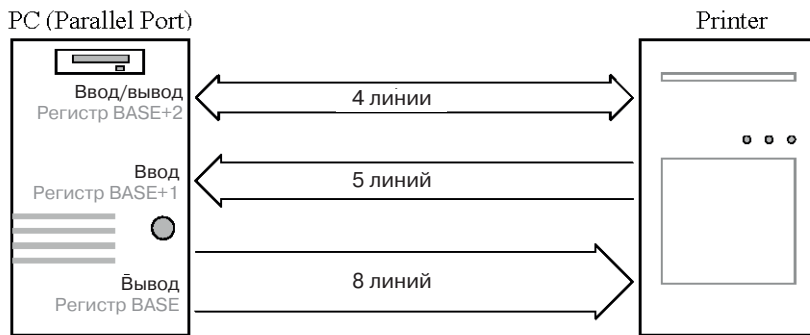


Рис. 2.2. Параллельный порт в общем виде

Три группы линий, показанные на **Рис. 2.2** изображают соединение между параллельным портом компьютера и внешним устройством, в данном случае с принтером. Каждая группа линий управляется или считывается через три регистра, последовательно расположенных в *адресном пространстве ввода/вывода*. Это адресное пространство образовано некоторым количеством ячеек памяти – *регистров*, позволяющих обмениваться данными с устройствами ввода/вывода. Это пространство памяти отличается от обычной памяти компьютера. Компьютер записывает данные в различные регистры ввода/вывода, где они сохраняются и могут быть считаны внешними устройствами. Другие регистры позволяют внешним устройствам записывать в них данные, которые затем могут быть считаны компьютером, а некоторые регистры допускают двунаправленный обмен данными.

Первый из этих трёх регистров в пространстве ввода/вывода имеет адрес BASE, **Рис. 2.3**. Этот регистр с наименьшим адресом и используется для обращения к другим регистрам параллельного порта путём прибавления констант к его значе-

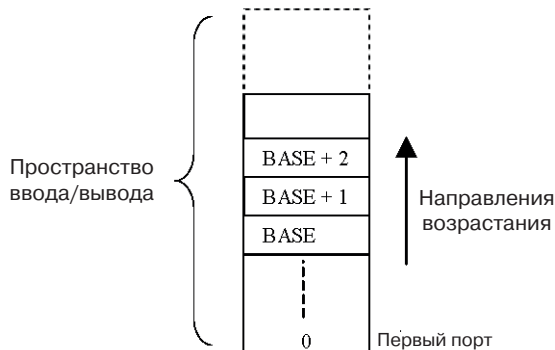


Рис. 2.3. Адресация пространства ввода/вывода

нию (т. е. адресу). Запись в регистр BASE приводит к выводу восьми бит данных (байта) в параллельный порт (**Рис. 2.2**), где каждый бит занимает отдельную линию.

Адрес следующего регистра группы на единицу больше адреса BASE и обозначается BASE+1. В регистр BASE+1 выведены пять входных линий параллельного порта. Можно только считывать состояния этих пяти сигналов.

Третий регистр обозначается BASE+2. Посредством этого регистра осуществляется управление четырьмя двунаправленными линиями параллельного порта. В этот регистр можно как записывать, так и считывать данные из него.

Примечание

Четыре двунаправленные линии регистра BASE+2 не являются непосредственными выходами логических схем. В параллельном порту к этим линиям часто подключаются резисторы и конденсаторы для уменьшения влияния помех. Это приводит к значительно более медленному переключению между состояниями и может вызвать ошибочное чтение данных при подключении логических схем различных семейств.

Также следует отметить, что разброс ёмкостей конденсаторов вызывает неодновременное (асинхронное) переключение состояний сигналов. Это асинхронное переключение выходов регистра BASE+2 может вызвать проблемы при передаче данных по синхронным интерфейсам.

В **Табл. 2.1** приведено соответствие битов данных во всех трёх адресах ввода/вывода, используемых параллельным портом, и контактов разъёма DB25. Каждый провод в соединительном кабеле между параллельным портом и внешним устройством (обычно это принтер) передаёт сигнал, соответствующий какому-либо регистру ввода/вывода. Биты в регистрах BASE и BASE+2 расположены последовательно, начиная с бита **D0**. Но в регистре BASE+1 биты располагаются с бита **D3**.

Таблица 2.1. Назначение контактов разъёма DB25 параллельного порта

Регистр BASE (8-битные выходные данные)	Регистр BASE+1 (5-битные входные данные)	Регистр BASE+2 (4-битные двунаправленные данные)
D0 – контакт 2 D1 – контакт 3 D2 – контакт 4 D3 – контакт 5 D4 – контакт 6 D5 – контакт 7 D6 – контакт 8 D7 – контакт 9	D3 – контакт 15 D4 – контакт 13 D5 – контакт 12 D6 – контакт 10 D7 – контакт 11	D0 – контакт 1 D1 – контакт 14 D2 – контакт 16 D3 – контакт 17
Примечание: D0̄ означает инверсию бита схемой параллельного порта.		

Некоторые биты по адресам BASE+1 и BASE+2 инвертируются схемой параллельного порта. Такие биты обозначаются чёрточкой над именем бита. Так же

обозначаются и сигналы на *принципиальной схеме* интерфейсной платы, на которой показаны все электрические соединения. При использовании таких битов в программе нужно учитывать их инверсию при чтении или записи данных порта.

Если нужно послать данные по одной из инвертированных линий порта, то в программе нужно инвертировать соответствующий бит. Такая двойная инверсия (в схеме порта и в программе) даёт в результате желаемый выходной сигнал. Аналогичным образом поступают при чтении инверсного бита порта, инвертируя нужные биты, чтобы получить их истинное значение. В программе инверсия реализуется всего одной строчкой кода, речь об этом пойдёт в разделе 3.6 следующей главы.

Контакты разъёма DB25 с 18 по 25 не показаны в Табл. 2.1. Они все подключены к земле компьютера и соединяются с интерфейсной платой кабелем, Рис. 2.4. Это кабель с вилками DB25 на концах, контакты соединяются отдельными проводами «один в один» (первый контакт одного разъёма соединён с первым контактом другого разъёма, аналогичным образом соединяются остальные контакты).

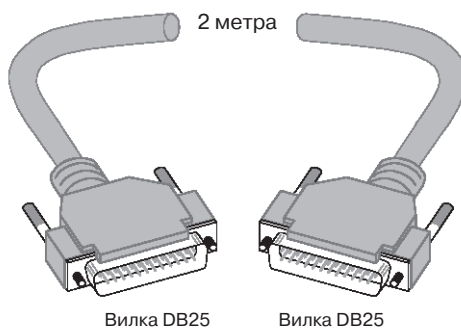


Рис. 2.4. Кабель DB25M-DB25M

Примечание

Биты данных **D0...D2** по адресу BASE+1 не подключены к схеме параллельного порта. Это относится и к битам **D4...D7** регистра BASE+2. Чтение этих битов даст неопределённый результат.

2.3. Представление данных

Ранее говорилось, что компьютеры представляют данные в виде *двух* состояний ВКЛЮЧЕНО или ВЫКЛЮЧЕНО (ВЫСОКИЙ или НИЗКИЙ уровни напряжения) и поэтому называемые *двоичными*. Таким образом, данные представляются только двумя логическими состояниями (ВКЛЮЧЕНО/ВЫКЛЮЧЕНО). Числа в двоичной системе счисления представляются в виде суммы целых степеней числа 2, которые являются весом двоичного разряда. Мы можем сравнить эту систему с известной нам десятичной системой счисления. Десятичные числа образуются целыми степенями числа 10.

Например, десятичное число 25 можно представить так:

$$\begin{aligned} 25 &= 2 \times 10^1 + 5 \times 10^0 \\ &= 2 \times 10 + 5 \times 1 \end{aligned}$$

Десятичное число 25 равно двоичному числу 11001:

$$\begin{aligned}
 11001 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
 &= 1 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1 \times 1 \\
 &= 25 \text{ (десятичное)}
 \end{aligned}$$

Примечание

В двоичном числе крайняя правая цифра имеет наименьший вес и называется *младший значащий бит* (**Least Significant Bit, LSB**). Наоборот, крайняя левая цифра имеет наибольший вес и называется *старшим значащим битом* (**Most Significant Bit, MSB**).

Многоразрядные двоичные числа трудночитаемы. Для упрощения чтения пользуются более удобным представлением чисел, называемым *шестнадцатеричным*. В этом случае числа представляются посредством шестнадцати цифр.

В десятичной системе счисления используются десять арабских цифр от 0 до 9. В шестнадцатеричной системе нужно шестнадцать разных цифр. В качестве первых десяти используются арабские от 0 до 9, но ещё нужны цифры с одиннадцатой по пятнадцатую. В качестве этих цифр используются заглавные буквы латинского алфавита A, B, C, D, E и F.

В **Табл. 2.2** приведено соответствие десятичных, двоичных и шестнадцатеричных чисел.

Таблица 2.2. Преобразование между системами счисления

Десятичная	Двоичная	Шестнадцатеричная
0 = 0×10^0	0 = 0×2^0	0 = 0×16^0
1 = 1×10^0	1 = 1×2^0	1 = 1×16^0
2 = 2×10^0	10 = $1 \times 2^1 + 0 \times 2^0$	2 = 2×16^0
3 = 3×10^0	11 = $1 \times 2^1 + 1 \times 2^0$	3 = 3×16^0
4 = 4×10^0	100 = $1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	4 = 4×16^0
5 = 5×10^0	101 = $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$	5 = 5×16^0
6 = 6×10^0	110 = $1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$	6 = 6×16^0
7 = 7×10^0	111 = $1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	7 = 7×16^0
8 = 8×10^0	1000 = $1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	8 = 8×16^0
9 = 9×10^0	1001 = $1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$	9 = 9×16^0
10 = $1 \times 10^1 + 0 \times 10^0$	1010 = $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$	A = 10×16^0
11 = $1 \times 10^1 + 1 \times 10^0$	1011 = $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	B = 11×16^0
12 = $1 \times 10^1 + 2 \times 10^0$	1100 = $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	C = 12×16^0
13 = $1 \times 10^1 + 3 \times 10^0$	1101 = $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$	D = 13×16^0
14 = $1 \times 10^1 + 4 \times 10^0$	1110 = $1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$	E = 14×16^0
15 = $1 \times 10^1 + 5 \times 10^0$	1111 = $1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	F = 15×16^0
16 = $1 \times 10^1 + 6 \times 10^0$	10000 = $1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	10 = $1 \times 16^1 + 0 \times 16^0$
17 = $1 \times 10^1 + 7 \times 10^0$	10001 = $1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$	11 = $1 \times 16^1 + 1 \times 16^0$

В программах иногда возникает необходимость выводить цифровые сигналы через параллельный порт в виде одного или нескольких байтов. Выходные сигналы являются набором логических сигналов, которые удобно представлять шестнадцатеричными цифрами. В других случаях нужно преобразовывать принятые от внешних устройств данные в шестнадцатеричные цифры. Следующие примеры демонстрируют преобразование двоичных чисел в шестнадцатеричные.

Мы можем получить шестнадцатеричное представление двоичного числа, если разобьём двоичное число на четырёхбитные группы (полубайты), начиная с младшего значащего бита. Заметим, что после разбиения байта на две группы по четыре бита мы получили две порции данных, называемых также *тетрадами*.

$$\begin{array}{rcl} 10001 & = & 1 \\ & = & 1 \end{array} \quad \begin{array}{rcl} 0001 & & \\ & = & 1 \end{array} \quad \text{(шестнадцатеричное)}$$

Шестнадцатеричные числа часто записываются с префиксом 0x, например 0x11.

$$\begin{array}{rcl} 1010001101 & = & 10 \\ & = & 2 \end{array} \quad \begin{array}{rcl} 1000 & & \\ & = & 8 \end{array} \quad \begin{array}{rcl} 1101 & & \\ & = & D \end{array} \quad \text{(шестнадцатеричное 0x28D)}$$

Шестнадцатеричные числа также обозначаются символами \$ или H, например \$11 или 11H.

Если нам нужно записать данные по двоичному адресу 1010001101, то мы сами должны преобразовать это число в шестнадцатеричный формат и использовать полученный адрес 0x28D в качестве адреса для записи данных. Если бы мы преобразовывали двоичное число в десятичное, то пришлось бы выполнять более сложное преобразование.

Теперь нам известно шестнадцатеричное представление чисел и в такой форме мы можем записывать адреса регистров параллельного порта. В большинстве компьютеров регистр BASE имеет адрес 0x378, но в некоторых моделях BASE может находиться по адресам 0x278, 0x3BC или 0x300. В самом распространённом случае, когда BASE равно 0x378, адрес BASE+1 будет 0x379, а BASE+2 0x37A.

2.4. Программа для отображения шестнадцатеричных и десятичных чисел

Программа, приведённая в **Листинге 2.1**, преобразовывает числа из десятичной системы в шестнадцатеричную и наоборот. Шестнадцатеричные числа зачастую оказываются удобнее десятичных при работе с сигналами параллельного порта. Входные данные, считываемые из порта, наоборот, иногда удобнее выводить на экран в десятичной форме. Изучайте взаимосвязь десятичной и шестнадцатеричной систем счисления при помощи этой программы.

Листинг. 2.1. Программа для отображения чисел в десятичном и шестнадцатеричном форматах

```
/*
*****
Эта программа выводит на экран введённое
```

```
    десятичное число в шестнадцатеричном формате
    *****/

#include <iostream.h>

void main()
{
    int Number;

    cout << "Enter an integer number -> ";
    cin >> Number;

    cout << "The number is:" << endl;
    cout << dec << Number << " in decimal" << endl;
    cout << hex << Number << " in hexadecimal" << endl;
}
```

В **Листинге 2.1** заголовочный файл `iostream.h` позволяет использовать `cout` и аргумент преобразования числа `hex`. Переменная `Number` служит для хранения вводимого целого числа по запросу "Enter an integer number ->" (Введите целое число). Это число затем выводится в двух следующих строках, сначала в десятичном виде, а затем в шестнадцатеричном. Шестнадцатеричный вид числа задаётся спецификатором формата `hex`.

2.5. Заключение

В этой главе было объяснены устройство параллельного порта и основные понятия цифровой логики, необходимые для использования параллельного порта с внешними схемами: двоичный формат, цифровые логические уровни, запас помехоустойчивости и семейства логических схем КМОП и ТТЛ.

Обмен данными по интерфейсу параллельного порта компьютера осуществляется посредством трёх регистров пространства ввода/вывода. Каждый регистр имеет размер один байт, при этом в двух регистрах некоторые биты не используются, а некоторые инвертируются схемой параллельного порта. Первый регистр используется только для вывода данных, второй – только для ввода, а последний – и для ввода и для вывода. Было объяснено представление чисел в десятичной, шестнадцатеричной и двоичной системах счисления. Эти знания нам понадобятся при создании программ в следующих главах.

2.6. Литература

1. Bergsman, P., *Controlling The World With Your PC*, HighText Publications, San Diego, 1994.
2. IBM, Technical Reference – Personal Computer AT, IBM Corporation, 1985.
3. NS CMOS, *CMOS Logic Databook*, National Semiconductor Corporation, 1988.
4. Wakerly, J. F., *Digital Design Principles and Practices*, Second Edition, Prentice Hall, 1994.

3 Тестирование параллельного порта

Содержание главы:

- Простое тестирование порта.
- Источник питания, интерфейс порта, буферы и управление светодиодами.
- Программирование в стиле C для полного использования параллельного порта.

3.1. Введение

Целью этой главы является создание программ, позволяющих изучить основные операции ввода/вывода параллельного порта компьютера. Контроль работы программы осуществляется при помощи простой тестовой схемы со светодиодами в качестве индикаторов. Перед тем, как приступить к работе с источником питания, параллельным портом и светодиодами, объясняется принцип их работы. Как только эти схемы будут собраны (в представленной последовательности), можно приступать к написанию и тестированию программ.

Сначала будут созданы программы, не являющиеся объектно-ориентированными, осуществляющие ввод/вывод данных через параллельный порт. Далее, в главе 5, мы познакомимся с объектно-ориентированным программированием (ООП). После прочтения этой главы вы будете знать принцип обмена данными по параллельному порту и станете лучше понимать простые программы на C++.

3.2. Источник питания интерфейсной платы

Для нормальной работы схемы на интерфейсной плате ей нужно электропитание постоянным стабилизированным напряжением. Источник питания в составе интерфейсной платы формирует все необходимые напряжения для питания её различных цепей. Эта часть платы должна быть собрана и отлажена раньше всех других схем. Другие схемы нужно собирать только когда источник питания будет давать нужные напряжения.

Схема источника питания показана на **Рис. 3.1**. Большая часть энергии поступает в интерфейсную плату из высоковольтного источника (из электросети) через трансформатор (силовой модуль). Выходы источника питания с напряжениями +5 В и +9 В могут поддерживать эти напряжения при токах нагрузки до 1 А. Для обеспечения максимальных значений тока каждый стабилизатор должен обладать соответствующим радиатором. Выход –8 В питается не от сети, а от батареи с напряжением +9 В, соответствующей требованиям этого выхода по напряжению и току. Выход –8 В предназначен для питания аналоговых цепей, работающих в

широком диапазоне питающих напряжений, вследствие этого нет необходимости в стабилизации этого напряжения.

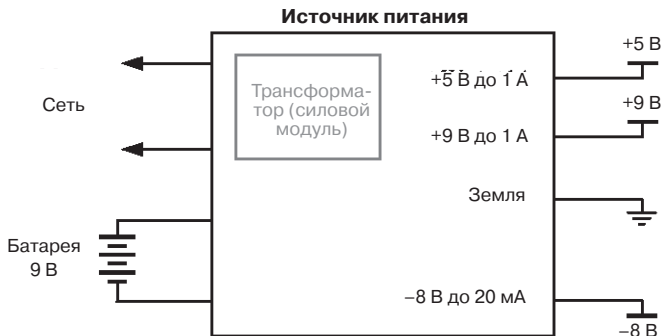


Рис. 3.1. Блок питания в обобщённом виде

На **Рис. 3.2** изображена более подробная функциональная схема источника питания. Батарея с напряжением 9 В питает схему через диод, проводящий ток только при соблюдении полярности подключения батареи. Проходящий через диод ток обуславливает падение напряжения на нём примерно на 1 В, поэтому на выходе получается -8 В.

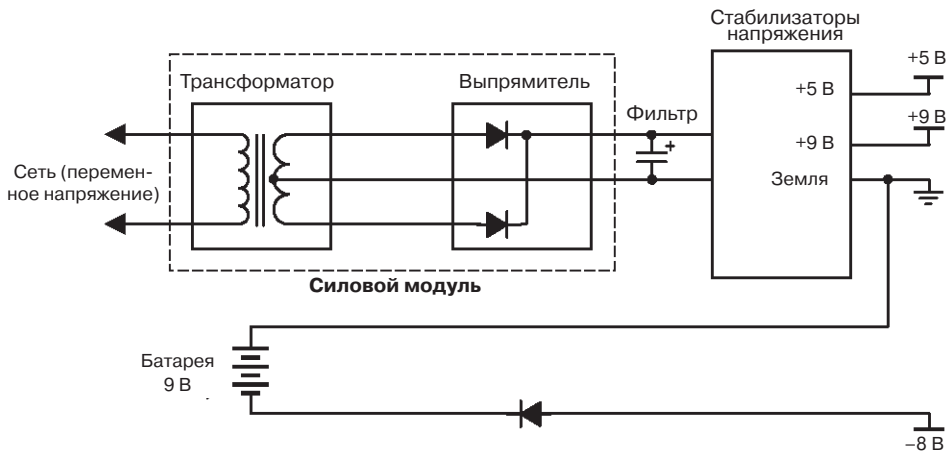


Рис. 3.2. Узлы источника питания

Сетевая часть источника питания состоит из четырёх узлов: трансформатора, выпрямителя с фильтром и двух стабилизаторов напряжения (на +5 В и +9 В). Совместная работа этих узлов приводит к выпрямлению сетевого напряжения и преобразованию его в стабилизированные напряжения +5 В и +9 В, где полностью отсутствуют колебания.

Переменное сетевое напряжение сначала должно быть уменьшено по амплитуде, затем выпрямлено и только потом уже может поступать на низковольтные схе-

мы. Эту функцию выполняет трансформатор внутри силового модуля. Там есть два диода, заставляющие протекать ток только в одном направлении, выпрямляя тем самым переменное синусоидальное напряжение трансформатора (**Рис. 3.3**).

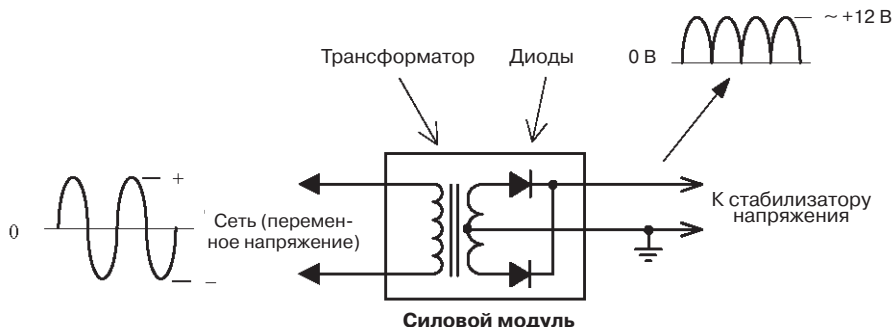


Рис. 3.3. Трансформаторный модуль источника питания (без конденсатора)

Для нормальной работы стабилизаторов напряжения необходимо подавать им на вход напряжение, превышающее выходное, по меньшей мере, на несколько вольт. К выходу силового модуля подключен конденсатор большой ёмкости, предотвращающий периодические «провалы» его выходного напряжения до нуля, обеспечивая тем самым сглаживание пульсаций напряжения до приемлемого уровня (**Рис. 3.4**).

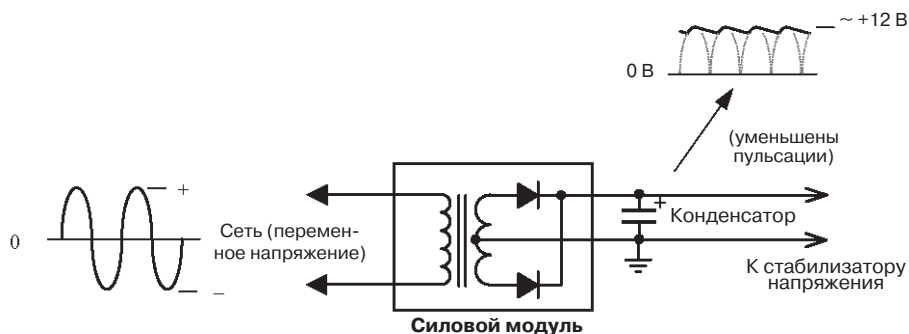


Рис. 3.4. Трансформаторный модуль с конденсатором на выходе.

На входы стабилизаторов напряжения подаётся пульсирующее напряжение (**Рис. 3.5**) и их внутренними схемами удерживается в пределах нескольких процентов от номинальных значений на их выходах. Стабилизатор +5 В, например, формирует выходное напряжение в диапазоне +4.75...+5.25 В. На интерфейсной плате ко входу и к выходу стабилизаторов подключено несколько конденсаторов, соединённых с землёй. Эти конденсаторы предотвращают появление на выходах стабилизаторов колебаний высоких частот и улучшают устойчивость стабилизаторов при быстро изменяющейся нагрузке.

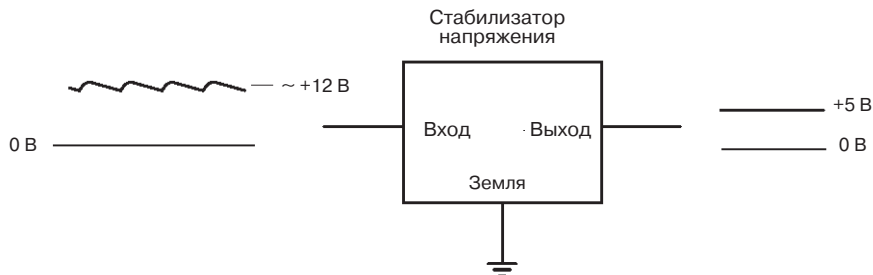


Рис. 3.5. Стабилизатор напряжения (без конденсаторов)

Теперь можно собрать и наладить источник питания. Инструкции по сборке схемы, пайке, чтению принципиальной схемы, а также способы проверки и отладки схем, способы подготовки и проверки печатной платы приведены в конце книги, **Приложение А**.

3.3. Интерфейс параллельного порта

Часто возникает необходимость проверки правильности работы программы во время её разработки. На интерфейсной плате есть схема, позволяющая наблюдать сигналы параллельного порта, обеспечивая тем самым возможность контролировать выполнение программы. Функциональная схема этого устройства показана на **Рис. 3.6**.

Сигналы параллельного порта подаются посредством соединительного кабеля на разъём D25 интерфейсной платы. От разъёма восемь сигналов подаются на входы микросхемы буфера. Эти сигналы формируются посредством записи данных в регистр BASE. Вторая буферная микросхема предназначена для передачи пяти сигналов в регистр BASE+1 компьютера при помощи пяти печатных дорожек, подключённых к разъёму D25. Остальные четыре сигнала, определяемые регистром BASE+2, подключаются к резисторам. Эта часть параллельного порта может вводить и выводить данные.

Микросхемы часто выходят из строя вследствие ошибочного соединения двух выходов без принятия специальных мер по ограничению возникающего при этом тока. В сигнальные линии регистра BASE+2 последовательно включены резисторы для ограничения тока и уменьшающие риск повреждения выходов параллельного порта при ошибочном подключении к другим выходам на интерфейсной плате.

Заметим, что ко входам двух 8-канальных буферных микросхем (**Рис. 3.6**) подключены подтягивающие резисторы (на рисунке они не показаны). Их назначение описано в **Приложении А**.

На **Рис. 3.6** есть выводы с точками на концах. Эти точки обозначают контакты на плате, обеспечивающие возможность соединения с другими цепями платы. Это выполняется соединительными проводниками (**Рис. 3.7**), подключаемым к этим контактам. Вы можете самостоятельно изготавливать такие проводники по мере необходимости (**Приложение А**).

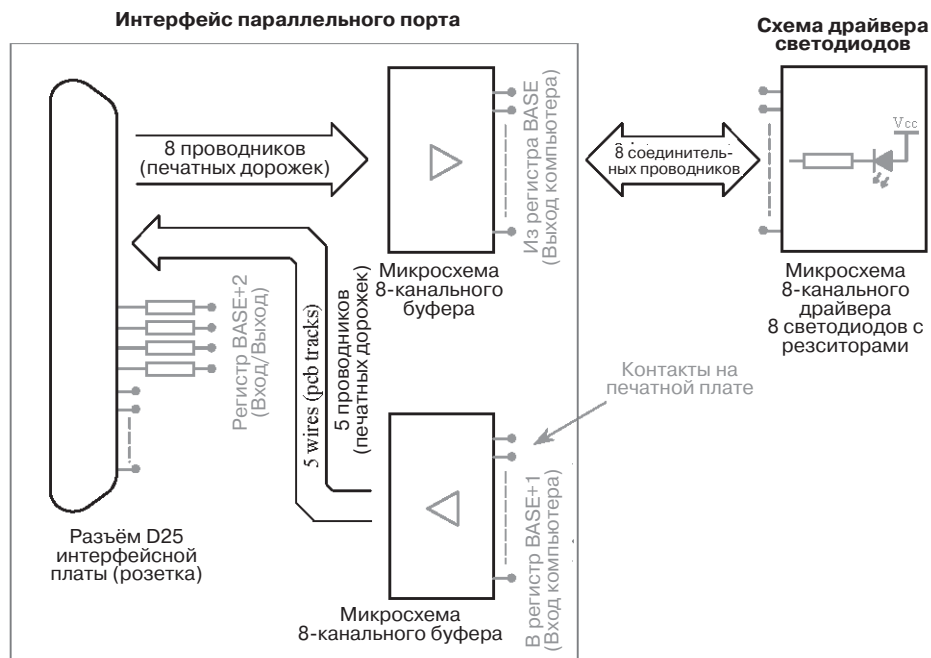


Рис. 3.6. Функциональная схема интерфейса параллельного порта и драйвера светодиодов

Каждое соединение выполняется путём соединения проводником *выхода* одной схемы и *входа* другой. Никогда не соединяйте выходы элементов схем друг с другом. Такое соединение приведёт, скорее всего, к выходу из строя этих элементов.

На схеме (**Рис. 3.6**) показаны также резистор и светодиод, представляющие одну из восьми таких цепей. Эта схема нужна для отображения уровней логических сигналов интерфейсной платы. Состояние сигналов определяется путём подключения нужных сигналов к соответствующим парам резистор-светодиод.

Для тестирования программ, считывающих данные параллельного порта, на интерфейсной плате есть несколько контактов, подключенных к цепям +5 В и земле. Это позволяет тестировать любые из четырёх или пяти бит входных данных в регистрах BASE+1 и BASE+2.

Соберите схему для подключения к параллельному порту и проверьте её работоспособность. В конце книги есть принципиальная схема, сборочный чертёж и описан способ наладки. Для экспериментов нужно изготовить несколько соединительных проводников.

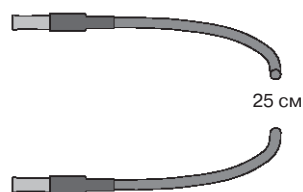


Рис. 3.7. Соединительный проводник

3.3.1. Схема драйвера светодиодов

На **Рис. 3.6** показаны восемь логических выходов (идущие от параллельного порта), соединённых с буфером печатными дорожками. К сожалению, подобно большинству логических схем, отдельный вывод порта не может обеспечить ток, достаточный для светодиода; здесь нужна микросхема-драйвер. Каждый выход буфера соединён с отдельным контактом на печатной плате. Контакты выходов буфера соединяются со входами микросхемы-драйвера светодиодов проводами.

На этой плате в качестве драйвера используется микросхема ULN2803A. Она представляет собой набор транзисторов, каждому из которых достаточно слабого выходного тока логической схемы для управления светодиодом. Большинство светодиодов светятся при токе 5...20 мА. Драйвер ULN2803A коммутирует ток каждой пары светодиод-резистор по отдельности и работает следующим образом:

1. Когда на вход драйвера (с левой стороны микросхемы) подаётся ВЫСОКИЙ логический уровень, соответствующий выходной контакт коммутируется внутри микросхемы на землю. Это вызывает протекание тока от источника питания +5 В через светодиод, через резистор и через выход драйвера на землю, заставляя светодиод светиться.
2. Когда на вход драйвера подаётся НИЗКИЙ логический уровень, то сопротивление между соответствующим выходным контактом микросхемы и землёй становится очень большим. Ток через светодиод крайне мал и он не светится.

3.3.2. Работа светодиода

Для свечения светодиода необходимо протекание через него некоторого минимального тока. Светодиоды, подобно большинству диодов, проводят ток только в одном направлении, и это направление соответствует рабочему включению светодиода. Ток протекает от анода (обозначаемого треугольником) к катоду (обозначаемому чёрточкой), **Рис. 3.8**.

Чтобы ток мог протекать в прямом направлении, к аноду нужно приложить большее положительное напряжение, чем к катоду, называемое прямым напряжением, V_F . Эта разность потенциалов (V_F) для большинства светодиодов составляет примерно 2 В и 0.7 В для обычных диодов, когда они включены в проводящем направлении.

Вольтамперная характеристика светодиода приведена на **Рис. 3.9**. Эта характеристика отражает зависимость тока через светодиод от напряжения на нём в рабочем режиме, соответствующий участок характеристики выделен жирной линией в первом квадранте характеристики. В рабочем режиме увеличение тока через светодиод приводит к яркости его свечения. Когда значение этого тока становится больше допустимого I_F , то он выходит из строя. Токоограничивающий резистор (**Рис. 3.10**) препятствует возрастанию тока и определяет его величину.

Если мы изменим полярность напряжения на светодиоде на обратную, то сможем достигнуть напряжения обратного пробоя V_{BR} (**Рис. 3.9**), на вольтамперной характеристике пробой наступает при -5 В. Как только будет превышено напря-



Рис. 3.8. Ток через светодиод в рабочем режиме

жение пробоя, ток через светодиод будет возрастать до значения I_R , при котором светодиод выйдет из строя.

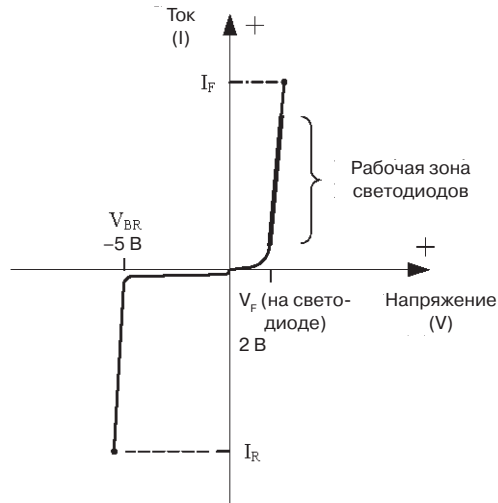


Рис. 3.9. Типичная вольт-амперная характеристика светодиода (без последовательного резистора)

Когда мы прикладываем к светодиоду прямое напряжение, обуславливающее его хорошую проводимость, то это называется его *смещением*. Выше говорилось, что без токоограничивающего резистора светодиод выходит из строя. Ток через резистор зависит от напряжения на нём. Если светодиод и резистор включены как на **Рис. 3.10**, то через них протекает один и тот же ток. Такое построение цепи называется *последовательным* соединением.

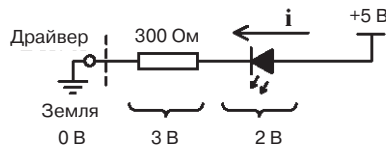


Рис. 3.10. Ток в последовательной цепи

Если нам известно напряжение на резисторе, то мы можем вычислить ток через него (а значит и через светодиод), поскольку

$$\text{Ток} = \frac{\text{Напряжение}}{\text{Сопротивление}}$$

Мы знаем, что прямое напряжение на светодиоде примерно 2 В и напряжение на всей цепи равно 5 В. Поэтому напряжение на резисторе составит:

$$5 \text{ В} - 2 \text{ В} = 3 \text{ В}.$$

Ток I через резистор определяется напряжением на нём, разделённым на его сопротивление в Омах (Ω), то есть:

$$I = \frac{3V}{330\Omega} = 0,009 A \quad (\text{амперы обозначаются буквой } A)$$

Токи в электронных схемах обычно составляют лишь малую долю Ампера и поэтому в качестве единицы измерения тока применяется миллиампер, составляющий 1/1000 Ампера. Таким образом, через светодиод и резистор течёт ток 0.009 Ампер или 9 миллиампер.

Соберите и проверьте схему драйвера светодиодов и изготовьте по меньшей мере восемь соединительных проводников (**Приложение А**).

3.4. Элементарный вывод через параллельный порт

В главе 2 говорилось, что параллельный порт представлен тремя регистрами ввода/вывода (обычно с адресами 0x378, 0x379, 0x37A). Поэтому под параллельным портом мы понимаем совокупность этих трёх регистров. Регистр по адресу 0x378 наиболее прост. Обычно этот регистр используется только для вывода данных, но старые компьютеры имеют возможность ввода данных через него. Тем не менее, наши программы будут только выводить данные через регистр 0x378 в целях их совместимости. Мы напишем программу, которая будет выводить данные через регистр 0x378 и зажигать тем самым соответствующие светодиоды на интерфейсной плате.

Чтобы проверить работу программы необходимо подключить интерфейсную плату к параллельному порту компьютера. Подключение осуществляется соединительным кабелем, описанным в главе 2. Остальные соединения выполняются согласно **Табл. 3.1**, приведённой ниже. Следует отметить, что первый вывод микросхемы имеет на печатной плате прямоугольную контактную площадку. Соединения выполняются изготовленными ранее проводниками. В результате сделанных соединений линии порта с адресом BASE (0x378) будут подключены через микросхему-буфер к схеме драйвера светодиодов, управляющей ими.

Таблица 3.1. Соединения для осуществления вывода

Регистр BASE (Буфер U13)	Вывод ULN2803A (Драйвер U3)
D0	1
D1	2
D2	3
D3	4
D4	5
D5	6
D6	7
D7	8

Программа на **Листинге 3.1** состоит из нескольких выражений, расположенных после комментариев. В программе использована библиотечная функция `outportb()` для вывода байта в регистр BASE. Адрес регистра и данные передаются в качестве фактических аргументов при вызове функции `outportb()`. Поскольку `outportb()` библиотечная функция, то нам не нужно писать её тело. Оно находится в библиотеке и будет там найдено редактором связей.

Листинг 3.1. Запись в регистр BASE

```

/*****
    ЗАПИСЬ В РЕГИСТР (вывод)

    Эта программа выводит битовые наборы в регистр BASE,
    зажигая соответствующие светодиоды на интерфейсной плате.
*****/

#include <dos.h>

#define BASE 0x378

void main()
{
    outportb(BASE,255); // в двоичном формате 255 = 1111 1111
    // Число 255 можно заменить любым другим целым числом
    // из диапазона 0...255, что приведёт к изменению
    // выходных сигналов, соответствующих двоичному значению
    // числа, например 65 = 0100 0001.
}

```

В разделе 1.2.2 было сказано, что строки, начинающиеся с символа «решётка» “#”, являются директивами компилятора. Первая директива заставляет компилятор просматривать заголовочный файл `dos.h` (являющийся исходным). В этом файле находится прототип функции `outportb()`. В заголовочном файле `dos.h` также содержится информация, касающаяся многих других функций. Но для этой программы нужна информация только о функции `outportb()`. Если этот заголовочный файл не будет включен в программу, то компилятор никак не сможет обработать эту функцию и нельзя будет её использовать в программе. Заметим, что прототип ничего не говорит о том, **как** выполняется функция. Другими словами, до этапа редактирования связей программе недоступны команды функции.

Второй директивой компилятора является выражение `define`. Эта директива заставляет компилятор все выражения `BASE` в программе заменять шестнадцатеричным числом `0x378`. Благодаря `BASE` вместо `0x378` облегчается чтение программы, так как в слове `BASE` больше смысла, чем числе. Если нужно постоянно менять базовый адрес, то нам достаточно изменить выражение `define`, а препроцессор автоматически выполнит все замены `BASE` во всей программе. Директива `define` в больших программах может упростить написание кода и улучшить его «читаемость».

Затем следует функция `main()` (единственная в данной программе), с которой начинают выполняться все программы на C++. Обычно в программе на C++ пи-

шется (определяется) много функций. Служебное слово `void` говорит о том, что функция `main()` ничего не возвращает. Тело функции `main()` начинается открывающейся фигурной скобки `"{"` сразу после строки `void main()`, а за ней располагаются составляющие его команды. Единственным исполнимым выражением в этой программе является `outportb(BASE, 255)`. Это выражение выводит байт данных из компьютера. После нескольких строк комментариев тело функции заканчивается закрывающейся фигурной скобкой `"}"`.

Функция `outportb()` принимает два аргумента: адрес регистра и данные для записи в этот регистр. В этой программе адресом регистра является `BASE`, заменяемый препроцессором на `0x378`. Поэтому данные будут записаны в регистр `0x378`. Данные представлены десятичным числом `255`. Нужно отметить, что в регистры чаще всего записывают данные размером один байт. Байт может принимать 256 различных значений. Когда байт равен 0, то все его биты равны 0. Когда байт равен 255 все биты имеют значение 1. Значения байта между 0 и 255 можно представить соответствующими комбинациями битов. Выполнив соединения по **Табл. 3.1** и подключив к компьютеру интерфейсную плату можно откомпилировать и запустить эту программу. В результате все биты станут равны 1 и это приведёт к зажиганию всех светодиодов. Если, например, вы поменяете строку с функцией `outportb()` на `outportb(BASE, 65)`, то будут светиться только светодиоды битов 0 и 6, а остальные будут погашены.

Если программа не работает и не зажигает светодиоды, то убедитесь в правильности используемого базового адреса при помощи программы `base_adr.exe`, находящейся на прилагаемом компакт-диске.

Несколько раз измените значение 255 в программе на другие числа (меньшими 255) и наблюдайте, как зажигаются светодиоды на интерфейсной плате. Этот опыт поможет понять, как связаны между собой наборы битов и десятичные числа, обсуждавшиеся в главе 2. Вместо числа 255 можно писать шестнадцатеричное число, например:

```
outportb(BASE, 0x0F);
```

3.5. Ввод через параллельный порт

В этом разделе мы узнаем, как написать программу для чтения регистра по адресу `BASE+1 (0x379)`. Это единственный из трёх регистров, предназначенный исключительно для ввода данных. Мы будем читать этот регистр и отображать принятые данные в виде числа на экране.

Повторим, что для проверки программы нужно подключить интерфейсную плату к компьютеру при помощи кабеля. Остается только сделать соединения согласно **Табл. 3.2**. Меняя подключения к шинам питания во втором столбце **Табл. 3.2**, можно легко модифицировать входные данные.

Соединительными проводниками выполните соединения по **Табл. 3.2**. Эти соединения приведут к подаче сигналов с интерфейсной платы в регистр `BASE+1 (0x379)`. Обратите внимание, что в этом регистре сигнальными являются только пять бит из восьми возможных. Биты **D0...D2** не имеют соответствующих сигнальных линий параллельного порта.

Таблица 3.2. Соединения для осуществления ввода

Регистр BASE+1 (Буфер U6)	Шины питания +5 В и Земля
D3	+5 В
D4	Земля
D5	Земля
D6	+5 В
<u>D7</u>	+5 В

Примечание: Надчеркивание означает инверсию бита схемой параллельного порта.

Программа должна выполняться в два этапа:

- 1. Чтение регистра.
- 2. Отображение результата на экране.

Оба этих действия реализуются очень простыми выражениями и не требуют написания дополнительных функций. Таким образом, будет достаточно одной функции `main()`. Для чтения восьми бит данных из регистра мы можем воспользоваться библиотечной функцией `inportb()`. Отображать результат можно библиотечной функцией `printf()`. Может оказаться полезным, если мы будем останавливать программу сразу после отображения результата, чтобы мы успели рассмотреть содержимое экрана. Это особенно необходимо, когда программа разрабатывается в интегрированной среде разработки (IDE). В IDE после завершения работы программы на экране появляется редактор IDE. Это мешает нам наблюдать за экраном во время выполнения программы. Но мы можем оставить видимым экран с результатами, если введём в конец программы ожидание нажатия клавиши. Это можно сделать при помощи библиотечной функции `getch()`. Тогда программа должна содержать следующие друг за другом функции в порядке их перечисления: `inportb()`, `printf()` и `getch()`. Для использования этих функций нужны их прототипы, содержащиеся в заголовочных файлах `dos.h`, `stdio.h` и `conio.h`, соответственно. Программа приведена в **Листинге 3.2**.

Листинг 3.2. Чтение регистра BASE+1

```
/*
*****
ЧТЕНИЕ РЕГИСТРА BASE+1

Эта программа читает регистр BASE+1 и прочитанное
значение печатает на экране. Это значение образовано
пятью сигналами, прочитанными из порта. Биты
0, 1 и 2 содержат неопределённые значения, так как не
относятся к параллельному порту. Биты 3, 4, 5 и 6
читаются как обычно. Бит 7 инвертируется схемой порта.
*****/

#include <dos.h>
#include <conio.h>
```

```
#include <stdio.h>

#define BASE 0x378

void main()
{
    unsigned char InputData; // Объявили тип для
                             // различных входных данных

    InputData = inportb(BASE+1); // Читаем порт BASE+1

    printf("%2X\n", InputData); // Выводим результат на экран
                                // в шестнадцатеричном виде

    getch(); // Ждём нажатия клавиши
}
```

Примечание

Подробное описание библиотечных функций находится в документации пакета разработки C++. Реже описания хранятся в файлах помощи и доступны из меню. В документации для каждой библиотечной функции указан соответствующий ей заголовочный файл. Вы можете определить, какой заголовочный файл нужно подключить в программу, чтобы можно было использовать ту или иную функцию.

В этой программе есть директивы компилятора; три из них являются выражениями `include` и одно выражение `define`. Три выражения `include` подключают заголовочные файлы `dos.h`, `stdio.h` и `conio.h`. Выражение `define` даёт возможность обозначать адрес `0x378` словом `BASE`.

Функция `main()` начинается в строке `void main()`, смысл которой обсуждался выше. Как и все функции, `main()` может иметь аргументы. В данном случае аргументов она не имеет и поэтому после слова `main` следует пара круглых скобок. Это говорит компилятору, что это функция и аргументов она не имеет.

Примером функции, принимающей параметры, является `outportb()` (**Листинг. 3.1**). Она принимает два аргумента – адрес регистра и данные. Поэтому её скобки не пусты.

Далее в программе (после фигурной скобки):

```
unsigned char InputData;
```

Ранее говорилось, что это объявление идентификатора. Компилятору сообщается имя идентификатора (в данном случае `InputData`) и тип данных, которые он представляет (в данном случае `unsigned char`).

Эта переменная объявлена для хранения возвращаемого функцией `inportb()` значения. Это хороший пример для изучения понятия возвращаемого значения функции. Имеющийся в файле `dos.h` прототип функции `inportb()` говорит, что эта функция возвращает значение типа `unsigned char`. Поэтому и переменная

InputData должна объявляться как `unsigned char`, чтобы ей можно было присваивать значение функции `inportb()`.

Следующее выражение в программе:

```
InputData = inportb(BASE+1);
```

Это выражение представляет собой вызов функции `inportb()` с одним параметром, являющимся адресом. Выражение `BASE` эквивалентно `0x378`. Поэтому `BASE+1` будет вычислено как `0x379`. Это адрес регистра, из которого нужно произвести чтение. Поэтому это выражение приведёт к чтению второго регистра `BASE+1`. Прочитанное функцией `inportb()` значение присваивается переменной `InputData`.

Результат чтения отображается следующим образом:

```
printf("%2X\n", InputData);
```

Функция `printf()` относится к языку C, а не C++. Но C++ допускает синтаксис C и поэтому эта функция может использоваться в программах на C++. Функция `printf()` обладает большой гибкостью и простотой использования. Приложения лучше писать на C++, но в наших программах задействованы все лучшие стороны языка C. Функция `printf()` необычна по сравнению с другими и обладает большими возможностями, она может принимать *переменное* количество параметров. Большинство создаваемых вами функций будут иметь *фиксированное* число параметров. В нашем примере функции `printf()` передаётся два параметра. В качестве первого аргумента выступает строка символов `"%2X\n"`, а вторым аргументом является значение переменной `InputData`. Эти два аргумента отделяются друг от друга запятой.

C	Примеры часто используемых спецификаторов формата
<code>%10.3f</code>	Вещественное число, ширина поля 10 символов, 3 десятичных цифры.
<code>%5d</code>	Целое число, ширина поля 5 символов.
<code>%c</code>	Символ.
<code>%s</code>	Строка (последовательность символов, например фраза).
<code>%X</code> или <code>%x</code>	Шестнадцатеричное число: <code>x</code> будет печатать шестнадцатеричные буквы в верхнем регистре, а <code>x</code> в нижнем.

Первый аргумент является спецификатором форматирования при печати значения `InputData` на экране. Символы `"%2X"` означают, что будет использоваться шестнадцатеричный формат с шириной поля в два символа. Возврат каретки и перевод строки обозначается символом `"\n"`, символ `"n"` называется символом новой строки.

Для отображения принятого байта достаточно двух шестнадцатеричных цифр, поскольку каждая шестнадцатеричная цифра представляет собой четыре бита. Поэтому достаточно ширины поля в два символа. После вывода на экран числа курсор будет установлен в начало следующей за числом строки.

Строка

```
getch();
```

Приостанавливает выполнение программы, пока не будет нажата кнопка на клавиатуре, предоставляя нам время для чтения содержимого экрана. Функция `getch()` ожидает ввод символа с клавиатуры, и поэтому программа выполняться не будет до нажатия на клавишу. Нажатие на клавишу приведёт к завершению программы, т. к. `getch()` является последней командой программы.

Работа программы проверяется путём сопоставления выводимого на экран шестнадцатеричного числа со значениями битов, определяемых сигналами с интерфейсной платы. Вы можете менять соединения на интерфейсной плате, отражённые в правом столбце **Табл. 3.2**. Это значит, что сигнальные линии можно подключать или к земле или к шине +5 В. После перезапуска программы на экране должен появиться уже другой результат.

3.6. Коррекция внутренней инверсии

Обратимся к программе из **Листинга 3.2**. Когда мы читаем регистр `BASE+1 (0x379)` один из считываемых с интерфейсной платы сигналов (бит **D7**) инвертируется схемой параллельного порта. Аналогично инвертируются и некоторые сигналы регистра `BASE+2 (0x37A)` при выводе данных через него (биты **D0**, **D1** и **D3**). В этом разделе мы рассмотрим способ коррекции такой инверсии. Коррекция выполняется программно путём инверсии соответствующих битов, восстанавливая тем самым аппаратно инвертированные данные.

3.6.1. Вывод данных

Программа из **Листинга 3.3** записывает данные в регистр `BASE+2`. Обратите внимание, что в этом регистре задействованы только биты 0, 1, 2 и 3; биты 4, 5, 6 и 7 портом не используются. Некоторые из задействованных битов инвертируются схемой параллельного порта, это биты 0, 1 и 3. Поэтому для устранения аппаратной инверсии нам в программе нужно инвертировать биты 0, 1 и 3. Бит 2 параллельным портом не инвертируется и инвертировать его не нужно.

На интерфейсной плате нужно выполнить соединения согласно **Табл. 3.3**.

Таблица 3.3. Подключение светодиодов

Регистр <code>BASE+2</code>	ULN2803A (драйвер U3)
<u>D3</u>	D0 (1)
<u>D4</u>	D1 (2)
<u>D5</u>	D2 (3)
<u>D6</u>	D3 (4)

Примечание: Надчеркивание означает инверсию бита схемой параллельного порта.

Листинг 3.3. Запись в регистр `BASE+2` с коррекцией внутренней инверсии порта

```

/*****
ЗАПИСЬ В РЕГИСТР BASE+2 С КОРРЕКЦИЕЙ ВНУТРЕННЕЙ ИНВЕРСИИ
*****/
```

Эта программа выводит четыре бита в регистр BASE+2, корректируя инверсию битов 0, 1 и 3. Вы можете менять отображаемые интерфейсной платой биты.

```

*****/

#include <dos.h>

#define BASE 0x378

void main()
{
// При выводе данных через регистр BASE+2 его биты 0, 1 и 3
// инвертируются схемой параллельного порта. Инверсию можно
// устранить программно путём выполнения операции "исключающее
// ИЛИ" над выводимыми данными и константой 0x0B (0000 1011).
// Биты 4...7 не имеют никакого значения, т. к. они не
// используются портом.

    outportb(BASE+2, 0x0B ^ 0x0F);

// В двоичном виде 0x0F = 0000 1111.
// Выводимое значение (0x0F) может быть заменено любым
// значением из диапазона 0x00...0x0F. Четыре выходных
// сигнала будут соответствовать битам выводимого числа.
// Например:
// № Бита: 7 6 5 4 3 2 1 0
// 0x0F      0 0 0 0 1 1 1 1
// 0x05      0 0 0 0 0 1 0 1
}

```

В этой программе нужно пояснить только одну строку:

```
outportb(BASE+2, 0x0B ^ 0x0F);
```

Функция `outportb()` служит для записи данных в регистр, как было рассмотрено выше. В данном случае адресом регистра является значение `BASE+2`. Выражение `define` ставит в соответствие `BASE` значение `0x378`. Поэтому первый параметр функции будет эквивалентен `0x378+2` или `0x37A`. Вторым параметром являются данные, выводимые в регистр. Это значение вычисляется следующим образом:

```
0x0B ^ 0x0F
```

Символ `^^` в этом выражении является оператором «исключающее ИЛИ» (XOR). Это один из битовых операторов языков C и C++, позволяющих работать с отдельными битами. Вы лучше поймёте работу битовых операторов на примере оператора «исключающее ИЛИ» в **Табл. 3.5**. Действие оператора поясняется при помощи **Табл. 3.4**. Оператору нужны два операнда.

Таблица 3.4. Оператор «исключающее ИЛИ»

Операнд А	0	0	1	1
Операнд В	0	1	0	1
Результат	0	1	1	0

Примечание

В обычной операции сложения:

3+5

оператором является «+», а операндами числа 3 и 5.

У битового оператора оба операнда должны быть *битами*. В **Табл. 3.4** два операнда **А** и **В**. Результат операции XOR для всех четырёх комбинаций значений операндов приведён в строке **Результат**.

Как видно из таблицы, 1 получается только в том случае, когда операнды **А** и **В** не одинаковы. Если оба операнда в колонке одинаковы, то в результате получается 0. В колонках 2 и 4 **Табл. 3.4** мы видим, что если операнд **В** равен 1, то результат будет инверсией операнда **А**. Операнд **В** является «фильтром», выделяющий биты, подлежащие инверсии в операнде **А**.

Мы будем выполнять программную инверсию для устранения аппаратной инверсии параллельного порта. В **Табл. 3.5** приведён пример выражения:

$0x0B \wedge 0x0F$

Здесь операнд **А** содержит данные для передачи, а операнд **В** является «фильтром» инверсии битов, которые инвертируются также параллельным портом.

Таблица 3.5. Вычисление $0x0B \wedge 0x0F$

№ бита	7	6	5	4	3	2	1	0
Операнд А (0x0F)	0	0	0	0	1	1	1	1
Действие	XOR							
Операнд В (0x0B)	0	0	0	0	1	0	1	1
Результат	0	0	0	0	0	1	0	0

Как говорилось выше, битовые операторы работают с отдельными битами. Другими словами бит 0 операнда **А** и бит 0 операнда **В** образуют соответствующий бит результата. Аналогично операция «исключающее ИЛИ» выполняется над битом 1 операндов **А** и **В**, и так далее.

В фильтре установлены биты, соответствующие тем битам данных, которые нужно инвертировать, а соответствующие неизменяемым битам данных равны 0. Поэтому для инверсии битов 0, 1 и 3 в фильтре должны быть установлены в 1 биты 0, 1 и 3. В строке **Результат** **Табл. 3.5** что биты 0, 1 и 3 имеют значения, противополо-

ложные значениям битов операнда **A**. Таким образом, когда подлежащие выводу данные являются операндом **A**, то заданные биты инвертируются в программе, образуя **Результат**. Когда эти инвертированные биты попадут в порт и там аппаратно инвертированы, то пришедшие на интерфейсную плату данные будут соответствовать операнду **A**, которые и требовалось передать.

Для проверки работы программы вы можете изменять данные (т. е. 0x0F) от 0x00 до 0x0F. Светящиеся светодиоды будут соответствовать состоянию битов в передаваемом значении. Заметим, что значение-фильтр менять не нужно, в противном случае будут инвертироваться не все из нужных битов.

3.6.2. Работа в качестве входа

В программе из **Листинга 3.2** при чтении регистра **BASE+1** один из сигналов инвертируется параллельным портом. Регистр **BASE+1** служит только для ввода битов с номерами 3, 4, 5, 6 и 7. Из них бит 7 претерпевает инверсию. Эта инверсия корректируется программно при помощи тех же действий, что и предыдущем разделе. Выполняется операция «исключающее ИЛИ» над фильтром, инвертирующим бит 7, и данными порта. В качестве фильтра используется значение:

0x80 = 1 0 0 0 0 0 0 0

Требуемая операция приведена в **Табл. 3.6**. Неиспользуемые и поэтому недействительные биты данных **D0**, **D1** и **D2** обозначены как «X». Эти биты могут иметь любые состояния и поэтому результат операции «исключающее ИЛИ» будет для соответствующих битов также неопределённым.

Таблица 3.6. Вычисление 0x0B ^ 0x0F

№ бита	7	6	5	4	3	2	1	0
Операнд A (входные данные)	1	0	1	1	1	X	X	X1
Действие	XOR							
Операнд B (фильтр)	1	0	0	0	0	0	0	0
Результат (корректированные данные)	0	0	1	1	1	X	X	X

Листинг 3.4. Чтение регистра BASE+1 с коррекцией аппаратной инверсии

```

/*****
    ЧТЕНИЕ РЕГИСТРА BASE+1 С КОРРЕКЦИЕЙ АППАРАТНОЙ ИНВЕРСИИ

    Эта программа читает данные из регистра BASE+1
    (0x379). После чтения данных выполняется
    коррекция аппаратной инверсии бита 7. Получается,
    что инверсии как бы не было.
*****/

#include <dos.h>
#include <conio.h>
```



```
#include <stdio.h>

#define BASE 0x378

void main()
{
    unsigned char InputPort1;

    InputPort1 = inportb(BASE+1);
    InputPort1 ^= 0x80;
    printf ("%2X\n", InputPort1);
    getch();
}
```

В **Листинге 3.4** нужно пояснить только одну строку:

```
InputPort1 ^= 0x80;
```

Оно эквивалентно следующему выражению:

```
InputPort1 = InputPort1 ^ 0x80;
```

И соответствует виду:

Результат = Операнд А ^ Фильтр;

Операнд **А** соответствует искажённым инверсией данным из порта, а **Результат** откорректированным данным. Рассмотрим выражение:

```
InputPort1 = InputPort1 ^ 0x80;
```

InputPort1 в правой части выражения содержит искажённые аппаратной инверсией данные. InputPort1 в левой части выражения принимает результат операции «исключающее ИЛИ» между искажёнными данными и значением фильтра 0x80. Другими словами, значение InputPort1, как говорят, «XOR-ится» со значением фильтра, а затем результат записывается обратно в переменную InputPort1. Выражение printf() печатает откорректированное значение на экране. В результате число на экране должно соответствовать сигналам на интерфейсной плате.

3.7. Заключение

В этой главе мы рассмотрели работу источника питания интерфейсной платы, интерфейс порта и драйвера светодиодов. Эти схемы позволяют параллельному порту компьютера взаимодействовать с интерфейсной платой и проверять работу программ.

Мы научились писать программы на C++ для приёма и передачи данных через три порта параллельного порта компьютера. Эти программы выводят на экран результаты либо при помощи объекта cout (как в главе 1), либо функциями семейства printf(). Также было приведено несколько спецификаторов формата для функции printf().

Для инверсии некоторых битов данных порта используется битовая операция «исключающее ИЛИ». Битовые операторы являются очень полезной частью языков С и С++, позволяя выполнять операции над отдельными битами байта.

3.8. Литература

1. Bergsman, P., *Controlling The World With Your PC*, High Text Publications, San Diego, 1994.
2. IBM, Technical Reference – Personal Computer AT, IBM Corporation, 1985.
3. NS LINEAR *Databook*, National Semiconductor Corporation, 1987.
4. Savant, C. J., et al., *Electronic Design Circuits and Systems*, Second Edition, Benjamin Cummins, 1995.
5. House, R., *Beginning with C – An Introduction to Professional Programming*, International Thompson Publishing, 1994.
6. Deitel H. M. and P. J. Deitel *C: How to Program*, Prentice Hall, 1994.
7. *C Programming Language – an applied perspective* by L Miller and A Quilici, John Wiley Publishing, 1987.
8. Hanly, J. R., E. B. Koffman and J. C. Horvath, *C Program Design for Engineers*, Addison Wesley, 1995.
9. Rudd, A., *Mastering C*, John Wiley, 1994.

4 Объектно-ориентированное программирование

Содержание главы:

- Что означает объектно-ориентированное программирование (ООП)?
- Инкапсуляция.
- Члены-данные и члены-функции объекта.
- Наследование и полиморфизм.
- Конструкторы и деструкторы.
- Абстрактные классы.
- Иерархии классов.

4.1. Введение

В этой главе мы рассмотрим основы объектно-ориентированного программирования в C++. Объектно-ориентированное программирование это новый способ создания программ. Это следующий уровень после *процедурного программирования*.

В процедурном программировании о данных и функциях можно сказать, что они «находятся в едином пространстве». Это означает, что данные могут быть использованы и/или изменены любой функцией; распространённой ошибкой является непреднамеренное использование их не по назначению, что приводит к неприятным эффектам во время выполнения программы. При доработке или модификации существующих программ могут возникнуть трудноустраняемые неполадки.

В этой главе на примерах из повседневной жизни изучаются понятия объектно-ориентированного программирования на качественном уровне. Затем эти понятия определяются в терминах объектно-ориентированного программирования, которые в свою очередь будут подробно истолкованы. После прочтения этой главы у вас должно сложиться хорошее понимание основных идей и терминов объектно-ориентированного программирования в следующей главе.

4.2. Воображаемые и реальные объекты

Объекты в объектно-ориентированном программировании устроены подобно объектам, с которыми мы имеем дело в жизни. Объект можно представить как самодостаточную вещь, имеющую некое описание и определённую функциональность. Описание может быть перечислением всех свойств объекта. Функциональность может определяться набором действий, которые объект выполняет для нас и/или мы сами выполняем над объектом. *Тип объекта* в терминах объектно-ориенти-

ванного программирования определяется *классом объекта*. В языке C++ можно перечислить все качества или свойства класса объекта и все его функции.

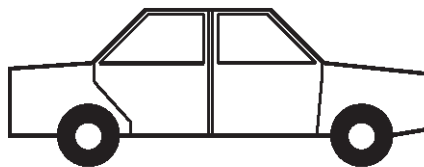
В реальной жизни мы можем описывать как реально существующие объекты, так и абстрактные, воображаемые или придумываемые нами. Реальные объекты ощущаемы нами, а воображаемые нет. В то время как ощущаемые объекты могут иметь дубликаты, умозрительные объекты продублировать нельзя, поскольку не бывает такого явления, как «два одинаковых понятия». Воображаемый объект (абстрактный объект) можно выразить в виде физически реализуемого объекта, точно определив все присущие ему черты, после чего может быть создан реально существующий объект. Таким образом, объект из абстрактного преобразуется в реальный.

Для лучшего усвоения этого понятия в качестве объекта рассмотрим *транспорт*. Очевидно, что транспортом является некое средство для перевозки товаров и людей,двигающееся при помощи колёс и приводимое в движение каким-либо видом энергии (**Рис. 4.1**). Производитель не может начать изготовление транспорта, пока не будут определены его характерные черты. Будет транспортное средство поездом, автобусом, машиной или чем-то ещё? Если машина, то это маленькая, средняя или большая машина? Каковы тип двигателя и его мощность? Сколько дверей должно быть? Такое уточнение производится до тех пор, пока не будут определены все детали, необходимые для изготовления машины. На этом этапе объект уже не является абстрактным. После определения всех свойств можно изготовить любое количество «настоящих» машин, являющихся реальными объектами.



АБСТРАКТНЫЙ ОБЪЕКТ – ТРАНСПОРТ

(а)



РЕАЛЬНЫЙ ОБЪЕКТ – ТРАНСПОРТ

(б)

Рис. 4.1. Понятия абстрактного и реального объектов

4.3. Реальные объекты

Хотя машины одного класса одинаковы после их изготовления, они всё же являются отдельными вещами. У каждой из машин будет свой собственный двигатель, топливная система, тормозная система и т. д. Если кому-то понадобится «Запустить двигатель!», то этого нельзя будет сделать до тех пор, пока не станет

известно, какую машину нужно завести. Соответственно, слово «двигатель» не означает «вот этот двигатель». Нужно сопоставить двигатель какой-либо машине, чтобы его можно было заводить. Предположим, что изготовлены три машины, обозначенные **А**, **В** и **С**. Тогда фраза «двигатель машины **А**» будет однозначно идентифицировать конкретный двигатель. Поэтому, если «автомобиль» это тип объекта, то «автомобиль **А**» является реальным объектом. Важно понимать различие между типом объекта и реальным объектом. Существование типа объекта ещё не означает наличия реального объекта этого типа. С другой стороны, объект не может не обладать типом.

Такие элементы, как двигатель, топливная система, тормозная система и т. д. могут служить их общим описанием. Например, чтобы описать свойства двигателя в машине какого-либо *типа*, мы не будем говорить, что «синяя машина обладает двигателем такими-то качествами», а скажем так: «машины этого типа обладают двигателем с такими свойствами». С другой стороны, если в машине сломается двигатель, то нам нужно выделить эту машину среди прочих фразой: «двигатель в *синей* машине» сломан и его нужно чинить. Поэтому, если известен тип объекта, то мы можем пользоваться его отдельными элементами, не обращаясь к конкретному объекту.

4.3.1. Внешний интерфейс объекта

Машина была сложным объектом даже в прежние времена. В машине есть узлы, в работе которых не требуется участие пользователя; система впрыска топлива, например. Пользователь не вмешивается напрямую в систему впрыска топлива, но она всё же скрыто функционирует. С другой стороны, водитель имеет доступ к рулю, педали тормоза, приборной панели и т. д. Это можно назвать внешним *интерфейсом* объекта «машина». Так же и в объектно-ориентированном программировании каждый объект должен обладать внешним интерфейсом, чтобы им можно было пользоваться. Объект без внешнего интерфейса похож на отлично изготовленную машину, полностью закрытую и запаянную так, что никто не может проникнуть внутрь, чтобы управлять ей.

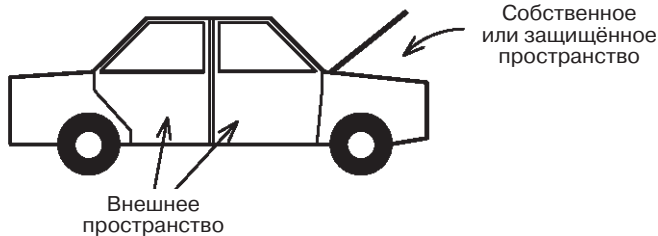


Рис. 4.2. Внешний интерфейс объекта

Программные объекты могут обладать собственными функциями для выполнения определённых действий, которые не выполняются непосредственно пользователем (подобно системе впрыска топлива в автомобиле). Поэтому в общем случае объект является закрытой сущностью с внешним интерфейсом, обеспечивающим доступ к его скрытым элементам. Программная реализация объектов станет понятной, когда мы разберёмся с созданием классов объектов в последующих главах.

4.3.2. Создание и уничтожение объекта

Все машины появляются в результате некоторого производственного процесса, который можно назвать «создающим процессом». При необходимости могут быть незначительные различия между машинами одной и той же конструкции – например, у машин может быть разный цвет. Даже сам производственный процесс может немного изменяться. На одном заводе машины могут собираться из готовых узлов, а на другом «с нуля». Но в обоих случаях будут изготавливаться машины одинакового *типа*.

После того, как машины «увидели свет», ими уже можно пользоваться. В конце срока службы машина должна пройти некий разрушающий процесс, имеющий место в пункте утилизации машин. Разрушение – это важный процесс, уничтожающий ненужные элементы в целях поддержания чистоты окружающей среды и эффективного управления ресурсами. В случае компьютерных программ уничтожение объектов необходимо для рационального использования памяти.

В объектно-ориентированном программировании существуют способы реализации этих аспектов. Хотя мы и можем провести аналогию между объектами из нашей повседневной жизни и программными объектами, всё же истинная мощь объектно-ориентированного программирования на C++ заключается в сочетании методов объектно-ориентированного программирования и приёмов программирования на C++.

4.4. Классы объектов

Класс объекта описывает некий *тип* объекта. Класс объекта не является самим объектом, но он тот тип, который имеют все создаваемые от него объекты. Если привести пример из жизни, то аналогом класса являются строительные чертежи здания, а не само здание. По строительным чертежам можно возвести любое количество зданий. Подобным образом и класс объекта может использоваться для создания любого количества объектов. Каждый объект, создаваемый в соответствии с его классом, размещается в памяти компьютера. Чем больше однотипных объектов создано, тем больший объём памяти они занимают.

У каждого класса должно быть имя, также называемое именем *типа* объекта. Слово `class` является служебным словом в C++. Оно служит для обозначения как бы подробного плана, у которого есть имя. Программисты сами придумывают имена классов при создании новых классов.

У каждого класса должно быть *определение класса*. Как и в случае объектов окружающего мира, свойства (качества) объекта и связанные с ним функции приводятся в определении класса (Рис. 4.3). Свойства (или качества) объекта называются *данными-членами* класса, а связанные с объектом функции называются *функциями-членами*.

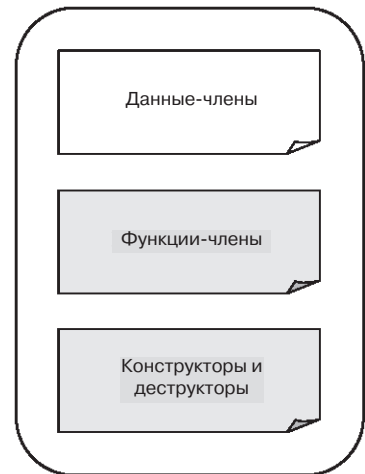


Рис. 4.3. Компоненты класса объекта

Наряду с функциями-членами существует два особых вида функций. Функций первого типа может быть несколько в классе и они называются *конструкторами*. Конструкторы служат для создания каждого отдельного объекта, размещаемого в памяти. Функция второго типа у каждого класса только одна и называется *деструктором*. Деструкторы предназначены для освобождения памяти, занятой данными, когда эти данные больше не нужны. Если конструктор не написан программистом, то компилятор автоматически создаст конструктор (невидимый для программиста). Аналогично, если не написан деструктор, то компилятор автоматически создаёт деструктор (также невидимый программисту).

Некоторые члены класса (и данные, и функции) могут быть доступными извне; к другим членам доступ ограничен. Доступные извне члены образуют внешний интерфейс объекта (**Рис. 4.4**). Он состоит из данных-членов и функций-членов. Члены с ограниченным доступом обладают атрибутами «собственный» или «защищённый».

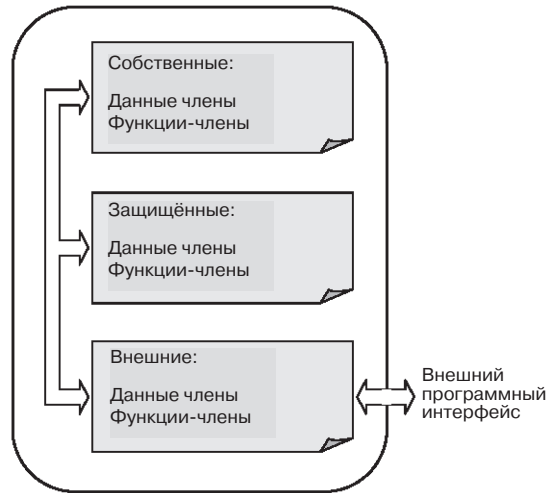


Рис. 4.4. Внешний интерфейс класса

4.5. Инкапсуляция

Объединение данных-членов и функций-членов в одно целое называется *инкапсуляцией*. Инкапсуляция обладает рядом достоинств. Прежде всего, инкапсуляция ограничивает доступ к внутренним элементам объекта. Доступ к данным является управляемым и контролируемым. Доступ ограничивается обычно к тем данным, которые не должны быть непосредственно доступны или изменяемыми. Доступ может ограничиваться и к некоторым функциям.

Взаимодействие с объектом возможно только посредством его внешнего интерфейса. Такая «ограниченная видимость» данных-членов и функций-членов известна также как скрытие данных. Иногда не нужно знать внутренние особенности объекта. В таких случаях важно знать только как пользоваться объектом. Хорошим примером является полоса прокрутки. Программисту нужно знать только способ её использования в программе для прокрутки экрана. Внутреннее устройство полосы прокрутки скрыто от программиста и он об этом не задумывается.

В то время как на внешние по отношению к классу функции накладываются ограничения доступа, любая функция-член имеет доступ к любому члену этого класса. Это приводит к эффективной работе объектно-ориентированных программ по следующим причинам. Отсутствует необходимость объявления дан-

ных членов внутри функции-члена, а также не нужно передавать данные-члены в функции-члены в виде аргументов. Обычная функция, не являющаяся членом класса, может возвращать только одно значение, тогда как функция-член может «возвращать» больше путём изменения любого количества данных-членов класса.

4.5.1. Инстанцирование объектов

Процесс создания объекта называется *инстанцированием* объекта. Для инстанцирования объекта какого-либо класса нужно вызвать какой-нибудь конструктор этого класса. Конструктор играет роль «производственного процесса» в нашей «машинной» аналогии. Конструктор, как и любая функция, является написанной на C++ функцией. Его нужно вызывать для создания объекта. Если созданы объекты класса типа «машина», то мы можем их назвать **A**, **B**, **C** и т. д. Они все равноценны, обладая лишь незначительными различиями, например в цвете. В объектно-ориентированном программировании класс не занимает памяти. Точно так же и строительные чертежи не занимают участка земли, в то время как зданиям нужна конечная площадь.

В объектно-ориентированном программировании для создания объекта нужно вызывать конструктор. При вызове конструктора ему можно передать параметры, подобно выбору цвета для изготавливаемой машины. Можно определить класс, используя несколько конструкторов. Аналогично, машину можно собрать из готовых узлов, а можно и «с нуля».

4.6. Абстрактные классы

Абстрактные классы представляют собой начальное смысловое определение класса, которого однако недостаточно для инстанцирования объекта. Большинство удачных иерархий классов берут начало от абстрактного класса. Все достоинства абстрактного класса не могут быть должным образом продемонстрированы, пока не будут объяснены некоторые характерные для C++ понятия в следующих главах.

Абстрактный класс подлежит доработке, в результате которой получается «настоящий» класс, с работающими функциями, в возможностью инстанцирования полнофункциональных объектов. Не существует «абстрактного объекта» по той простой причине, что ничего абстрактного физически не существует. В нашем примере с транспортом его можно характеризовать такой величиной, как скорость. Но мы не можем говорить о том, как увеличить или уменьшить скорость, пока не получим дополнительных сведений о конкретном транспортном средстве. Для увеличения и уменьшения скорости мы можем ввести функции. Мы не сможем сказать, какие действия должна предпринимать функция для увеличения или уменьшения скорости, пока не будем знать, что это за транспорт – автомобиль, поезд или что-то другое. Аналогично и в определении класса такие функции называются «чистыми виртуальными» функциями. «Чистые» потому, что их действия (тела) на данный момент не определены, а *виртуальные* из-за необходимости определить их тела в производных классах, чтобы они могли выполнять свои задачи.

4.7. Иерархии классов

Иерархия классов это набор классов, созданных на базе одного или нескольких классов, называющихся *базовыми классами* (они являются корневым элементом иерархии). Классы, созданные на основе базовых классов, называются *производными классами*. Производные классы являются более полными и детальными по сравнению с их базовыми классами. Одним из самых важных достоинств объектно-ориентированного программирования является *многократное использование кода*. Создание иерархий классов способствует многократному использованию кода, поскольку не переписывается базовый код. Объектно-ориентированное программирование позволяет повторно использовать уже написанный код столько, сколько нужно.

Базовый класс может быть абстрактным классом, но это не является обязательным условием. Как говорилось выше, от абстрактного класса не может быть инстанцирован объект. Тем не менее, разработку иерархии классов лучше всего начинать с абстрактного класса, состоящий из основных данных-членов и функций-членов, которые должны быть у всех объектов производных классов иерархии.

В качестве примера рассмотрим графическую программу, работающую с такими геометрическими телами, как линии, окружности, треугольники, квадраты и т. д.: можно определить базовый класс для перемещения, масштабирования, стирания, рисования, деформирования *всех* этих объектов. Давайте назовём этот класс `Shape`. Создаваемый нами класс, не подразумевающий какой-либо конкретный объект, может быть абстрактным классом. Мы уже говорили, что базовый класс не обязательно должен быть абстрактным. Абстрактный класс должен определять функциональность всех объектов иерархии, не конкретизируя сам объект.

Производные классы, в отличие от базового класса, должны быть более подробными. Например, можно создать производный класс, предназначенный для работы с окружностью. Давайте назовём его `Circles`. Тогда такие функции класса `Circles`, как `Move`, `Scale`, `Hide`, `Show` и `Stretch` пишутся специально для работы с объектом «окружность». Вы можете себе представить, насколько трудно перенести всю функциональность класса `Circles` в новый класс `Squares`, предназначенный для работы с квадратами! Нужно сделать массу изменений, чтобы вместо окружностей можно было работать с квадратами. Проще напрямую от класса `Shape` создать производный класс `Squares`, а затем дать определения необходимым для класса `Squares` функциям. Хотя от абстрактного класса и нельзя инстанцировать объект, он обладает очень полезными возможностями, выражающимися в *виртуальных функциях* и *позднем связывании*. Эти два понятия будут обсуждаться в главе 8.

4.8. Наследование

Наследование тесно связано с иерархией классов, когда от базового класса создаётся производный класс, говорят, что новый класс *наследует* все данные-члены и все функции-члены от базового класса. Этот механизм лежит в основе многократного использования кода. Функции и данные базового класса автоматически переходят в производный класс. Их не нужно писать заново. Нам нужно только

добавить дополнительные данные-члены и функции-члены к новому классу, расширяющие его функциональность по сравнению с его базовым классом. Можно также изменить унаследованные функции в соответствии с возлагаемыми на них задачами. Но может возникнуть необходимость ограничения доступа к некоторым унаследованным данным-членам и функциям-членам.

В качестве примера рассмотрим ранее упоминавшуюся графическую программу (**Рис. 4.5**). Мы обсуждали абстрактный класс `Shape`. В этом классе были такие графические функции, как `Move` (перемещение), `Scale` (масштабирование), `Hide` (скрыть), `Show` (показать) и `Stretch` (растянуть) и т. д. Если нам нужен класс для работы с окружностями, то мы можем от базового класса `Shape` создать производный класс `Circles`. Говорят, что производный класс `Circles` является подклассом абстрактного класса `Shape` и наследует все его функции-члены (данные-члены, если они есть). В классе `Shape` нет характерных для окружности свойств, таких как координаты центра и радиус. Эти свойства добавляются в качестве новых данных-членов в производный класс.

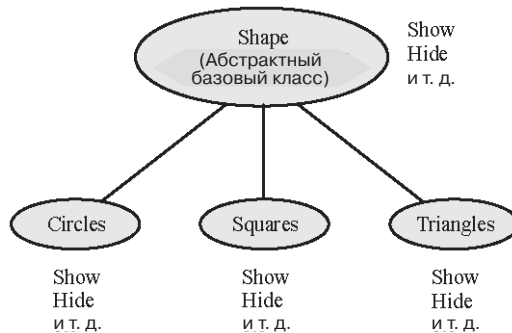


Рис. 4.5. Наследование от абстрактного класса

4.9. Множественное наследование

Производный класс может иметь несколько базовых классов. В этом случае производный класс наследует все данные-члены и все функции-члены от всех базовых классов. Например, можно создать класс `Colours`, позволяющий устанавливать цвета объектов и фона, выполняя заливку. Из классов `Shape` и `Colours` мы можем образовать новый производный класс `Circles` (**Рис. 4.6**). Теперь класс `Circles` способен рисовать на экране цветные окружности!

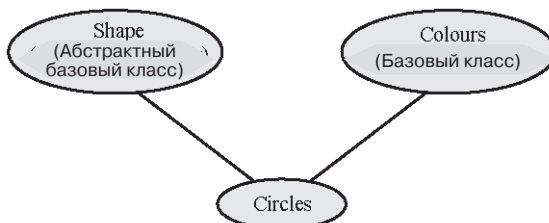


Рис. 4.6. Множественное наследование

4.10. Полиморфизм

В принципе, все понятия объектно-ориентированного программирования основаны на механизмах инкапсуляции, наследования и *полиморфизма*. Полиморфизм является более сложным понятием, чем уже кратко описанные инкапсуляция и наследование. В объектно-ориентированном программировании полиморфизм означает возможность существования функций с одинаковыми именами, одинаковыми типами возвращаемых значений, одинаковым количеством и типами параметров в классах одной иерархии. Тела этих функций разные для удовлетворения нужд каждого класса. Не обязательно каждый класс иерархии должен обладать такой функцией. В одной иерархии может быть любое количество полиморфных функций, полиморфизм обеспечивает единый интерфейс для выполнения действий одного рода. Сильной стороной объектно-ориентированного программирования является полиморфизм виртуальных функций (речь о них пойдёт в главе 8).

Давайте разберём пример с полиморфными функциями. Поскольку полиморфная функция «одинакова» во всей иерархии, может сложиться впечатление, что она выполняет одни и те же действия в каждом классе. Возвращаясь к нашему графическому примеру, мы можем в качестве полиморфной функции рассматривать функцию `Show()`. Функция `Show()` принадлежит классам `Shape`, `Circles`, `Squares` и т. д. Другой полиморфной функцией является функция `hide()`. Предположим, что `Show()` показывает объект на экране. Если в объекте `Circle` функция `Show()` рисует окружность, то во объекте `Square` она будет рисовать квадрат. Поэтому, несмотря на одинаковые названия функций, работают они по-разному, в зависимости от типа объекта.

Полиморфизм является средством, позволяющим в полной мере воспользоваться преимуществами виртуальных функций. Программист может использовать в программе виртуальные функции, не зная при этом, какому объекту они будут принадлежать во время выполнения программы. Программа пишется в общем виде, отвечая требованиям всех классов иерархии. Программисту не нужно создавать сложную логику выбора соответствующей объекту функции, когда пользователь выбирает объект во время выполнения программы. Эта задача возлагается на компилятор и редактор связей, что значительно облегчает программирование. Это особенно актуально для программ с большим количеством классов и сложными иерархиями.

Например, программист может написать типичную программу, где виртуальная функция `Show()` в иерархии используется для отображения объекта на экране. Пользователь выбирает во время выполнения программы объект, с которым должна работать функция `Show()`. Не нужна сложная логика определения типа объекта класса `Shape()`, выбираемого пользователем в произвольный момент времени. Соответствующая типу объекта функция выбирается автоматически во время выполнения программы. Это будет в полной мере продемонстрировано в главе 8.

4.11. Пример иерархии объектов

Что помочь вам усвоить вышеописанные понятия, создадим иерархию объектов, не прибегая к синтаксису C++ и его служебным словам. Хотя нижеприведённые

определения классов нельзя откомпилировать в составе реальной программы на C++, они иллюстрируют принципы иерархии объектов. Начнём с класса `Vehicle` (Транспорт), оспуждавшийся ранее:

Абстрактный класс **Транспорт**

Данные-члены:

Скорость

Мощность

Функции-члены:

Стоп

Поехали

Этот класс является самым главным для всех остальных классов в этом примере. Он инкапсулирует самые характерные элементы транспорта. Среди них должны быть Мощность, обуславливающая его способность к движению, и Скорость, характеризующая это движение. Функция-член Поехали будет заставлять транспорт двигаться, а другая функция-член Стоп будет его останавливать.

От класса Транспорт путём добавления специфичных деталей можно образовать новые производные классы. Таким образом мы создадим два новых класса Пассажирский Транспорт и Грузовой Транспорт. Эта иерархия классов показана на **Рис. 4.7** в виде двух ответвлений от корневого элемента (базового класса). Определение класса выглядит так:

Производный класс **Пассажирский Транспорт**: произведён от класса **Транспорт**

Дополнительные данные-члены:

Количество пассажиров

Производный класс **Грузовой Транспорт**: произведён от класса **Транспорт**

Дополнительные данные-члены:

Допустимая нагрузка в килограммах

Следует заметить, что оба класса наследуют все данные-члены и функции-члены базового класса Транспорт. Если, например, мы перечислили все элементы класса Пассажирский Транспорт, то получили бы следующее определение класса:

Производный класс **Пассажирский Транспорт**:

Данные-члены:

Скорость

Мощность

Количество пассажиров

Функции-члены:

Стоп

Поехали

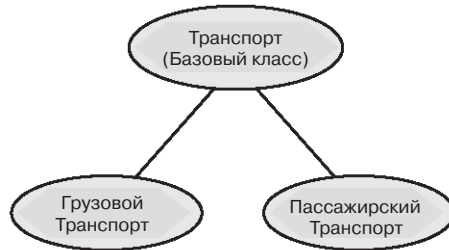


Рис. 4.7. Создание классов на основе базовых классов

Хоть мы и не упоминали члены *Скорость*, *Мощность*, *Стоп*, *Поехали* они всё же появились в результате наследования. Класс *Грузовой Транспорт* является прямым наследником класса *Транспорт* и поэтому ничего не наследует от класса *Пассажирский Транспорт*.

Таким образом, иерархия представлена двумя ответвлениями от корневого объекта, как показано на **Рис. 4.7**. Классы *Пассажирский Транспорт* и *Грузовой Транспорт* могут быть, а могут и не быть абстрактными классами. От любого из этих классов можно создать производный класс, вводя дополнительные уточняющие элементы.

Следующее определение класса *Пассажирский Поезд* является производным от базового класса *Пассажирский Транспорт*:

Производный класс **Пассажирский Поезд**: произведён от класса **Пассажирский Транспорт**

Дополнительные данные-члены:

Количество пассажирских вагонов

Дополнительные функции-члены:

Открыть двери

Закрыть двери

Кондиционирование воздуха

В этом определении члены класса *Пассажирский Поезд* добавлены к уже существующим членам класса *Пассажирский Транспорт*, которые были унаследованы (они не видны). Полное определение класса *Пассажирский Поезд*, если бы мы могли видеть его, выглядело бы следующим образом (унаследованные члены показаны *жирным курсивом*):

Производный класс **Пассажирский Поезд**

Данные-члены:

Скорость

Мощность

Количество пассажиров

Количество пассажирских вагонов

Функции-члены:

Стоп

Поехали

Открыть двери
Закрыть двери
Кондиционирование воздуха

После того, как нами в качестве базового класса был использован *Пассажирский Транспорт*, мы можем создать производный класс от класса *Грузовой Транспорт*. Приведём пример нового класса *Товарный Поезд*, произведённого от класса *Грузовой Транспорт*:

Производный класс **Товарный Поезд**: произведён от класса **Грузовой Транспорт**

Дополнительные данные-члены:
Количество крытых вагонов
Количество цистерн
Количество платформ

Автомобиль обычно рассматривается как средство для перевозки пассажиров. Поэтому при необходимости создать класс, представляющий автомобили, нам лучше всего начать с класса *Пассажирский Транспорт*. Дадим пример определения нового класса *Автомобиль*:

Производный класс **Автомобиль**: произведён от класса **Пассажирский Транспорт**

Дополнительные данные-члены:
Объём двигателя
Цвет кузова
Цвет внутренней отделки
Количество цилиндров
Размер колёс
Количество дверей

Дополнительные функции-члены:
Рулить
Тормозить

Если нам после этого понадобится определить класс, представляющий престижные автомобили, то лучшей отправной точкой будет класс *Автомобиль*. Класс *Автомобиль* выбран вместо класса *Пассажирский Транспорт* потому, что объекты класса *Автомобиль* более похожи на престижные автомобили. Если мы будем создавать класс *Престижные Автомобили* на базе класса *Пассажирский Транспорт*, то нам придётся заново вводить такие члены, как *Объём двигателя*, *Цвет кузова*, *Цвет внутренней отделки* и т. д. Для этого придётся совершать ненужную работу, есть риск внести ошибку и не реализуется возможность многократного использования кода.

Производный класс **Престижный Автомобиль**: произведён от класса **Автомобиль**

Дополнительные данные-члены:

Температура воздуха в салоне
Географические координаты автомобиля

Дополнительные функции-члены:
Кондиционирование воздуха
Регулировка зеркал
Стеклоподъемник
Круиз-контроль
Управление антенной
Управление CD-проигрывателем

В иерархии классов изменения могут быть сделаны с минимальной модернизацией программы. Если изменения очень специфичны, то их лучше сделать в поздних производных классах. Если изменения имеют общий характер, то их лучше сделать вблизи корня иерархии. Например, если все престижные машины в будущем будут оборудоваться системами автоматического управления, то нам нужно в класс Престижный Автомобиль добавить новую функцию-член Следовать по маршруту. Напротив, если все транспортные средства будут оснащаться средствами автоматического управления, то функцию Следовать по маршруту нужно внести в класс Транспорт.

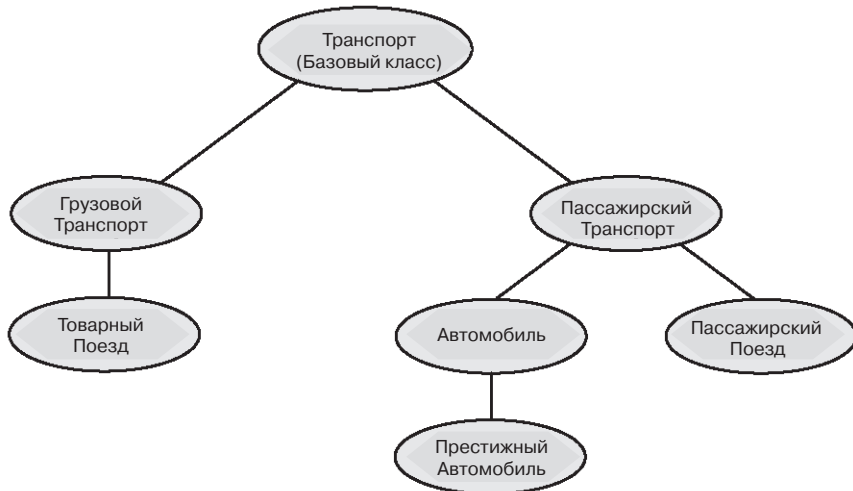


Рис. 4.8. Пример иерархии классов

Набор классов, образующих иерархию, всегда имеет *аддитивный* характер, а не *мультипликативный*. Аддитивный выражается в добавлении дополнительных данных-членов и функций-членов к производным классам. Мультипликативность подразумевает дублирование однотипных объектов в составе новых классов.

Например, класс, представляющий дом, и класс, представляющий более совершенный дом, образуют иерархию. Напротив, класс, представляющий дом, и класс, представляющий несколько домов предыдущего класса, иерархией не являются. В новых классах должны быть новые члены, отличающиеся от имеющихся в базовых классах.

Теперь хоть и понятно, как образуются классы, всё же пока не ясно, как с ними работать. В следующих главах мы будем создавать классы и начнём работать с ними. Пока нам достаточно знать, что такое иерархия классов и как она формируется.

4.12. Преимущества объектно-ориентированного программирования

Большая часть программирующего сообщества уже приняло объектно-ориентированное программирование как более совершенный способ программирования. Одним из его главных преимуществ является высокая надёжность программ, что является прямым следствием инкапсуляции. Изменения в классе не затрагивают других частей программы. А внутренние элементы класса хорошо изолированы от внешнего мира. Это значительно облегчает сопровождение программ. Если в дальнейшем понадобится улучшить функциональность класса, то связанное с этим программирование затронет только сам класс и никак не отразится в классах, непричастных к вносимым изменениям. Изменения в классе необязательно приведут к изменению внешнего интерфейса объекта. В качестве примера из жизни рассмотрим автомобиль: сейчас водители управляют автомобилем так же, как и раньше. Но топливная система претерпела изменения от карбюраторной до инжекторной. Техническая реализация объекта улучшилась, а автомобиль по-прежнему управляется по-прежнему посредством внешнего интерфейса (педали газа).

Наследование позволяет многократно использовать код. Сокращается время программирования и отладки. Это позволяет быстрее выводить программу на рынок и уменьшает стоимость разработки. Естественная связь между объектами из повседневной жизни и программных объектов упрощает понимание классов. Благодаря такой связи в этой главе было дано интуитивно-понятное определение класса.

Самое полезное и самое мощное средство объектно-ориентированного программирования заключается в виртуальных функциях и иерархиях объектов. Виртуальные функции позволяют программам выбирать подходящую функцию для выполнения действий над объектом, который указывается во время выполнения программы. Это ограждает программиста от написания длинных программ, призванных обслуживать отдельные объекты, выбираемые пользователем во время выполнения программ.

4.13. Недостатки объектно-ориентированного программирования

Программисты, хорошо владеющие процедурным программированием в целом неприязненно относятся к объектно-ориентированному программированию. Объектно-ориентированный подход радикально отличается от процедурного и требует иного мышления, особенно это касается начинающих программистов.

Объектно-ориентированное программирование является не всегда подходящим при написании небольших программ. Также объектно-ориентированное про-

граммирование может не удовлетворять приложения, критичные к времени выполнения. Хотя с ростом производительности компьютеров этот недостаток становится менее значимым. Операционная система обычно нагружает компьютер больше, чем объектно-ориентированная программа.

4.14. Заключение

В этой главе были приведены примеры из окружающего мира для пояснения объектно-ориентированных понятий. Мы начали различать два метода программирования: процедурный и объектно-ориентированный. Процедурное программирование оставляет возможность случайного неправильного использования данных и функций, что приводит к нежелательным последствиям и усложняет отладку. В объектно-ориентированном программировании применяется скрытие и защита данных от случайного ошибочного использования посредством инкапсуляции данных и функций, образующих класс. Внешний интерфейс инкапсулированного класса пояснен на примерах из окружающего мира.

Было дано качественное определение абстрактных и действительных классов. Затем абстрактные и действительные классы были описаны в терминах объектно-ориентированного программирования. Наследование пояснено на примере иерархии объектов, демонстрирующей, как можно создавать иерархии классов.

Дано краткое описание конструкторов и деструкторов, более подробно они будут рассматриваться в следующих главах. Инстанцирование объектов находится в тесной связи с конструкторами, а класс может обладать любым количеством конструкторов.

Обсуждались важные понятия полиморфизма, а виртуальные функции были рассмотрены недостаточно подробно из-за их сравнительной сложности. Они будут описаны и повсеместно использоваться в главе 8. Наконец, были рассмотрены достоинства и недостатки объектно-ориентированного программирования.

4.15. Литература

1. Meyer, B., *Object Oriented Software Construction*, Prentice Hall, 1998.
2. Firesmith, D. G., *Object Oriented Requirements, Analysis and Logical Design*, John Wiley, 1993.
3. Staugaard A. C. (Jr), *Structured and Object Oriented Techniques*, Prentice Hall, 1997.
4. Gray, N. A. B. *Programming with Classes*, John Wiley, 1994.

5

Объектно-ориентированное программирование

Содержание главы:

- Создание класса.
- Разработка объекта для портов.
- Доступ к атрибутам.
- Разработка класса Parallel Port.

5.1. Введение

Цель этой главы – научить вас создавать классы для использования их в программах на C++. Будут расширены понятия объектно-ориентированного программирования, данные в предыдущей главе, в определениях классов будет использоваться синтаксис C++. Созданные в данной главе классы будут использоваться при разработке программ, осуществляющих взаимодействие с различными устройствами на интерфейсной плате через параллельный порт компьютера.

В начале главы в три этапа разработаем класс параллельного порта `ParallelPort`. Каждый этап разработки будет расширять функциональность класса `ParallelPort`, в конечном итоге обеспечивая возможность двусторонней передачи данных с использованием большинства возможностей параллельного порта.

5.2. Правила обозначения

В целях улучшения «читаемости» программ, мы зададим правила обозначения, применяемые к именам функций и идентификаторам. Как объяснялось в главах 2 и 3, параллельный порт компьютера представлен тремя последовательно расположенными регистрами ввода/вывода. В большинстве случаев эти регистры расположены по адресам `0x378`, `0x379` и `0x37A`. Имя `BASE` обозначает первый из этих адресов (в данном случае `0x378`). Правила обозначений сведены в **Табл. 5.1**.

Таблица 5.1. Правила обозначения идентификаторов в программах

Суффикс идентификаторов	Смещение относительно базового адреса (BASE)	Физические адреса (наиболее распространённые)
Port0	0	0x378
Port1	1	0x379
Port2	2	0x37A

Примечание

Адреса регистров могут отличаться от 0x378, 0x379 и 0x37A. Но они так же будут размещаться в трёх следующих друг за другом адресах. Создаваемые нами классы будут иметь возможность менять адрес BASE на необходимый в конкретном случае.

В соответствии с правилами обозначения (**Табл. 5.1**), все данные-члены и функции-члены, заканчивающиеся словом `Port0` должны соответствовать регистру `BASE`. Например, функция с именем `WritePort0()` записывает данные в регистр `BASE`. Аналогично, функция `WritePort2()` записывает данные в регистр `BASE+2`.

5.3. Разработка класса

В конце этой главы нами будет создан класс `ParallelPort`, реализующий большую часть функциональности порта. Разработка будет происходить в три этапа; на первом этапе мы создадим класс, работающий с регистром `BASE`. Этот класс в двух последующих этапах будет дополнен функциональностью регистра `BASE+1`, а затем и `BASE+2`.

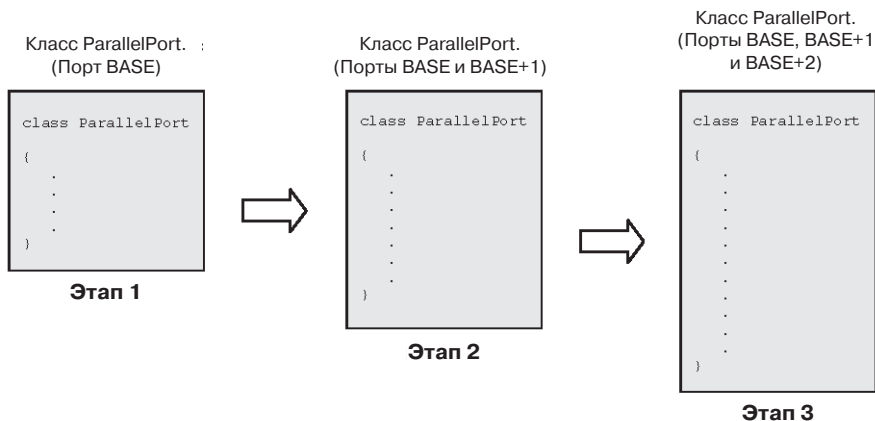


Рис. 5.1. Разработка класса параллельного порта

Порт `BASE` наиболее прост в использовании и может пересылать один байт данных за один раз. Обратите внимание, что современные компьютеры могут и принимать, и передавать данные посредством этого регистра. Для совместимости со старыми компьютерами через регистр `BASE` мы будем только выводить данные.

Создание класса:

1. Придумать имя класса. Оно должно быть коротким и в то же время отражать назначение и содержимое класса.
2. Определить данные-члены класса (характерные черты или свойства объекта).
3. Определить функции-члены класса (действия объекта).

4. Создать конструктор(ы) и деструктор класса.

В предыдущей главе данные-члены рассматривались как особенности или свойства объекта. Функции-члены отождествлялись с действиями объекта. На нашем обычном языке описание класса регистра BASE будет выглядеть примерно так:

«Это регистр по некоторому адресу, который может выводить один байт данных»

В соответствии с этим описанием регистр характеризуется адресом. Объект предназначен для вывода байта данных на внешнее устройство посредством этого регистра. Теперь, когда мы установили характерные черты и действия, мы можем попытаться определить класс. Мы инкапсулируем эти данные-члены и функции-члены в фундаментальный класс `ParallelPort`.

5.3.1. Данные-члены

В классе `ParallelPort` должен быть известен базовый адрес регистра, чтобы можно было работать с ним. Эти данные должны храниться как данные-член с соответствующим названием `BaseAddress`. Адреса регистров параллельного порта лежат в диапазоне от 0 до 0x3FF. Восьмибитного числа (0...0xFF) недостаточно для хранения таких значений, поэтому мы будем пользоваться 16-битным числом (0...0xFFFF). Для этого мы объявим данные-член `BaseAddress` типа `unsigned int`, который сможет хранить любой адрес регистра из диапазона 0...0x3FF:

```
unsigned int BaseAddress;
```

5.3.2. Функции-члены

Поскольку класс `ParallelPort` должен уметь записывать данные в регистр BASE, то для этого будет создана функция `WritePort0()`. Один байт представляется типом данных `unsigned char`. Функция только выводит данные, ей не нужно возвращать значение. Это означает, что тип её возвращаемого значения будет `void`. Функция будет такой:

```
void WritePort0(unsigned char data)
{
    outportb(BaseAddress, data);
}
```

Функция `WritePort0()` принимает один параметр, а именно `data` типа `unsigned char`. Тело функции состоит из единственного выражения; она выводит значение параметра `data` в регистр, адрес которого содержится в `BaseAddress`.

Тип возвращаемого значения у функции `WritePort0()` определён как `void`. Если функция должна возвращать значение, то она должна содержать выражение `return` в своём теле.

5.3.3. Атрибуты доступа

Атрибуты доступа определяют доступность членов класса для прочих функций программы. Есть три различных типа атрибутов доступа; далее описаны атрибуты *private*, *protected* и *public*.

private

Функции и данные, перечисленные под атрибутом `private`, доступны только для функций-членов этого же класса. Другие функции, где бы они не находились, не имеют возможности ни прочитать, ни модифицировать любые из таких собственных данных-членов, ни вызывать собственные функции-члены. Важное замечание: если в определении класса для каких-либо членов не указан атрибут доступа, то по умолчанию устанавливается атрибут `private`.

protected

Функции-члены и данные-члены перечисленные под атрибутом `protected` доступны функциям-членам содержащего их класса и функциям-членам *производных* классов, для которых этот класс является базовым. Классы, не связанные с этим классом, не имеют доступа к защищённым данным-членам и функциям-членам. Образование классов продолжает обсуждаться в главе 6.

public

Функции и данные перечисленные под атрибутом `public` доступны любой функции, независимо от её отношения к классу.

5.3.4. Определение класса

Более привлекательный способ разработки классов и обладающий большей защищённостью, заключается в использовании *определений классов*. Определение классов начинается со служебного слова `class`, за которым следует имя типа объекта (в данном случае `ParallelPort`). Выбор имени типа целиком ложится на вас. В C++ также возможно определить объект в виде совокупности взаимосвязанных данных, известных как *структуры*.

Как видно из **Рис.5.2**, тело определения класса начинается с первой открывающейся фигурной скобки (`{`), а заканчивается перед последней закрывающейся скобкой (`}`). Не забывайте ставить точку с запятой (`;`) после закрывающейся фигурной скобки, завершая тем самым определение класса.

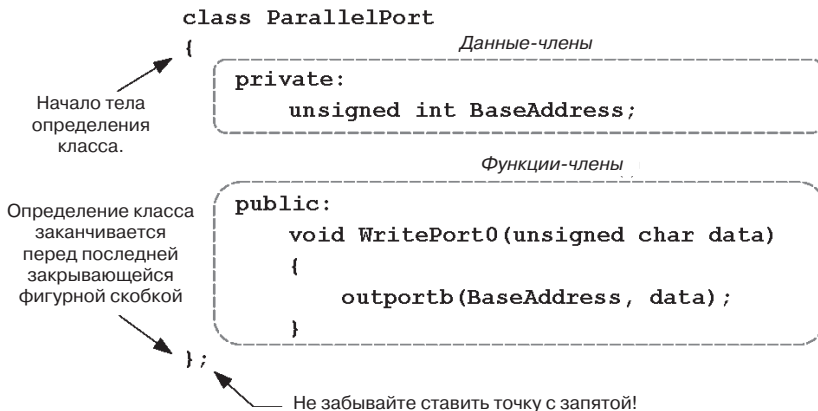


Рис. 5.2. Определение класса

В определении класса есть два раздела. Первый раздел связан с данными-членами, а второй с функциями-членами. Хотя данные-члены и функции-члены могут быть вперемешку друг с другом, является полезным хранить их раздельно.

Как говорилось ранее, атрибутом доступа по умолчанию является `private`, когда не указан никакой другой атрибут доступа. Поэтому служебное слово `private` не является обязательным в определении класса. Если оно было опущено, то переменная `BaseAddress` осталась бы по-прежнему собственным членом класса. Тем не менее лучше указывать служебное слово `private`, что несколько проясняет код.

У класса есть автоматические конструктор и деструктор, созданные компилятором и невидимые здесь. Конструктор создаёт объекты для использования их в программе, а деструктор освобождает память от объектов, когда они больше не нужны в программе.



Рис. 5.3. Класс `ParallelPort` в общем виде

В классе `ParallelPort` `BaseAddress` является собственным членом данных. Поэтому к нему есть доступ только из функции `WritePort0()`, принадлежащей этому же классу. Эта функция объявлена как `public` и вследствие этого может вызываться любой функцией в программе. Таким образом, обеспечивается интерфейс между объектом и внешним миром. Если бы функция была бы объявлена как `private`, то её нельзя было бы вызвать из не относящейся к классу функции (включая функцию `main()`). В результате объектом типа `ParallelPort` нельзя было бы пользоваться (вспомните запаянный автомобиль, которым никто не может управлять!).

5.3.5. Конструктор

Задачей *конструктора* является создание объектов для их использования в программе. Имеющееся определение типа ещё не означает наличия реального

объекта в памяти. Чтобы объект мог хранить данные и выполнять функции, и те и другие должны быть размещёнными в памяти. Это происходит, когда вызывается конструктор. Он создаёт объект и выделяет память для всех данных-членов в процессе, известном как *инстанцирование*. Во время инстанцирования выполняются все выражения тела конструктора. Программист может написать конструктор в соответствии с предъявляемыми к процессу создания объекта требованиями.

Имя конструкторов всегда совпадает с именем типа объекта – в нашем случае это `ParallelPort`. Конструктор, хоть и является функцией, но **никогда** не возвращает результат. Это характерная черта конструкторов. Есть только два типа функций в C++, никогда не возвращающих значения, даже типа `void`. Второй такой функцией является деструктор.

5.3.6. Автоматический конструктор

Автоматический конструктор – это как конструктор, не принимающий никаких параметров (т. е. пара круглых скобок пуста). При отсутствии пользовательского конструктора, компилятор предоставляет конструктор, являющийся автоматическим конструктором. Главная задача автоматического конструктора, созданного компилятором, состоит в *выделении памяти* для всех данных-членов объекта. Если у объекта есть базовый класс, то он также вызывается автоматическим конструктором – это происходит **до того**, как конструктор производного класса выделит память для его данных-членов.

Программист может сам создать автоматический конструктор. Написанный программистом автоматический конструктор тоже не принимает никаких параметров, но **может** содержать в своём теле какие-либо выражения. Когда программист пишет сам конструктор, автоматический или любой другой, компилятор **не создаёт** конструктор. Конструктор, написанный программистом, тоже вызывает конструктор базового класса **перед** выполнением собственных действий.

Компилятор также создаёт специальный конструктор, называемый конструктором копии, и перегружает оператор присвоения (знак `=`). Отложим обсуждение конструкторов копии и перегруженные операторы присвоения до главы 12.

5.3.7. Перегрузка конструкторов

Термин *перегрузка* означает наличие множества функций с одинаковыми именами, но выполняющих разные действия. В случае перегруженных конструкторов в определении класса содержится больше одного конструктора, все с одинаковыми именами (с именем класса), и, как все конструкторы, без возвращаемого значения. В классе допускается любое количество конструкторов. Поэтому конструкторы могут различаться только по количеству и типам передаваемых каждому из них параметров и выражениями, составляющими их тела.

5.3.8. Деструкторы

Роль деструктора противоположна задачам конструктора. Конструктор начинает жизненный цикл объекта, создавая его в памяти. Деструктор удаляет объект из памяти, освобождая пространство. Имя деструктора такое же, как у класса, а в начале имени символ тильды (`~`). Например, деструктор класса `ParallelPort` будет называться `~ParallelPort()`. Разработчик программы может включить де-

структур в определение класса. Если разработчик не предоставляет деструктор, то компилятор создаёт автоматический деструктор.

В отличие от конструкторов, деструкторы явно не вызываются. Он вызывается автоматически, когда перестаёт существовать динамический объект или когда для удаления объекта из памяти применяется оператор `delete`. В иерархиях классов деструкторы производных классов вызываются *перед* деструкторами базовых классов.

В то время как не бывает виртуальных конструкторов, деструкторы могут быть виртуальными функциями. Виртуальные деструкторы будут подробно описаны в разделе 8.5. Сейчас мы приступим к созданию класса `ParallelPort` в три этапа, как было объяснено в начале раздела 5.3 и как показано на **Рис. 5.1**.

5.4. Класс ParallelPort – этап 1

На этом этапе разработки класса `ParallelPort` будет создан класс, обладающий простейшими элементами из этих трёх этапов. Будет использоваться только одна возможность регистра `BASE`, позволяющая записывать байт данных (8 бит, **D0...D7**) во внешнее устройство.

Сначала будет использоваться автоматический конструктор класса (создаваемый компилятором). Такая конструкция неполноценна, что обсуждалось выше. Класс будет разрабатываться дальше для преодоления этих недостатков.

5.4.1. Определение класса

Выше говорилось, что `WritePort0()` является *публичной функцией-членом* (т. е. доступной из функций, не относящихся к классу). Поэтому любая функция в программе может вызывать эту функцию. Функция `WritePort()` в свою очередь вызывает известную нам функцию `inportb()` для передачи данных в регистр, определяемый значением `BaseAddress`.

Если мы сами не создадим конструктор в определении класса, то он будет автоматически создан компилятором.

C++ Синтаксис функций-членов

Компилятору можно сообщить о функциях-членах двумя способами:

1. Определение функции-члена находится вне класса. В этом случае используется тот же синтаксис, что и для обычных функций, за исключением того, что перед именем функции помещается имя класса со следующими за ним двумя двоеточиям (`::`), как показано в **Листинге 5.1**.
2. Определение функции-члена находится внутри класса. Здесь не обязательно пользоваться синтаксисом из первого пункта (**Листинг 5.2**).

Определение функции-члена `WritePort0()`, показанное в **Листинге 5.1**, сделано вне класса.

Листинг 5.1. Определение функции-члена класса `ParallelPort`

```
void ParallelPort::WritePort0(unsigned char data)
```



```
{  
    outportb(BaseAddress,data);  
}
```

Если функция-член `WritePort0()` определена внутри самого класса, то такое определение класса показано в **Листинге 5.2**. Любые функции-члены, которые определены внутри класса и невелики, будут рассматриваться как *in-line* функции (они будут обсуждаться далее). Если функции объявляются вне класса, то они являются обычными функциями.

Листинг 5.2. Определение функций-членов внутри класса

```
#include <dos.h>  
  
class ParallelPort  
{  
    private:  
        unsigned int BaseAddress;  
  
    public:  
        void WritePort0(unsigned char data)  
        {  
            outportb(BaseAddress,data);  
        }  
};
```

C++ In-Line функции-члены

Компилятор считает *in-line* функциями небольшие функции-члены, определённые внутри класса. Где бы в исходном файле не вызывалась *in-line* функция, компилятор заменяет вызов инструкциями из её тела. Так происходит, когда *in-line* функция мала по размеру и не требуется сложной (долгой) передачи параметров.

Это означает, что при каждом вызове *in-line* функции из программы, компилятор вставляет в исполняемый код очередной экземпляр тела функции, что приводит к увеличению размера программы. Тем не менее, программа выигрывает в скорости выполнения, когда выполняется только тело *in-line* функции без затрат времени на вызов функции.

Обычные функции имеют только один экземпляр функции в памяти. В этом случае, вызов функции приводит к сохранению текущего контекста выполнения программы и переходу к области памяти, содержащей функцию. Выполнение кода функции завершается возвратом в место её вызова и восстановлением контекста выполнения программы. Эти дополнительные действия по сохранению, переходам и восстановлению увеличивают время её выполнения и обуславливают более медленную работу по сравнению с *in-line* функциями (хотя исполняемые программы меньше по размеру).

Обратите внимание, Листинг 5.1 и Листинг 5.2 являются неполными. Глядя на параметры функции `WritePort0()` мы видим, что можем передать фактический аргумент (например, `0x7F`) через параметр `data`, но при этом нет возможности присвоить значение для `BaseAddress`.

Если не будет механизма инициализации переменной `BaseAddress`, то функция `outportb()` не будет работать. Использование фиксированного адреса неэффективно. Например:

```
void WritePort0(unsigned char data)
{
    outportb(0x378, data);
}
```

Вышеприведённый пример не будет работать у пользователей, имеющих параллельный регистр с адресом `BASE 0x378`. Другим пользователям придётся редактировать функцию-член так, чтобы она соответствовала адресу `BASE` параллельного порта компьютера. Наконец, в рассмотренном примере не используется член `BaseAddress`.

Определение класса – первая модернизация

Нам нужно предоставить пользователю возможность указывать базовый адрес параллельного порта своего компьютера. Это лучше всего сделать во время создания объекта `ParallelPort` – т. е. во время вызова конструктора. Автоматический конструктор нужно заменить конструктором, который позволит нам присваивать нужное значение члену `BaseAddress` во время создания объекта.

В новом конструкторе: ему передаётся параметр `baseaddress`, а затем это значение присваивается члену данных `BaseAddress`. Теперь функция `outportb()` будет работать как задумывалось. В **Листинге 5.4** будет отмечено, что функция-член определяется вне класса. Причина такого решения приведена выше под заголовком «In-line функции-члены».

Листинг 5.3. Определение класса `ParallelPort` с конструктором

```
class ParallelPort
{
    private:
        unsigned int BaseAddress;

    public:
        ParallelPort(int baseaddress); // конструктор
        void WritePort0(unsigned char data);
};
```

Листинг 5.4. Определения функций-членов в модернизированном классе `ParallelPort`

```
ParallelPort::ParallelPort(int baseaddress) // конструктор
{
    BaseAddress = baseaddress;
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress, data);
}
```

Примечание

Член `BaseAddress` и `baseaddress` это разные вещи. Аргумент `baseaddress` означает место ввода фактического аргумента при вызове конструктора и нужен только в это время. В противоположность ему член `BaseAddress` находится в памяти на протяжении существования объекта; в большинстве случаев до конца выполнения программы.

Определение класса – вторая модернизация

Поскольку у большинства пользователей параллельный порт будет расположен по адресу `0x378`, то было бы полезным, если конструктор будет по умолчанию задавать `BASE` как `0x378`. Ещё нам нужно обеспечить возможность указывать другой адрес регистра, как было реализовано в предыдущем определении класса.

Для этого добавим второй конструктор (перегруженный конструктор), присваивающий члену `BaseAddress` значение `0x378` по умолчанию. У этого нового автоматического конструктора нет параметров (по определению автоматического конструктора), а в своём теле он присваивает члену `BaseAddress` значение `0x378`. В Листингах 5.5 и 5.6 приведены определения класса и функции-члена.

Листинг 5.5. Определение класса `ParallelPort` с автоматическим конструктором

```
class ParallelPort
{
    private:
        unsigned int BaseAddress;

    public:
        ParallelPort(); // автоматический конструктор
        ParallelPort(int baseaddress); // конструктор
        void WritePort0(unsigned char data);
};
```

Листинг 5.6. Определение функций-членов для класса из Листинга 5.5

```
ParallelPort::ParallelPort() // автоматический конструктор
{
    BaseAddress = 0x378;
}

ParallelPort::ParallelPort(int baseaddress) // конструктор
{
    BaseAddress = baseaddress;
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress, data);
}
```

Если в программе мы вызовем конструктор без параметров, то автоматический конструктор сделает по умолчанию адрес BASE равным 0x378. Также можно в качестве BASE использовать адрес, скажем, 0x3BC, передав это значение в качестве параметра при создании объекта типа `ParallelPort`:

```
ParallelPort(0x3BC);
```

В этом случае для создания объекта типа `ParallelPort` вызывается второй конструктор (тот, что с аргументом). Несмотря на то, что конструкторы обычно используются для присвоения некоторым или всем членам-данным начальных значений, они могут выполнять также разнообразные действия как обычные функции.

Следует отметить важную деталь – функции-члену `WritePort0()` явно не сообщалось о существовании члена `BaseAddress`, т. е. в теле функции нет объявления `BaseAddress`. В этом объявлении нет необходимости, поскольку `BaseAddress` и `WritePort0()` являются членами одного класса, чем и обусловлена возможность доступа к `BaseAddress` (раздел 5.3.3).

5.5. Программирование с классами

Теперь мы будем пользоваться созданным в предыдущей главе классом `ParallelPort` для вывода байта данных в регистр BASE. Выполняющая эти действия программа хоть и кажется длинной, тем не менее в следующих главах будут реализованы преимущества объектно-ориентированного программирования.

Листинг 5.7. Запись в регистр BASE средствами объектно-ориентированного программирования

```

/*****
    ЗАПИСЬ В ПОРТ (объектно-ориентированный подход)

    Программа выводит байт данных на интерфейсную плату,
    используя фундаментальный класс ParallelPort.
*****/

#include <dos.h>

class ParallelPort
{
private:
    unsigned int BaseAddress;

public:
    ParallelPort();
    ParallelPort(int baseaddress);
    void WritePort0(unsigned char data);
};

ParallelPort::ParallelPort() // автоматический конструктор

```

```

{
    BaseAddress = 0x378;
}

ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress,data);
}

void main()
{
    ParallelPort OurPort; // инстанцирование объекта

    OurPort.WritePort0(255); // вызов функции-члена
}

```

Здесь использованы ранее объяснённые объявления класса и функции. Непонятным пока является тело функции `main()`. Таким образом используются классы в приложениях. Первая из двух строк функции `main()` создаёт объект `OurPort` типа `ParallelPort`. Во второй строке вызывается функция `WritePort0()` объекта `OurPort` и в регистр выводится значение 255. Фактический адрес регистра, используемый функцией `WritePort0()` назначается при создании объекта `OurPort` одним из двух конструкторов. Рассмотрим этот процесс более подробно.

Создание объекта (инстанцирование)

Первая строка функции `main()`:

```
ParallelPort OurPort;
```

Эта строка определения подобна следующему определению:

```
int a; // Тип данных int, переменная a
```

Аналогично, `ParallelPort` тип данных, являющийся типом объекта. Имя переменной `OurPort`. Различие только в том, что `int` является встроенным типом данных, а `ParallelPort` мы создали сами.

В C++ строки с объявлениями не только сообщают компилятору имя (`OurPort`, например) и тип (`ParallelPort`), но ещё и вызывают конструктор этого объекта, чтобы создать этот объект. Создание объекта `OurPort` было объяснено ранее как инстанцирование – создание реального объекта в памяти, которым можно пользоваться. Важно понимать разницу между «объектом» (также известном как экземпляр класса) и «типом объекта» (классом объекта). Объектом является `OurPort`, а типом объекта `ParallelPort`. Таким образом, мы создали *объект* с именем `OurPort` *типа* `ParallelPort`.

Обращаясь к **Листингу 5.5**, мы видим две функции-конструктора. Это:

```
ParallelPort();
ParallelPort(int baseaddress);
```

Какой из этих конструкторов использован в строке создания объекта `OurPort`? Компилятор автоматически выбирает нужный конструктор по передаваемым ему

параметрам (или по их отсутствию). Поскольку мы не передавали параметр `base-address` при создании объекта `OurPort`, то компилятор вызовет первый из двух конструкторов, т. е. автоматический конструктор.

С другой стороны, строка объявления в функции `main()` могла бы выглядеть так:

```
ParallelPort OurPort(0x3BC);
```

В этом случае компилятор не будет вызывать автоматический конструктор, поскольку ему был передан параметр. Вместо него для создания объекта `OurPort` будет вызван другой конструктор (с одним аргументом):

```
ParallelPort(int baseaddress);
```

В предыдущем примере при создании объекта типа `ParallelPort` формальный аргумент `baseaddress` заменён фактическим аргументом `0x3BC`. Затем значение `0x3BC` присваивается члену `BaseAddress` (смотрите тело конструктора в **Листинге 5.6**).

Созданный в программе из **Листинга 5.7** объект `OurPort` будет обладать всеми перечисленными в определении класса качествами. У него есть собственный член данных `BaseAddress` и три публичных функции-члена. Две из этих функций являются конструкторами с именем `ParallelPort`, а другая функция `WritePort0()`.

Передача данных в регистр

У созданного нами объекта `OurPort` используется функция-член `WritePort0()` для передачи в регистр значения 255. Доступ к функции-члену осуществляется при помощи точки между *именем объекта* и именем члена:

```
OurPort.WritePort0(255);
```

Остальные члены объекта адресуются аналогично, хотя атрибуты доступа могут препятствовать этому. Например, несмотря на правильный синтаксис, попытка выполнить следующее выражение из любой функции вне класса приведёт к ошибке:

```
OurPort.BaseAddress = 0; // не работает!
```

Это вызвано тем, что `BaseAddress` является собственным членом данных класса и функции снаружи класса не могут изменять его (функция `main()` к классу не относится). Этот неправомерный доступ к члену `BaseAddress` будет обнаружен во время компиляции и возникнет ошибка компиляции. В следующем разделе будут представлены дополнительные примеры атрибутов доступа, используемые в программе.

Заметим, что функция `WritePort0()` принимает один параметр. Этот параметр заменяется фактическим аргументом 255. Число 255 в двоичном представлении соответствует восьми единицам.

Работа программы может быть проверена подключением интерфейсной платы к компьютеру, как показано в **Табл. 3.1**. Передача числа 255 в качестве параметра функции `WritePort0()` приведёт к свечению восьми светодиодов. Это значение можно изменять от 0 до 255 включительно. Перезапуская программу можно наблюдать зажигающиеся в соответствии числу светодиоды.

5.5.1. Примеры с атрибутами доступа

В предыдущем разделе была сделана попытка запрещённого доступа при присвоении члену `BaseAddress` значения 0; приведём этот фрагмент ещё раз:

```
OurPort.BaseAddress = 0; // не работает!
```

В этой программе `BaseAddress` объявлен как собственный член данных, что препятствует функции `main()` изменить его значение.

Член `BaseAddress` можно было бы объявить не собственным, а публичным членом. Тогда к нему можно было бы обращаться из любой функции программы, и ошибка компиляции не возникала бы. В **Листинге 5.8** показано, как это можно сделать. Отметим, что если в вашем компьютере нет второго параллельного порта (LPT2), то не существует и регистра `0x3BC`, и программа не будет работать. Но компилироваться она будет без ошибок. Также убедитесь, что кабель интерфейсной платы включен в разъём DB25 нужного порта.

Листинг 5.8. Объявление `BaseAddress` публичным членом данных

```

/*****
    Обратите внимание, что член BaseAddress объявлен
    теперь с атрибутом публичного доступа public.
    Поэтому его значение можно менять из функции
    main() и при этом не будет ошибок компиляции.

    Если в вашем компьютере нет второго параллельного
    порта, то попытка обращения к регистру 0x3BC приведёт
    к ошибке!
*****/

#include <dos.h>

class ParallelPort
{
    public:        // Атрибут доступа "собственный" заменён
                  // на "публичный".

    unsigned int BaseAddress;

    public:
        ParallelPort();
        ParallelPort(int baseaddress);
        void WritePort0(unsigned char data);
};

ParallelPort::ParallelPort()
{
    BaseAddress = 0x378;
}

ParallelPort::ParallelPort(int baseaddress)
```

```

{
    BaseAddress = baseaddress;
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress,data);
}

void main()
{
    ParallelPort OurPort;

    OurPort.BaseAddress = 0x3BC; // Не приводит
                                // к ошибке компиляции

    OurPort.WritePort0(255);
}

```

Объявление публичных данных-членов является плохой практикой программирования. Главная цель объектно-ориентированного программирования заключается как раз в инкапсуляции, что обеспечивает устойчивость объекта к неправильному использованию. Эти цели не будут достигнуты при объявлении публичных данных-членов.

Если нужно изменить член данных, то это можно сделать посредством специально для этого написанной функции. Такая возможность продемонстрирована функцией `ChangeAddress()` в **Листинге 5.9**.

Листинг 5.9. Допустимый способ менять член данных объекта

```

/*****
    ЗАПИСЬ В ПОРТ (Добавлена функция-член для
    изменения собственного члена данных).

    Внимание: попытка обращения к регистру 0x3BC
    вызовет ошибку, если в вашем компьютере нет
    второго параллельного порта.
*****/

#include <dos.h>

class ParallelPort
{
    private:        // Атрибут доступа изменён
                   // обратно на "собственный"
        unsigned int BaseAddress;

    public:
        ParallelPort();
        ParallelPort(int baseaddress);
}

```



```

void WritePort0(unsigned char data);

// Добавлена новая публичная функция-член.
void ChangeAddress(unsigned int newaddress);
};

ParallelPort::ParallelPort()
{
    BaseAddress = 0x378;
}

ParallelPort::ParallelPort(int baseaddress)
{
    BaseAddress = baseaddress
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress,data);
}

// Определена новая публичная функция-член.
void ParallelPort::ChangeAddress(unsigned int newaddress)
{
    BaseAddress = newaddress;
}

void main()
{
    ParallelPort OurPort;

// Правильный способ манипулирования с членом данных
    OurPort.ChangeAddress(0x3BC);
    OurPort.WritePort0(255);
}

```

Ниже приведёно выражение из предыдущего листинга, позволяющее поменять значение члена BaseAddress на 0x3BC без каких-либо нарушений:

```
OurPort.ChangeAddress(0x3BC);
```

Этим примером мы продемонстрировали, как можно изменять собственные данные-члены при помощи публичной функции-члена. Публичная функция-член ChangeAddress() имеет полный доступ к собственному члену BaseAddress, поскольку оба этих члена принадлежат одному и тому же классу. В законченном классе ParallelPort функции ChangeAddress() не будет. Вместо этого будет использоваться конструктор, позволяющий задавать желаемое значение базового адреса, отличающееся от значения по умолчанию 0x378, присваиваемого члену BaseAddress автоматическим конструктором.

5.6. Класс ParallelPort – этап 2

На первом этапе мы создали класс `ParallelPort`, реализующий требуемую функциональность регистра `BASE`. Теперь нам нужен такой объект для регистра `BASE+1`. Вспомним, что порт `BASE` является выходным регистром, а `BASE+1` входным. К объекту предъявляются следующие требования:

- Возможность указания базового адреса параллельного порта.
- Передавать данные в регистр `BASE`.
- Принимать данные из регистра `BASE+1`.

Расширение функциональности имеющегося объекта является хорошим поводом для создания производного класса. Тем не менее разработка иерархии классов нецелесообразна, поскольку элементы параллельного порта просты в использовании. Гораздо лучше рассматривать параллельный порт как *один* объект. Поэтому на втором этапе классу `ParallelPort` будет добавлена возможность работы с регистром `BASE+1`.

В **Листинге 5.10** приведено определение обновлённого класса `ParallelPort`; новые элементы выделены жирным шрифтом. В нём содержатся объявления данных-членов и функций-членов. Все данные-члены класса `ParallelPort` объявлены как собственные. Все функции-члены публичные. Как и прежде `BaseAddress` один из членов данных, а функция `WritePort0()` пересылает данные в регистр `BASE`.

Листинг 5.10. Новое определение класса `ParallelPort`

```
class ParallelPort
{
    private:
        unsigned int BaseAddress;
        unsigned char InDataPort1;

    public:
        ParallelPort(); // автоматический конструктор
        ParallelPort(int baseaddress); // конструктор
        void WritePort0(unsigned char data);
        unsigned char ReadPort1();
};
```

Порт `BASE+1` является входным (данные поступают в компьютер). Для чтения данных из этого регистра в класс `ParallelPort` введена функция `ReadPort1()`. Для хранения прочитанных из регистра данных объявлен член данных `InDataPort1`. Цифры в названиях членов соответствуют смещению от адреса `BASE`. Например, функция `WritePort0()` записывает данные в регистр с нулевым смещением от базового адреса – в данном случае `BASE+0`, являющийся собственно базовым адресом. Аналогично функция `ReadPort1()` читает данные из регистра со смещением 1. Поэтому чтение выполняется из регистра `BASE+1`.

Определения всех функций этого дополненного класса приведены в **Листинге 5.11**.

Листинг 5.11. Определения функций класса `ParallelPort`

```
ParallelPort::ParallelPort() // автоматический конструктор
```

```

{
    BaseAddress = 0x378;
    InDataPort1 = 0;
}

ParallelPort::ParallelPort(int baseaddress) // конструктор
{
    BaseAddress = baseaddress;
    InDataPort1 = 0;
}

void ParallelPort::ReadPort1()
{
    InDataPort1 = inportb(BaseAddress+1);
    // Инвертируем старший бит для компенсации внутренней
    // инверсии схемы порта принтера.
    InDataPort1 ^= 0x80;
    // Используем маску (или фильтр) для обнуления
    // неиспользуемых битов D0, D1 и D2.
    InDataPort1 &= 0xF8;
    return InDataPort1;
}

```

В конструкторах появилось только одно выражение, инициализирующее член данных регистра `BASE+1 InDataPort1` значением 0. Если не инициализировать этот член, то в нём будет находиться неопределённое значение. Тем не менее, его инициализация не является обязательной. Его можно инициализировать каким-либо другим значением или оставить неинициализированным, приняв меры по предотвращению его использования до того, как из регистра будет прочитано значение члена `InDataPort1`.

Функция `ReadPort1()` читает данные из регистра `BASE+1` и возвращает значение типа `unsigned char`. В связи с этим в теле функции содержится выражение `return`, которое и возвращает значение `InDataPort1`. В то время как эта функция сохраняет результат операций ввода в члене `InDataPort1`, она же служит интерфейсом между другими функциями вне класса и этим членом данных. Это улучшает гибкость объекта. В последующих главах класс `ParallelPort` будет использоваться для написания многих программ. Гибкость объекта типа `ParallelPort` способствует написанию хороших и эффективных программ.

Вызываемая в теле функции `ReadPort1()` функция `inportb()` читает данные из указанного регистра, в данном случае это `BASE+1`. Из этого регистра могут быть прочитаны только с 3 по 7 биты. Бит 7 ещё и инвертируется схемой параллельного порта. В функции `ReadPort1()` предусмотрены компенсация инверсии (раздел 3.6) и обнуление неиспользуемых битов **D0...D2** логическим оператором «И» (`&`). Шестнадцатеричное число `F8` является битовой маской `1111 1000` и обнуляет биты **D0...D2** у любого числа, являющегося его вторым операндом. Результат этой корректирующей операции сохраняется в члене данных `InDataPort1`. В последней строке функции `ReadPort1()` находится выражение `return`, возвращающее значение члена `InDataPort1`.

Вся программа приведена в **Листинге 5.12**. Проверьте работу программы, подключив интерфейсную плату к компьютеру согласно **Табл. 3.1** и **Табл.3.2**.

Листинг 5.12. Запись данных в регистр BASE и чтение данных из регистра BASE+1

```

/*****
    Фундаментальный класс ParallelPort дополнен
    возможностью ввода данных из регистра BASE+1.
    Получившийся класс по-прежнему называется
    ParallelPort и используется для записи в регистр
    BASE и чтения регистра BASE+1.
*****/

#include <stdio.h>
#include <dos.h>

class ParallelPort
{
    private:
        unsigned int BaseAddress;
        unsigned char InDataPort1;

    public:
        ParallelPort(); // автоматический конструктор
        ParallelPort(int baseaddress); // конструктор
        void WritePort0(unsigned char data);
        unsigned char ReadPort1();
};

ParallelPort::ParallelPort() // автоматический конструктор
{
    BaseAddress = 0x378;
    InDataPort1 = 0;
}

ParallelPort::ParallelPort(int baseaddress) // конструктор
{
    BaseAddress = baseaddress;
    InDataPort1 = 0;
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress, data);
}

void ParallelPort::ReadPort1()
{
    InDataPort1 = inportb(BaseAddress+1);
}

```

```
// Инвертируем старший бит для компенсации внутренней
// инверсии схемы порта принтера.
    InDataPort1 ^= 0x80;
// Используем маску (или фильтр) для обнуления
// неиспользуемых битов D0, D1 и D2.
    InDataPort1 &= 0xF8;
    return InDataPort1;
}

void main()
{
    unsigned char BASE1Data;
    ParallelPort OurPort;

    OurPort.WritePort0(255);
    BASE1Data = OurPort.ReadPort1();
    printf("\nData Read from Port at BASE+1 %2X\n",BASE1Data);
}
```

Листинг 5.12 является объединением объяснённых ранее **Листинга 5.10** и **Листинга 5.11**, и добавлена функция `main()`.

Первая строка функции `main()`:

```
unsigned char BASE1Data;
```

Эта строка объявляет переменную `BASE1Data` для хранения данных типа `unsigned char`. Выражаясь терминами C++, в этой строке инстанцируется объект типа `unsigned char` с именем `BASE1Data`. Вследствие этого `BASE1Data` будет теперь находиться в памяти. Переменная `BASE1Data` предназначена для хранения прочитанного значения из регистра `BASE+1`. Как это происходит станет ясно после изучения остальных выражений функции `main()`.

Следующая строка функции `main()`:

```
ParallelPort OurPort;
```

В этой строке инстанцируется объект `OurPort` типа `ParallelPort`. Поэтому у объекта `OurPort` будет два члена данных: `BaseAddress` и `InDataPort1`. При выполнении этой строки вызывается автоматический конструктор (не имеющий аргумента, используемого в качестве базового адреса). В результате переменная `BaseAddress` будет иметь значение `0x378`, а `InDataPort1` будет равна 0.

В следующих двух строках вызываются функции-члены:

```
OurPort.WritePort0(255);
BASE1Data = OurPort.ReadPort1();
```

В первой строке в регистр `BASE` выводится байт данных (число 255). Это приводит к свечению всех восьми светодиодов. Вторая строка выполняет чтение регистра `BASE+1` и корректирует инверсию бита **D7**. Функция `ReadPort1()` сохраняет этот результат в члене `InDataPort1` для дальнейшего использования и возвращает в функцию `main()` значение члена `InDataPort1`. Принятое функцией `main()` значение сохраняется в переменной `BASE1Data`.

Последняя строка функции `main()` отображает значение `BASE1Data` в шестнадцатеричном формате на экране с шириной поля вывода 2 символа. Перед отображением значения на экран при помощи символов “\n” выводятся символы новой строки и возврата каретки. В этом примере функции `main()` её переменной `BASE1Data` присваивается значение, возвращаемое функцией `ReadPort1()`. Наши программы в дальнейшем не всегда будут работать подобным образом.

Также отметим необходимость члена данных `InDataPort1`, что позволяет внутри объекта хранить принятое из регистра значение. Если бы не было этой переменной, то программа имела бы зависимость от переменной `BASE1Data`, принадлежащей функции `main()`, чтобы иметь возможность сохранять принятое от `ReadPort1()` значение. В дальнейшем в программах иногда будет нежелательны переменные функции `main()`, использующиеся подобным образом. В таких случаях при отсутствии у объекта типа `ParallelPort` члена `InDataPort1`, сохраняющего принятые от `ReadPort1()` значения, то принятые из регистра данные будут утеряны сразу после завершения функции `ReadPort1()`.

5.7. Класс `ParallelPort` – этап 3

На этом заключительном этапе мы доработаем класс `ParallelPort` до полной поддержки всех возможностей ввода/вывода параллельного регистра компьютера, но с одним исключением. Не все компьютеры позволяют принимать данные через регистр `BASE+2`. Этот класс будет выводить данные через регистр `BASE`, вводить данные через `BASE+1` и выводить данные через `BASE+2`. Он также корректирует внутреннюю инверсию параллельного порта.

5.7.1. Полнофункциональный класс `ParallelPort`

От законченного класса `ParallelPort` требуется следующее:

- Возможность указывать адрес `BASE` параллельного порта.
- Выводить данные через регистр `BASE`.
- Вводить данные через регистр `BASE+1`.
- Выводить данные через регистр `BASE+2`.

Определение финального класса `ParallelPort` показано в **Листинге 5.13**.

Листинг 5.13. Определение класса `ParallelPort`

```
class ParallelPort
{
    private:
        unsigned int BaseAddress;
        unsigned char InDataPort1;

    public:
        ParallelPort();
        ParallelPort(int baseaddress);
        void WritePort2(unsigned char data);
        unsigned char ReadPort1();
};
```

В определении класса содержатся функции, соответствующие вышеприведённым требованиям к классу. Определения функций-членов даны в **Листинге 5.14**. Дополнения к ранее созданному классу `ParallelPort` выделены жирным шрифтом в **Листингах 5.13** и **5.14**.

Листинг 5.14. Определения функций-членов класса `ParallelPort`

```
ParallelPort::ParallelPort()
{
    BaseAddress = 0x378;
    InDataPort1 = 0;
}

ParallelPort::ParallelPort(int baseaddress)
{
    BaseAddress = baseaddress;
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress,data);
}

void ParallelPort::WritePort2(unsigned char data)
{
// Инвертирование битов 0, 1 и 3 для компенсации
// внутренней инверсии схемы порта принтера.
    outportb(BaseAddress+2,data ^ 0x0B);
}

unsigned char ParallelPort::ReadPort1()
{
    InDataPort1 = inportb(BaseAddress+1);
    // Инвертирование старшего бита для компенсации
    // внутренней инверсии схемой порта принтера.
    InDataPort1 ^= 0x80;
    // При помощи маски обнуляются неиспользуемые биты D0, D1 и D2.
    InDataPort1 &= 0xF8;
    return InDataPort1;
}
```

Класс `ParallelPort` используется в программе (**Листинг 5.15**) для обмена данными по всем трём регистрам параллельного порта компьютера. Работу программы можно проверить при помощи интерфейсной платы. Необходимые соединения на интерфейсной плате приведены в **Табл. 3.1** и **Табл. 3.2**. Перед пошаговым тестированием программы на регистре `BASE+2` отсоедините драйвер светодиодов от выходов регистра `BASE` и подключите его к выходам `BASE+2` (**Табл. 3.3**).

Листинг 5.15. Операции ввода/вывода с классом `ParallelPort`

```
/******
Класс, работавший с регистрами BASE и BASE+1
доработан для работы с регистром BASE+2. Получившийся
```

```

    класс по-прежнему называется ParallelPort.
    *****/

#include <dos.h>
#include <conio.h>
#include <stdio.h>

class ParallelPort
{
    private:
        unsigned int BaseAddress;
        unsigned char InDataPort1;

    public:
        ParallelPort();
        ParallelPort(int baseaddress);
        void WritePort0(unsigned char data);
        void WritePort2(unsigned char data);
        unsigned char ReadPort1();
};

ParallelPort::ParallelPort()
{
    BaseAddress = 0x378;
    InDataPort1 = 0;
}

ParallelPort::ParallelPort(int baseaddress)
{
    BaseAddress = baseaddress;
    InDataPort1 = 0;
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress, data);
}

void ParallelPort::WritePort2(unsigned char data)
{
    outportb(BaseAddress+2, data ^ 0x0B);
}

unsigned char ParallelPort::ReadPort1()
{
    InDataPort1 = inportb(BaseAddress+1);
    // Инвертирование старшего бита для компенсации его
    // внутренней инверсии схемой параллельного порта.
    InDataPort1 ^= 0x80;
}

```



```
// Сброс неиспользуемых битов D0, D1 и D2 при помощи маски.
InDataPort1 &= 0xF8;
return InDataPort1;
}

void main()
{
    unsigned char BASE1Data;
    ParallelPort OurPort;

    OurPort.WritePort0(0x55);
    printf("\n\nData sent to Port at Base\n");
    getch();

    BASE1Data = OurPort.ReadPort1();
    printf("\nData read from Port at BASE+1: %2X\n",BASE1Data);

    getch();

    OurPort.WritePort2(0x00);
    printf("\nData sent to Port at BASE+2\n");
    getch();
}
```

Первая строка тела функции `main()` инстанцирует объект `BASE1Data` типа `unsigned char`, используемый для сохранения прочитанных из регистра `BASE+1` данных. Во второй строке вызывается конструктор класса `ParallelPort` для создания объекта `OurPort`. Этот объект обладает функциями для записи и чтения данных через все три регистра параллельного порта.

Остальные выражения функции `main()` выполняют ряд операций ввода/вывода. При помощи функции `getch()` организовано ожидание нажатия клавиши для обеспечения возможности увидеть вывод на экран. Если не будет выражения `getch()`, то содержимое экрана будет или прокручено вверх или произойдёт возврат в IDE, прежде чем пользователь успеет рассмотреть экран. Функция `getch()` не является членом объекта `OurPort`. Поэтому к объекту она не имеет никакого отношения и вызывается как обычная функция.

5.8. Заключение

В начале этой главы мы создали класс `ParallelPort`. Этот класс содержит только самые важные данные-члены и функции-члены для работы с портом. Членам класса были присвоены различные атрибуты доступа и была разъяснена важность правильного применения этих атрибутов.

На примере различных программ показано различие между автоматическим конструктором и остальными конструкторами. Затем класс `ParallelPort` был доработан для поддержки работы с регистрами `BASE+1` и `BASE+2`. Работа с объектом этого класса показана на примере программы, выполняющей обмен данными с интерфейсной платой. Теперь, когда у нас есть полнофункциональный класс `ParallelPort`, мы можем его интенсивно использовать в последующих главах.

5.9. Литература

1. Borland, *Borland C++ Getting Started*, Borland International, 1991.
2. Winston, P. H., *On to C++*, Addison Wesley, 1994.
3. Lipman, S. B., *C++ Primer*, Addison Wesley, 1991.
4. Dench, D. and B. Prior, *Introduction to C++*, Chapman and Hall, 1994.
5. Etter, D. M., *Introduction to C++ – For Engineers and Scientists*, Prentice Hall, 1997.

6 Цифро-аналоговое преобразование

Содержание главы:

- Что такое цифро-аналоговое преобразование?
- Формирование напряжений при помощи цифро-аналогового преобразователя (ЦАП).
- Основные сведения об операционных усилителях.
- Наследование и производные классы.
- Класс для ЦАП.
- Спецификаторы доступа.

6.1. Введение

Цифро-аналоговые преобразователи (ЦАП) являются неотъемлемой частью многих систем автоматического управления. В этой главе описывается принцип работы ЦАП и его типичное использование для формирования аналогового напряжения или тока. Вы научитесь создавать производные классы от уже существующих при разработке программ для управления цифро-аналоговым преобразователем. Новые классы унаследуют уже имеющиеся и добавляют дополнительные возможности, а также будут изменены унаследованные функции. По ходу изложения этого материала будет продолжено рассмотрение атрибутов и спецификаторов доступа.

6.2. Цифро-аналоговое преобразование

Цифро-аналоговое преобразование представляет собой формирование аналогового напряжения или тока при помощи нескольких цифровых сигналов, управляющих аналоговым выходом (**Рис. 6.1**). У некоторых типов цифро-аналоговых преобразователей данные вводятся в последовательной форме, а у других ЦАП входные данные представляются в параллельном виде.

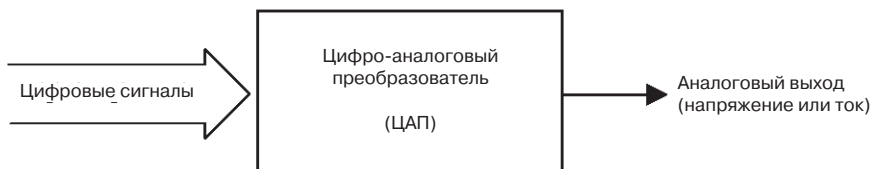


Рис. 6.1. Цифро-аналоговое преобразование

На практике применяются различные типы цифро-аналоговых преобразователей. Некоторые из них формируют выходное аналоговое напряжение путём интегрирования последовательности входных импульсов (например, преобразователи напряжение-частота). Другие используют распространённый способ цифро-аналогового преобразования, обсуждаемый здесь на примере ЦАП, используемого в этой главе. Для понимания принципа работы ЦАП нужно сначала изучить его главный элемент – *операционный усилитель*.

6.2.1. Основные сведения об операционном усилителе

Операционный усилитель является главным элементом многих аналоговых электронных систем. Условное графическое обозначение *операционного усилителя* показано на **Рис. 6.2**.

Операционный усилитель обладает очень высоким внутренним усилением и ему для работы достаточно очень малого входного тока, протекающего через его входы (обозначенные знаками $+$ и $-$). Отрицательный вход (со знаком $-$) является *инвертирующим входом*, вход со знаком $+$ *неинвертирующим входом*. В устройстве с операционными усилителями также имеются шины питания (обычно их не изображают), подключаемые к выводам питания верхнего и нижнего плеч. Если разность (дифференциал) входных напряжений ΔV положительна, когда на входе со знаком $+$ напряжение больше, чем на входе со знаком $-$, то на выходе операционного усилителя будет положительное напряжение. И наоборот, напряжение на выходе отрицательно при отрицательной разности ΔV .

Поскольку внутреннее усиление операционного усилителя очень велико, например, 1 миллион, то выходной сигнал величиной 10 В будет получен при входном сигнале всего 10 миллионных В между двумя входами ($\Delta V = 10$ мкВ, приставка «мк» означает «микро», т. е. 10^{-6}). Входной ток операционного усилителя крайне мал, например 200 нА, «н» означает «нано», 10^{-9} .

Операционные усилители лежат в основе многих схем. Расчёт большинства схем с операционными усилителями производится в предположении их очень большого усиления и практически полного отсутствия входного тока. Операционный усилитель на **Рис. 6.2** показан без подключенных к нему элементов. В таком включении на практике он не используется, между выходом и входом нужно подключать внешние элементы, так называемую *обратную связь*, улучшающую свойства схемы. Термин «обратная связь» означает передачу выходного сигнала обратно на вход операционного усилителя. Теперь давайте рассмотрим схему преобразования тока в напряжение (**Рис. 6.3**), которая положена в основу ЦАП на интерфейсной плате.

Входное напряжение V_{IN} вызывает протекание тока i через резисторы R_1 и R_F , как

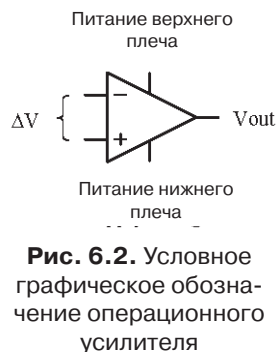


Рис. 6.2. Условное графическое обозначение операционного усилителя

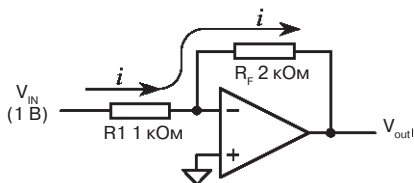


Рис. 6.3. Схема преобразователя тока в напряжение

показано на рисунке, обуславливая выходное напряжение V_{OUT} . Для расчёта выходного напряжения нам нужно знать величину тока i и значение напряжения на инвертирующем входе (и сопротивление резисторов R_1 и R_F разумеется).

Выходное напряжение V_{OUT} не может выходить за пределы напряжения питания для формирования напряжений вне диапазона питающего напряжения. Большинство операционных усилителей питаются напряжением не более $+15\text{ В}$ в верхнем и не менее -15 В в нижнем плечах. Опираясь на эти сведения, а также учитывая огромное внутреннее усиление операционного усилителя и пренебрегая его выходным сопротивлением, а входное дифференциальное напряжение ΔV будем считать не превышающим некоторого значения, скажем, 15 мкВ :

$$V_{\text{OUT}}(\text{max}) = \Delta V \times \text{Внутреннее усиление} \\ \pm 15\text{ В} = 15\text{ мкВ} \times 10^{-6}$$

В вышеприведённом примере выходное напряжение V_{OUT} не может выходить за пределы $\pm 15\text{ В}$, поэтому ΔV всегда будет меньше $\approx 15\text{ мкВ}$, если принять собственное усиление операционного усилителя равным миллиону.

Напряжение на инвертирующем входе равно напряжению на неинвертирующем входе плюс ΔV . Поскольку напряжение на неинвертирующем входе равно потенциалу земли (0 В , земля обозначена черточкой) и нам известно что разность напряжений (ΔV) между обоими входами не превышает 15 мкВ , то и напряжение на инвертирующем входе не будет превышать $0\text{ В} + 15\text{ мкВ} = 15\text{ мкВ}$. Это напряжение на инвертирующем входе настолько мало, что его можно считать нулевым или *виртуальной землёй*. Теперь, когда нам известно напряжение на инвертирующем входе, можно вычислить значение тока i и определить выходное напряжение V_{OUT} .

Ток течёт от высокого потенциала к более низкому и поэтому ток i потечёт как показано на **Рис.6.4**, поскольку напряжение V_{IN} больше напряжения на инвертирующем входе операционного усилителя, обладающего потенциалом виртуальной земли ($\approx 0\text{ В}$). Если напряжение на одном выводе резистора R_1 составляет 1 В и 0 В на другом, то ток i будет вычисляться так:

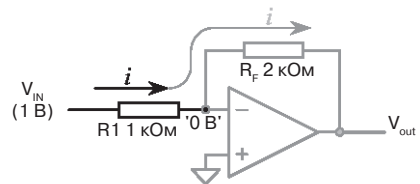


Рис. 6.4. Вычисление входного тока i

$$\begin{aligned} \text{Ток } (i) &= \frac{\text{Напряжение}}{\text{Сопротивление}} \\ &= \frac{1\text{ В} - 0\text{ В}}{1\text{ кОм}} \\ &= 1\text{ мА} \end{aligned}$$

Буква «к» обозначает одну тысячу Ом.

Поскольку входного тока у операционного усилителя практически нет, то весь ток i должен протекать через R_F через выход усилителя и его нагрузку (она не показана).

Напряжение V_{OUT} равно напряжению на инвертирующем входе плюс падение напряжения на резисторе R_F . По формуле $U = I \times R$ напряжение на резисторе R_F равно произведению тока i и сопротивления резистора R_F .

$$\begin{aligned} V_{R2} &= 1 \text{ мА} \times 2 \text{ кОм} \\ &= 2 \text{ В} \end{aligned}$$

Как упоминалось выше, ток течёт от высокого потенциала к более низкому, поэтому с левой по схеме стороны резистора R_F напряжение больше, чем с правой стороны R_F . Поскольку на левой стороне потенциал составляет 0 В, а падение напряжения на R_F равно 2 В, то с правой стороны R_F будет потенциал –2 В. Так как правая сторона R_F соединена с выходом V_{OUT} , то V_{OUT} тоже будет равно –2 В.

Ток, протекающий через резистор обратной связи R_F в точке соединения с выходом операционного усилителя разделяется на две части: часть ответвляется в нагрузку операционного усилителя (подключена к выходу V_{OUT} , не показана) и оставшаяся часть тока протекает через выход операционного усилителя. Как происходит протекание тока через выход операционного усилителя станет понятно после рассмотрения *отрицательной обратной связи*.

В принципе, выход операционного усилителя может либо потреблять, либо отдавать ток, удерживая на инвертирующем входе потенциал 0 В. В схеме с отрицательной обратной связью (**Рис. 6.3** и **Рис. 6.4**) это происходит автоматически. Для организации отрицательной обратной связи нужно подать напряжение V_{OUT} обратно на инвертирующий вход, напрямую или посредством внешних элементов, обычно резисторов. Отрицательная обратная связь работает следующим образом.

Если предположить, что напряжение V_{IN} возрастает, то это приведёт к ослабленному возрастанию напряжения на инвертирующем входе; другими словами увеличится отрицательная разность потенциалов ΔV между неинвертирующим и инвертирующим входами. Такое возрастающее отрицательное дифференциальное напряжение вызывает значительное уменьшение выходного напряжения V_{OUT} , что, в свою очередь, приведёт к уменьшению напряжения на инвертирующем входе. Входное дифференциальное напряжение ΔV уменьшается и в конечном итоге наступает равновесное состояние. Такой процесс в операционных усилителях с отрицательной обратной связью происходит автоматически и очень быстро.

Положительная обратная связь, напротив, не обеспечивает саморегулирующегося выходного напряжения. Вместо этого выходное напряжение близко к значению источника питания либо одного, либо другого плеча. Если сменить полярность напряжения между неинвертирующим и инвертирующим входами, то выходное напряжение также поменяет свою полярность. Такой тип обратной связи используется в компараторах напряжения.

Зная, как работает схема преобразователя тока в напряжение, можно приступить к рассмотрению и изучению работы практических схем ЦАП.

6.2.2. Принципы работы ЦАП

Будут рассмотрены два способа цифро-аналогового преобразования, когда выходное аналоговое напряжение определяется несколькими цифровыми сигналами. В обоих случаях используется преобразование тока в напряжение, которое

было рассмотрено в предыдущем разделе. Различие заключается в способе построения входной цепи резисторов. Сначала рассмотрим схему ЦАП с суммирующим усилителем.

6.2.2.1. ЦАП с суммирующим усилителем

Схема на **Рис. 6.5** работает аналогично рассмотренному ранее преобразователю ток-напряжение (**Рис. 6.3** и **Рис. 6.4**). В предыдущей схеме ток i_T задавался при помощи только одного резистора, а в рассматриваемом случае суммирующего усилителя используются четыре резистора, каждый из которых формирует свой входной ток. В суммирующем усилителе происходит сложение токов всех входных резисторов, в результате получается ток i_T .

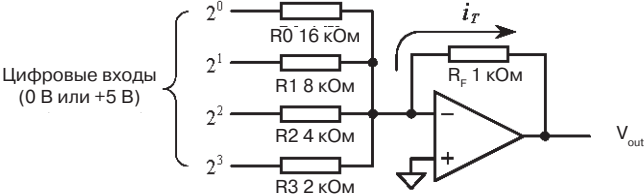


Рис. 6.5. ЦАП на основе суммирующего усилителя

Если бы все входные резисторы имели одинаковое сопротивление, то они могли бы давать либо нулевой ток (при нулевом значении на цифровом входе) или $\frac{1}{4}$ полного тока i_T (ВЫСОКИЙ логический уровень). Поэтому на выходе ЦАП могут быть следующие значения напряжений:

Выход ЦАП	Цифровые входы
0 В	На всех входах 0 В.
–1 В	На одном входе +5 В, на остальных 0 В.
–2 В	На двух входах +5 В, на остальных 0 В
–3 В	На трёх входах +5 В, на остальных 0 В
–4 В	На всех входах +5 В

Четыре одинаковых резистора дадут только пять уровней выходного напряжений с шагом 1 В. Зная, что четыре бита с разным весом (значениями) могут представлять шестнадцать различных чисел, то можно подобрать сопротивления резисторов таким образом, чтобы обеспечить шестнадцать различных значений тока i_T . Входные резисторы подобраны так, что они различаются между собой пропорционально целой степени числа 2.

Резистор $R0$ эквивалентен числу 2^0 , $R1$ эквивалентен 2^1 и т. д. (**Рис. 6.5**). Ток от логического входа 2^0 составляет $\frac{1}{8}$ от тока входа 2^3 , ток от входа 2^1 составляет $\frac{1}{4}$ от тока входа 2^3 и ток от входа 2^2 составляет $\frac{1}{2}$ от тока входа 2^3 . Комбинируя входные сигналы можно получить шестнадцать различных уровней выходного напряжения.

Этот способ цифро-аналогового преобразования обладает одним недостатком: для изготовления ЦАП с большей разрешающей способностью (т. е. с большим

числом цифровых входов) требуются резисторы с большей точностью сопротивления. Например, для 8-битного ЦАП (256 уровней) нужны резисторы с допуском сопротивления много меньше $1/256$ для избежания перекрытия уровней во всём диапазоне выходного напряжения ЦАП. Также значения сопротивлений не принадлежат стандартному ряду – это является более сложным препятствием. Другая схема называется *матрицей $R-2R$* и не имеет вышеуказанных недостатков.

6.2.2.2. ЦАП с матрицей $R-2R$

В этой схеме также используется преобразование ток-напряжение как в ЦАП с суммирующим усилителем. Эти два способа цифро-аналогового преобразования различаются схемой включения входных резисторов.

Входным напряжением массива резисторов является высокостабильный и точный источник опорного напряжения, обозначенный V_{REF} на **Рис. 6.6**. Он нужен для формирования точного значения протекающего по цепи тока. На схеме матрицы $R-2R$ можно увидеть четыре ключа (обычно это полупроводниковые ключи), каждый управляется индивидуально одним из четырёх цифровых входов. Ключи коммутируются либо с аналоговой землей (обозначенной чёрточкой на схеме) с потенциалом 0 В, либо с инвертирующим входом операционного усилителя, имеющий потенциал *виртуальной земли*. Термин «виртуальная земля» используется из-за того, что его потенциал очень близок к потенциалу 0 В (это описано в предыдущем разделе 6.2.1). Почему матрица $R-2R$ в таких ЦАП всегда коммутируется на нулевой потенциал объясняется далее.

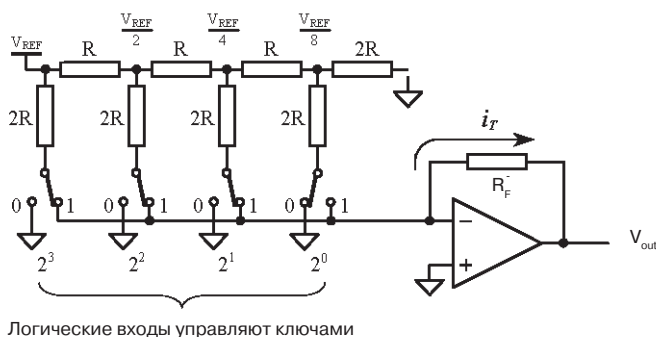


Рис. 6.6. ЦАП с матрицей $R-2R$

Ток протекающий через ключ, управляемый младшим значащим битом 2^0 составляет $1/8$ от тока ключа, управляемого старшим значащим битом 2^3 . Аналогично, ток ключа бита 2^1 составляет $1/4$ тока ключа 2^3 , а ток ключа 2^2 составляет $1/2$ тока ключа 2^3 . Когда ключ находится в положении «0», то ток через резистор $2R$ стекает на землю, минуя инвертирующий вход операционного усилителя. В положении «1» ток через резистор $2R$ протекает на инвертирующий вход операционного усилителя, суммируясь с токами других ключей, также находящихся в положении «1». Эти токи все вместе образуют ток i_T , а выходное напряжение получается так же, как и в схеме преобразователя ток-напряжение. Как упоминалось выше, протекающий через R_F ток приводит к отрицательному напряжению на выходе. Часто в ЦАП нужен положительный выход, и это достигается введением инвер-

тирующего усилителя, который меняет полярность выхода на противоположную. Схема, выполняющая такую операцию, находится и на интерфейсной плате, ей также будет уделено внимание.

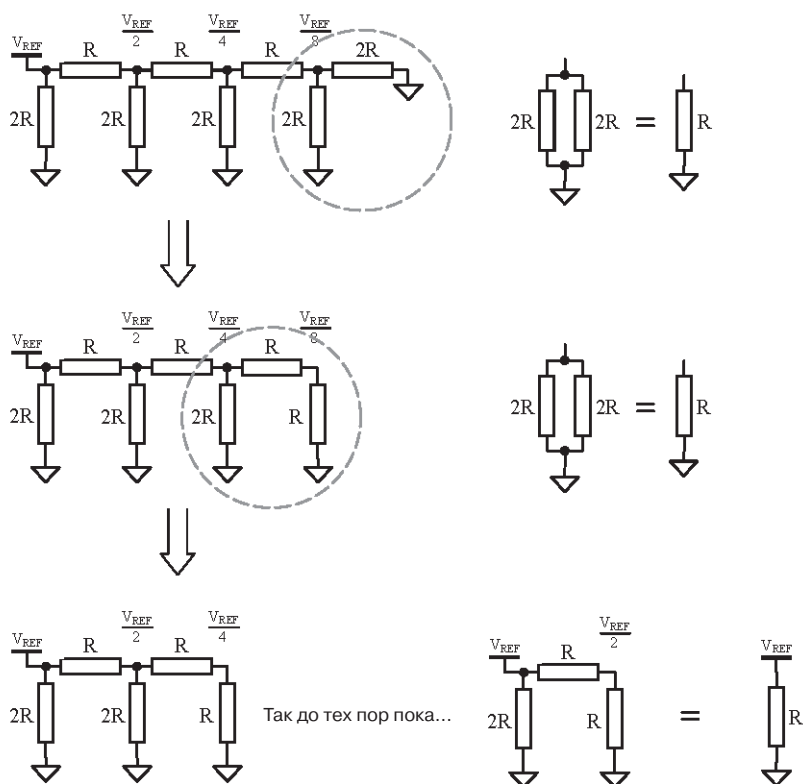


Рис. 6.7. Нагрузка для опорного напряжения

О матрице $R-2R$ следует сделать последнее замечание. Эта схема обладает очень хорошим качеством, выражающимся в том, что для источника опорного напряжения сопротивление матрицы остаётся всегда неизменным и равно R , независимо от состояний ключей. Это исключает помехи во время переключения, когда ключ находится «в воздухе» между двумя контактами. Источники опорного напряжения и стабилизаторы незначительно меняют своё напряжение при изменении сопротивления нагрузки. Поскольку сопротивление матрицы $R-2R$ неизменно, то напряжение источника будет более стабильным. На **Рис. 6.7** показано постоянное сопротивление R нагрузки источника опорного напряжения – помните, что ключи на концах резисторов $2R$ всегда имеют нулевой потенциал, независимо от того, в каком положении они находятся.

В заключении отметим, что матрица $R-2R$ обладает полезным свойством, заключающимся в простом соотношении между резисторами, равном 2. Также матрица обеспечивает неизменное сопротивление нагрузки для источника опорного напряжения, а его напряжение при этом получается более стабильным.

6.2.3. Работа DAC0800

На интерфейсной плате использован ЦАП DAC0800 – 8-битное устройство с токовым выходом. Этот выходной ток будет преобразовываться в напряжение преобразователем ток-напряжение. На **Рис. 6.8** показана блок-схема этого ЦАП.

Выход DAC0800 управляет протекающим через него током, который на самом деле втекает в выходной вывод ЦАП. Для получения выходного напряжения нужен преобразователь ток-напряжение, в данном случае единственный резистор R , один вывод которого подключен к выходу ЦАП, а другой к источнику опорного напряжения 0 В (Рис. 6.8). Протекающий через резистор ток i вызывает на нём падение напряжения ΔV ($\Delta V = i \times R$). Зная, что ток протекает от высокого потенциала к более низкому, то на выводе резистора, подключенном к выходу ЦАП, напряжение будет равно $0 \text{ В} + (-\Delta V) = -\Delta V$. Если ток i равен нулю, то и напряжение на резисторе R также равно нулю.

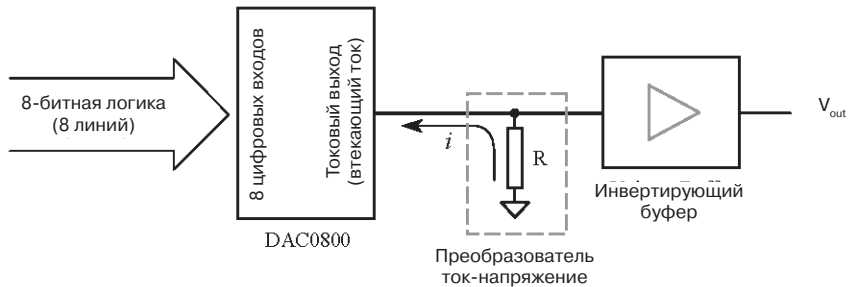


Рис. 6.8. Блок-схема ЦАП DAC0800 на интерфейсной плате

Таким образом, мы теперь располагаем программируемым выходным напряжением от нуля до $-\Delta V$, причём значение ΔV , соответствующее входному цифровому значению, зависит от сопротивления резистора R . Если выход ЦАП формирует напряжения всегда одного знака относительно нуля (в данном случае меньше нуля), то такой выход называется *однополярным (униполярным) выходом*. Другой тип выхода может давать как отрицательные, так и положительные напряжения и называется *биполярным выходом*. На **Рис. 6.9** показано подключение резистора для работы выхода в биполярном режиме.

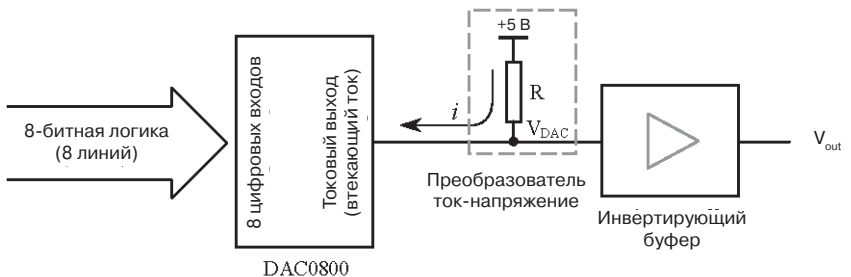


Рис. 6.9. Биполярное преобразование ток-напряжение

Выходное напряжение V_{DAC} биполярного преобразователя ток-напряжение рассчитывается по формуле:

$$\begin{aligned} V_{DAC} &= +5 \text{ В} - \text{Напряжение на } R \\ &= +5 \text{ В} - i \times R \end{aligned}$$

Знак минус в предыдущей формуле появился оттого, что ток i *втекает* в DAC0800, и оттого что ток течёт от высокого потенциала к более низкому – поэтому напряжение на выходе преобразователя ток-напряжение будет равно +5 В минус падение напряжения на резисторе R . При возрастании тока, втекающего в ЦАП, напряжение на его выходе падает от +5 В (на всех восьми цифровых входах 0), переходя через ноль и стремясь к –5 В (на всех восьми цифровых входах 1, и при соответствующем сопротивлении резистора R). Выход преобразователя ток-напряжение на интерфейсной плате может быть сконфигурирован либо как однополярный, либо как биполярный соответствующим подключением резистора.

Таблица 6.1. Выход преобразователя ток-напряжение

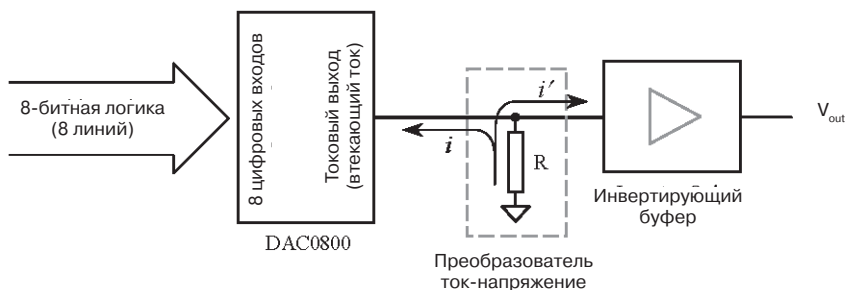
Цифровой вход ЦАП	Однополярный режим	Биполярный режим
255	–5 В	–5 В
0	0 В	+5 В

В **Табл. 6.1** сведены выходные напряжения преобразователя ток-напряжение для однополярного и биполярного подключений резистора R (сопротивление которого имеет соответствующее значение).

Обычно мы считаем, что наименьшему цифровому значению на входе ЦАП должно соответствовать наименьшее выходное напряжение, наибольшему входному значению (255 в случае 8-битного ЦАП) наибольшее выходное напряжение. Для удовлетворения этому условию выходное напряжение преобразователя ток-напряжение нужно инвертировать, чтобы в однополярном режиме на выходе было +5 В, когда все цифровые входы в единичном состоянии и 0 В при нуле на всех цифровых входах. В биполярном режиме на выходе должно быть +5 В, когда на всех входах единица, и –5 В при нуле на всех входах. На **Рис. 6.11** и **Рис. 6.12** показаны схемы буфера и инвертора, которые и выполняют такое преобразование.

Типичный инвертор имеет некоторый входной ток i' (**Рис. 6.10**). Если этот ток достаточно велик, то он будет заметно влиять на напряжение на резисторе R . Это получается из-за того, что теперь уже становится два тока i и i' , протекающие через резистор R , при этом i' создаёт напряжение ошибки $\Delta V_{\text{ERROR}} = i' \times R$. Полное напряжение на резисторе R будет равно $\Delta V = (i' + i) \times R$, равное истинному напряжению $(i \times R)$ плюс напряжение ошибки ΔV_{ERROR} .

Чтобы исключить протекание существенного тока с выхода ЦАП в инвертор между ними включают буфер. Буфер потребляет крайне малый ток (порядка 200 нА), которого недостаточно для сколько-нибудь заметного влияния на точность преобразования ток-напряжение в ЦАП.

Рис. 6.10. Влияние инвертора на напряжение ΔV

6.2.3.1. Схема буфера

Как упоминалось выше, этот узел выполняет буферирование напряжения с выхода преобразователя ток-напряжение в ЦАП. Буфер представляет собой специальную схему на операционном усилителе, обеспечивающую почти нулевой входной ток. Эта схема показана на **Рис. 6.11**.

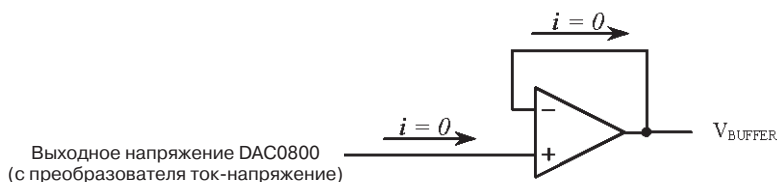


Рис. 6.11. Схема буфера

Вспомним, что входной ток операционного усилителя практически равен нулю, вследствие чего и протекающий от выхода преобразователя ток-напряжение DAC0800 ток на неинвертирующий вход операционного усилителя также равен нулю. Этот операционный усилитель охвачен отрицательной обратной связью. Выходное напряжение операционного усилителя V_{BUFFER} будет всегда обеспечивать практически нулевую разность между неинвертирующим и инвертирующим входами. Поэтому выходное напряжение будет в точности повторять входное или, иначе говоря, *буферизировать* выходное напряжение ЦАП, подаваемое на неинвертирующий вход.

6.2.3.2. Схема инвертора

Увеличивающееся цифровое значение на входах ЦАП вызывает уменьшение выходного напряжения (**Табл. 6.1**). Инвертор напряжения выполняет инверсию напряжения выхода преобразователя ток-напряжение ЦАП, что приводит к увеличению выходного напряжения при увеличении цифрового значения на входе ЦАП. Мы уже рассматривали схему, которая изменяет полярность входного сигнала: это преобразователь ток-напряжение. Эта схема инвертирует и увеличивает напряжение, причём увеличение (усиление) равно отношению сопротивлений резисторов R_F/R_1 . Если сопротивления резисторов R_F и R_1 будут одинаковыми, то будет единичное усиление — у нас получился *инвертор* (**Рис. 6.12**). Инвертор рассматривается так же, как это сделано в разделе 6.2.1.

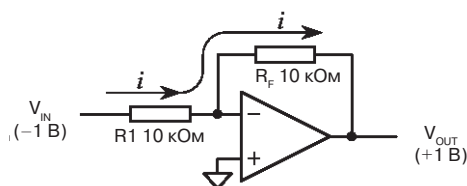


Рис. 6.12. Схема инвертора на операционном усилителе

6.2.4. Характеристики и параметры ЦАП

Качество ЦАП определяется несколькими основными характеристиками. Это время установления выхода, нелинейность и погрешность диапазона. Время установления выхода определяет, насколько быстро может изменяться и стабилизироваться в пределах половины младшего значащего бита сигнал на выходе ЦАП (младший значащий бит соответствует наименьшему изменению выходного напряжения, обусловленному сигналом цифрового входа 2^0). Типичное время установления выхода для DAC0800 составляет 100 нс.

Линейность характеризует максимальное отклонение от идеальной передаточной характеристики ЦАП во всём диапазоне входных значений. Передаточная характеристика определяет взаимосвязь между входными и выходными значениями ЦАП. В идеальном случае эта зависимость выражена прямой линией, когда выходное значение увеличивается пропорционально увеличению входного значения. На **Рис. 6.13** показана идеальная передаточная характеристика с двумя её независимыми искажениями: *ошибкой смещения* и *ошибкой крутизны*.

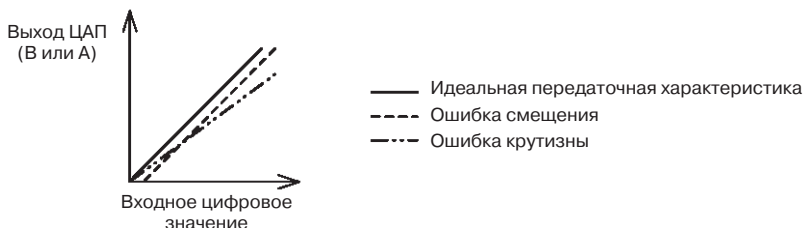


Рис. 6.13. Передаточная характеристика ЦАП

Ошибка смещения это напряжение (или ток) на выходе ЦАП при нулевом значении на входе. В идеальном случае ошибка смещения должна быть нулевой. Ошибка крутизны возникает при неодинаковом изменении выходного значения относительно входного сигнала.

Используемый в интерфейсной плате DAC0800 характеризуется следующими данными: его нелинейность в диапазоне допустимых температур составляет $\pm 0.1\%$ (соответственно линейность равна 99.9%), а погрешность диапазона ± 1 младшего значащего бита, означающая, что максимальный выходной сигнал будет отличаться от идеального (соответствующего идеальной передаточной характеристике) не более чем на ± 1 наименьшего возможного шага выходного сигнала.

6.3. Работа с цифро-аналоговым преобразователем

В предыдущей главе мы создали класс `ParallelPort`. Этот класс позволяет вводить и выводить данные через все три регистра параллельного порта компьютера. Он может использоваться для управления цифро-аналоговым преобразователем. Для работы с цифро-аналоговым преобразователем на интерфейсной плате нужна лишь часть возможностей класса `ParallelPort`. ЦАП должен лишь принимать 8-битное число от компьютера. Это можно сделать передачей 8-битного числа в регистр `BASE`. Посредством этого регистра можно одновременно выводить восемь цифровых сигналов, которыми можно будет воспользоваться на интерфейсной плате. Поскольку у нас уже есть полностью завершённый класс `ParallelPort`, то для управления цифро-аналоговым преобразователем можно воспользоваться им.

Далее в главе будет сделано следующее. Во-первых, будет написана программа с объектом класса `ParallelPort`, управляющая ЦАП. После запуска этой программы на выходе ЦАП появится напряжение, пропорциональное переданному ему 8-битному числу. Затем мы продолжим изучение механизма наследования на примере производного класса, представляющего ЦАП. Будет уделено внимание ограничениям, накладываемых атрибутами доступа. Затем будет подробно описано правильное использование атрибутов. Рассмотрим несколько различных версий этой программы, каждый раз с более интенсивным повторным использованием кода. Конечная программа будет использоваться в последующих главах в качестве объектно-ориентированной программы, управляющей ЦАП.

Первая программа работы с цифро-аналоговым преобразователем приведена в **Листинге 6.1**. Определение класса `ParallelPort` и его функций-членов такие же, как и в предыдущей главе. В функции `main()` создаётся объект класса `ParallelPort` с именем `D_to_A`. Затем следуют несколько пересылок различных данных в параллельный порт. После каждого вызова `WritePort0()` используется функция `getch()`. Функция `getch()` заставляет программу ждать реакции пользователя перед выполнением очередной инструкции. Это даёт время, чтобы произвести измерения на интерфейсной плате и убедиться в адекватности значения выходного напряжения ЦАП.

Таблица 6.2. Подключение ЦАП

Порт <code>BASE</code> (буфер <code>U13</code>)	<code>DAC0800</code> (<code>U3</code>)
D0	D0 (12)
D1	D1 (11)
D2	D2 (10)
D3	D3 (9)
D4	D4 (8)
D5	D5 (7)
D6	D6 (6)
D7	D7 (5)

Перед работой с программой для ЦАП нужно подготовить интерфейсную плату нижеописанным способом. К клеммнику J14 нужно подключить исправный гальванический элемент с напряжением 9 В. На контакты *LINK1* нужно установить джампер, установив тем самым однополярный режим (0...5 В). Соединениями по Табл. 6.2 параллельный порт подключается к 8-битному входу ЦАП.

Листинг 6.1. Цифро-аналогоое преобразование при помощи класса `ParallelPort`

```

/*****
    В этой программе используется класс ParallelPort,
    разработанный в предыдущей главе, для вывода
    байта данных в цифро-аналоговый преобразователь
    (ЦАП). ЦАП формирует напряжение, пропорциональное
    принятому байту.
*****/

#include <iostream.h>
#include <conio.h>

class ParallelPort
{
private:
    unsigned int BaseAddress;
    unsigned char InDataPort1;

public:
    ParallelPort(); // автоматический конструктор
    ParallelPort(int baseaddress); // конструктор
    void WritePort0(unsigned char data);
    void WritePort2(unsigned char data);
    unsigned char ReadPort1();
};

ParallelPort::ParallelPort()
{
    BaseAddress = 0x378;
    InDataPort1 = 0;
}

ParallelPort::ParallelPort(int baseaddress)
{
    BaseAddress = baseaddress;
    InDataPort1 = 0;
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress,data);
}

```

```
}

void ParallelPort::WritePort2(unsigned char data)
{
    outportb(BaseAddress+2,data ^ 0x0B);
}

unsigned char ParallelPort::ReadPort1()
{
    InDataPort1 = inportb(BaseAddress+1);
    // Инвертирование старшего бита для компенсации его
    // внутренней инверсии схемой параллельного порта.
    InDataPort1 ^= 0x80;
    // Сброс неиспользуемых битов D0, D1 и D2 при помощи маски.
    InDataPort1 &= 0xF8;
    return InDataPort1;
}

void main()
{
    ParallelPort D_to_A;

    cout << "Press a key ... " << endl;
    getch();
    D_to_A.WritePort0(0);

    cout << "Press a key ... " << endl;
    getch();
    D_to_A.WritePort0(32);

    cout << "Press a key ... " << endl;
    getch();
    D_to_A.WritePort0(64);

    cout << "Press a key ... " << endl;
    getch();
    D_to_A.WritePort0(128);

    cout << "Press a key ... " << endl;
    getch();
    D_to_A.WritePort0(255);
}
```

Аналоговый сигнал на интерфейсной плате можно измерить на контакте *VDAC*, соединённом с выводом 7 операционного усилителя LM357 (*U10B*). После каждого нажатия на клавишу программа будет выводить увеличенное значение, а ЦАП формировать соответствующее ему более высокое напряжение. Напряжение измеряется вольтметром на контактах *VDAC* и земли.

Выше упоминалось, что на интерфейсной плате расположены дополнительные узлы, реализующие одно- и биполярные типы выхода ЦАП. Переставьте джампер в положение *LINK2* и проверьте работу ЦАП в биполярном режиме, перезапустив программу.

6.4. Производные классы

В последнем примере для работы с ЦАП был использован класс `ParallelPort`. Этот класс создавался для работы с параллельным портом. Возможностей класса `ParallelPort` более чем достаточно для ЦАП. Тем не менее класс `ParallelPort` не очень подходит для цифро-аналогового преобразования. Вместо него лучше было бы иметь класс, у которого хотя бы название имеет отношение к цифро-аналоговому преобразованию. В таком случае очень удобно создать производный класс. На данный момент нужно поменять имя класса, что является первым шагом на пути к производному классу. Далее в книге мы столкнёмся с более сложными производными классами.

Одной из сложностей объектно-ориентированного программирования является повторное использование ранее написанного кода. В C++ есть превосходные механизмы для расширения функциональности существующего класса путём создания нового суперкласса. Как говорилось в главе 4, такие суперклассы называются *производными* классами. Производный класс создаётся на основе *базового* класса. Подразумевается, что производный класс имеет более конкретизированное назначение, в отличие от своего базового класса. В нашем случае классом общего назначения выступает `ParallelPort`, который можно рассматривать в качестве базового класса. Новый класс должен в большей степени соответствовать цифро-аналоговому преобразователю. В **Листинге 6.2** показан простейший путь создания нового класса.

Листинг 6.2. Образование класса ЦАП

```
class ParallelPort
{
    private:
        unsigned int BaseAddress;
        unsigned char InDataPort1;

    public:
        ParallelPort(); // автоматический конструктор
        ParallelPort(int baseaddress); // конструктор
        void WritePort0(unsigned char data);
        void WritePort2(unsigned char data);
        unsigned char ReadPort1();
};

class DAC : public ParallelPort
{
};
```

Класс создаётся несложным, чтобы мы могли сосредоточить внимание на принципах образования классов, а не на функционировании класса ЦАП. Класс `ParallelPort` определяется так же, как и в главе 5. В вышеприведённом листинге жирным шрифтом выделено только определение класса `DAC`. Первая строка определения нового класса:

```
class DAC : public ParallelPort
```

Здесь мы образуем новый класс `DAC`, а класс `ParallelPort` используем в качестве базового. Служебное слово `public` перед `ParallelPort` называется *спецификатором доступа*. Мы вернёмся к спецификаторам доступа после рассмотрения функций-членов и данных-членов, унаследованных от базового класса.

Унаследованные члены в классе `DAC`

Здесь под наследованием подразумевается тот факт, что создаваемый класс становится обладателем всех данных-членов и функций-членов. Взглянув на класс `DAC`, можно увидеть, что определение класса пустое. В пятой главе говорилось, что если мы сами не создадим конструктор и деструктор для класса, то они будут предоставлены компилятором. Итак, поскольку класс `DAC` унаследован от класса `ParallelPort`, то он наследует все его члены (функции и данные).

Перечислим все данные-члены класса `DAC`:

```
unsigned int BaseAddress;  
unsigned char InDataPort1;
```

Функции-члены класса `DAC`:

```
DAC(); // Конструктор, предоставленный компилятором (скрыт)  
ParallelPort();  
ParallelPort(int baseaddress);  
void WritePort0(unsigned char data);  
void WritePort2(unsigned char data);  
unsigned char ReadPort1();  
~ParallelPort(); // Унаследованный деструктор,  
                // созданный компилятором (скрыт).  
~DAC(); // Созданный компилятором деструктор (скрыт).
```

Примечание

Обратите внимание, что мы не писали деструкторы в определениях классов `ParallelPort` и `DAC`. Поэтому компилятор создаёт автоматические деструкторы, перечисленные выше, для каждого из этих классов. Созданные компилятором конструкторы и деструкторы названы скрытыми от программиста, потому что их нет в исходном коде. Более подробно деструкторы рассматриваются в главе 8.

Класс `DAC` имеет ту же структуру, что и `ParallelPort`, только у него автоматические конструктор и деструктор, созданные компилятором. Понятно, что сделанное нами образование класса является просто переименованием, поскольку класс

DAC не имеет никаких дополнительных возможностей. Этот класс может делать то же, что и `ParallelPort` и ничего более. Но если нам понадобится усовершенствовать класс DAC, то нужно просто добавить новые данные-члены и функции-члены в новый класс DAC, а затем дать определения добавленным функциям.

Инстанцирование объектов класса DAC (вызов конструкторов)

Здесь будет рассматриваться вопрос «Мы можем использовать этот класс?» Чтобы инстанцировать объект типа DAC нужно вызвать конструктор класса DAC. Его можно вызвать в выражении объявления нового объекта `D_to_A`:

```
DAC D_to_A;
```

Когда вызывается конструктор производного класса происходит вызов конструктора унаследованного базового класса, принимающий какие-либо параметры, переданные ему конструктором производного класса (это будет объяснено позже). Конструктор базового класса инстанцирует свои члены и выполняет все требуемые от него действия. После завершения своей работы он передаёт управление обратно в начало тела конструктора производного класса. Затем конструктор производного класса инстанцирует свои члены и выполняет все предписанные ему действия.

В нашей программе класс DAC мы сами не писали конструктор, поэтому компилятор создал свой автоматический компилятор (не принимающий параметров). Это тот самый конструктор, который вызывается в вышеприведённом выражении. *До того*, как выполнить свои действия, он вызывает конструктор унаследованного класса. Так как конструктор производного класса не передаёт аргументы конструктору базового класса, от будет вызван конструктор `ParallelPort()`, а не `ParallelPort(int baseaddress)` для выделения памяти для унаследованных членов базового класса. Если взглянуть на тело конструктора в **Листинге 6.1**, то можно увидеть, что он инициализирует `BaseAddress` значением `0x378` (а `InDataPort1` нулём). Затем автоматический конструктор класса DAC выделяет память для каких-либо добавленных членов (их нет) и выполняет пустое тело.

Внешняя функция `WritePort0()` унаследована от класса `ParallelPort`. Поэтому её можно использовать для передачи данных (в нашем случае числа 255) через параллельный порт, базовый адрес которого инициализирован значением `0x378`, при помощи следующего выражения:

```
D_to_A.WritePort0(255);
```

У этой программы есть недостаток – класс DAC не предоставляет возможности указать адрес регистра BASE параллельного порта. Это не подходит тем пользователям, у которых регистр BASE расположен по адресу, отличающемуся от `0x378`. Поэтому нам нужен некоторый механизм, позволяющий инициализировать член данных `BaseAddress` подходящим значением. Обычно это удобнее всего делать в конструкторе. Так же, как и на ранних стадиях развития класса `ParallelPort`, создаваемый компилятором конструктор класса DAC не подходит для нашей программы. Нам нужно самим написать конструкторы и обеспечить возможность указания значения `BaseAddress`. Усовершенствованный класс DAC приведён в **Листинге 6.3**.

Листинг 6.3. Усовершенствованный класс ЦАП

```
class ParallelPort
{
    private:
        unsigned int BaseAddress;
        unsigned char InDataPort1;

    public:
        ParallelPort();
        ParallelPort(int baseaddress);
        void WritePort0(unsigned char data);
        void WritePort2(unsigned char data);
        unsigned char ReadPort1();
};

class DAC : public ParallelPort
{
    public:
        DAC(); // автоматический конструктор.
        DAC(int baseaddress); // конструктор.
};
```

На этот раз мы пользуемся уже известным нам из класса `ParallelPort` способом и создаём таким образом два конструктора класса `DAC`. Обратите внимание, что компилятор не будет создавать автоматический конструктор, поскольку мы предоставили конструктор `DAC(int baseaddress)`, **Листинг 6.3**. Поэтому автоматический конструктор `DAC()` должны написать сами.

Теперь мы можем определить эти конструкторы путём введения выражений, которые должны задавать адрес регистр `BASE` инициализацией унаследованного собственного члена данных `BaseAddress`. Первая попытка определения конструкторов приведена в **Листинге 6.4**.

Листинг 6.4. Неработающий конструктор

```
DAC::DAC() // Работает.
{
}

DAC::DAC(int baseaddress)
{
    BaseAddress = baseaddress; // Не работает!
}
```

В следующем выражении использован пользовательский автоматический конструктор `DAC()` для инстанцирования объекта `D_to_A` типа `DAC`:

```
DAC D_to_A;
```

Этот автоматический конструктор класса `DAC` будет работать так же, как и автоматический конструктор, создаваемый компилятором. Сначала он вызовет унаследованный конструктор базового класса, который тоже не принимает ар-

гументы – автоматический конструктор базового класса `ParallelPort()`. Этот конструктор инстанцирует унаследованные члены базового класса, инициализирует собственный член данных `InDataPort1` нулём, а собственный член данных `BaseAddress` значением `0x378`. Потом выполняется автоматический конструктор класса `DAC` для инстанцирования добавленных им членов (их нет), затем выполняется его тело – в данном случае оно пустое.

Производный класс `DAC` наследует члены своего базового класса `ParallelPort`, но у него нет доступа к собственным членам унаследованного базового класса (независимо от использованного модификатора доступа – в данном случае `public`). Это говорит о том, что конструктор `DAC(int baseaddress)` из **Листинга 6.4** не будет иметь доступа к унаследованному члену данных базового класса `BaseAddress` (объявленного как «собственный»). А значит он не может быть откомпилирован, и, как следствие, не будет работать!

Эта задача имеет следующее решение. Нужно не напрямую обращаться к собственному члену данных, унаследованному от базового класса `ParallelPort` из функции производного класса `DAC`, а вызывать для этого публичную функцию, унаследованную от базового класса, которая может изменять член данных `BaseAddress` своего же класса, как показано на **Рис. 6.14**. Работающее определение конструктора класса `DAC` дано в **Листинге 6.5**.

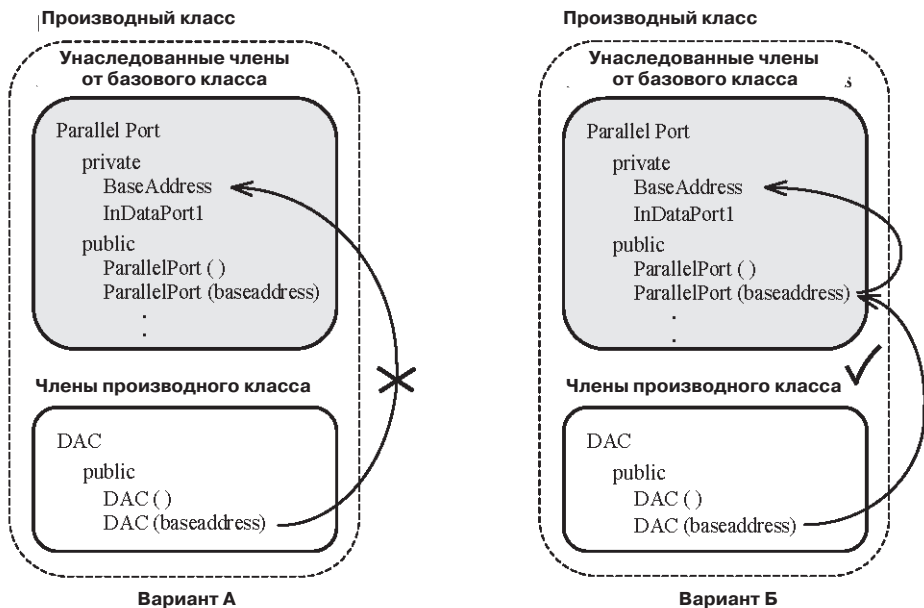


Рис. 6.14. Доступ к унаследованному собственному члену данных `BaseAddress`

Листинг 6.5. Исправленное определение конструктора класса `DAC`

```
DAC::DAC()
{
```

```

}

DAC::DAC(int baseaddress) : ParallelPort(baseaddress)
{
}

```

Выделенная жирным часть строки в **Листинге 6.5** представляет собой новый механизм, позволяющий собственному члену данных `BaseAddress` принимать значения аргумента, передаваемого во время создания объекта `D_to_A` типа `DAC`:

```
DAC D_to_A(0x3BC);
```

Это выражение приводит к вызову соответствующего конструктора класса `DAC`. Так как адрес регистра `BASE` параллельного порта передан в виде аргумента во время инстанцирования объекта `D_to_A`, то будет вызван конструктор `DAC(int baseaddress)`, а не автоматический конструктор `DAC()`. Сразу после вызова конструктор вызывает соответствующий конструктор унаследованного базового класса, передавая ему свой аргумент. Выделенная строка:

```
DAC::DAC(int baseaddress) : ParallelPort(baseaddress)
```

сообщает компилятору, что конструктор производного класса `DAC(int baseaddress)` передаёт параметр `baseaddress` унаследованному конструктору базового класса `ParallelPort(baseaddress)`. Этот конструктор базового класса инстанцирует свои члены, а затем инициализирует собственный член данных `BaseAddress` значением параметра `baseaddress` (и инициализирует `InDataPort1` нулём). Сразу после этого конструктор `DAC(int baseaddress)` должен инстанцировать свои дополнительные члены (их нет), а затем выполнить инструкции, содержащиеся в его теле.

Так можно инициализировать недоступный собственный член данных `BaseAddress`, унаследованный от класса `ParallelPort` – при минимуме кода благодаря наследованию.

Теперь мы можем вернуться к вопросам использования этого класса в программе для выполнения цифро-аналогового преобразования. Вся программа приведена в **Листинге 6.6**. Вместо инстанцирования объекта `ParallelPort` эта программа инстанцирует объект типа `DAC`. Программа работает так же, как и программа из **Листинга 6.1**.

Листинг 6.6. Цифро-аналоговое преобразование при помощи класса `DAC`

```

/*****
    Новый класс DAC используется в функции main()
    для записи последовательности байтов данных
    в цифро-аналоговый преобразователь.
*****/

#include <iostream.h>
#include <conio.h>

class ParallelPort
{

```

```

private:
    unsigned int BaseAddress;
    unsigned char InDataPort1;

public:
    ParallelPort(); // автоматический конструктор
    ParallelPort(int baseaddress); // конструктор
    void WritePort0(unsigned char data);
    void WritePort2(unsigned char data);
    unsigned char ReadPort1();
};

ParallelPort::ParallelPort()
{
    BaseAddress = 0x378;
    InDataPort1 = 0;
}

ParallelPort::ParallelPort(int baseaddress)
{
    BaseAddress = baseaddress;
    InDataPort1 = 0;
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress, data);
}

void ParallelPort::WritePort2(unsigned char data)
{
    outportb(BaseAddress+2, data ^ 0x0B);
}

unsigned char ParallelPort::ReadPort1()
{
    InDataPort1 = inportb(BaseAddress+1);
    // Инвертирование старшего бита для компенсации его
    // внутренней инверсии схемой параллельного порта.
    InDataPort1 ^= 0x80;
    // Сброс неиспользуемых битов D0, D1 и D2 при помощи маски.
    InDataPort1 &= 0xF8;
    return InDataPort1;
}

class DAC : public ParallelPort
{
public:
    DAC();
    DAC(int baseaddress);

```

```
};

DAC::DAC()
{
}

DAC::DAC(int baseaddress) : ParallelPort(baseaddress)
{
}

void main()
    DAC D_to_A;

    cout << "Press a key ... " << endl;
    getch();
    D_to_A.WritePort0(0);

    cout << "Press a key ... " << endl;
    getch();
    D_to_A.WritePort0(32);

    cout << "Press a key ... " << endl;
    getch();
    D_to_A.WritePort0(64);

    cout << "Press a key ... " << endl;
    getch();
    D_to_A.WritePort0(128);

    cout << "Press a key ... " << endl;
    getch();
    D_to_A.WritePort0(255);
}
```

6.5. Добавление членов к производному классу

После рассмотрения простого определения класса для усвоения принципов наследования и образования классов, мы можем приступить к добавлению новой функциональности к производному классу. В следующей программе к классу DAC будут добавлены член данных и функция-член. Член данных будет запоминать последнее выведенное в ЦАП значение. Функция будет представлять собой внешний интерфейс, позволяющий узнать, какое значение выводилось в ЦАП в последний раз. В классе будет модифицированная функция унаследованной функции WritePort0(), которая будет запоминать последнее выведенное значение. Сейчас мы посмотрим, как реализовать следующее:

1. Добавить новые члены к производному классу.
2. Модифицировать унаследованную функцию.

Новое определение класса DAC и определение его функций-членов приведены в **Листинге 6.7**. Заметьте, что в этом листинге мы повторно объявили функцию `WritePort0()` в качестве функции-члена класса DAC. Этого требует язык C++ при изменении тела унаследованной функции `WritePort0()`. Дополнительную информацию смотрите в разделе 6.5.2.

Листинг 6.7. Класс DAC с добавлением новых членов

```
class DAC : public ParallelPort
{
    private:
        unsigned char LastOutput;

    public:
        public:
            DAC();
            DAC(int baseaddress);
            void WritePort0(unsigned char data);
            unsigned char GetLastOutput();
};

DAC::DAC()
{
    LastOutput = 0;
}

void DAC::DAC(int baseaddress) : ParallelPort(baseaddress)
{
    LastOutput = 0;
}

void DAC::WritePort0(unsigned char data)
{
    outportb(BaseAddress,data); // Не будет работать!
    LastOutput = data;
}

unsigned char DAC::GetLastOutput()
{
    return LastOutput;
}
```

Собственный член данных `LastOutput` добавлен к классу DAC для хранения значения, выведенного функцией `WritePort0()`. Добавлена также функция-член `GetLastOutput()` к производному классу. Она возвращает значение `LastOutput`. Поэтому единственным выражением в теле функции `GetLastOutput()` будет:

```
return LastOutput;
```

Любая функция в программе, которой понадобится последнее выведенное в ЦАП значение (через регистр `BASE`), должна вызывать функцию `GetLastOutput()`

класса DAC. Это единственный способ доступа к значению члена данных `LastOutput`, который гарантирует, что никакие внешние по отношению к классу DAC функции не могут получить к нему прямой доступ. Это ещё раз демонстрирует управление доступом к собственным данным-членам класса в объектно-ориентированном программировании.

Оба конструктора были изменены так, чтобы они могли инициализировать `LastOutput` нулём путём добавления выражения:

```
LastOutput = 0;
```

Поэтому во время instantiation объекта типа DAC конструктор инициализирует новый член данных `LastOutput` нулём.

Тем не менее, определение функции `WritePort0()` в **Листинге 6.7** работать не будет. Так происходит из-за того, что функция `WritePort0()` производного класса DAC пытается использовать унаследованный член данных `BaseAddress`, являющийся собственным членом базового класса. Хотя собственные члены базового класса и наследуются, но производные классы к ним доступа не имеют, что иллюстрирует **Рис. 6.15**.

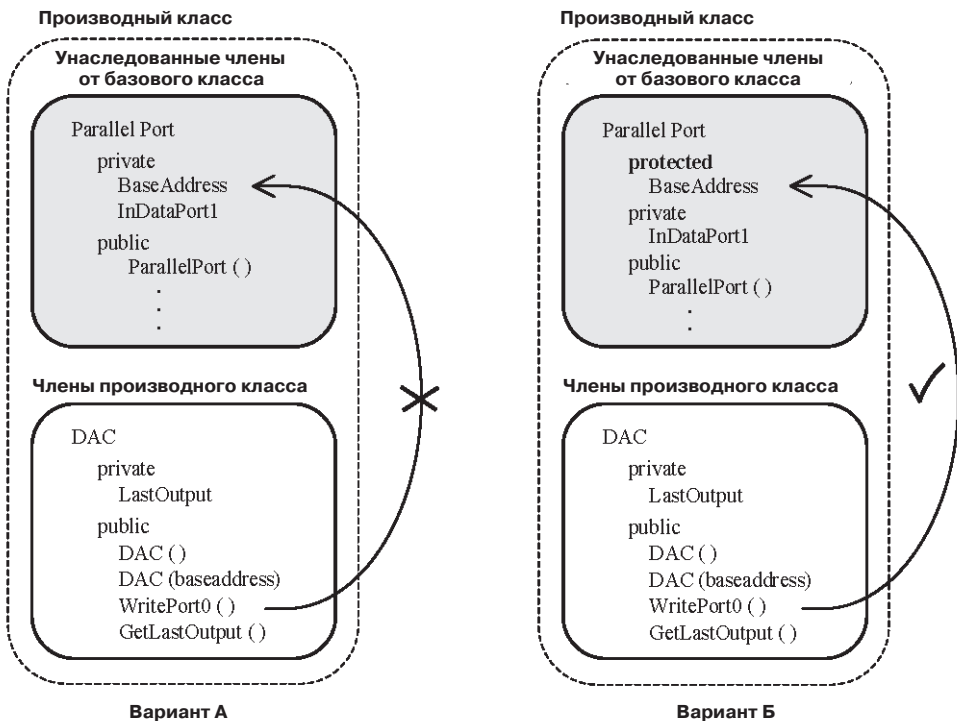


Рис. 6.15. Собственные и защищённые члены

Одним из способов обеспечения доступа к `BaseAddress` является «ослабление» его атрибутов доступа. Это можно сделать, объявив `BaseAddress` в базовом классе как `protected`. Тогда `BaseAddress` доступен всем функциям всех производных классов.

водных классов, образованных от базового со *спецификатором доступа* `public` или `protected`. Спецификаторы доступа описаны далее более подробно в разделе 6.5.1. Изменённое определение базового класса приведено в **Листинге 6.8**.

Примечание

Объявляйте переменные как `private`, если вы не планируете создавать от этого класса производные классы. Когда вы хотите использовать класс в качестве базового при создании новых классов, то нужен тщательный подход к выбору тех переменных, к которым должны иметь доступ производные классы. Только эти переменные нужно объявлять как `protected`.

Листинг 6.8. Базовый класс `ParallelPort` – `BaseAddress` защищённый член

```
class ParallelPort
{
    protected:
        unsigned int BaseAddress;

    private:
        unsigned char InDataPort1;

    public:
        ParallelPort();
        ParallelPort(int baseaddress);
        void WritePort0(unsigned char data);
        void WritePort2(unsigned char data);
        unsigned char ReadPort1();
};

class DAC : public ParallelPort
{
    private:
        unsigned char LastOutput;

    public:
        DAC();
        DAC(int baseaddress);
        void WritePort0(unsigned char data);
        unsigned char GetLastOutput();
};
```

Вышеприведённое определение класса используется в программе в **Листинге 6.9**.

6.5.1. Спецификаторы доступа

Рассмотрим строку:

```
class DAC : public ParallelPort
```

Служебное слово `public` в этой строке является *спецификатором доступа*. Спецификаторы доступа изменяют атрибуты доступа следующим образом. Защищённые переменные класса `ParallelPort` доступны в функциях-членах класса `DAC` тогда, и только тогда, когда класс `DAC` образован от `ParallelPort` со спецификатором доступа базового класса `public` или `protected`. Существует также спецификатор доступа `private`. На **Рис. 6.16** показано, как спецификаторы доступа влияют на атрибуты доступа унаследованных членов.

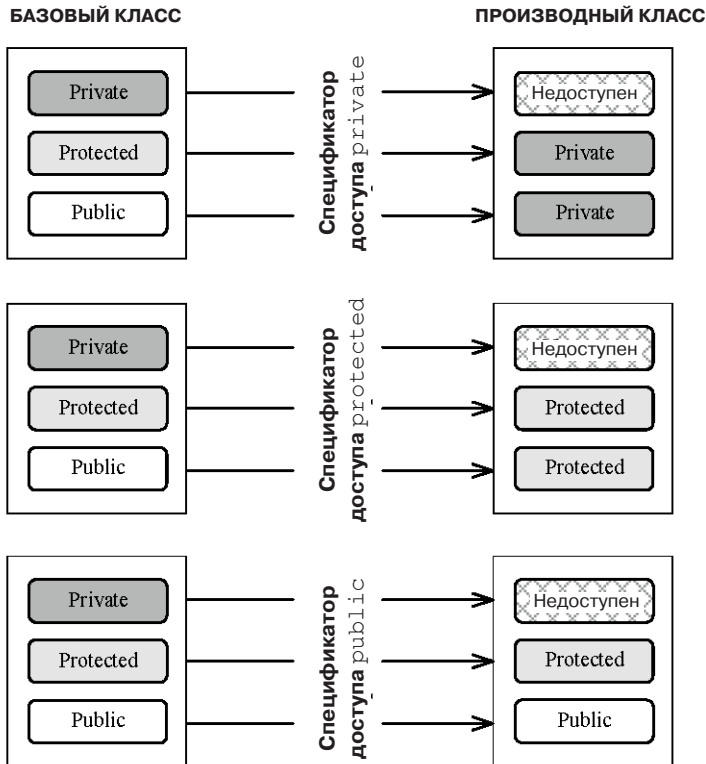


Рис. 6.16. Спецификаторы доступа определяют атрибуты доступа в производном классе

Спецификатор доступа `public`

Если образование класса происходит с использованием спецификатора доступа базового класса `public`, то все унаследованные публичные члены базового класса с атрибутом `public` останутся `public` и в производном классе, а все защищённые члены с атрибутом `protected` останутся `protected`. Все унаследованные собственные члены `private` будут по-прежнему собственными членами базового класса и поэтому будут недоступными для производного класса.

Спецификатор доступа `protected`

При образовании класса спецификатором доступа базового класса `protected` все унаследованные публичные члены с атрибутом `public` и все защищённые

члены с атрибутом `protected` базового класса в производном классе будут иметь атрибут `protected`. Все унаследованные собственные члены с атрибутом `private` останутся собственными членами базового класса и производный класс не будет иметь к ним доступа.

Спецификатор доступа `private`

Когда класс образуется со спецификатором доступа базового класса `private`, то все унаследованные публичные члены с атрибутом `public` и все защищённые члены с атрибутом `protected` станут собственными членами с атрибутом `private` в производном классе. Все унаследованные собственные члены с атрибутом `private` останутся собственными членами базового класса и производный класс не будет иметь к ним доступа.

6.5.2. Полиморфные функции

Попытка переопределить функцию-член в **Листинге 6.9** вполне допустима. Функция-член `WritePort0()` класса `DAC` унаследована им от базового класса `ParallelPort`. Для того, чтобы можно было переопределять унаследованные функции в производных классах, эти функции нужно явно указать в определении производного класса, как было сделано в **Листингах 6.9** и **6.7**. В этом примере две функции `WritePort0()`: одна из них принадлежит классу `ParallelPort`, а другая классу `DAC`. Существование в иерархии классов нескольких разных функций с одинаковыми именами представляет собой *полиморфизм*. Эти функции имеют не только одинаковые имена, но и одинаковое количество и тип параметров, расположенных в одинаковой последовательности.

Объявление класса `DAC` представлено на **Листинге 6.8**. Хотя функция `WritePort0()` и унаследована от класса `ParallelPort`, тем не менее она заново определяется в классе `DAC`. Это позволяет переопределить тело функции `WritePort0()` так, чтобы она соответствовала потребностям класса `DAC`.

Примечание

Между перегрузкой функций и полиморфизмом есть чёткое различие. Перегруженные функции имеют одинаковые имена. Они различаются по количеству и типу передаваемых им параметров. Перегруженные функции также могут не быть членами класса.

Полный текст программы, которая компилируется без ошибок, приведён в **Листинге 6.9**.

Листинг 6.9. Цифро-аналоговое преобразование с использованием доработанного объекта класса `DAC`

```

/*****
В этой программе ликвидирована ошибка компилирования
путём замены атрибутов доступа к члену BaseAddress
в базовом классе ParallelPort с private на protected.
Теперь функции производного класса, образованного

```

со спецификатором доступа `public`, имеют доступ к унаследованному члену данных `BaseAddress`. Доступ есть только у производных классов от этого базового класса и у самого базового класса. Функция `WritePort0()` переобъявлена в производном классе и теперь её изменять, не вызывая при этом ошибок компиляции.

*****/

```
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include <dos.h>

class ParallelPort
{
    protected:
        unsigned int BaseAddress;

    private:
        unsigned char InDataPort1;

    public:
        ParallelPort();
        ParallelPort(int baseaddress);
        void WritePort0(unsigned char data);
        void WritePort2(unsigned char data);
        unsigned char ReadPort1();
};

ParallelPort::ParallelPort()
{
    BaseAddress = 0x378;
    InDataPort1 = 0;
}

ParallelPort::ParallelPort(int baseaddress)
{
    BaseAddress = baseaddress;
    InDataPort1 = 0;
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress,data);
}

void ParallelPort::WritePort2(unsigned char data)
{
    outportb(BaseAddress+2,data ^ 0x0B);
}
```

```

}

unsigned char ParallelPort::ReadPort1()
{
    InDataPort1 = inportb(BaseAddress+1);
    // Инвертирование старшего бита для компенсации его
    // внутренней инверсии схемой параллельного порта.
    InDataPort1 ^= 0x80;
    // Сброс неиспользуемых битов D0, D1 и D2 при помощи маски.
    InDataPort1 &= 0xF8;
    return InDataPort1;
}

class DAC : public ParallelPort
{
private:
    unsigned char LastOutput;

public:
    DAC();
    DAC(int baseaddress);
    void WritePort0(unsigned char data);
    unsigned char GetLastOutput();
};

DAC::DAC()
{
    LastOutput = 0;
}

DAC::DAC(int baseaddress) : ParallelPort(baseaddress)
{
    LastOutput = 0;
}

void DAC::WritePort0(unsigned char data)
{
    outportb(BaseAddress,data);
    LastOutput = data;
}

unsigned char DAC::GetLastOutput()
{
    return LastOutput;
}

void main()
    DAC D_to_A;

    D_to_A.WritePort0(0);

```

```
// printf("\nDAC byte: %3d    ", D_to_A.LastOutput); // Не работает.
// Почему?
printf("\nDAC byte: %3d    ", D_to_A.GetLastOutput());
cout << "    Measure voltage and press a key" << endl;
getch();

D_to_A.WritePort0(32);
printf("\nDAC byte:%3d    ", D_to_A.GetLastOutput());
cout << "    Measure voltage & then press a key" << endl;
getch();

D_to_A.WritePort0(64);
printf("\nDAC byte:%3d    ", D_to_A.GetLastOutput());
cout << "    Measure voltage and press a key" << endl;
getch();

D_to_A.WritePort0(128);
printf("\nDAC byte:%3d    ", D_to_A.GetLastOutput());
cout << "    Measure voltage and press a key" << endl;
getch();

D_to_A.WritePort0(255);
printf("\nDAC byte:%3d    ", D_to_A.GetLastOutput());
cout << "    Measure voltage and press a key" << endl;
getch();
}
```

В вышеприведённой программе во время инстанцирования объекта `D_to_A` типа `DAC` вызывается конструктор `DAC()`. Рассмотрим определение функции автоматического конструктора `DAC()`; там происходит вызов автоматического конструктора класса `ParallelPort` перед выполнением тела конструктора `DAC()`. Автоматический конструктор класса `ParallelPort` инициализирует член `BaseAddress` значением `0x378` (а `InDataPort1` инициализирует нулём). Затем начинается выполняться тело конструктора `DAC()`. Там происходит инициализация собственного члена `LastOutput` класса `DAC` нулём.

Заметим, что когда на экран нужно вывести значение `LastOutput`, то вызывает функцию `GetLastOutput()`. Так нужно делать из-за того, что функция `printf()` не может получить прямой доступ к собственному члену данных `LastOutput`.

Определение функции `WritePort0()` можно немного изменить, чтобы восстановить атрибут `private` для `BaseAddress` по следующим причинам. Посмотрите на функцию `WritePort0()` из **Листинга 6.9**, приведённую в **Листинге 6.10**.

Листинг 6.10. Функция `WritePort0()` класса `DAC`

```
Void DAC::WritePort0(unsigned char data)
{
    outportb(BaseAddress, data);
    LastOutput = data;
}
```


Обращение к `BaseAddress` происходит только во время передачи данных в порт. Это может происходить в полиморфной функции `WritePort0()` класса `ParallelPort`. Она может беспрепятственно обращаться к члену `BaseAddress`, поскольку функция и этот член данных принадлежат одному классу. Можно вызвать полиморфную функцию `WritePort0()` класса `ParallelPort` из полиморфной функции `WritePort0()` класса `DAC` при помощи оператора разрешения контекста, представляющего собой два двоеточия (`::`). Эта процедура показана на **Рис. 6.17**, это модифицированный фрагмент из **Листинга 6.10** и он приведён в **Листинге 6.11**.

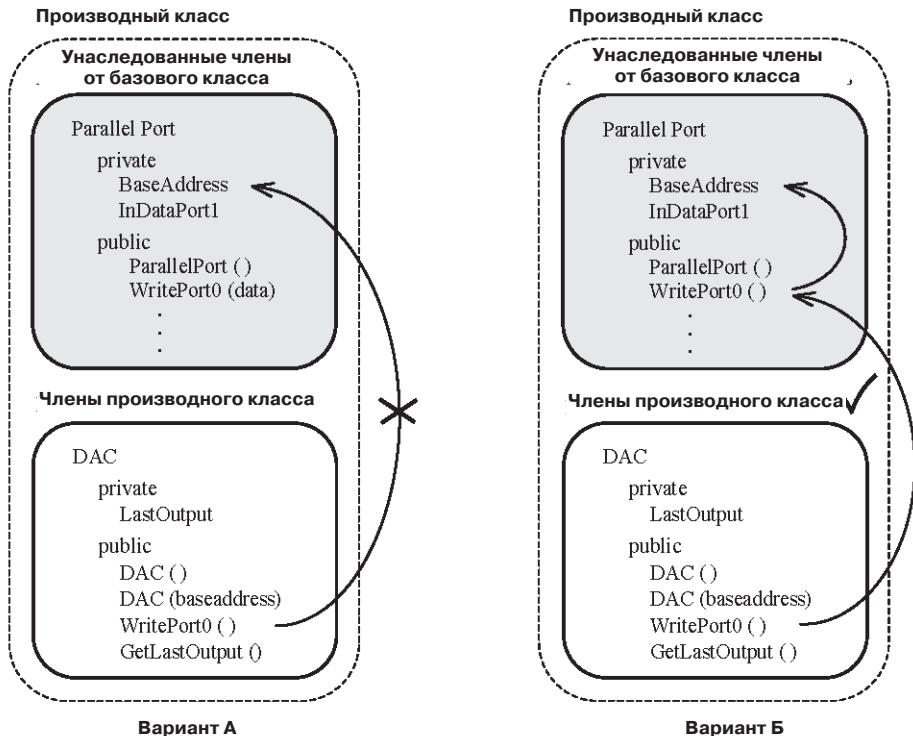


Рис. 6.17. Использование унаследованных полиморфных функций (`BaseAddress` снова `private`)

Листинг 6.11. Вызов полиморфной функции базового класса

```
void DAC::WritePort0(unsigned char data)
{
    ParallelPort::WritePort0(data);
    LastOutput = data;
}
```

В определении класса `ParallelPort` у члена данных `BaseAddress` снова можно установить атрибут доступа `private`, как показано на **Рис. 6.17**. Новая и более совершенная программа представлена в **Листинге 6.12**.

Листинг 6.12. Использование полиморфных функций

```

/*****
В этой программе атрибут доступа члена данных
обратно изменён на private. Доступ к BaseAddress
осуществляется посредством полиморфной функции
WritePort0() базового класса, которая имеет
доступ к этому члену.
*****/

#include <iostream.h>
#include <stdio.h>
#include <conio.h>

class ParallelPort
{
    private:
        unsigned int BaseAddress;
        unsigned char InDataPort1;

    public:
        ParallelPort();
        ParallelPort(int baseaddress);
        void WritePort0(unsigned char data);
        void WritePort2(unsigned char data);
        unsigned char ReadPort1();
};

ParallelPort::ParallelPort()
{
    BaseAddress = 0x378;
    InDataPort1 = 0;
}

ParallelPort::ParallelPort(int baseaddress)
{
    BaseAddress = baseaddress;
    InDataPort1 = 0;
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress,data);
}

void ParallelPort::WritePort2(unsigned char data)
{
    outportb(BaseAddress+2,data ^ 0x0B);
}

```

```
}

unsigned char ParallelPort::ReadPort1()
{
    InDataPort1 = inportb(BaseAddress+1);
    // Инвертирование старшего бита для компенсации его
    // внутренней инверсии схемой параллельного порта.
    InDataPort1 ^= 0x80;
    // Сброс неиспользуемых битов D0, D1 и D2 при помощи маски.
    InDataPort1 &= 0xF8;
    return InDataPort1;
}

class DAC : public ParallelPort
{
private:
    unsigned char LastOutput;

public:
    DAC();
    DAC(int baseaddress);
    void WritePort0(unsigned char data);
    unsigned char GetLastOutput();
};

DAC::DAC()
{
    LastOutput = 0;
}

DAC::DAC(int baseaddress) : ParallelPort(baseaddress)
{
    LastOutput = 0;
}

void DAC::WritePort0(unsigned char data)
{
    ParallelPort::WritePort0(data);
    LastOutput = data;
}

unsigned char DAC::GetLastOutput()
{
    return LastOutput;
}

void main()
{
    DAC D_to_A;

    D_to_A.WritePort0(0);
}
```

```

printf("\nDAC byte: %3d    ", D_to_A.GetLastOutput());
cout << "    Measure voltage and press a key" << endl;
getch();

D_to_A.WritePort0(32);
printf("\nDAC byte:%3d    ", D_to_A.GetLastOutput());
cout << "    Measure voltage and press a key" << endl;
getch();

D_to_A.WritePort0(64);
printf("\nDAC byte:%3d    ", D_to_A.GetLastOutput());
cout << "    Measure voltage and press a key" << endl;
getch();

D_to_A.WritePort0(128);
printf("\nDAC byte:%3d    ", D_to_A.GetLastOutput());
cout << "    Measure voltage and press a key" << endl;
getch();

D_to_A.WritePort0(255);
printf("\nDAC byte:%3d    ", D_to_A.GetLastOutput());
cout << "    Measure voltage and press a key" << endl;
getch();
}

```

Освоив элегантный способ работы с собственными членами данных базового класса из производного класса, мы можем уже завершить совершенствование класса DAC, изменив имя функции WritePort0() на более подходящее. Давайте назовём функцию WritePort0() класса DAC SendData().

Определение класса и полный текст программы, выполняющей те же действия, что и программа из **Листинга 6.12**, приведён в **Листинге 6.13**. Этот законченный класс DAC мы будем использовать, когда понадобится воспользоваться цифро-аналоговым преобразователем на интерфейсной плате в следующих главах.

Листинг 6.13. Функция WritePort0() класса DAC заменена на SendData()

```

/*****
    В этой программе функция WritePort0() класса
    DAC называется SendData(), что является более
    подходящим для класса DAC.
*****/

#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include <dos.h>

class ParallelPort
{
    private:

```

```

        unsigned int BaseAddress;
        unsigned char InDataPort1;

    public:
        ParallelPort();
        ParallelPort(int baseaddress);
        void WritePort0(unsigned char data);
        void WritePort2(unsigned char data);
        unsigned char ReadPort1();
};

ParallelPort::ParallelPort()
{
    BaseAddress = 0x378;
    InDataPort1 = 0;
}

ParallelPort::ParallelPort(int baseaddress)
{
    BaseAddress = baseaddress;
    InDataPort1 = 0;
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress,data);
}

void ParallelPort::WritePort2(unsigned char data)
{
    outportb(BaseAddress+2,data ^ 0x0B);
}

unsigned char ParallelPort::ReadPort1()
{
    InDataPort1 = inportb(BaseAddress+1);
    // Инвертирование старшего бита для компенсации его
    // внутренней инверсии схемой параллельного порта.
    InDataPort1 ^= 0x80;
    // Сброс неиспользуемых битов D0, D1 и D2 при помощи маски.
    InDataPort1 &= 0xF8;
    return InDataPort1;
}

class DAC : public ParallelPort
{
    private:
        unsigned char LastOutput;

    public:
        DAC();

```

```

    DAC(int baseaddress);
    void SendData(unsigned char data);
    unsigned char GetLastOutput();
};

DAC::DAC()
{
    LastOutput = 0;
}

DAC::DAC(int baseaddress) : ParallelPort(baseaddress)
{
    LastOutput = 0;
}

void DAC::SendData(unsigned char data)
{
    ParallelPort::WritePort0(data);
    LastOutput = data;
}

unsigned char DAC::GetLastOutput()
{
    return LastOutput;
}

void main()
{
    DAC D_to_A;

    clrscr() // очистка экрана

    D_to_A.SendData(0);
    printf("\nDAC byte: %3d   ", D_to_A.GetLastOutput());
    cout << "    Measure voltage and press a key" << endl;
    getch();

    D_to_A.SendData(32);
    printf("\nDAC byte: %3d   ", D_to_A.GetLastOutput());
    cout << "    Measure voltage and press a key" << endl;
    getch();

    D_to_A.SendData(64);
    printf("\nDAC byte: %3d   ", D_to_A.GetLastOutput());
    cout << "    Measure voltage and press a key" << endl;
    getch();

    D_to_A.SendData(128);
    printf("\nDAC byte: %3d   ", D_to_A.GetLastOutput());
    cout << "    Measure voltage and press a key" << endl;
    getch();

    D_to_A.SendData(255);

```

```
printf("\nDAC byte:%3d    ", D_to_A.GetLastOutput());
cout << "    Measure voltage and press a key" << endl;
getch();
}
```

6.6. Заключение

Операционный усилитель, обсуждавшийся в этой главе, входит в состав многих электронных систем. На интерфейсной плате он совместно с микросхемой DAC0800 образует схему цифро-аналогового преобразователя. Были рассмотрены две основных схемы ЦАП, а также характеристики и параметры ЦАП.

В этой главе объяснялись важные понятия наследования и полиморфизма. Также было описано, что такое атрибуты доступа и как на них влияют спецификаторы доступа. Мы также научились пользоваться оператором разрешения контекста (области видимости) для доступа к полиморфным функциям базового класса. Созданный в конце главы класс DAC имеет всё необходимое для работы с цифро-аналоговым преобразователем и защищает свои члены данных и члены данных базового класса атрибутом `private`.

6.7. Литература

1. *NS DATA CONVERSION/ACQUISITION Databook*, National Semiconductor Corporation, 1984.
2. Bentley, J., *Principles of Measurement Systems*, Second edition, Longman Scientific & Technical, Essex, 1988.
3. Horowitz, P. And Hill, W., *The Art of Electronics*, Cambridge University Press, Cambridge, 1989.
4. Loweday, G., *Microprocessor Sourcebook*, Pitman Publishing Limited, London, 1986.
5. Savant, C. J. et al., *Electronic Design Circuits and Systems*, Second edition, Benjamin-Cummings, Redwood City, 1987.
6. Webb, R. E., *Electronics for Scientists*, Ellis Horwood, New York, 1990.
7. Wobschall, D., *Circuit Design for Electronic Instrumentation*, McGraw-Hill, 1987.
8. Lafore, R. *Object Oriented Programming in MICROSOFT C++*, Waite Group Press, 1992.
9. Wang, P. S., *C++ with Object Oriented Programming*, PWS Publishing, 1994.
10. Pohl, I., *Object Oriented Programming Using C++*, Benjamin Cummins, 1993.
11. Johnsonbaugh, R. and M. Kalin, *Object Oriented Programming in C++*, Prentice Hall, 1995.
12. Barton, J. J. and L. R. Nackman, *Scientific and Engineering C++ – An Introduction with Advanced Techniques and Examples*, Addison Wesley, 1994.

7 Управление светодиодами

Содержание главы:

- Циклы.
- Условные переходы.
- Классы для управления светодиодами.
- Массивы.
- Аргументы по умолчанию.
- Указатели.
- Динамическое выделение памяти.

7.1. Введение

В этой главе сначала рассмотрим использование таких широко распространённых конструкций в C/C++, как циклы и условные переходы. Далее поговорим об указателях, часто применяющихся во многих программах на C++. Представление об указателях необходимо для динамического выделения памяти и работы с виртуальными функциями, которые будут рассмотрены в следующей главе. Вы познакомитесь с указателями более подробно, когда они будут использоваться для последовательного просмотра массива чисел. Эти числа далее будут использованы для зажигания светодиодов на интерфейсной плате, отображая процесс перебора значений массива.

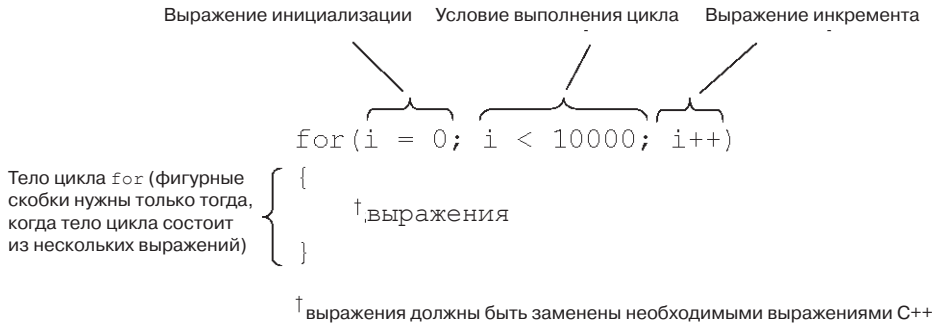
7.2. Циклы

7.2.1. Цикл `for`

Цикл `for` является итеративным циклом. Он приводит к многократному выполнению одного или нескольких выражений. В общем случае цикл `for` имеет вид, изображённый на **Рис. 7.1**. Фигурные скобки нужны только в том случае, если телом цикла является *составной оператор*. Если тело цикла представлено одним выражением, то фигурные скобки ставить не обязательно, хотя это и не запрещается.

C++ Составной оператор

Составной оператор представляет собой совокупность нескольких выражений, находящихся между парой фигурных скобок `{ }`.

Рис. 7.1. Пример цикла `for`

Между круглыми скобками заключены три выражения, относящихся к циклу `for`. Первое выражение:

```
i = 0;
```

Это выражение выполняется только один раз в самом начале цикла `for` и называется *выражением инициализации*. Инициализирующее выражение может быть более сложным. Оно может инициализировать несколько переменных. Вообще говоря, эти переменные называются *счётчиками цикла*. В предыдущем примере, переменная `i` служит для подсчёта количества выполнений цикла `for`; отсюда и название. В C++ в выражении инициализации можно даже объявлять переменные (например, `int i = 0;`). Следует отметить, что в случае отсутствия выражения инициализации нужно всё равно ставить точку с запятой.

Второе выражение:

```
i < 10000;
```

называется *условием цикла*. Это выражение проверяется перед выполнением тела цикла `for`. Результатом проверки является логическое значение. Это значит, что происходит проверка значения этого выражения, оно либо *истинно* (единица), либо *ложно* (ноль).

C++ Истина или ложь

В программе все нулевые значения считаются ложью; все ненулевые значения считаются истиной.

Когда программа вычисляет логическое выражение, то истине будет соответствовать результат, равный 1. В случае ложного значения условия результат будет нулевым.

В данном случае условие цикла проверяется, меньше ли `i`, чем 10000. Если значение `i` меньше чем 10000 то выражение будет истинным, в противном случае выражение будет ложным. Тело цикла будет выполнено сразу же после проверки условия цикла, но только в том случае, если условие истинно. Если условие ложно, то выполнение цикла `for` завершается, а его тело не выполняется.

Левая угловая скобка (<) является оператором «меньше». Он относится к классу *операторов отношения*. Из-за наличия оператора отношения и всё выражение (`i < 10000`) называется выражением отношения.

C++ Операторы отношения

<	меньше
>	больше
<=	меньше или равно
>=	больше или равно

Наряду с выражениями отношения можно использовать также и выражения равенства. В этих выражениях содержатся *операторы равенства*.

C++ Операторы равенства

==	равно
!=	не равно

Третье выражение в цикле `for`:

```
i++;
```

Это выражение называется *выражением инкремента*. Оно выполняется сразу после тела цикла `for`. Оно обычно используется для инкремента одного или нескольких счётчиков цикла. В нашем случае значение `i` увеличивается на единицу.

Оператор `++` можно использовать по-разному. Если он стоит перед идентификатором (например, `++i`), это *преинкремент*. Если он после идентификатора, то это *постинкремент*. Приставка «пре» в названии операции (подразумеваются операции инкремента и декремента) означает, что эта операция выполняется **до** проверки значения идентификатора. Если операция типа «пост», то она выполняется **после** проверки значения идентификатора в условном выражении. Оператор `--` используется аналогичным образом, только он выполняет декремент. Эти операторы относятся к категории *унарных операторов*. Они называются унарными, потому что выполняют действие над единственным аргументом.

C++ Унарные операторы

+	унарный плюс
-	унарный минус
++	преинкремент или постинкремент
--	предекремент или постдекремент
~	побитовая инверсия. Инвертирует каждый бит.
!	логическое отрицание. Преобразует истину в ложь и наоборот.

Приведённый ниже фрагмент программы иллюстрирует работу цикла `for`. Также показана работа одного цикла `for` в составе другого (*вложенный цикл for*):

```
int i, j;

for (i = 0; i < 5; i++)
{
    for (j = 0; j < i; j++)
        cout << '*';
    cout << endl;
}
```

В программе этот фрагмент вывел бы на экран следующее:

```
*
**
***
****
*****
```

Рис. 7.2. Результат выполнения вложенного цикла `for`

Тело внешнего цикла `for` начинается открывающей фигурной скобкой и заканчивается закрывающей фигурной скобкой. Между этими скобками находится внутренний цикл `for`. У внутреннего цикла `for` нет фигурных скобок, поскольку его тело состоит только из одного выражения:

```
cout << '*';
```

Второе выражение внешнего цикла `for`:

```
cout << endl;
```

и выполняется после завершения итераций внутреннего цикла `for`. Каждая итерация внутреннего цикла `for` печатает на экране символ `'*'` и инкрементирует счётчик цикла `j`. Печать строки завершается, когда значение `j` становится равным счётчику внешнего цикла. Поэтому каждая итерация внешнего цикла состоит из `i` итераций внутреннего цикла `for`.

7.2.2. Циклы `while` и `do-while`

Повторяющиеся действия, выполнявшиеся в цикле `for`, можно выполнять в цикле `while`. Цикл `while` похож на цикл `for`, у которого нет выражений инициализации и инкремента. У него есть только выражение условия цикла, заключённое в круглые скобки, проверяющееся в начале цикла. Чтобы цикл `while` выполнялся это выражение должно быть истинным. Если оно ложно (равно нулю), то выполнение цикла прекращается. Этот цикл может ни разу не выполниться, если условие цикла уже в начале цикла ложно или равно нулю.

Поскольку нет выражений инициализации и инкремента, то в цикле `while` не обязательно наличие счётчика цикла. Тем не менее цикл `while` можно использовать как цикл `for` и наоборот. Циклы `while` используются, как правило, в тех случаях, если заранее неизвестно, сколько итераций должно быть сделано.

Выражение `do-while` очень похоже на выражение `while`. Различие в том, что условие цикла `do-while` проверяется после выполнения тела цикла, поэтому цикл будет выполнен по крайней мере один раз. Тело цикла находится между служеб-

ным словом `do` и служебным словом `while`. Фигурные скобки должны использоваться, когда телом цикла является составной оператор. На **Рис. 7.3** представлена структура обоих циклов.

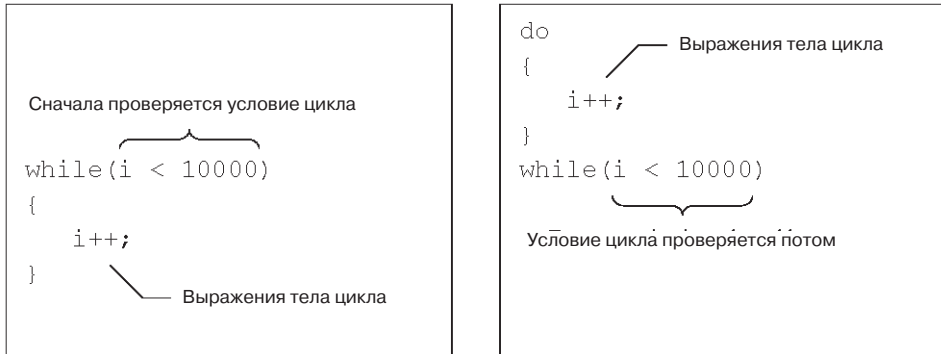


Рис. 7.3. Циклы `while` и `do-while`

7.3. Переходы

7.3.1. Выражение `if`

У выражения `if` есть *выражение условия*, заключённое в пару круглых скобок, расположенных непосредственно за служебным словом `if`. В более общем виде выражение `if` имеет *истинный блок* и *ложный блок*, отделённый служебным словом `else`. Истинный блок расположен перед `else`, а ложный после него. Условие может быть истинным или ложным. Если оно истинно, то выполняется истинный блок, ложный блок игнорируется, а если оно ложно, то выполняется только ложный блок, истинный блок игнорируется.

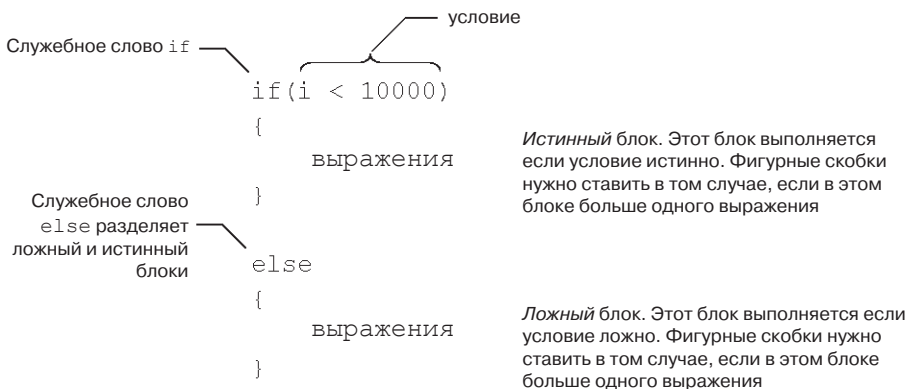


Рис. 7.4. Выражение `if` в общем виде

Если в каком-либо блоке содержится несколько выражений, то их нужно заключить в фигурные скобки (`{` и `}`), чтобы получился составной оператор. Внутри этих составных операторов могут находиться другие выражения `if`. Этот случай называется *вложенным* выражением `if`. Во вложенных выражениях `if` служебное слово `else` относится к ближайшему выражению `if`.

Допускается использовать выражения `if` с одним блоком, как показано на **Рис. 7.5**. Это должен быть истинный блок.

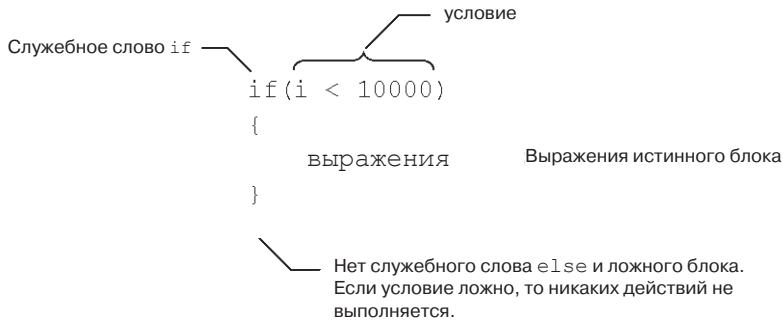


Рис. 7.5. Выражение `if` с истинным блоком

Выражение `if` может быть вложенным, как показано на **Рис. 7.6**. В двух приведённых примерах служебные слова `else` относятся к разным выражениям `if`. Пары `if-else` лучше видны при их соответствующем выделении отступами, **Рис. 7.7**.

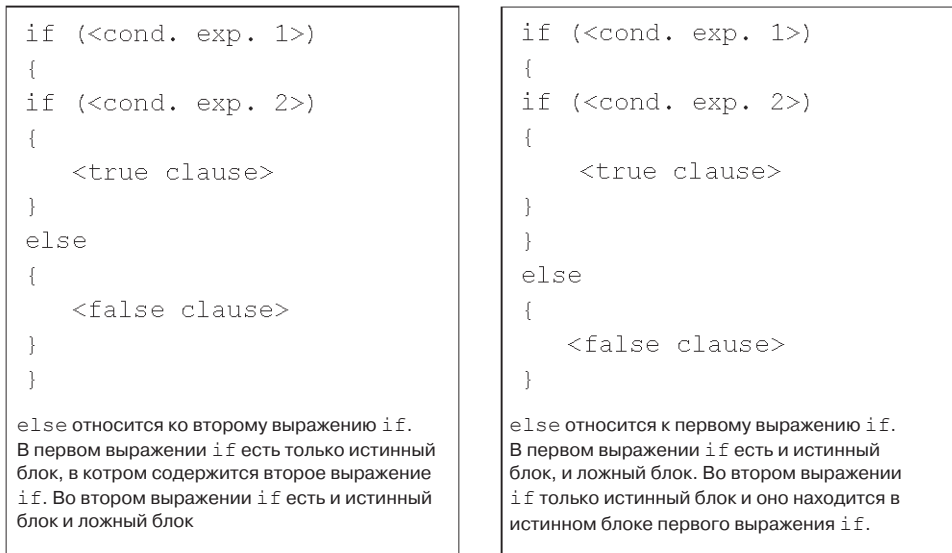


Рис. 7.6. Вложенные выражения `if` без выделения отступами

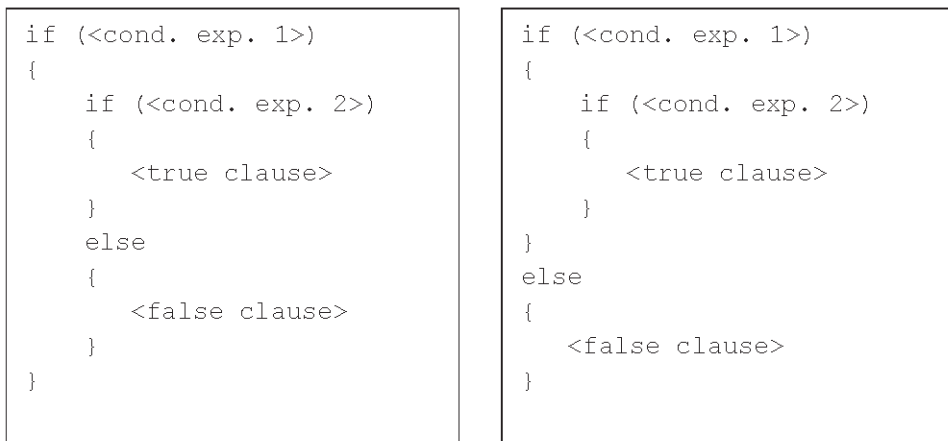


Рис. 7.7. Выделение выражений if при помощи отступов

Пример использования выражений if приведён ниже во фрагменте программы:

```
int Number;
cout << "Enter an integer Number ";
cin >> Number;

if (Number > 50)
    cout << "Number is greater than 50" << endl;
else
    cout << "Number is less than or equal to 50" << endl;
```

Число, введённое пользователем, проверяется в выражении if и по результату проверки на экран выводится соответствующий текст.

Существует компактный вариант выражения if, так называемый *тернарный оператор* (? :). Например, если переменная Switch, то её состояния ВКЛЮЧЕНО/ВЫКЛЮЧЕНО можно отображать на экране следующим образом:

```
Switch == 1 ? cout << "on" : cout << "off";
```

Это выражение эквивалентно следующему:

```
if (Switch == 1)
    cout << "on";
else
    cout << "off";
```

7.3.2. Выражения break и continue

Выражения break и continue могут существенно улучшить функциональные возможности ваших программ. Из них break используется наиболее часто. Их синтаксис очень прост, они всегда выглядят следующим образом:

```
break;
continue;
```

Выражение `break` используется для прекращения выполнения или цикла (`while`, `do-while` или `for`) или выражения `switch`. Выражение `continue` прекращает выполнение текущей итерации цикла и начинает следующую итерацию. На **Рис. 7.8** показаны оба случая.

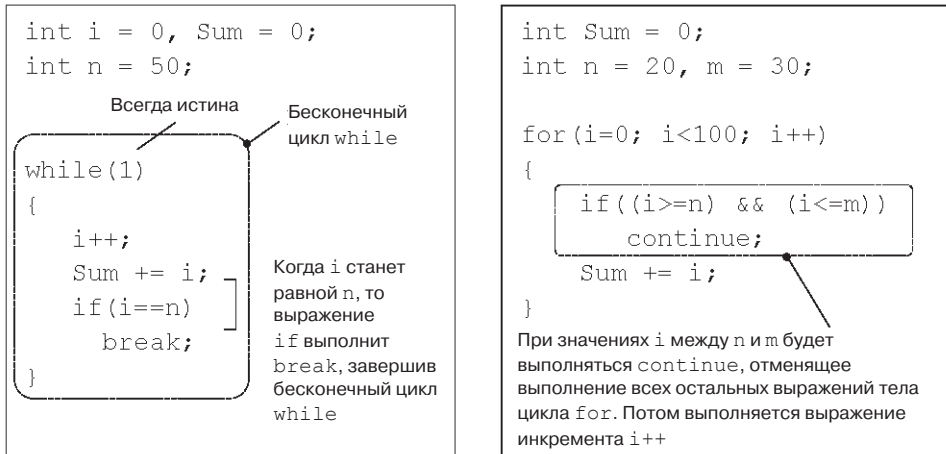


Рис. 7.8. Выражения `break` и `continue`

Ранее упоминалось, что циклы могут быть вложенными; т. е. один цикл внутри другого. Так же и выражение `switch` может находиться внутри цикла. В таких случаях выражение `break` будет относиться к ближайшему циклу и выражению `switch`. Выражение `continue` будет относиться к ближайшему циклу и в выражении `switch` использоваться не может.

В C++ можно использовать также `goto` для перехода к какой-либо метке. Применение `goto` может серьёзно нарушить структуру программы. Его использование не приветствуется и здесь не описывается, смотрите ссылки в разделе 7.11.

7.3.3. Выражение `switch-case`

Выражение `switch-case` используется для выполнения одного из нескольких возможных фрагментов кода. Выбор осуществляется *выражением выбора*, расположенного в круглых скобках после служебного слова `switch` (**Рис. 7.9**).

Выражение выбора для `switch` должно быть целого типа, например, `char`, `unsigned char`, `int`, `unsigned int` и т. д. Будет выполняться то выражение `case`, которое совпадёт с выражением выбора `switch`.

Все возможные варианты перечисляются в теле выражения `switch`. Непосредственно после служебного слова `case` должна быть константа целого типа, отличающаяся от других своим значением. Для каждого `case` могут быть соответствующие ему выражения, включая пустые выражения. Все выражения после `case` выполняются последовательно.

Выражение `break` нужно для выхода из `switch` в конце какого-либо `case`. Если не будет `break`, то будет выполняться следующий `case`. Необязательным элементом является `default` для выполнения действий, не соответствующих ни одному из имеющихся `case`.

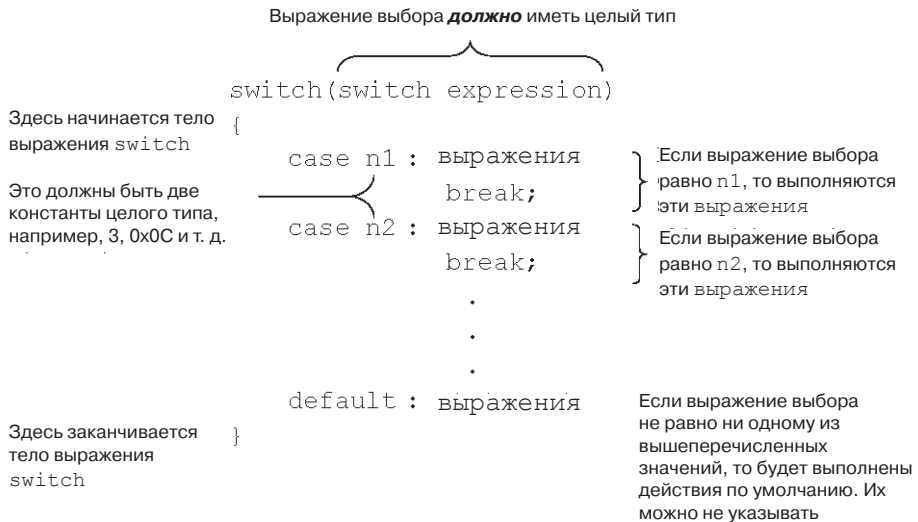


Рис. 7.9. Выражение `switch`

C++ Константное выражение целого типа

Константное выражение должно приводить в результате к целому типу и не может содержать никаких переменных.

```
#define TWIN 2
```

Таким образом определена константа `TWIN`, которая будет заменяться числом 2. Тогда `TWIN+1` будет константным выражением целого типа. Обратите внимание, что `TWIN` не является переменной.

Но если `int a=0;`

Тогда `a+1` не будет константным целым выражением, поскольку `a` является переменной.

7.4. Массивы

Массивы представляют собой наборы объектов одинакового типа. Объекты могут иметь как один из базовых типов, так и тип, определяемый пользователем. Каждый отдельный объект набора называется *элементом массива*. В программах на протяжении этой главы в массивах мы будем хранить наборы состояний светодиодов.

Массивы могут иметь различную структуру или несколько измерений, как показано на **Рис. 7.10**. Каждая ячейка может хранить объект известного типа. Практически, массивы могут иметь неограниченное количество измерений.

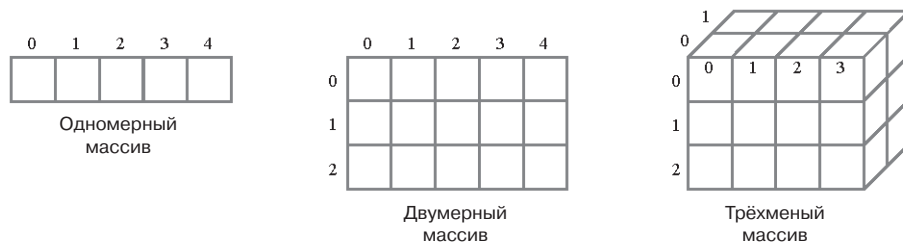


Рис. 7.10. Схематичное изображение массивов

Размер массива задаётся числом элементов (ячеек) для каждого измерения. На **Рис. 7.10** можно видеть массивы следующих размеров:

- Одномерный массив: 5 элементов.
- Двумерный массив: 3 строки, 5 элементов в строке. Размер 15 элементов.
- Трёхмерный массив: 3 строки, 4 элемента в строке, каждый из которых в свою очередь состоит из двух элементов. Размер 24 элемента.

Хотя массивы и можно представить как на **Рис. 7.10**, тем не менее в памяти компьютера они хранятся в последовательном виде, строка за строкой, что называется, в развёрнутом виде. В случае хранения в памяти трёхмерного массива, сначала в развёрнутом виде идёт первый слой, затем второй слой и т. д.

Одномерные массивы

При объявлении одномерного массива сразу после его имени в квадратных скобках указывается его размер. Индекс массива всегда начинается с 0 и до размера массива минус единица. Пример объявления массива:

```
int a[10];
```

Так объявляется массив, состоящий из 10 объектов типа `int`. Они хранятся в последовательно расположенных ячейках памяти, начиная с элемента `a[0]` и заканчивая `a[9]`, представляя собой набор из 10 элементов, изображённый на **Рис. 7.11**. Обратите внимание, что элемента `a[10]` не существует. Нельзя обращаться к этой ячейке памяти, поскольку это уже не массив `a`.

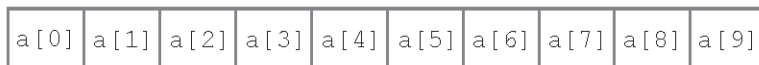


Рис. 7.11. Схематичное изображение массивов

Если доступ к элементам массива осуществляется посредством индексной переменной, например, `a[i]`, то `i` должна быть целого типа, и, как было сказано выше, не должна превышать наибольший индекс, в данном случае она должна быть от 0 до 9.

Каждый элемент в отдельности может быть инициализирован в ходе выполнения программы присвоением ему значения. Например, следующий фрагмент присваивает всем элементам ноль:

```
for (int i = 0; i < 10; i++)
    a[i] = 0;
```

Значения элементам массива также можно присвоить во время его объявления. Значения элементов массива должны быть перечислены в через запятую и заключены в фигурные скобки, например:

```
int a[8] = {2,3,7,4}; // 8 элементов
```

В этом примере массив частично заполнен значениями, перечисленными в фигурных скобках; элемент $a[0]=2$, $a[1]=3$, $a[2]=7$ и $a[3]=4$. Элементы с $a[4]$ по $a[7]$, которые не были явным образом проинициализированы, инициализируются по умолчанию значением 0. Поэтому, чтобы проинициализировать массив нулями можно поступить так:

```
int a[8] = {};
```

Доступ к элементам массива

Доступ к отдельным элементам массива осуществляется при помощи *индекса*. В следующем примере индекс массива принимает значения от 0 до 7 (8 элементов).

```
int a[8];           // объявлен размер 8 элементов
int Result;         // объявлена переменная Result

a[0] = 2;           // обращение к a[0], чтобы присвоить 2
a[1] = 3;           // обращение к a[1], чтобы присвоить 3

Result = a[0]*a[1]; // 2*3 = 6
```

Двумерные массивы

Двумерный массив можно представить как массив одномерных массивов. Для двумерных размеров указывается два размера. Общее количество элементов равно произведению этих двух размеров. Например, двумерный массив можно объявить, как показано на **Рис. 7.12**:

```
int b[2][5];
```

Рис. 7.12. Объявление двумерного массива

Для доступа к элементам массива используются два индекса. В массиве b одна размерность может принимать значения от 0 до 1, а другая размерность от 0 до 4. Этот массив можно изобразить в виде структуры, изображённой на **Рис. 7.13**.

Отметим, элементы массива хранятся в последовательно расположенных ячейках памяти в развёрнутой форме. Это значит, что сначала идёт первая строка, а за ней вторая и т. д. Поэтому за элементом `b[0][4]` находится элемент `b[1][0]`.

<code>b[0][0]</code>	<code>b[0][1]</code>	<code>b[0][2]</code>	<code>b[0][3]</code>	<code>b[0][4]</code>
<code>b[1][0]</code>	<code>b[1][1]</code>	<code>b[1][2]</code>	<code>b[1][3]</code>	<code>b[1][4]</code>

Рис. 7.13. Схематичное изображение двумерного массива

Элементы массива можно проинициализировать во время выполнения программы, присвоив значения каждому элементу. Например, следующий фрагмент программы присваивает всем элементам нулевое значение.

```
int i, j;

for (i = 0; i < 2; i++)
    for (j = 0; j < 5; j++)
        b[i][j] = 0;
```

Инициализацию можно выполнить также во время объявления массива. В следующем примере показано, как это можно сделать.

```
int b[2][5] = {{0,0,0,0,0}; {0,0,0,0,0}};
```

Значения инициализации элементов строк заключены во внутренние фигурные скобки, а значения инициализации строк отделяются точкой с запятой.

7.5. Указатели

Указатель представляет собой адрес данных, находящихся в памяти. В качестве примера таких данных можно привести: объекты классов, данные таких базовых типов, как `int`, `float`, `char`, `long` и т. д., массивы (набор однотипных данных) и функции. Указатель в C++ определяет *где* находится объект, и чаще всего несёт информацию о *типе* объекта. Например, указатель на `integer` «знает», где в памяти находятся данные типа `integer`, но при этом неизвестно, какое именно значение там хранится.

Указатели являются важным инструментом повышения эффективности программ на C++. Существует несколько основных задач, где применение указателей наиболее выгодно: передача функциям больших объектов, динамическое выделение памяти и использование виртуальных функций. Чаще всего при передаче параметров в функции происходит замена параметра копией фактического аргумента. Если фактический аргумент слишком велик по объёму, то на создание его копии уходит очень много памяти. Гораздо лучше создать копию с адресом большого объекта, а не копировать сам объект. Это делается при помощи указателя, который хранит в себе адрес объект и занимает мало места в памяти.

Динамическое выделение памяти подразумевает предоставление памяти во время выполнения программы. Обычно динамическое выделение осуществляется по мере необходимости, т. е. в ходе выполнения программы может быть запрошена дополнительная память и предоставлена, если есть такая возможность. В результате динамического выделения памяти возвращается указатель на выделенную область памяти. Затем этот указатель можно использовать для действий над данными в выделенной области.

Наверно, самым непонятным является аспект использования указателей с виртуальными функциями; этот вопрос будет подробно рассмотрен в главе 8. В следующих разделах рассматриваются общие вопросы использования указателей в C++.

Для работы с указателями предназначены два унарных оператора. Ранее говорилось, что унарные операторы принимают только один аргумент. Эти операторы приведены в **Табл. 7.1**.

Таблица 7.1. Унарные операторы для указателей

Оператор	Функция
&	Взятие адреса аргумента
*	Ссылка на адрес (разыменование)

Оператор *адрес операнда* применяется для определения адреса объекта в памяти.

Оператор *ссылки на адрес* можно использовать для получения значения, хранящегося по данному адресу. Это также называется *разыменованием*.

7.5.1. Объявление указателей

Как уже известно, существует специальный тип данных `int`, представляющий целые числа и несколько других базовых типов. Программисты также могут сами создавать свои собственные типы данных, как, например, `DAC`, созданный в главе 6. Тем не менее, не существует такого одного типа данных, как *указатель*. Области памяти, обозначаемые указателями, могут содержать любой тип данных и функций.

Звёздочка означает, что идентификатор является переменной-указателем

```
data type *identifier;
```

Здесь должен быть тип данных, такой как `int`, `char` или класс, например `DAC`

Рис. 7.14. Синтаксис объявления указателя

При объявлении переменной-указателя в C++ необходимо указать тип данных или функцию, на которые указывает такая переменная. Мы увидим, какое большое значение имеет тип данных указателя, когда будем изучать арифметические операции с указателями (иногда это называется адресной арифметикой)

в разделе 7.5.6. В простейших случаях указатели объявляются как на **Рис. 7.14**. Указатели на различные объекты объявляются не так, как было описано выше.

7.5.2. Указатели на скалярные величины

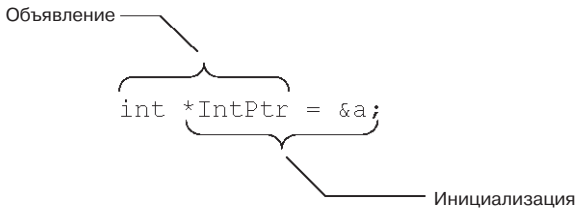
Одиночный элемент является скалярной величиной. Если переменная объявлена как указатель на одиночное целое число, то говорят, что этот указатель указывает на скалярную величину. В противоположность им можно привести указатели на массивы. Примеры объявлений одиночных переменных и объявления указателей на скалярные величины показаны в **Табл. 7.2**.

Таблица 7.2. Объявление скалярных величин и указателей на скалярные величины

Объявление скалярных величин	Объявление указателей на скалярные величины
<code>int a;</code>	<code>int a;</code> <code>int *IntPtr = &a;</code>
<code>int b = 0;</code>	<code>int b = 0;</code> <code>*IntPtr = b;</code>
<code>float p = 0.0;</code>	<code>float p = 0.0;</code> <code>float *FIntPtr = &p;</code>

В следующей строке объединены объявление и инициализация указателя на `int`:

```
int *IntPtr = &a;
```



К аналогичному результату приведут следующие две строки:

```
int *IntPtr;  
IntPtr = &a;
```

Первое выражение объявляет указатель на `int`. Во втором выражении при помощи оператора взятия адреса `&` определяется адрес переменной `a` целого типа, который затем присваивается указателю `IntPtr`. Обратите внимание, что переменная `a` типа `int` должна объявляться до присвоения её адреса `IntPtr`.

Выражение:

```
*IntPtr = b;
```

выполняет разыменованье и операцию присвоения. Выражение `*IntPtr` звучит так: «содержимое области памяти, на которую указывает `IntPtr`». Это называется

разыменованием. Таким образом, всё выражение будет означать: «содержимому области памяти, на которую указывает `IntPtr`, присвоить значение `b`». Поскольку `IntPtr` указывает на `a`, то результат будет таким же, как и в таком случае:

```
a = b;
```

Пример объявления указателя на переменную типа `float` и присвоение ей значения приведено в **Табл. 7.2**.

Примечание

Даны следующие объявления:

```
int a = 0;
float* FltPtr;
```

Тогда присвоение в виде:

```
FltPtr = &a; // Неправильно!
```

будет неверным. Подразумевается, что `FltPtr` хранит адрес объекта типа `float`. С другой стороны, `&a` даёт адрес объекта типа `int`. Эти два типа несовместимы между собой и поэтому присвоение будет неверным.

7.5.3. Указатели на объекты классов

Указатели на объекты классов объявляются аналогично указателям на скалярные величины. Пример такого объявления:

```
ParallelPort *PortPtr;
```

Здесь тип данных `ParallelPort`, а указатель `PortPtr`. Указатель на объект класс `DAC` можно объявить следующим образом:

```
DAC *DACPtr;
```

Объект типа `DAC` можно объявить следующим образом:

```
DAC Dac;
```

Тогда следующее присвоение будет правильным:

```
DACPtr = &Dac;
```

Операторы доступа к членам

При использовании объекта класса `DAC` можно вызывать функцию `SendData()` при помощи оператора «точка» (`.`), известном также, как оператор *доступа к членам*:

```
Dac.SendData(255);
```

Функцию `SendData()` можно вызвать также и при использовании указателя `DACPtr`, хотя синтаксис при этом будет другим. В этом случае используется опера-

тор доступа к членам по указателю (\rightarrow), состоящий из знака «минус» ($-$) и правой угловой скобки ($>$):

```
DACPtr->SendData(255);
```

Указатели на базовые объекты могут указывать на объекты производных классов

Это одно из наиболее важных понятий в объектно-ориентированном программировании, широко использующееся на практике. В предыдущих разделах говорилось, что указатель на тип `float` не может указывать на `int`. Это правило не относится к базовому и производным классам. Хотя такие объекты и различаются, указатель на базовый класс может указывать на объекты производных классов:

```
ParallelPort *PortPtr;
DAC Dac;
PortPtr = &Dac; // так можно!
```

Мы только что рассмотрели преимущества такого использования указателей. Их основное назначение для работы с виртуальными функциями и будет обсуждаться в разделах 8.5 и 8.6.

7.5.4. Указатели на массивы

Указатели на одномерные массивы

Если объявлен массив, аналогичный изображённому на **Рис. 7.15**, то `a` (без индекса) будет адресом массива, т. е. указывать на первый элемент массива. Поэтому выражения `a` и `&a[0]` эквивалентны и оба указывают на первый элемент. Следует отметить важную деталь, что `a` является *константным указателем*. Его нельзя инкрементировать, декрементировать или присваивать ему другие значения. Поскольку массив постоянно хранится в выделенной области памяти, то и адрес является неизменным.

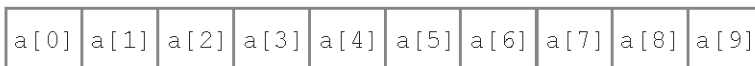


Рис. 7.15. Изображение одномерного массива

Следующие выражения верны:

```
int a[10];
int *ElementPtr;
int b = 0;
ElementPtr = a; // аналогично ElementPtr = &a[0];
*a = b;          // значение b размещено в a[0]
```

Некоторые из нижеприведённых выражений неправильны:

```
int a[10];
float *FltPtr;
int b = 0;
```

```
FltPtr = a; // неправильно - несоответствие типов
a = &b;      // неправильно - a является константой
```

Указатели на двумерные массивы

Как говорилось ранее, двумерный массив можно рассматривать как массив одномерных массивов. В качестве примера можно привести такой двумерный массив:

```
int a[5][5];
```

Количество строк

Количество элементов в строке

Вспомним, что элементы хранятся в памяти последовательно. Например, элемент `a[1][0]` находится сразу после `a[0][4]`.

В отличие от одномерных массивов, имя массива `a` является указателем на всю строку, начиная с `a[0][0]` и заканчивая `a[0][4]`. Указатель `a` по-прежнему остаётся константным.

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
.				
.				
.				
a[4][0]	a[4][1]	a[4][2]	a[4][3]	a[4][4]

Рис. 7.16. Изображение двумерного массива

Указатель на строку из пяти элементов можно объявить следующим образом:

```
int (*RowPtr)[5];
```

Обратите внимание на незначительные отличия в вышеприведённом выражении, выраженные наличием или отсутствием круглых скобок. Сравните это объявление с объявлением массива указателей, обсуждаемыми далее.

Если `a` будет разыменован, то в результате получится указатель на первый элемент первой строки, т. е. `&a[0][0]`. Этот указатель тоже является константным. Для получения значения `a[0][0]` указатель нужно разыменовать дважды. Следующий фрагмент иллюстрирует сказанное:

```
int *ElementPtr;
int b;
int a[5][5];
int (*RowPtr)[5];
RowPtr = a;      // указатель на первую строку
```



```
ElementPtr = *a; // указатель на первый элемент
                // первой строки
b = **a;         // то же, что b = *ElementPtr;
ElementPtr = a;  // неправильно - несоответствие типов
RowPtr = *a;     // неправильно - несоответствие типов
*a = &b;         // неправильно - *a является константой
```

Важное наблюдение состоит в том, что если разыменовывается имя массива, то он указывает на элемент более низкого уровня. Например, если разыменовывать имя двумерного массива, то получим указатель на одномерный массив. Если разыменовано имя одномерного массива, то получается содержимое первого элемента массива. Больше такой массив разыменовывать нельзя.

7.5.5. Массивы указателей

Также возможно объявлять массивы указателей. В таком массиве каждый элемент является указателем. В качестве примера приведём такое объявление массива указателей:

```
int *IntPointers[20];
```

В этом объявлении `IntPointers` является константным указателем. Он указывает на первый элемент массива указателей. Если воспользоваться оператором разыменования, как показано ниже, то получится содержимое первого элемента, которое, в свою очередь, является указателем на `int`. Поэтому это значение присваивается совместимому указателю. Рассмотрим следующие объявления:

```
int a;
int *IntPtr;
int *IntPointers[20];
```

`IntPointers` представляет собой адрес начала массива указателей на `int`. Другими словами он содержит адрес памяти. По этому адресу размещается указатель на `int`. Содержимое любого элемента массива может быть присвоено указателю на `int`, например `IntPtr`. Чтобы получить содержимое первого элемента массива `IntPointers` нужно разыменовывать `IntPointers`. После разыменования результат можно присвоить `IntPtr`, как показано ниже:

```
IntPtr = *IntPointers; // содержимое первого элемента
```

После разыменования `IntPointers` (т. е. `*IntPointers`) мы получим адрес, по которому расположен указатель на `int`. Чтобы получить содержимое по адресу, на который указывает этот указатель на `int`, нужно ещё раз разыменовывать `*IntPointers` и получить значение целого типа. Такое значение можно присвоить переменной `a` типа `int`:

```
a = **IntPointers; // аналогично a = *IntPtr;
```

7.5.6. Арифметические операции над указателями

Как уже известно, переменная-указатель может быть инкрементирована и декрементирована. Также можно прибавлять и вычитать целые числа. Но получа-

ющийся при этом результат действий над указателями отличается от результата обычной арифметики. Перед дальнейшим объяснением разберёмся, когда нужно использовать арифметические операции над указателями.

Одномерные массивы

Арифметические операции над указателями предназначены для доступа к элементам массивов. Рассмотрим пример:

```
int a[5];
```

Объявлен массив из 5 целых чисел. Имя массива `a` является константным указателем и указывает на первый элемент массива. На **Рис. 7.17** показан пример размещения такого массива в памяти.

Адрес памяти	Элемент массива и его значение
↓	↓
600000	a[0] is 4
600002	a[1] is 9
600004	a[2] is 8
600006	a[3] is 3
600008	a[4] is 5

Рис. 7.17. Пример размещения 5 целых чисел в памяти

На примере (**Рис. 7.17**) адреса памяти изменяются на 2 при перемещении от одного элемента к соседнему. Так происходит оттого, что размер целого числа принят равным двум байтам. Поэтому элемент `a[0]` будет занимать адреса 600000 и 600001. Следующий элемент `a[1]` начинается с адреса 600002 и так далее.

Значением `a` будет 600000. Указатель `a` не является переменной, он константа. Он указывает на первый элемент массива и поэтому других значений принимать не может. Его, как и другие указатели, можно разыменовывать:

`*a` → `a[0]` имеет значение 4

Хоть `a` и нельзя изменить, можно добавить к нему целое число и получить новое значение:

`a + 1` правильное выражение

Если это выражение рассматривать с точки зрения обычной арифметики, то результат должен быть равен 600001. Но в случае адресной арифметики с указателями получится 600002. Поскольку `a` является указателем на `int`, то `+1` означает «плюс один элемент типа `int`», размер которого в нашем случае равен двум байтам. Заботу о размере данных в адресной арифметике берёт на себя компилятор. Результат выражения `a + 1` также является указателем и указывает на следующий элемент массива:

a + 1 → 600002
a + 2 → 600004
a + 3 → 600006

Как и другие указатели, вышеперечисленные три указателя тоже можно разыменовывать:

*(a + 1) → a[1] имеет значение 9
*(a + 2) → a[2] имеет значение 8
*(a + 3) → a[3] имеет значение 3

В вышеприведённом примере скобки имеют очень большое значение. Если не применять скобки, то результаты будут такими:

*a + 1 → a[0] + 1 получится значение 5
*a + 2 → a[0] + 2 получится значение 6
*a + 3 → a[0] + 3 получится значение 7

Как видно, адресная арифметика может использоваться для доступа к элементам массива в одномерном массиве. Для массива a, i-ый элемент можно обозначить так:

*(a + 1)

Двумерные массивы

Адресная арифметика для двумерных массивов немного отличается. Рассмотрим пример:

int a[3][2];

Это объявлен массив из 6 элементов, которые хранятся в памяти в последовательно расположенных ячейках памяти, **Рис. 7.18.**

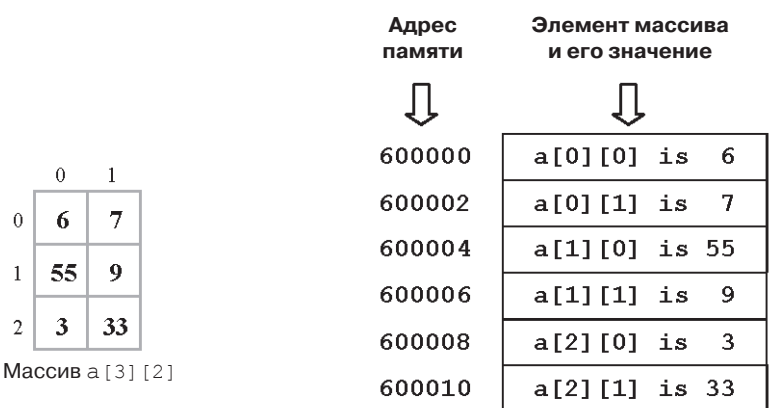


Рис. 7.18. Двумерный массив в памяти

Как и в случае одномерного массива, имя массива a тоже является указателем. Он имеет значение 600000. Он является константой и не может быть изменён.

От одномерного массива он отличается тем, что *a* указывает на *первую строку* элементов, а не на первый элемент. Поэтому он представляет данные, имеющие размер как 2 целых числа, согласно второму индексу в объявлении массива. Указатель *a* указывает на 2 целых, т. е. на 4 байта памяти. Зная это, рассмотрим арифметические операции над указателями.

$a + 1 \rightarrow 600004$ указывает на вторую строку
 $a + 2 \rightarrow 600008$ указывает на третью строку

В такой форме записи, указатель на *i*-ю строку может быть получен сложением *i* и *a*:

$a + i \rightarrow$ указывает на *i*-ю строку

Как и другие указатели, их тоже можно разыменовывать. После разыменования таких указателей получается также указатель:

$*(a + 1) \rightarrow 600004$ указывает на первый элемент второй строки
 $*(a + 2) \rightarrow 600008$ указывает на второй элемент третьей строки
 $*(a + i) \rightarrow$ указывает на первый элемент строки $i + 1$

Размер данных такого разыменованного указателя уже не является целой строкой. Теперь это уже элементы, являющиеся целыми числами. Теперь в основе арифметических вычислений будут лежать целые двухбайтовые числа.

$*(a + 1) + 1 \rightarrow 600006$ указывает на $a[1][1]$
 $*(a + 2) + 1 \rightarrow 600010$ указывает на $a[2][1]$

Тогда указатель на *j*-й элемент *i*-й строки будет таким:

$*(a + i) + j \rightarrow$ указывает на $a[i][j]$

Разыменовав вышеприведённые указатели можно получить значения элементов:

$*(*(a + 1) + 1) \rightarrow a[1][1]$ имеет значение 9
 $*(*(a + i) + j) \rightarrow a[i][j]$

Те же правила распространяются на массивы больших размерностей.

7.5.7. Указатели на функции

Указатели можно объявить для указания на функции. По сравнению с ранее рассмотренными указателями, указатели на функции тоже хранят её адрес. Это адрес первой инструкции функции.

Пример объявления указателя на функцию:

```
int (*CalcFunctionPtr)(int,int);
```

В предыдущем примере указатель на функцию имеет имя *CalcFunctionPtr*. Этот указатель может указывать только на функции одного вида, определяемым объявлением указателя. Объявление говорит, что функция, на которую указывает *CalcFunctionPtr*, должна принимать два целочисленных параметра и возвращать целый результат. Пример функции, на которую может указывать *CalcFunctionPtr*:

```
int Add(int a, int b)
{
    return (a + b);
}
```

Другая функция, подходящая для CalcFunctionPtr:

```
int Sub(int a, int b)
{
    return (a - b);
}
```

Как и в случае с массивами, когда имя массива является константным указателем, имена функций также представляют собой константные указатели по той простой причине, что во время выполнения программы расположение функции в памяти остаётся неизменным.

В двух вышеприведённых функциях Add и Sub являются константными указателями. Каждый из них указывает на первую инструкцию соответствующей функции. Эти константные указатели могут быть присвоены объявленной переменной-указателю на функцию. Пример программы приведён в **Листинге 7.1**.

Листинг 7.1. Использование указателей на функции

```
#include <iostream.h>
#include <conio.h>

int Add(int a, int b)
{
    return (a + b);
}

int Sub(int a, int b)
{
    return (a - b);
}

void main()
{
    int a, b, Result;
    char key;
    int (*CalcFunctionPtr)(int,int);

    cout << "Enter two integer values" << endl;
    cin >> a >> b;
    cout << "Press '+' or '-' key" << endl;

    key = getch(); // getch() считывает нажатые кнопки

    switch(key)
    {
```

```

        case '+' : CalcFunctionPtr = Add;
        break;
        case '-' : CalcFunctionPtr = Sub;
    }

    Result = CalcFunctionPtr(a,b);
    cout << "The result is " << Result << endl;
}

```

Если нажата кнопка '+', то в выражении `switch` указателю `CalcFunctionPtr` присваивается адрес функции `Add()`. Если нажата кнопка '-', то указатель `CalcFunctionPtr` будет указывать на функцию `Sub()`. В предпоследней строке выполняется или функция `Add()` или функция `Sub()`, в зависимости от нажатой кнопки. Это пример того, когда указатель на функцию используется для выполнения различных задач для различных случаев в одном выражении (`Result = CalcFunctionPtr(a,b)`).

В более сложных программах значительная часть программы может быть написана с использованием указателей на функцию. Если нужно изменить возможные варианты, то нет необходимости менять ту часть программы, которая выполняет вычисление результата. В **Листинге 7.2** показано, как можно добавить возможность умножения и при этом результат вычисляется в том же выражении, что и в **Листинге 7.1**, т.е. `Result = CalcFunctionPtr(a,b)`.

Листинг 7.2. Расширение функциональности программы из **Листинга 7.1**

```

#include <iostream.h>
#include <conio.h>

int Add(int a, int b)
{
    return (a + b);
}

int Sub(int a, int b)
{
    return (a - b);
}

int Mult(int a, int b)
{
    return (a * b);
}

void main()
{
    int a, b, Result;
    char key;
    int (*CalcFunctionPtr)(int,int);

    cout << "Enter two integer values" << endl;
}

```

```

cin >> a >> b;
cout << "Press '+', '-' or '*' key" << endl;

key = getch(); // getch() считывает нажатые кнопки

switch(key)
{
    case '+' : CalcFunctionPtr = Add;
                break;
    case '-' : CalcFunctionPtr = Sub;
                break;
    case '*' : CalcFunctionPtr = Mult;
}

Result = CalcFunctionPtr(a,b);
cout << "The result is " << Result << endl;
}

```

Функции, возвращающие указатели

Теперь рассмотрим функции, возвращающие указатели, так как они объявляются похожим образом. Функция с именем `AnyFunction`, принимающая два параметра типа `int` и возвращающая указатель на `int`, объявляется следующим образом:

```
int *AnyFunction(int,int);
```

Сравните это объявление с объявлением `FunctionPtr`, что является указателем на функцию, принимающую два параметра типа `int` и возвращающую значение типа `int`:

```
int (*FunctionPtr)(int,int);
```

В одном случае круглые скобки есть, а в другом их нет. Хотя различие и невелико, но эти два объявления сильно отличаются. `AnyFunction` это имя функции и, соответственно, константный указатель. `FunctionPtr` это переменная-указатель.

7.5.8. Указатели на void

Указатели можно объявлять как «указатель на void». У этих указателей нет ограничений на тип данных или функции, на которые они могут указывать. Следующий фрагмент иллюстрирует сказанное:

```

int a;           // объявление переменной типа int
float b;         // объявление переменной типа float
void *VoidPtr;   // объявление указателя на void
int Add(int,int); // объявление функции
.
.
.

```

```

VoidPtr = &a;      // адрес переменной типа int
                  // присвоен указателю на void
VoidPtr = &b;      // адрес переменной типа float
                  // присвоен указателю на void
VoidPtr = Add;     // адрес функции
                  // присвоен указателю на void

```

Преимуществом использования указателей на `void` является возможность указывать ими на различные объекты без создания специальных указателей на эти объекты.

7.5.9. Указатель `this`

В объектно-ориентированном программировании в каждом объекте есть скрытый указатель с именем `this`, указывающий на сам объект. Хотя он и скрыт, при необходимости его можно использовать так же, как и другие указатели. Функции-члены какого-либо класса могут прямо указывать при помощи указателя `this` с каким объектом должна работать функция. Рассмотрим использование указателя `this` на примере функции `GetLastOutput()` класса `DAC`, описанного в главе 6:

```

unsigned char DAC::GetLastOutput()
{
    return LastOutput;
}

```

При использовании указателя `this` эта функция будет выглядеть так:

```

unsigned char DAC::GetLastOutput()
{
    return this->LastOutput;
}

```

Теперь предположим, что созданы два объекта класса `DAC`:

```
DAC Dac1, Dac2;
```

Для каждого объекта вызовем функцию `GetLastOutput()`:

```

Dac1.GetLastOutput();
Dac2.GetLastOutput();

```

При выполнении первой из этих функций указатель `this` будет указывать на адрес объекта `Dac1` и поэтому `this->LastOutput` будет соответствовать члену `LastOutput` объекта `Dac1`. Когда выполнится функция с префиксом `Dac2`, то `this` будет указывать на адрес объекта `Dac2`, а `this->LastOutput` будет членом `LastOutput` объекта `Dac2`.

Покажем случай, когда нужно явное использование указателя `this` на примере конструктора класса `ParallelPort`:

```

ParallelPort::parallelPort(int baseaddress)
{
    BaseAddress = baseaddress;
}

```


Параметр `baseaddress` был назван так осознанно. В самом деле, если бы его назвали `BaseAddress`, то конструктор выглядел бы так:

```
ParallelPort::parallelPort(int BaseAddress)
{
    BaseAddress = BaseAddress; // путаница!
}
```

Как видно параметр невозможно отличить от члена данных. В качестве решения проблемы можно функцию записать так:

```
ParallelPort::parallelPort(int BaseAddress)
{
    this->BaseAddress = BaseAddress;
}
```

В теле функции выражение `this->BaseAddress` явно указывает на член данных.

7.6. Работа с указателями

Для изучения работы с указателями создадим массив чисел, которые затем можно будет передать на интерфейсную плату, чтобы зажигать светодиоды в нужных комбинациях. Из этого массива при помощи указателя будут последовательно извлекаться значения. Для вывода данных на интерфейсную плату используется регистр `BASE`.

7.6.1. Массивы чисел для светодиодов

Для начала создадим программу, реализующую эффект «бегущий огонь» на восьми светодиодах. В этой программе используются заранее определённые маски свечения светодиодов.

«Бегущие огни» – массив констант, определён внутри класса

Массив будет просматриваться в цикле, а его элементы будут выводиться через регистр `BASE` для зажигания светодиодов. В результате циклического опроса массива на линейке из 8 светодиодов будет виден «бегущий огонь». Массив состоит из восьми элементов, каждый элемент предназначен для зажигания только какого-то одного светодиода из группы. Когда происходит выборка следующего элемента массива, то горящий светодиод гаснет, а следующий за ним по направлению «бега» зажигается.

В **Табл. 7.3** приведены шестнадцатеричные и соответствующие двоичные значения элементов массива, которые нужно один за другим выводить в порт.

Создадим новый класс `LEDs`, где массив `Pattern` будет определён в качестве члена данных, а также будет обеспечена возможность его инициализации нужными значениями. Содержимое массива `Pattern` будет неизменным для класса и пользователь не сможет изменить его в теле функции `main()`. Класс будет иметь функцию для последовательного вывода значений `Pattern` в регистр `BASE`. Класс `LEDs` будет производным класса `ParallelPort`, чтобы унаследовать его функциональность. Определение класса приведено в **Листинге 7.3**.

Таблица 7.3. Маски свечения светодиодов в массиве Pattern

Элемент массива	Двоичное число								Шестнадцатеричное число
	D7	D6	D5	D4	D3	D2	D1	D0	
Pattern[0]	0	0	0	0	0	0	0	1	0x01
Pattern[1]	0	0	0	0	0	0	1	0	0x02
Pattern[2]	0	0	0	0	0	1	0	0	0x04
Pattern[3]	0	0	0	0	1	0	0	0	0x08
Pattern[4]	0	0	0	1	0	0	0	0	0x10
Pattern[5]	0	0	1	0	0	0	0	0	0x20
Pattern[6]	0	1	0	0	0	0	0	0	0x40
Pattern[7]	1	0	0	0	0	0	0	0	0x80

Листинг 7.3. Определение класса LEDs

```
class LEDs : public ParallelPort
{
    private:
        unsigned char Pattern[8];
        int PatternIndex;

    public:
        LEDs();
        LEDs(int baseaddress);
        void LightLEDs();
};
```

Собтвенный член данных PatternIndex нужен для хранения номера маски свечения светодиодов, которая использовалась в предыдущий раз, таким образом можно перебирать элементы массива для образования эффекта «бегущего огня». Pattern является массивом из восьми элементов типа unsigned char. Элементы типа unsigned char представляют собой просто 8-битные данные и не влекут за собой сложностей, присущих числам со знаком.

Функции-члены класса можно определить теперь так, как показано на **Листинге 7.4.**

Листинг 7.4. Функции-члены класса LEDs

```
LEDs::LEDs()
{
    // Заполнение массива Pattern
    for(int i = 0; i < 8; i++)
        *(Pattern + i) = 1 << i;

    PatternIndex = 0; // инициализация значением 0
}

LEDs::LEDs(int baseaddress) : ParallelPort(baseaddress)
```

```

{
    // Заполнение массива Pattern
    for(int i = 0; i < 8; i++)
        *(Pattern + i) = 1 << i;

    PatternIndex = 0; // инициализация значением 0
}

void LEDs::LightLEDs()
{
    while(!kdhit()) // функция завершается после
                    // нажатия на кнопку
    {
        WritePort0(*(Pattern + PatternIndex++));

        // Сброс PatternIndex в исходное состояние,
        // когда его значение будет равно 8
        if(PatternIndex == 8) PatternIndex = 0;

        delay(500);
    }
}

```

Имя массива `Pattern` (без индекса) является указателем на первый элемент массива. Поэтому:

```
Pattern + i
```

будет указывать на i -ый элемент массива. Для получения значения, на которое указывает `Pattern + i`, его нужно разыменовать:

```
*(Pattern + i)
```

В конструкторах класса `LEDs` происходит инициализация массива значениями, полученными путём сдвига числа 1 влево i раз:

```
1 << i
```

Конструкторы инициализируют массив `Pattern` значениями согласно **Табл. 7.3** в каждой итерации цикла `for`:

```
*(Pattern + i) = 1 << i;
```

Цикл `while` в функции `LightLEDs()` имеет условие `!kbhit()` и будет выполняться, пока не будет нажата какая-нибудь клавиша. В цикле `while` унаследованная функция `WritePort0()` используется для вывода массива `Pattern` в регистр `BASE` по одному элементу. Обращение к каждому элементу происходит при помощи `PatternIndex` относительно начального адреса массива `Pattern`. К константному указателю `Pattern` каждый раз прибавляется значение `PatternIndex`. Это позволяет перебирать элементы массива от начала к концу. По достижении конца массива `PatternIndex` сбрасывается в исходное состояние 0, что вызывает перебор значений массива `Pattern` с начала. Следует обратить внимание, что выполняется постинкремент `PatternIndex`:

```
*(Pattern + PatternIndex++)
```

В этом выражении происходит вычисление адреса текущего элемента. После вычисления адреса происходит инкремент `PatternIndex`. Задержка выполнения программы на 500 миллисекунд нужна для того, чтобы можно было увидеть «бегущий огонь». Подключите сигналы регистра `BASE (U13)` к драйверу светодиодов (`U3`) для проверки программы, полный текст которой приведён ниже.

Листинг 7.5. Полный текст программы «бегущий огонь»

```
// Полный текст программы "бегущий огонь"
#include <iostream.h>
#include <conio.h>
#include <dos.h>

class ParallelPort
{
    private:
        unsigned int BaseAddress;
        unsigned char InDataPort1;

    public:
        ParallelPort();
        ParallelPort(int baseaddress);
        void WritePort0(unsigned char data);
        void WritePort2(unsigned char data);
        unsigned char ReadPort1();
};

ParallelPort::ParallelPort()
{
    BaseAddress = 0x378;
    InDataPort1 = 0;
}

ParallelPort::ParallelPort(int baseaddress)
{
    BaseAddress = baseaddress;
    InDataPort1 = 0;
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress,data);
}

void ParallelPort::WritePort2(unsigned char data)
{
    outportb(BaseAddress+2,data ^ 0x0B);
}

unsigned char ParallelPort::ReadPort1()
```

```

{
    InDataPort1 = inportb(BaseAddress+1);
    // Инвертирование старшего бита для
    // коррекции аппаратной инверсии порта принтера
    InDataPort1 ^= 0x80;
    // Маскируем неиспользуемые биты данных
    // D0, D1 и D2 нулём
    InDataPort1 &= 0xF8;
    return InDataPort1;
}

class LEDs : public ParallelPort
{
private:
    unsigned char Pattern[8];
    int PatternIndex;

public:
    LEDs();
    LEDs(int baseaddress);
    void LightLEDs();
};

LEDs::LEDs()
{
    // Заполнение массива Pattern
    for(int i = 0; i < 8; i++)
        *(Pattern + i) = 1 << i;

    PatternIndex = 0; // инициализация значением 0
}

LEDs::LEDs(int baseaddress) : ParallelPort(baseaddress)
{
    // Заполнение массива Pattern
    for(int i = 0; i < 8; i++)
        *(Pattern + i) = 1 << i;

    PatternIndex = 0; // инициализация значением 0
}

void LEDs::LightLEDs()
{
    while(!kdhit()) // функция завершается после
                    // нажатия на кнопку
    {
        WritePort0(*(Pattern + PatternIndex++));

        // Сброс PatternIndex в исходное состояние,

```

```

        // когда его значение будет равно 8
        if(PatternIndex == 8) PatternIndex = 0;

        delay(500);
    }
}

void main()
{
    LEDs leds;
    leds.LightLEDs(); // эффект "бегущий огонь"
    getch();

    cout << endl << "Halted !" << endl;
    cout << "Press a key to continue" << endl;
    getch();

    leds.LightLEDs(); // "Бег" продолжается
                     // со следующей позиции
}

```

«Бегущие огни» – пользователь определяет содержимое массива постоянного размера

Программа из **Листинга 7.5** не обладает гибкостью. Содержимое массива `Pattern` определено в классе. Гораздо лучше предоставить пользователю возможность самому определять содержимое массива, поэтому внесём изменения в класс `LEDs`.

Пользователь будет сам определять содержимое массива, который использует для управления светодиодами. Поэтому нет необходимости инициализировать этот массив в конструкторах. Вместо этого в классе будет указатель на массив, который будет определяться пользователем. Такой массив может использоваться функцией `LightLEDs()`, если будут известны его начальный адрес и размер. Поэтому в классе `LEDs` нужны два члена данных для хранения адреса массива и его размера.

В этом случае проще всего заменить член данных `Pattern` (массив значений типа `unsigned char`) на `PatternPtr`, представляющим собой указатель на `unsigned char`. Также нужна функция-член, которая вычисляла бы адрес массива и присваивала его этому указателю на массив. Поскольку размер массива может теперь выбираться произвольно, то появляется необходимость в члене данных, где хранилось бы максимально возможное количество элементов массива. При помощи этого члена класса будем выяснять, не достигнут ли конец массива, а затем сбрасывать `PatternIndex` в исходное состояние 0. Изменения класса `LEDs` показаны в **Листинге 7.6**.

Листинг 7.6. Изменения в классе `LEDs`

```

class LEDs : public ParallelPort
{

```

```

private:
    unsigned char* PatternPtr;
    int PatternIndex;
    int MaxIndex;

public:
    LEDs();
    LEDs(int baseaddress);
    void SetPatternAddress(unsigned char* pattern, int maxidx);
    void LightLEDs();
};

```

Определения функций-членов этого класса приведены в **Листинге 7.7**.

Листинг 7.7. Определения функций модифицированного класса

```

LEDs::LEDs()
{
    MaxIndex = 0;
    PatternIndex = 0;
}

LEDs::LEDs(int baseaddress) : ParallelPort(baseaddress)
{
    MaxIndex = 0;
    PatternIndex = 0;
}

void LEDs::SetPatternAddress(unsigned char* pattern, int maxidx)
{
    PatternPtr = pattern; // Присвоение адреса массива
                        // указателю PatternPtr
    MaxIndex = maxidx;
}

void LEDs::LightLEDs()
{
    if(MaxIndex <= 0)
    {
        cout << "No Patterns to display" << endl;
        return;
    }

    while(!kbhit())
    {
        WritePort0(*(PatternPtr + PatternIndex++));

        // Сброс PatternIndex в исходное состояние
        // по достижении конца массива
    }
}

```

```

        if(PatternIndex == MaxIndex) PatternIndex = 0;

        delay(500);
    }
}

```

В **Листинге 7.8** приведена функция `main()`, позволяющая пользователю ввести маски свечения светодиодов в массив.

Листинг 7.8. Функция `main()`, позволяющая ввести маски свечения светодиодов

```

void main()
{
    unsigned char LightPattern[8];
    int UserPattern;

    LEDs Leds;
    int i;

    cout << "Enter 8 user patterns in the range 0x00-0xFF ";
    cout << endl;
    for(i = 0; i < 8; i++) // заполнение массива из 8 элементов
    {
        cin >> UserPattern;
        *(LightPattern + i) = UserPattern;
    }

    Leds.SetPatternAddress(LightPattern, 8);
    Leds.LightLEDs();
    getch();
    Leds.LightLEDs();
}

```

Обратите внимание, что в функции `main()` из **Листинга 7.8** был объявлен массив `LightPattern[8]`, где каждый элемент предназначен для хранения маски свечения светодиодов. Во время выполнения программы пользователь последовательно вводит нужные маски. В локальную переменную `UserPattern` типа `int` осуществляется ввод целых чисел. Затем эти данные присваиваются элементам массива `LightPattern` типа `unsigned char`, при этом имя массива `LightPattern` используется в качестве указателя.

Выражение в **Листинге 7.8**:

```
Leds.SetPatternAddress(LightPattern, 8);
```

можно записать так:

```
Leds.SetPatternAddress(LightPattern, sizeof(LightPattern));
```

Использование функций часто может быть более эффективным, если вызывать их при помощи предварительно определённых макросов. Как видно из предыдущего выражения, в вызове функции слово `LightPattern` встречается дважды. Чтобы минимизировать возможность ошибки при вызове можно определить ма-

крос, принимающий единственный аргумент `LightPattern`. В следующем разделе показано определение макроса.

7.7. Макросы

Макрос можно представить как шаблон, который заменяется препроцессором на выражение. Представим, например, что компилятору встречается следующее выражение:

```
y = x*x*x;
```

Тогда для упрощения программирования можно определить такой макрос:

```
#define CUBE(x) ((x)*(x)*(x))
```

Препроцессор будет заменять все вхождения `CUBE(x)` на `((x)*(x)*(x))`. Поэтому в программе можно смело использовать `CUBE(x)`. Дополнительные скобки нужны для жёсткого задания порядка выполнения операций. Например, если `CUBE(x)` определить так:

```
#define CUBE(x) (x * x * x)
```

То выражение `y = CUBE(x)`; будет заменено на `y = 3*3*3`, дающее в результате 27.

С другой стороны, выражение `y = CUBE(2+1)`; будет развёрнуто препроцессором в виде `y = 2+1*2+1*2+1`, приводящее к неверному значению 7.

Если же каждый параметр `x` поместить в скобки, т. е. `(x)*(x)*(x)`, то такое определение будет развёрнуто так:

```
y = (2+1)*(2+1)*(2+1);
```

Результат в этом случае будет правильным и равен 27. Если предыдущее определение макроса использовать в следующем выражении, то результат будет неправильным.

```
y = 81/CUBE(3);
```

Это будет заменено на:

```
y = 81/(3)*(3)*(3);
```

и в результате получится 243, а не ожидаемое значение 3. Дополнительная пара наружных скобок исправит и этот недостаток.

Усовершенствовать программу из **Листинга 7.8** можно введением следующего макроса:

```
#define SetArray(x) SetPatternAddress((x), sizeof(x))
```

Функция `main()` показана в **Листинге 7.9**.

Листинг 7.9. Функция `main()`, позволяющая ввести маски свечения светодиодов

```
#define SetArray(x) SetPatternAddress((x), sizeof((x)))
```

```
void main()
```

```

{
    unsigned char UserPattern, LightPattern[4];
    LEDs Leds;
    int i;

    cout << "Enter" << sizeof(LightPattern);
    cout << " user patterns in the range 0x00-0xFF ";
    cout << endl;
    for(i = 0; i < sizeof(LightPattern); i++)
    {
        cin >> UserPattern;
        *(LightPattern + i) = UserPattern;
    }
    Leds.SetArray(LightPattern);
    Leds.LightLEDs();
    getch();
    Leds.LightLEDs();
}

```

7.8. Динамическое выделение памяти

Часто возникает ситуация, когда в программе на С++ нужно динамически выделять память во время выполнения, путём запроса и получения запрашиваемой памяти. Динамическое выделение памяти широко применяется в программах на С++ и позволяет значительно уменьшить размер исполняемого файла программы. Динамическое выделение памяти особенно актуально, когда во время написания программы неизвестен объём данных, которые нужно хранить в памяти. Например, программа обработки оценок группы студентов должна работать при различном количестве студентов в группе, а это количество во время программирования неизвестно.

Выделение памяти бывает статическим и динамическим. В случае статического выделения компилятор выделяет требуемый объём памяти во время компиляции. Программы со статическим выделением памяти, как правило, имеют больший объём исполняемого файла, в отличие от программ с динамическим выделением памяти. Статически выделенная память располагается в *области данных*, которая предназначена для хранения данных. Область памяти, предназначенная для хранения инструкций, называется *областью кода* и располагается отдельно от области данных. Обычно инструкции программы, или код, не изменяется во время выполнения программы. А вот данные изменяются программой.

Временные данные создаются и удаляются во время выполнения программы в другой области памяти, называемой *стеком*. Стек представляет собой очередь типа «последний вошёл, первым вышел» (или *LIFO, Last In, First Out*) в специально выделенной области памяти компьютера. Некоторые примеры временных данных: передаваемые в функцию параметры, локальные переменные, объявленные внутри функций, возвращаемые функциями значения. В случае динамического выделения памяти запрашиваемая память предоставляется из ещё одной области памяти, которая называется *кучей* (или *свободной памятью*). Следует отметить,

что динамически выделенная память не возвращается системой в кучу автоматически. Ответственность за возврат динамической памяти обратно в систему (её освобождение), чтобы её можно было использовать в иных целях, ложится на программиста; существуют специальные команды для освобождения памяти.

В **Табл. 7.4** приведены два оператора, использующиеся при динамическом выделении памяти. Оператор `new` выделяет запрашиваемую память и возвращает указатель на начало выделенной области памяти. Приведём простейший пример использования оператора `new`:

```
int *IntPtr;  
IntPtr = new int;
```

К тому же результату приведёт и следующее выражение:

```
int *IntPtr = new int;
```

Таблица 7.4. Операторы динамического выделения памяти

Оператор	Действие
<code>new</code>	Запрашивает память
<code>delete</code>	Освобождает память

В этом примере была запрошена память из кучи для данных типа `int`. Выделенный блок памяти не имеет имени, вместо этого используется указатель `IntPtr`, который «знает», где этот блок находится. Поскольку мы уже умеем пользоваться указателями для работы с данными, то можем пользоваться этой памятью. Когда эта память станет ненужной, её следует освободить оператором `delete`:

```
delete IntPtr;
```

Эта операция не приводит к удалению указателя `IntPtr`. Вместо этого происходит освобождение или возврат в кучу участка памяти, на который указывает `IntPtr`, а значит динамически выделенная переменная целого типа становится недоступной в программе. Теперь память, ранее занятая этой переменной, может быть предоставлена в последующих запросах выделения памяти. Если память не была освобождена оператором `delete`, то её нельзя будет получить по запросу в оставшейся части программы. В этом случае мы допустили ситуацию, называемую *утечкой памяти*, когда уменьшается объём доступной памяти.

Теперь приведём более сложный пример, когда запрашивается память для массива из десяти целых чисел:

```
int *IntPtr;  
IntPtr = new int[10];
```

Когда выделенная память станет ненужной, её нужно освободить оператором `delete`:

```
delete IntPtr;
```

В следующем примере запрашивается память для двумерного массива из 100 элементов типа `int`:

```
int (*RowPtr[10]);  
RowPtr = new int[10][10];
```

RowPtr является указателем на строку из 10 элементов. Оператор new запрашивает память для хранения 10 наборов по 10 элементов типа int. Выделенную память можно освободить следующим образом:

```
delete RowPtr;
```

Таким же образом можно выделить память для объекта какого-либо класса. Чтобы динамически создать объект класса DAC, можно воспользоваться следующим выражением:

```
DAC *DACPtr;  
DACPtr = new DAC;
```

У динамически созданного (т. е. в предоставленной из кучи памяти) объекта класса DAC нет имени. Вместо имени используется указатель DACPtr, указывающий его размещение в памяти. Указатель на объект тоже можно использовать при работе с ним, как и его имя.

Выделенная память должна быть освобождена:

```
delete DACPtr;
```

Во всех этих случаях выделения памяти оператор new вызывает конструктор объекта. Хотя мы и не создавали конструктор типа int, у него есть свой конструктор. Например, при помощи следующих выражений можно выделить память для переменной типа int с начальным значением 0:

```
int *IntPtr;  
IntPtr = new int(0);
```

То же самое можно записать одной строкой:

```
int *IntPtr = new int(0);
```

Аналогично, если нужно создать новый объект класса DAC, который будет в качестве адреса BASE использовать 0x3BC вместо 0x378, то воспользуемся следующим выражением:

```
DAC* DACPtr = DAC(0x3BC);
```

Когда оператор new вызывает конструктор класса DAC, то ему будет передан указанный параметр (в данном случае 0x3BC).

Когда речь идёт об иерархии классов, то указатель не обязательно должен указывать на тип производного класса, он может указывать на один из базовых классов. В случае класса DAC следующие выражения будут правильными:

```
ParallelPort *Ptr;  
Ptr = new DAC;
```

Тот же указатель может указывать на другой класс из той же иерархии. Предположим, глубже по иерархии есть класс DAC16Bit. Тогда доступны следующие выражения:

```
Ptr = new DAC16Bit;
```

Эта возможность будет использована совместно с виртуальными функциями в разделе 8.6 в более совершенных программах.

Операции выделения памяти могут завершаться отказом в предоставлении памяти. Это может быть вызвано недостаточным объемом свободной памяти. Если выделения памяти не происходит, то указатель, возвращаемый оператором `new`, будет иметь значение `NULL`, являющееся предопределённой константой. Отказ в выделении памяти можно обнаружить следующим простым выражением:

```
if(Ptr == NULL)
{
    cout << "Memory allocation failed";
    exit(1);
}
```

Функция `exit()` является библиотечной функцией, вызываемой для прекращения выполнения программы (если вы считаете, что при недостатке памяти программу нужно завершить). Другим вариантом является генерация исключений в программе, как описано в разделе 7.9. Любые указатели, получаемые от оператора `new`, можно проверять перед их использованием. Обычно функции `exit()` в качестве параметра передаётся 1. Это информирует систему об аварийном завершении программы.

Приведение типов

Приведение типов используется для преобразования значения одного типа к другому типу, если это возможно. Следующий пример демонстрирует приведение базовых типов:

```
int a;
float b = 8.73;
a = (int) b; // a будет равно 8
```

Переменная `b` типа `float` *приводится* в соответствии типу переменной `a` (это тип `int`) и поэтому значение округляется до 8. Таким же образом можно приводить и типы указателей, как показано в следующем примере:

```
int *IntPtr;
IntPtr = (*int) new int[10][10];
```

Указатель, получаемый от оператора `new`, приводится в соответствие типу указателя `IntPtr` при помощи выражения `(*int)`, интерпретирующегося как «указатель на `int`». Обратите внимание, что оператор `new` возвращает указатель на массив из 10 элементов (поскольку строка созданного массива состоит из 10 элементов). Тем не менее, указатель `IntPtr` будет указывать на переменную типа `int`.

7.9. Обработка исключений

Обработка исключений позволяет программе выполнять необходимые действия при возникновении исключительных ситуаций. Такие ситуации обычно возникают, когда программа не может продолжать работу вследствие обстоятельств, приводящих к невозможности нормального выполнения программы. Например,

нормальному выполнению программы может препятствовать недостаток памяти в системе для удовлетворения запроса выделения памяти.

Есть множество причин, вызывающих аварийное завершение программы; это может быть недостаток дискового пространства для записи в файл, попытка записи в файл, который был уже открыт для чтения и т. д. Такие ситуации возникают вследствие независимых от программы обстоятельств, связанных с состоянием среды выполнения программы и не подразумевающих ошибок в коде программе. Для работы в таких ситуациях в C++ используется *обработка исключений*. Могут обрабатываться только те исключения, которые генерируются во время выполнения программы. Не могут обрабатываться такие исключения, как завершение программы по нажатию клавиш «Ctrl+C».

При обработке исключений используются служебные слова `try`, `throw` и `catch`. Словом `try` обозначается *блок try* (блок попытки). Блок `try` состоит из служебного слова `try`, за которым в фигурных скобках следуют выражения этого блока:

```
try
{
    ...
}
```

Внутри этого блока содержатся выражения, выполнение которых может вызвать исключительную ситуацию. Каждая исключительная ситуация должна иметь обозначение и генерироваться выражением `throw`. В примере из **Листинга 7.10** производится попытка выделить память из кучи в размере `n` переменных типа `unsigned char`. Эта попытка может завершиться неудачно в следующих случаях: если значение `n` меньше 1 или при недостатке свободной памяти.

Листинг 7.10. Пример блока `try` для динамического выделения памяти

```
unsigned char* LightPattern;
try
{
    if (n < 1)
        throw(n);
    LightPattern = new unsigned char[n];
    if (LightPattern == NULL)
        throw("Memory error");
}
```

В этом примере важным является синтаксис выражений `throw`. Первое выражение `throw` имеет только одно число `n`. Второе выражение `throw` генерирует строку. Сразу за блоком `try` следуют соответствующие *выражения-ловушки*. В нашем случае должны быть выражения `catch`, соответствующие целому числу из первого выражения `throw` и строке из второго выражения. Вместе с этими двумя ловушками **Листинг 7.10** приобретает вид, показанный в **Листинге 7.11**.

Листинг 7.11. Блок `try` с ловушками

```
unsigned char* LightPattern;

try
```

```

{
    if((n < 1) || (n >4)) // Примечание: || это логическое ИЛИ
        throw(n);

    LightPattern = new unsigned char[n];
    if(LightPattern ==NULL)
        throw("Memory error");
}

catch(int n) // ловит сгенерированное целое число
{
    cout << "Illegal number of elements requested" << endl;
    cout << "Array size defaults to 4" << endl;
    n = 4;
    LightPattern = new unsigned char[n];
}

catch(char* memerror)
{
    cout << "Memory allocation failed" << endl;
    cout << "Terminating program" << endl;
    exit(1);
}

```

«Бегущие огни» – пользователь определяет массив произвольного размера

Можно усовершенствовать программу из **Листинга 7.9** так, чтобы пользователь мог определять не только содержимое массива масок свечения светодиодов, но его размер. Каждое число, введённое пользователем в массив, будет по порядку выводиться на линейку светодиодов с небольшой задержкой. Поэтому функция `main()` может быть переписана следующим образом с использованием динамического выделения памяти и механизма обработки исключений, как показано в **Листинге 7.12**.

Листинг 7.12. Функция `main()` для динамического выделения памяти и механизма обработки исключений

```

#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <dos.h>

#define SetArray(x) SetPatternAddress((x), sizeof((x)))

class ParallelPort
{
private:
    unsigned int BaseAddress;
    unsigned char InDataPort1;

public:

```

```
ParallelPort();
ParallelPort(int baseaddress);
void WritePort0(unsigned char data);
void WritePort2(unsigned char data);
unsigned char ReadPort1();
};

ParallelPort::ParallelPort()
{
    BaseAddress = 0x378;
    InDataPort1 = 0;
}

ParallelPort::ParallelPort(int baseaddress)
{
    BaseAddress = baseaddress;
    InDataPort1 = 0;
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress,data);
}

void ParallelPort::WritePort2(unsigned char data)
{
    outportb(BaseAddress+2,data ^ 0x0B);
}

unsigned char ParallelPort::ReadPort1()
{
    InDataPort1 = inportb(BaseAddress+1);
    // Инвертируем старший бит для компенсации внутренней
    // инверсии схемы порта принтера.
    InDataPort1 ^= 0x80;
    // Используем маску (или фильтр) для обнуления
    // неиспользуемых битов D0, D1 и D2.
    InDataPort1 &= 0xF8;
    return InDataPort1;}

class LEDs : public ParallelPort
{
private:
    unsigned char* PatternPtr;
    int PatternIndex;
    int MaxIndex;

public:
```



```

        LEDs();
        LEDs(int baseaddress);
        void SetPatternAddress(unsigned char* pattern, int maxidx);
        void LightLEDs();
};

LEDs::LEDs()
{
    MaxIndex = 0;
    PatternIndex = 0;
}

LEDs::LEDs(int baseaddress) : ParallelPort(baseaddress)
{
    MaxIndex = 0;
    PatternIndex = 0;
}

void LEDs::SetPatternAddress(unsigned char* pattern, int maxidx)
{
    PatternPtr = pattern; // указателю PatternPtr присваивается
                          // адрес массива с масками
    MaxIndex = maxidx;
}

void LEDs::LightLEDs()
{
    if(MaxIndex == 0)
    {
        cout << "No patterns to display" << endl;
        return;
    }

    while(!kbhit())
    {
        WritePort0(*(PatternPtr + PatternIndex++));

        // Сброс PatternIndex в исходное состояние 0
        // по достижении конца массива
        if(PatternIndex == MaxIndex) PatternIndex = 0;
        delay(500);
    }
    getch(); // считать нажатие кнопки
}

void main()
{
    LEDs leds;
    unsigned char* LightPattern;

```

```
int TempPattern;
int n, i;

cout << "Pass in the desired size of LightPattern => ";
cin >> n;

try
{
    if(n < 1)
        throw(n);
    LightPattern = new unsigned char[n];
    if(LightPattern == NULL)
        throw("Memory error");
}

catch(int n) // ловушка для integer
{
    cout << "Illegal number of elements requested" << endl;
    cout << "Array size defaults to 4" << endl;
    n = 4;
    LightPattern = new unsigned char[n];
}

catch(char*) // ловушка для строки
{
    cout << "Memory allocation failed" << endl;
    cout << "Terminating program" << endl;
    return;
}

cout << "Enter " << n;
cout << " numbers in the range (0x00 - 0xFF)" << endl;

for(i = 0; i < n; i++)
{
    cin >> TempPattern;
    *(LightPattern + i) = TempPattern;
}

Leds.SetPatternAddress(LightPattern,n);
Leds.LightLEDs();
}
```

7.10. Заключение

В этой главе была рассмотрена работа различных циклов, таких как `for`, `while` и `do-while`. Цикл `for` обычно используется в тех случаях, когда число итераций

известно во время написания программы. Если этот цикл не подходит, то можно использовать циклы `while` и `do-while`. В некоторых случаях тело цикла `while` так и не выполнится ни разу, тогда как цикл `do-while` выполняется по меньшей мере один раз. Различные способы ветвления, такие как `if`, `switch-case`, `break` и `continue` могут совместно использоваться для организации выполнения программы наиболее удобным образом. Выражение `switch` позволяет выполнять разные действия в зависимости от условия.

Указатели являются важной и мощной частью языка C++ и о них было рассказано в этой главе. Они содержат адрес памяти, по которому в памяти размещаются различные объекты и функции. Адресная арифметика, применяемая к указателям, принимает во внимание размер указываемого объекта. Указатели могут эффективно использоваться для перебора значений массива, что и было продемонстрировано в программах этой главы. Самое главное, что указатели на объекты базового класса могут указывать и на объекты производных классов.

Для размещения в памяти вновь создаваемых объектов и массивов из них во время выполнения программы было использовано динамическое выделение памяти. При динамическом выделении памяти оператор `new` возвращает указатель на предоставленную память. Для освобождения выделенной памяти нужно использовать оператор `delete`.

Была рассмотрена обработка исключений для обработки возможных ошибок времени выполнения. Выражения, которые могут вызвать ошибки времени, содержатся внутри блока `try`, при возникновении ошибки выполняется выражение `catch`. Обработка исключений применялась для выполнений действий в ошибочных ситуациях, когда задаётся неверный размер массива или при недостаточном объёме свободной памяти для размещения памяти.

7.11. Литература

1. Kelley, A. and I. Pohl, *A Book on C – programming in C*, Benjamin Cummins, 1995.
2. House, R., *Beginning with C – An Introduction to Professional Programming*, International Thompson Publishing, 1994.
3. Deitel H. M. And P. J. Deitel *C: How to program*, Prentice Hall, 1994.
4. L. Miller and A. Quilci, *C programming Language – an applied perspective*, John Wiley Publishing, 1987.
5. Hanly, J. R., E. B. Koffman and J. C. Horvath, *C Program Design for Engineers*, Addison Wesley, 1995.
6. Rudd, A., *Mastering C*, John Wiley, 1994.
7. Lafore, R., *Object Oriented Programming in C++*, Waite Group Press, 1995.

8 Управление шаговыми и коллекторными электродвигателями

Содержание главы:

- Коллекторные двигатели: типы, характеристики и управление.
- Шаговые двигатели и способы управления ими.
- Виртуальные функции и иерархии классов.
- Использование указателей и динамическое выделение памяти.
- Управление двигателями с клавиатуры.

8.1. Введение

Электродвигатели находят широкое применение в бытовых и промышленных устройствах. Электродвигатели, управляемые компьютерами, становятся важной частью многих систем управления движением. В этой главе описаны основные конструкции электродвигателей постоянного тока, их характеристики, способы управления и их поведение под нагрузкой. Рассматривается способ управления коллекторным электродвигателем (электродвигателем постоянного тока), известный как широтно-импульсная модуляция (ШИМ). Затем будут рассмотрены типы шаговых двигателей, их устройство, различные режимы работы и способы управления ими.

Программирование в этой главе начинается с создания абстрактного класса, представляющего двигателя в общем виде. На базе этого абстрактного класса и класса `ParallelPort` создаётся новый класс `Motor` при помощи множественного наследования. Новый класс будет иметь возможность взаимодействовать с параллельным портом компьютера. От этого класса создаются другие классы двигателей постоянного тока и различные варианты шаговых двигателей, демонстрируя разработку иерархий классов.

В программах управления различными типами двигателей используется динамическое выделение памяти. На протяжении этой главы на примерах программ управления двигателями с клавиатуры будут демонстрироваться возможности виртуальных функций и преимущества позднего связывания.

8.2. Двигатели постоянного тока

Большинство двигателей, использующихся для управляемого привода, являются двигателями постоянного тока с постоянными магнитами. Они все характеризуются линейной зависимостью крутящего момента от скорости вращения и бывают нескольких распространённых видов.

8.2.1. Конструкция и характеристики двигателей постоянного тока

Основная конструкция коллекторного двигателя показана на **Рис 8.1**. Ток течёт от источника питания (изображённого в виде батареи) и по обмотке двигателя через щетки и коллектор. Этот ток создаёт магнитное поле в обмотке, противоположное магнитному полю постоянных магнитов, при этом возникает крутящий момент, способный произвести работу. Чтобы крутящий момент был равномерным, в двигателях обычно делают несколько обмоток. Каждая обмотка подключается к диаметрально противоположным сторонам коллектора, все контакты коллектора электрически изолированы друг от друга.

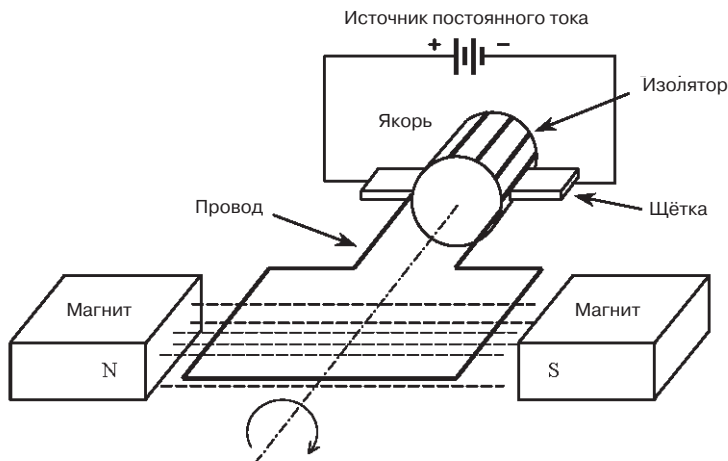


Рис. 8.1. Общая конструкция двигателя постоянного тока

В настоящее время в электродвигателях широко используются постоянные магниты. Эти магниты изготавливаются из сплава AlNiCo (алюминий, никель и кобальт), а магниты мощных двигателей изготавливают из сплава редкоземельных элементов самария и кобальта.

Есть другой тип двигателя постоянного тока, который называется безщёточным двигателем, у него нет коллекторных щёток, вместо этого ток обмоток коммутируется электронным способом. У этих двигателей постоянные магниты установлены в роторе, а обмотки находятся в статоре. Такое расположение гораздо выгоднее с точки зрения отвода тепла от двигателя и приводит к более высокой надёжности, низкому шуму, высокой скорости и большим предельным значениям крутящего момента.

Производительность двигателя зависит от приложенной к двигателю нагрузки и напряжения на двигателе. Трение в подшипниках двигателя, потери на щётках, потери в железе и потери на токи короткого замыкания (при перекрытии щётками одновременно двух соседних контактов коллектора) ограничивают скорость вращения на холостом ходу и снижают крутящий момент на валу, ограничивая минимальную скорость вращения. При увеличении нагрузки двигателя скорость

вращения уменьшается пропорционально возрастанию нагрузки. Воздушное охлаждение двигателя или его работа в течение коротких промежутков времени позволяют увеличить нагрузку на нём. При выборе двигателя необходимо учитывать инерционность нагрузки, крутящий момент и мощность, развиваемую под нагрузкой.

8.2.2. Управление коллекторным двигателем

Управление коллекторными двигателями осуществляется посредством изменения величины постоянного напряжения питания двигателя. Источники питания двигателя делятся на два типа: линейные и импульсные. В основе линейных источников питания лежит мощный транзистор, работающий в линейном режиме и обеспечивающий возможность плавного изменения выходного напряжения, соответствующего нагрузке на двигателе. В этом случае транзистор играет роль переменного резистора, проводимость которого зависит от величины входного управляющего сигнала. Для этого режима характерны потери мощности на транзисторе, обусловленные падением напряжения на нём.

В импульсных источниках питания напряжение на двигателе формируется импульсами напряжения, что приводит к некоторому среднему значению напряжения на двигателе. Импульсное управление делится на два вида: широтно-импульсная модуляция (ШИМ) и частотно-импульсная модуляция (ЧИМ), **Рис. 8.2.**

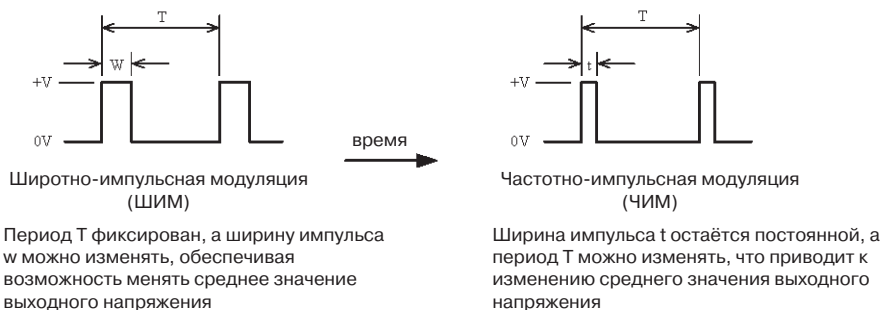


Рис. 8.2. Формирование напряжения разными импульсными способами

Управление двигателями осуществляется, как правило, при помощи широтно-импульсной модуляции. Такой способ подразумевает неизменный период T следования импульсов, тогда как ширина w импульсов (иначе говоря, *скважность*) меняется, приводя к изменению среднего значения напряжения на двигателе.

В случае частотно-импульсной модуляции изменение частоты следования импульсов постоянной длительности t приводит к изменению среднего напряжения на двигателе.

Обычно двигатели постоянного тока включаются по мостовой схеме, **Рис. 8.3.** Для подачи напряжения на двигатель нужно замкнуть два ключа. Когда ключи $S1$ и $S4$ замкнуты, а $S2$ и $S3$ разомкнуты, как изображено на схеме, то направление вращения вала двигателя будет одним. Если переключить все ключи, что будет соответствовать замкнутым $S2$ и $S3$ и разомкнутым $S1$ и $S4$, то ток через двигатель будет протекать в противоположном направлении, а вал вращаться в обратную сторону.

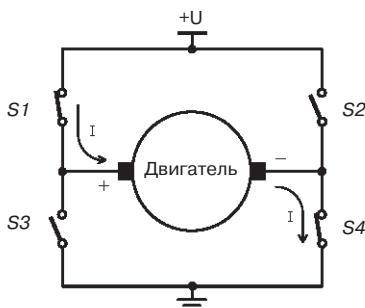


Рис. 8.3. Мостовая схема включения двигателя

Среднее напряжение на двигателе можно регулировать при помощи широтно-импульсного управления ключами. На практике в качестве ключей используются силовые транзисторы в ключевом режиме работы. При размыкании цепи ключа на обмотках двигателя возникает высокое напряжения, известное как *электро-движущая сила (ЭДС) самоиндукции*. Это напряжение необходимо ограничивать, чтобы не вышли из строя транзисторные ключи. Пробой транзисторов предотвращается подключением к транзисторам диодов, понижающих напряжение до безопасного уровня.

Управление двигателями осуществляется как по схемам с обратной связью, так и без неё. При отсутствии обратной связи не снимается никакого сигнала о вращении вала двигателя. В этом случае необходимая скорость вращения вала определяется исходя из известных нагрузки на валу и характеристик двигателя. Если нужно точно выдерживать скорость двигателя, а нагрузка на валу неизвестна или переменная, то обычно используют схемы с обратной связью. При управлении скоростью вращения с двигателя снимается сигнал скорости вращения вала. Сигнал скорости обычно формируется тахометром, выходное напряжение которого пропорционально скорости двигателя. Бывают также тахометры, генерирующие импульсы.

Когда нужно управлять положением вала двигателя используют определители положения (их называют ещё резольверами) или цифровыми шифраторами (энкодерами) для получения сигнала положения в цепи обратной связи. Резольверы генерируют два противофазных синусоидальных сигнала, амплитуды которых несут информацию о положении. У оптических энкодеров выходным сигналом являются два цифровых сигнала, взаимно сдвинутых по фазе примерно на 90° , где фазовый сдвиг несёт информацию о направлении вращения, **Рис. 8.4**. Направление вращения определяют по порядку следования фронтов импульсов выходных сигналов.

8.3. Шаговые двигатели

Шаговые двигатели часто используются в тех устройствах, где необходима принудительная фиксация (удержание) и хороший крутящий момент на низких скоростях вращения. Если двигатель не перегружен, то скорость вращения вала и его угловое положение предсказуемы, а управление осуществляется простым пере-

ключением тока в обмотках и часто не возникает необходимости в применении энкодеров.

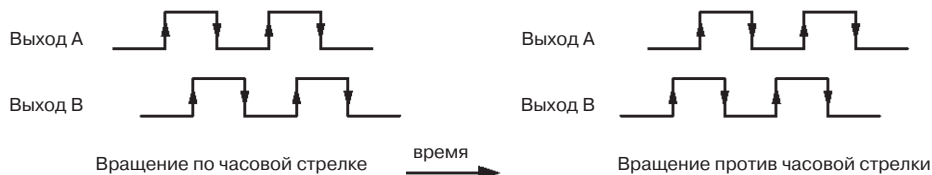


Рис. 8.4. Выход шифратора вращения со сдвигом на 90°

8.3.1. Конструкции шаговых двигателей

Находят применение три основных типа шаговых двигателей: с *постоянным магнитом*, с *переменным магнитным сопротивлением* и *комбинированные*.

В конструкциях с **постоянным магнитом** ротор представляет собой постоянный многополюсный магнит. Двигатели этого типа обладают остаточным моментом сопротивления вращению при отсутствии тока в обмотках, а их энергопотребление меньше, чем у других типов. Они недороги, их крутящий момент невысок и скорость вращения невелика. Тем не менее, они подвержены резонансам, время установления сравнительно велико и нестабильная работа на низких скоростях вращения. Главным образом, они применяются в непромышленных устройствах.

В двигателях с **переменным магнитным сопротивлением** ротор изготавливается из магнитомягких сортов стали, а количество их полюсов не равно количеству полюсов статора. Когда по обмоткам статора поочередно протекает ток, ротор в магнитном поле статора занимает такое положение, при котором сопротивление магнитному потоку минимально. У таких двигателей хорошее соотношение крутящего момента и инерции, хотя остаточный момент сопротивления вращению невелик. Двигатели с переменным магнитным сопротивлением редко используются в промышленных устройствах и им нужны другие схемы управления в отличие от шаговых двигателей других типов.

В двигателях **комбинированной** конструкции используются узлы, характерные и для типов с постоянным магнитом и для типов с переменным магнитным сопротивлением. Двигатели этого типа наиболее часто применяются в промышленных устройствах из-за их большого крутящего момента и остаточного момента сопротивления вращению.

8.3.2. Устройство шаговых двигателей

Принцип работы шагового двигателя будем рассматривать на упрощённом примере, приведённого на Рис. 8.5. Ротор этого двигателя представляет собой постоянный магнит и обладает двумя парами полюсов, а статор с четырьмя полюсами. Работа двигателя проиллюстрирована так называемыми *полными фазами*, каждая фаза соответствует повороту вала на 90° . Статор двигателя имеет две независимые обмотки, поэтому этот двигатель является *двухфазным*.

Если обмотки двигателя питать током согласно приведённой в Табл. 8.5 последовательности, то вал будет вращаться по часовой стрелке, Рис. 8.1. В каждой фазе вращения через обмотки протекает ток.

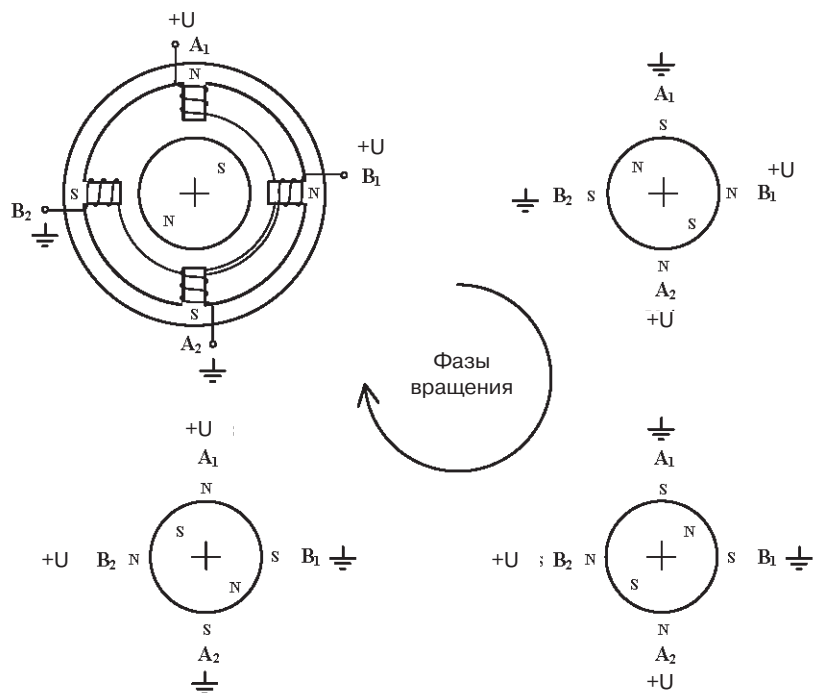


Рис. 8.5. Полный цикл двухфазного шагового двигателя

Таблица 8.1. Последовательность фаз двухфазного двигателя

Фаза	Питание обмоток			
	A1	A2	B1	B2
1	+U	Земля	+U	Земля
2	Земля	+U	+U	Земля
3	Земля	+U	Земля	+U
4	+U	Земля	Земля	+U

Фазы вращения могут составлять меньший угол, если питать обмотки в *полуфазной* последовательности. Под этим подразумевается питание только одной обмотки в каждой второй фазе вращения, Рис 8.6 и Табл. 8.2.

Таблица 8.2. Последовательность полуфаз двухфазного двигателя

Фаза	Питание обмоток			
	A1	A2	B1	B2
1	+U	Земля	+U	Земля
2			+U	Земля

Таблица 8.2. Последовательность полуфаз двухфазного двигателя (окончание)

Фаза	Питание обмоток			
	A1	A2	B1	B2
3	Земля	+U	+U	Земля
4	Земля	+U		
5	Земля	+U	Земля	+U
6			Земля	+U
7	+U	Земля	Земля	+U
8	+U	Земля		

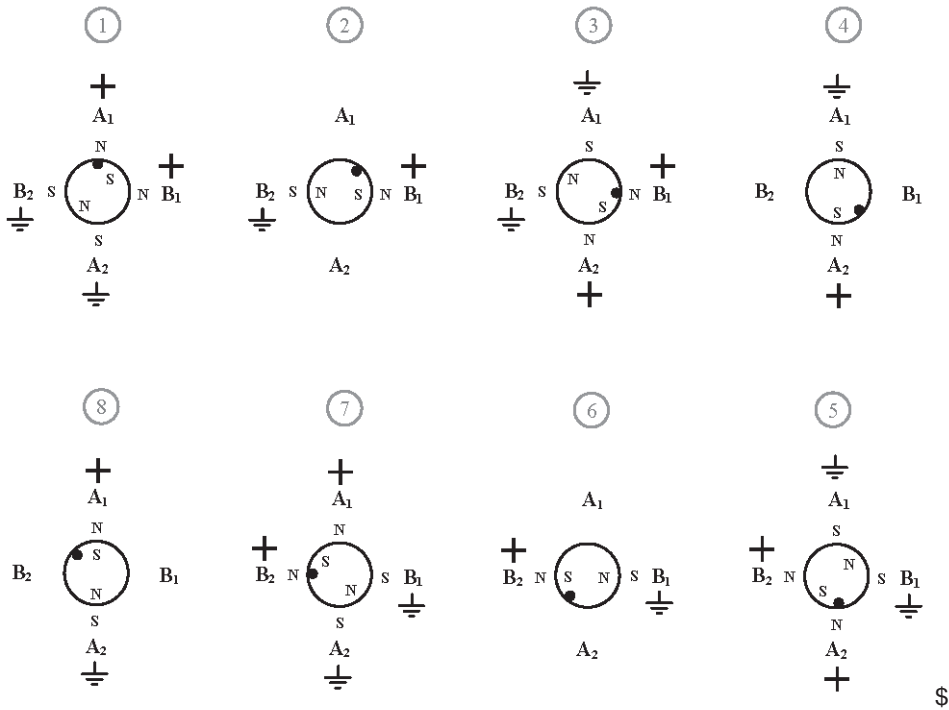


Рис. 8.6. Полуфазный полный цикл двухфазного шагового двигателя

Шаговые двигатели классифицируются по последовательности питания током их обмоток, сейчас рассмотрим это.

Биполярный двигатель

В таких двигателях в течении цикла вращения токи в обмотках меняют своё направление, что было описано ранее. Направление тока меняется путём изменения полярности напряжения на обмотках. Питая такой двигатель можно при помощи двух источников напряжения +U и -U относительно общей земли и четырёх ключевых транзисторов. На самом деле, питание осуществляется от одного источника

питания и при помощи восьми ключевых транзисторов, образующих два моста, **Рис 8.7.** Биполярные двигатели легко узнать, поскольку они обладают всего четырьмя выводами.

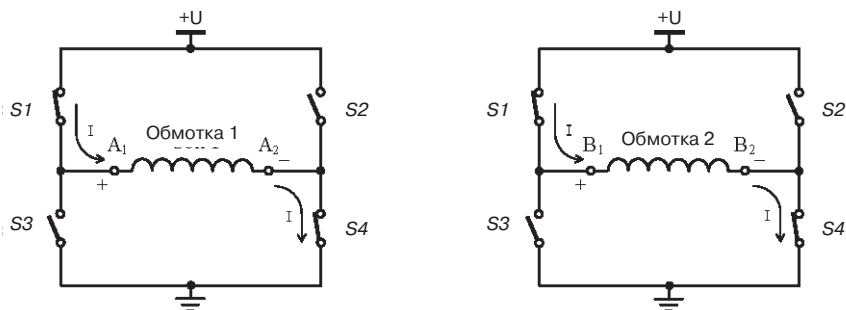


Рис. 8.7. Питание двухфазного двигателя по мостовой схеме

Униполярный двигатель

По обмоткам таких двигателей ток протекает всегда только в одном направлении. Магнитное поле на полюсах статора меняется на противоположное благодаря тому, что каждая обмотка состоит из двух секций, **Рис. 8.8.** Одна секция навита по часовой стрелке на полюсе статора, а другая против часовой стрелки, это так называемая *бифилярная* намотка.

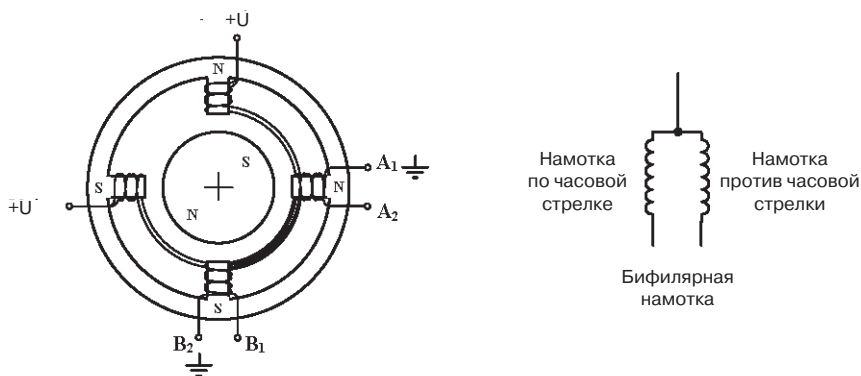


Рис. 8.8. Униполярный двигатель с бифилярной намоткой

Средний вывод бифилярной обмотки подключается к положительному полюсу источника питания. В любой момент времени ток протекает только по одной из двух половин обмотки путём подключения одного из крайних выводов к земле. Магнитное поле противоположной направленности получается подключением к земле другого крайнего вывода обмотки; **Рис. 8.9, Рис. 8.10 и Табл. 8.3.** Заметим, что северный полюс получается при подключении к земле секций A1 или B1, и наоборот, заземление A2 и B2 образует южный полюс магнитного поля.

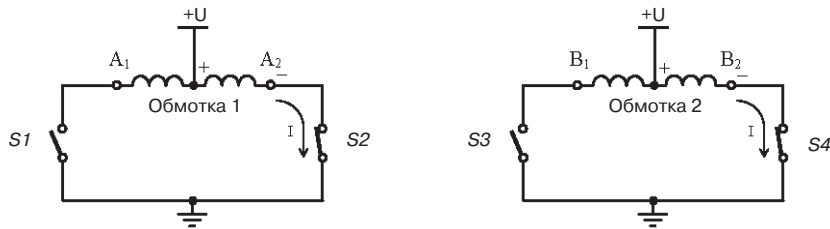


Рис. 8.9. Питание униполярного двигателя

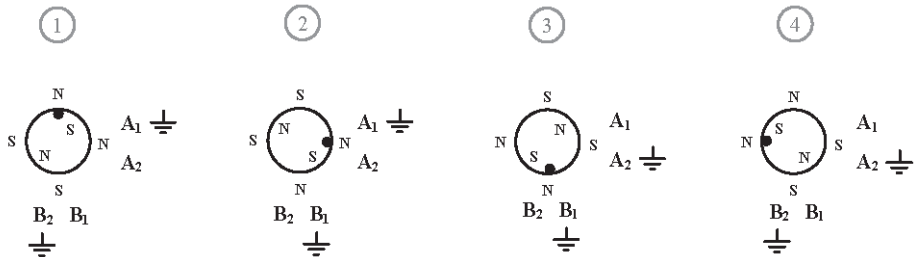


Рис. 8.10. Цикл полных фаз в униполярном двигателе

Таблица 8.3. Последовательность полных фаз униполярного двигателя

Фаза	Питание обмоток			
	A1	A2	B1	B2
1	Земля			Земля
2	Земля		Земля	
3		Земля	Земля	
4		Земля		Земля

Полуфазное питание достигается пропуском тока по очереди: сначала в одной секции обеих обмоток, а затем только в секции одной обмотки, при этом получается магнитное поле, как и при полуфазном питании биполярного двигателя. На Рис. 8.11 и в Табл. 8.4 приведена последовательность полуфазного питания униполярного двигателя.

Таблица 8.4. Последовательность полуфаз в униполярном двигателе

Фаза	Питание обмоток			
	A1	A2	B1	B2
1	Земля			Земля
2	Земля			
3	Земля			
4			Земля	
5			Земля	
6		Земля		
7		Земля		Земля
8		Земля		Земля

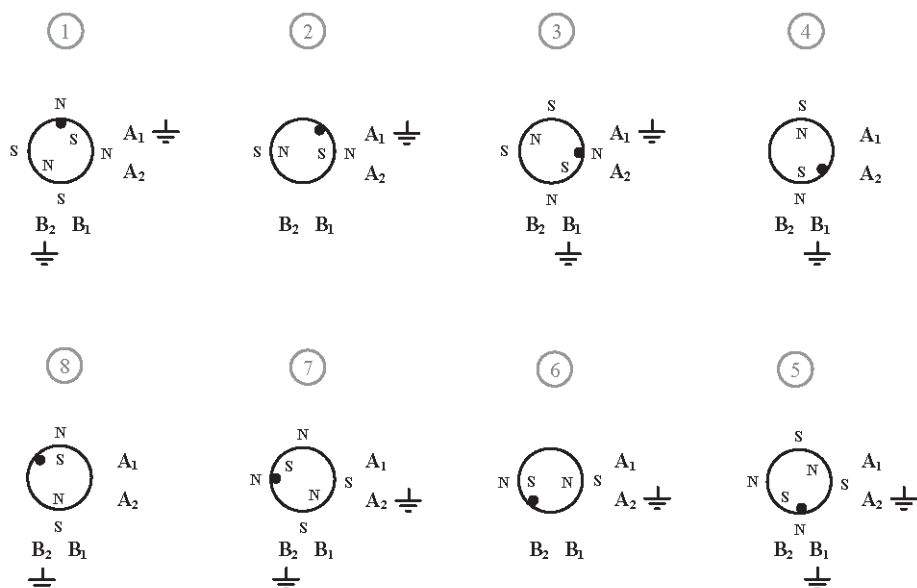


Рис. 8.11. Цикл полуфаз в униполярном двигателе

В униполярном двигателе для намотки того же количества витков, как в биполярном двигателе, нужно использовать провод меньшего диаметра из-за бифилярной намотки. Уменьшение диаметра провода приводит к увеличению сопротивления обмоток и снижению крутящего момента примерно на 30% на низких скоростях. На высоких скоростях характеристики униполярных двигателей превосходят аналогичные у биполярных двигателей.

Различные варианты исполнения обмоток шаговых двигателей приведены в **Табл. 8.5**. Пятивыводной униполярный двигатель эквивалентен шестивыводному униполярному двигателю, у которого два вывода питания объединены внутри, а наружу выведен только один контакт питания.

Таблица 8.5. Варианты обмоток шаговых двигателей

Количество выводов	Тип двигателя
4 вывода	Биполярный, двухфазный;
4 вывода	С переменным магнитным сопротивлением, трёхфазный;
5 выводов	Униполярный, двухфазный;
6 выводов	Униполярный, двухфазный;
8 выводов	Биполярный, четырёхфазный

В восьмивыводных биполярных двигателях независимые обмотки образуют две двухфазные пары. Такая схема допускает последовательное или параллельное включение фазовых пар. При последовательном включении увеличивается в два раза число ампер-витков, что приводит к удваиванию крутящего момента на низ-

ких скоростях. Суммарная индуктивность получившейся двойной обмотки пропорциональна квадрату количества витков, а значит, индуктивность увеличивается в четыре раза. Сопротивление обмотки удвоилось, следовательно, ток через обмотки ограничен и не превысит допустимого.

Параллельное включение обмоток восьмивыводного двигателя не изменяет суммарного количества витков и индуктивность не увеличивается. Сопротивление обмоток уменьшается в два раза, а ток через обмотки увеличивается в два раза при том же значении рассеиваемой мощности. Это даёт выигрыш в крутящем моменте при таком повышенном токе.

Последовательное включение обмоток приводит к быстрому снижению крутящего момента с ростом скорости вращения. Так происходит из-за того, что постоянная времени обмотки, равная отношению её индуктивности к сопротивлению, становится в два раза больше, чем при одиночном включении обмоток. При параллельном включении обмоток характеристики значительно лучше на высоких скоростях вращения по сравнению с последовательным включением, поскольку постоянная времени в этом случае в два раза меньше, чем при одиночном включении, и, соответственно, в четыре раза меньше, чем при последовательном включении. Также следует отметить, что кривая крутящего момента более плоская для параллельно включенных обмоток, обеспечивая повышенную мощность на валу.

8.3.3. Управление шаговым двигателем

При управлении шаговым двигателем при помощи мостовой схемы, приведённой ранее в этой главе, его характеристики ухудшаются. Для того, чтобы ток в обмотке достиг номинального значения, нужно конечное время, после того как ключ замкнётся (откроется транзистор). Поскольку обмотка обладает индуктивностью L и сопротивлением R , то ток будет возрастать экспоненциально во времени, скорость нарастания зависит от постоянной времени обмотки (τ), **Рис. 8.12**.

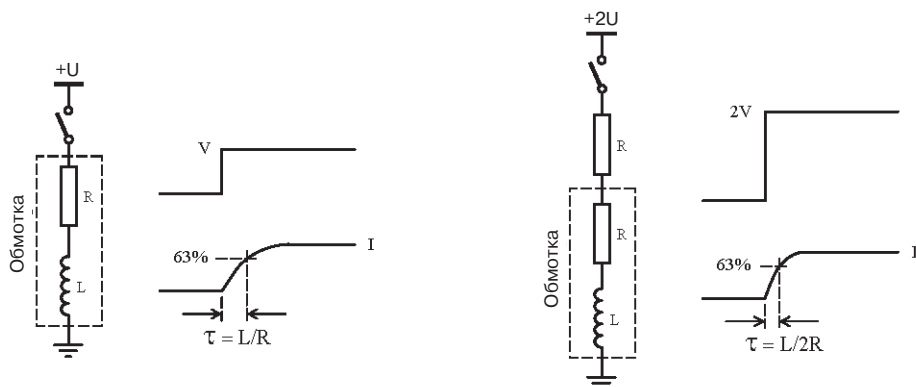


Рис. 8.12. Обычное включение и включение с последовательным сопротивлением

Одним из средств снижения постоянной времени является использование схемы с последовательным сопротивлением. В этой схеме последовательно с обмоткой включается резистор. Если предположить, что сопротивление резистора

равно сопротивлению обмотки, а напряжение питания удвоено, то номинальный ток I через обмотку останется неизменным, хотя постоянная времени обмотки уменьшится в два раза. Несмотря на увеличение крутящего момента на высоких скоростях, эта схема неэффективна из-за потерь мощности на дополнительном резисторе.

Более эффективным способом управления является *импульсное питание*. Как и схеме с последовательным сопротивлением увеличивают напряжение питания обмоток, но при этом не используют резистор. Это уменьшает время нарастания тока в обмотке. Без резистора ток через обмотку превысил бы предельно допустимый, если не принять мер по ограничению напряжения. Для предотвращения чрезмерного увеличения тока, последовательно с обмоткой включают токоизмерительный резистор с незначительным сопротивлением. Когда величина тока обмотки достигает номинального значения, то напряжение питания отключается, прекращая тем самым дальнейшее нарастание тока. Известно, что при отсутствии напряжения на обмотке ток в ней будет уменьшаться. По сигналу с токоизмерительного резистора напряжение питания снова подключается к обмотке, когда ток в ней уменьшится ниже номинального. Такие циклы ограничения тока продолжаютсся до тех пор, пока не будет снято питание с обмотки в следующей фазе работы двигателя, **Рис. 8.13**.

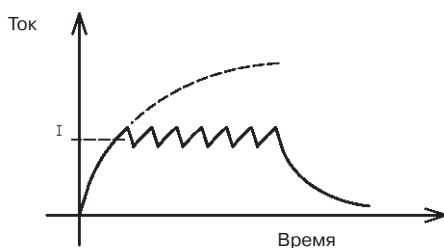


Рис. 8.13. Ток в обмотке при импульсном способе питания

Более точным способом управления является *микрошаговый метод*. В этом случае величина тока в каждой обмотке устанавливается индивидуально так, чтобы обеспечить промежуточные значения фазы между двумя полными фазами. При точном управлении током можно один оборот вала разбить на тысячи фаз.

Есть готовые контроллеры двигателей на микросхемах, которые упрощают задачу управления двигателями. Такие устройства состоят из схемы формирования сигналов управления, силовых ключей с защитными диодами, предотвращающими пробой транзисторов ЭДС самоиндукции, возникающей при размыкании цепи обмотки с током.

8.3.4. Характеристики шаговых двигателей

Шаговые двигатели характеризуются при помощи нескольких величин, таких как *момент выключенного двигателя*, *крутящий момент работающего двигателя*, *момент потери фазы*, *момент выпадения из синхронизма* и *фаза синхронной работы*. Эти термины кратко пояснены в **Табл. 8.6**.

Таблица 8.6. Параметры шаговых двигателей

Момент выключенного двигателя (holding torque)	Момент, препятствующий вращению, на валу остановленного двигателя.
Крутящий момент работающего двигателя (dynamic torque)	Момент, создаваемый валом работающего двигателя. Его величина уменьшается с ростом скорости вращения вала из-за влияния постоянной времени обмоток, мешающих нарастанию тока в обмотках до номинального значения.
Момент синхронной работы (pull-in torque)	Момент, создаваемый валом двигателя на нагрузке при запуске двигателя из промежуточного положения или при его остановке без потери фазы вращения. Инерционность собственно двигателя при этом не учитывается.
Момент выпадения из синхронизма (pull-out torque)	Максимальный момент, снимаемый с вала двигателя на установившейся скорости без потери фазы вращения. Этот момент превышает момент потери фазы, так как отсутствует ускорение вращения и, соответственно, момент не затрачивается на ускорение.
Скорость синхронной работы (ramped step rate)	Скорость, при которой не происходит потери фазы в периоды ускорения/замедления вращения.

Пока мы занимались изучением принципов работы коллекторных и шаговых двигателей. Теперь займёмся созданием объектно-ориентированных программ для управления этими двигателями с использованием схем на интерфейсной плате. В следующих разделах мы познакомимся с самой сильной стороной объектно-ориентированного программирования: виртуальными функциями.

Метод программирования, реализованный в последующих секциях, в первую очередь подразумевает создание иерархии классов, представляющих двигатели всех рассмотренных ранее типов. Затем эти классы этой иерархии будут использоваться при создании универсальной программы, способной управлять любым двигателем.

8.4. Иерархия классов для двигателей

Двигатели всех типов в иерархии классов будут управляться при помощи интерфейсной платы. Поэтому, хоть иерархия классов создаётся для двигателей, интерфейс также имеет большое значение. Можно выделить два типа двигателей из описанных в предыдущей главе. Это

1. Двигатели постоянного тока (коллекторные),
2. Шаговые двигатели.

Шаговые двигатели в свою очередь делятся на *униполярные* и *биполярные*. Этими двигателями можно управлять двумя способами: *полными фазами* и *половинными фазами*. Двигатели постоянного тока других подтипов не имеют.

Для начала представим двигатели «в общем виде», как абстрактные объекты. Двигатель будет оставаться абстрактным понятием, пока не будут описаны все признаки, характеризующие их особенности. Поэтому хорошей отправной точ-

кой иерархии будет создание абстрактного класса двигателя, включающего в себя наиболее общие признаки двигателей в иерархии. Все типы двигателей также будут характеризоваться типом интерфейса. В нашем случае все двигатели будут управляться через параллельный порт и поэтому можно использовать ранее разработанный класс `ParallelPort`. Поскольку интерфейс необходим всем типам двигателей в иерархии, то класс `ParallelPort` необходимо включить в иерархии на самых ранних этапах. Предлагаемая иерархия показана на **Рис. 8.14**.

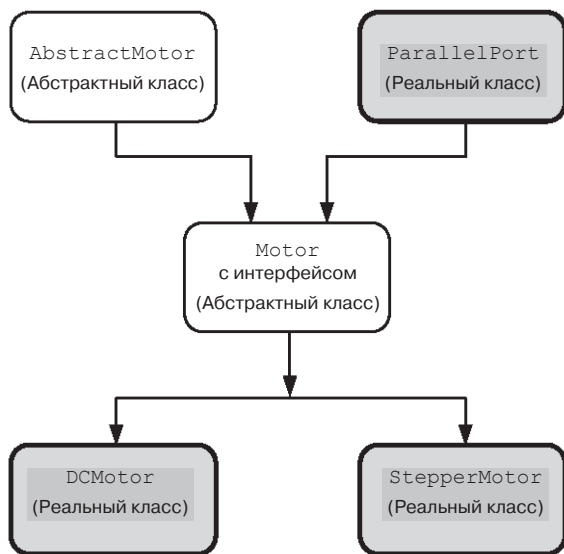


Рис. 8.14. Иерархия классов двигателя

Корнем иерархии является абстрактный класс `AbstractMotor`, представляющий собой двигатели всех типов. Класс `ParallelPort` находится на том же уровне, что и `AbstractMotor`. Но разработанный ранее класс не является абстрактным. Это обычный класс, от которого можно инстанцировать объекты.

Класс `Motor` образован способом множественного наследования от классов `AbstractMotor` и `ParallelPort`. Класс `Motor` тоже является абстрактным, поскольку он все ещё не полностью описывает объект иерархии. Это значит, что в классе опущены характерные признаки конкретных типов двигателей, без которых невозможно завершить определение функций-членов. Тем не менее, класс `Motor` обладает большими возможностями по сравнению с классом `AbstractMotor`, а именно: он обладает возможностью обмениваться информацией с устройствами по параллельному порту.

От класса `Motor` наследуются два класса `DCMotor` и `StepperMotor`. На этом уровне классы `DCMotor` и `StepperMotor` полностью описаны и должны быть реальными классами. На этом иерархия классов заканчивается.

Возникает вопрос: почему не образованы другие классы, например, представляющие двухфазные биполярные шаговые двигатели в полужазном режиме? Ответ будет следующим. Функциональные характеристики двухфазного биполярного шагового двигателя в полужазном режиме и двухфазного униполярного двигате-

ля в полнофазном режиме различны. Но с точки зрения программирования это аналогично двум автомобилям разного цвета. Конечно же, разные схемы управления двигателем не являются аналогами обычного автомобиля и престижного автомобиля. Когда нужны автомобили двух разных цветов, то цвет можно выбрать при помощи значения параметра. Для каждого нового цвета не нужно создавать отдельный класс. Такой же подход будет использован для различных типов шаговых двигателей.

8.5. Введение в виртуальные функции

Виртуальная функция это такая функция, сигнатура которой (объявление) одинакова на протяжении всей иерархии классов, но выполняемые действия меняются в соответствии её определению в каждом классе. Виртуальные функции полиморфны по своей природе, но с небольшим отличием. Как говорилось в главе 4, полиморфными являются функции с одинаковыми именами, количеством параметров, последовательностью параметров и типами параметров. Тела таких функций различаются, чтобы удовлетворять потребностям каждого класса. Виртуальные функции обладают всеми свойствами полиморфных функций, за исключением служебного слова `virtual` справа в начале объявления функции в классе. Хотя виртуальные и не виртуальные функции выглядят очень похоже, виртуальные функции обладают важными преимуществами. Они позволяют реализовать такое мощное средство, как *позднее связывание*, которое будет объяснено далее в этой главе.

Преимущества позднего связывания непосредственно связаны с виртуальными функциями в иерархии класса. Виртуальные функции сначала добавляются в базовый класс, а затем определяются в производных классах иерархии. Такой порядок обеспечивает прямую связь, необходимую между виртуальной функцией базового класса и виртуальной функцией производного класса где-либо ниже в иерархии классов. Любая виртуальная функция любого производного класса может быть вызвана при помощи указателя на базовый класс. Несмотря на использование указателя на базовый класс вместо указателя на собственно производный класс, во время выполнения будет выбрана соответствующая производному классу функция, благодаря механизму связывания виртуальных функций в иерархии классов.

Если для выбора соответствующей объекту функции виртуальные функции не используются, то разработчику нужно самому обеспечить механизм определения соответствия функции и класса. Этот дополнительный код должен включать в себя все необходимые действия, внутри типовых конструкций `if-then-else` или `switch`. Если иерархии классов большие и сложные, то это дополнительное программирование может оказаться чрезмерно трудной задачей и приводить к программам, которые трудно отлаживать и обслуживать. При добавлении к иерархии нового класса придётся вносить изменения во всю программу. Как можно себе представить, это не самый лучший способ программирования.

Виртуальные функции и позднее связывание избавляют разработчика программы от написания избыточного кода. Программист пишет программу в общем виде, используя виртуальные функции. Теперь этой задачей занимаются компилятор и линкер (редактор связей). В результате заметно уменьшается время про-

граммирования и отладки. Также требуются минимальные изменения кода при необходимости добавления чего-либо в иерархию классов. Эти преимущества станут очевидными по мере разработки.

Класс *AbstractMotor*

Как упоминалось ранее, в основе разработки класса *AbstractMotor* в качестве базового класса иерархии лежит необходимость образования цепочек виртуальных функций. Все классы, произведённые от класса *AbstractMotor* унаследуют все его функции. Не все функции могут быть полностью определены на этом этапе, тогда как остальные функции будут играть роль каркаса из-за недостающих деталей, относящихся к конкретному двигателю. Продуманный набор данных и функций базового класса будет выглядеть следующим образом:

1. Член данных для хранения установленной скорости двигателя;
2. Механизм чтения установленной скорости при помощи публичных функций;
3. Функция вращения «вперёд»;
4. Функция вращения «назад»;
5. Функция торможения двигателя;
6. Функция отключения двигателя (отключение напряжения).

Определение класса *AbstractMotor* приведено в **Листинге 8.1**.

Листинг 8.1. Класс *AbstractMotor*.

```
class AbstractMotor
{
    private:
        int Speed;

    public:
        AbstractMotor();
        void SetSpeed(int speed);
        int GetSpeed();
        virtual void Off() = 0;
        virtual void Forward() = 0;
        virtual void Reverse() = 0;
        virtual void Brake() = 0;
};
```

Нам знаком код, выделенный жирным шрифтом в **Листинге 8.1**. Он будет рассмотрен в последующих разделах.

В классе *AbstractMotor* есть собственный член данных *Speed* типа *int*. Функция *SetSpeed()* принимает *speed* в качестве параметра и присваивает члену *Speed* значение *speed*. Внешняя функция-член *GetSpeed()* может вызываться из любой функции программы и получать значение собственного члена данных *Speed*. Функция *Off()* снимает напряжение питания с двигателя. Функция *Forward()* заставляет вал двигателя вращаться в направлении «прямо» со скоростью, заданной в члене *Speed*. Похожим образом действует функция *Reverse()*,

приводящая к вращению вала в обратную сторону со скоростью `Speed`. Функция `Brake()` служит для короткого замыкания обмоток двигателя, что приводит к его быстрой остановке.

Теперь определим все функции-члены класса `AbstractMotor`. Класс `AbstractMotor` не управляет каким-либо конкретным типом двигателя. Вместо этого он содержит функции общего назначения, которые будут перекрыты в его производных классах, выполняя действия, соответствующие конкретному типу двигателя. Не зная особенностей двигателя, невозможно определить функции `Off()`, `Forward()`, `Reverse()` и `Brake()`. Если нет возможности определить функцию, то зачем тогда она входит в состав класса? Первая причина в том, что полезно определить состав объектов иерархии в самом её начале, чтобы строго контролировать единообразие всех объектов иерархии. Это помогает писать код, связанный со всеми объектами иерархии. Например, все классы иерархии должны обладать функцией `Forward()`. Производные от них классы должны переопределять унаследованную функцию `Forward()`, в которых будет отражаться их специфика. Вторая причина обусловлена *виртуальными функциями и поздним связыванием*.

Та часть программы, которая управляет двигателем, может быть очень сложной. В ней могут содержаться замысловатые операции переключения скорости, перемены направления вращения и торможения двигателя. При помощи виртуальных функций программу управления двигателем в функции `main()` можно написать полностью, не зная особенностей двигателей. Когда пользователь выберет какое-нибудь действие для выбранного двигателя, то автоматически будет задействована нужная функция этого класса для выполнения действий над объектом нужного типа (программисту нет необходимости писать код для этого). Использование виртуальных функций приводит к формированию компилятором такого кода, в котором реализован такой механизм вызова функций. В дальнейшем будет показано поведение таких функций. Эта часть программы будет работать даже над теми объектами, которые будут добавлены в иерархию в будущем. Становится возможным гибкое расширение иерархии классов без необходимости переписывать ту часть, которая управляет двигателем.

После запуска программы пользователь должен выбрать тип объекта (`DCMotor` или `StepperMotor`) из имеющихся типов двигателей. Например, пользователь может выбрать тип `DCMotor` и заставить его вал вращаться в направлении «прямо». В этом случае программа автоматически свяжет объект типа `DCMotor` с функцией `Forward()` класса `DCMotor`. Такое отложенное принятие решение о принадлежности функции называется *поздним связыванием*. Другими словами программа выбирает нужную функцию «прямого» вращения, руководствуясь типом объекта, выбираемого пользователем во время выполнения. Позднее связывание известно также как *динамическое связывание*. Динамическое потому, что связывание имеет место во время выполнения программы. Заметим, что полиморфные функции не обладают свойствами позднего связывания; это особенность виртуальных функций.

Если бы в программе не использовались виртуальные функции, то программисту пришлось бы писать дополнительный код, который выбирал бы нужную функцию. В этом случае каждая ветвь логики должна содержать код, связывающий указанный объект с соответствующей функцией. Связывание теперь не откладывается. В этом случае компилятор находит и связывает нужную функцию во время компилирования. Это называется *ранним связыванием* или *статическим связыванием*. Статическое связывание означает, что оно происходит до выполне-

ния программы: программа уже написана с учётом всех возможных комбинаций объект/функция, которые могут быть выбраны пользователем во время выполнения.

На первый взгляд виртуальные функции и позднее связывание может показаться слишком сложным. Тем не менее, без них очень сложно писать качественные программы общего назначения, которые должны работать в различных ситуациях, зависящих от действий пользователя во время выполнения. Применение виртуальных функций может значительно уменьшить объём программы. Программисту больше не нужно писать те участки кода, в которых происходит выполнение действий над каким-либо объектом. Вместо этого пишется один блок кода, который выполняет операции со всеми типами объектов в иерархии. Таким образом, пользователь (или сама программа) имеет возможность выбирать тип объекта во время выполнения.

8.5.1. Чистая виртуальная функция

Ранее говорилось, что не зная конструкции и способа подключения двигателя, невозможно определить тела функций `Off()`, `Forward()`, `Reverse()` и `Brake()`. Для того, чтобы проинформировать компилятор о невозможности определить тело виртуальной функции, такие функции определяются как «чистые» функции. Заметим, что только виртуальные функции могут объявляться чистыми. Чистая функция объявляется путём добавления `=0` в конце объявления такой функции.

Обычная функция объявляется так:

```
void Forward();
```

Объявление виртуальной функции:

```
virtual void Forward();
```

А это объявление чистой виртуальной функции:

```
virtual void Forward() = 0;
```

Чистые виртуальные функции и абстрактные классы

Если в классе объявить чистую виртуальную функцию (у которой нет исполняемого кода), то этот класс становится абстрактным классом. В некоторых абстрактных классах есть полезные функции. Если в классе присутствует функция, тело которой не определено, то от такого класса нельзя инстанцировать объекты. Причина в том, что программа может попытаться вызвать чистую виртуальную функцию, у которой нет кода. Поэтому существует правило, запрещающее инстанцирование объектов от абстрактного класса. Мы ещё вернёмся к виртуальным функциям, когда будем использовать их для управления коллекторными и шаговыми двигателями в иерархии классов, **Рис. 8.14**.

Возвращаясь к определению класса в **Листинге 8.1**, можно определить только три функции-члена: `AbstractMotor()`, `SetSpeed()` и `GetSpeed()`. Как объяснялось ранее, остальные четыре функции: `Off()`, `Forward()`, `Reverse()` и `Brake()` невозможно пока определить, поэтому они объявлены чистыми виртуальными функциями. В **Листинге 8.2** приведено определение функций-членов.

Листинг 8.2. Определение функций-членов класса `AbstractMotor`

```

AbstractMotor::AbstractMotor()
{
    Speed = 0;
}

void AbstractMotor::SetSpeed(int speed)
{
    Speed = speed;
    if (Speed > 255) Speed = 255; // Ограничение максимального значения
    if (Speed < 0) Speed = 0; // Ограничение минимального значения
}

int AbstractMotor::GetSpeed()
{
    return Speed;
}

```

Конструктор класса `AbstractMotor` инициализирует член данных `Speed` значением 0.

Внешняя функция `SetSpeed()` может вызываться из любой функции и позволяет изменить скорость любого двигателя из иерархии. Функция выполняет присвоение значения фактического аргумента `speed` члену данных `Speed`. Два выражения `if` в функции `SetSpeed()` гарантируют, что значение, передаваемое через `speed`, будет находиться в допустимом диапазоне значений от 0 до 255. Например, если будет передано значение 300, то первое выражение `if` определит выход за диапазон и присвоит значение 255. Поэтому член данных `Speed` будет иметь значение 255. Другими словами, любое значение, превышающее 255 будет заменено на 255. Второе выражение `if` будет заменять на 0 любые отрицательные значения. Значения, входящие в разрешённый диапазон будут оставаться неизменными.

Функция `GetSpeed()` возвращает текущее значение `Speed` и тоже может вызываться из любой функции. Это единственный способ получить значение `Speed` из внешних по отношению к классу функций.

Класс *Motor*

Затем в иерархии следует класс `Motor`. Этот класс наследует функции и данные от двух классов `AbstractMotor` и `ParallelPort`. Определение этого класса приведено в **Листинге 8.3**.

Листинг 8.3. Класс `Motor`.

```

class Motor : public AbstractMotor, public ParallelPort
{
    public:
        Motor(int baseaddress = 0x378);
        void Off();
        virtual void Forward() = 0;
        virtual void Reverse() = 0;
        virtual void Brake() = 0;
};

```

Обратите внимание на строку:

```
class Motor : public AbstractMotor, public ParallelPort
```

Новый класс называется `Motor` и образован при помощи спецификатора доступа `public` от двух базовых классов `AbstractMotor` и `ParallelPort`. Базовые классы разделяются запятыми. Новые члены данных не добавлялись. Должны иметь определения конструктор класса `Motor` и функция `Off()`. Остальные три функции остаются чистыми виртуальными функциями. Выше упоминалось, что пока в классе есть хотя бы одна чистая виртуальная функция, то такой класс является абстрактным. Поэтому класс `Motor` является абстрактным и от него нельзя инстанцировать объекты. Разница между классами `AbstractMotor` и `Motor` в том, что последний обладает возможностью обмениваться данными по параллельному порту с внешними устройствами. Отсюда следует, что все производные от него объекты могут обмениваться данными по параллельному порту.

В **Листинге 8.4** приведены определения конструктора класса `Motor` и функции `Off()`.

Листинг 8.4. Определения функций класса `Motor`

```
Motor::Motor(int baseaddress) : ParallelPort(baseaddress)
{
    Off();
}

void Motor::Off()
{
    WritePort0(0x00);
}
```

Все двигатели управляются либо одним, либо обоими мостами, находящимися на интерфейсной плате. Каждый из мостов образован четырьмя «ключами», каждым из которых можно управлять логическим (цифровым) сигналом. Отсюда следует, что для управления обоими мостами нужно восемь сигнальных линий и поэтому в программах для этого будет использоваться регистр `BASE`.

Двигатель можно выключить, разомкнув ключи. Для коллекторных двигателей нужно разомкнуть четыре ключа, а в случае биполярных шаговых двигателей замыкают все восемь ключей в двух мостах. В общем случае, можно выключить любой тип двигателя, разомкнув все ключи или, другими словами, если в регистр `BASE` записать ноль во все биты. Так и происходит в функции `Off()`. Это не виртуальная функция и в производных классах её переопределять не нужно.

Обратите внимание, что в конструкторе вызывается функция `Off()`. Таким образом снижается риск короткого замыкания питания на землю, путём закрытия всех транзисторов в мостах. При этом подразумевается, что пользователь запустит программу *до* подачи питания на интерфейсную плату и двигатель.

Значения по умолчанию для параметров функции

Конструктор класса `Motor`, в отличие от ранее созданных конструкторов, написан более совершенным способом. В предыдущих главах создавались два конструктора, в одном случае конструктор не имел параметров, а в другом имел один параметр

`baseaddress` типа `int`. Параметр конструкторе `Motor(int baseaddress=0x378)` передаётся в конструктор базового класса `ParallelPort(int baseaddress)` и при этом происходит его инициализация значением `0x378`, если при создании объекта класса `Motor` этот параметр опускается. Таким образом, такой конструктор может выполнять функции обоих конструкторов из предыдущих глав.

Значение по умолчанию параметра `baseaddress` выделено жирным шрифтом в **Листинге 8.3**. Выражение «по умолчанию» говорит о том, что если при объявлении экземпляра объекта не будет указано значение параметра, то ему будет присвоено значение `0x378`. Обратите внимание на то, что значение по умолчанию параметра конструктора указывается в круглых скобках в определении класса, **Листинг 8.3**. Это первый способ указания значений по умолчанию. Другой способ подразумевает указание этого значения в определении функции, **Рис. 8.15**. В этом случае значение по умолчанию в определении класса уже не указывается, как в **Листинге 8.5**. В двух местах одновременно указывать значение по умолчанию нельзя!

```
Motor::Motor(int baseaddress=0x378):AbstractMotor(),
    ParallelPort(baseaddress)
{
    Off();
}
```

Значение по-умолчанию
параметра **baseaddress**

Рис. 8.15. Альтернативный способ указания значения по умолчанию

Листинг 8.5. Определения функций класса `Motor`

```
class Motor : public AbstractMotor, public ParallelPort
{
public:
    Motor(int baseaddress);
    void Off();
    virtual void Forward(int speed) = 0;
    virtual void Reverse(int speed) = 0;
    virtual void Brake() = 0;
};
```

Конструктор класса `Motor` можно вызвать двумя способами, **Рис. 8.16** и **Рис. 8.17**:

```
Motor();
```

Аргумент не передаётся, поэтому используется значение по умолчанию. Вследствие этого унаследованный член данных **BaseAddress** будет иметь значение `0x378`

Рис. 8.16. Вызов конструктора класса `Motor` без аргументов

```
Motor(0x3BC);
```

Аргумент передан. Вследствие этого унаследованный член данных **BaseAddress** будет иметь значение `0x3BC`

Рис. 8.17. Вызов конструктора класса `Motor` с аргументом

Так как функция `Motor()` является конструктором, то её можно вызывать так, как показано на **Рис. 8.18**.

`Motor;`

Конструктор можно вызывать без круглых скобок. Результат будет таким же, как на **Рис. 8.16**

Рис. 8.18. Вызов конструктора класса `Motor` с аргументом по умолчанию

На вышеприведённых примерах показано, как задавать значение по умолчанию для одного параметра. Если у функции несколько параметров, то они тоже могут иметь значения по умолчанию с учётом следующего правила: справа от параметра, который должен иметь значение по умолчанию, нужно тоже указать значения по умолчанию для остальных параметров. Правильный и неправильный варианты объявления функций приведены на **Рис. 8.19**.

```
void AnyFunction(int x, int y=0, int z=1);    // правильно
```

Это правильный вариант. Для параметра **y** можно указать значение по умолчанию, поскольку параметр **z** имеет значение по умолчанию

```
void AnyFunction(int x, int y=0, int z);    // неправильно
```

Это неправильный вариант. У параметра **y** не может быть значения по умолчанию из-за отсутствия значения по умолчанию у параметра **z**

Рис. 8.19. Несколько параметров со значением по умолчанию

Класс `DCMotor`

Класс `DCMotor` находится в конце иерархии классов, изображённой на **Рис. 8.14**. Этот класс должен представлять коллекторные двигатели. Если так, то это уже не абстрактный класс. Определение класса `DCMotor` приведено в **Листинге 8.6**.

Листинг 8.6. Класс `DCMotor`

```
class DCMotor : public Motor
{
    public:
        DCMotor(int baseaddress = 0x378);
        virtual void Forward();
        virtual void Reverse();
        virtual void Brake();
};
```

Класс `DCMotor` произведён от класса `Motor`. Определение класса `DCMotor` хоть и выглядит небольшим, оно наследует все члены данных и функции-члены от классов `Motor`, `AbstractMotor` и `ParallelPort`. В этом классе нет чистых виртуальных функций. Поэтому этот класс не абстрактный, а обычный класс, но чтобы от

него можно было инстанцировать объекты, необходимо определить его функции-члены.

Для определения функций-членов класса нужно знать как управлять коллекторным двигателем и подключить его к интерфейсной плате. На **Рис. 8.20** показана схема включения коллекторного двигателя по мостовой схеме. Управлять ключами *S1–S4* можно сигналами битов **D0–D3** регистра **BASE**, выполнив соединения согласно **Табл. 8.7**.

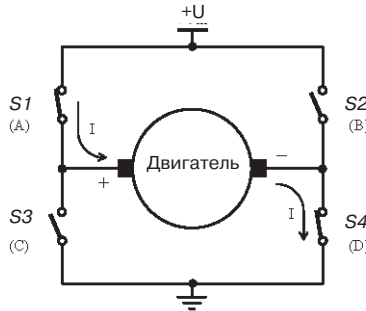


Рис. 8.20. Включение двигателя по мостовой схеме

Таблица 8.7. Подключение моста с коллекторным двигателем к параллельному порту

Ключ	Маркировка на интерфейсной плате	Бит регистра BASE
S1	A	D0
S2	B	D1
S3	C	D2
S4	D	D3

Ключ замыкается когда бит равен 1 и размыкается при нулевом значении бита. Вращение в направлении «прямо» включается записью в биты **D0** и **D3** единицы и нуля в биты **D1** и **D2**. Если инвертировать сигналы управления, то вращение будет происходить в направлении «назад», то есть биты **D1** и **D2** равны единице, а биты **D0** и **D3** нулю.

Торможение двигателя осуществляется замыканием его выводов. Для этого можно записать 1 в биты **D0** и **D1**, а в биты **D2** и **D3** записать 0, или записать 1 в биты **D2** и **D3**, а в биты **D0** и **D1** записать 0. Выберем в данном случае замыкание выводов на землю. Для этого в биты **D2** и **D3** запишем 1, а биты **D0** и **D1** обнулим. В **Табл. 8.8** сведены значения битов **D0–D3**, необходимые для вращения «прямо», «назад» и торможения.

Внимание!

НИКОГДА не допускайте одновременного включения ключей *S1* и *S3*. НИКОГДА не допускайте одновременного включения ключей *S2* и *S4*. Если это случится, то выйдут из строя транзисторы, и, возможно, блок питания.

Таблица 8.8. Значения регистра BASE для управления двигателем

Действие	Биты данных								Значение в регистре BASE
	D7	D6	D5	D4	D3	D2	D1	D0	
Вращение «прямо»	x	x	x	x	1	0	0	1	0000 1001 = 0x09
Вращение «назад»	x	x	x	x	0	1	1	0	0000 0110 = 0x06
Торможение	x	x	x	x	1	1	0	0	0000 1100 = 0x0C
Примечание: значения битов, обозначенные знаком x могут принимать оба значения 0 и 1. Они не подключены к мосту с двигателем и поэтому не играют никакой роли. Это касается битов D4...D7.									

Коллекторный двигатель подключается к интерфейсной плате согласно **Табл. 8.9** и **Рис. 8.21**. Интерфейсная плата рассчитана на подключение двигателей с напряжением питания +12 В и потребляемым током не более 1 А. В то же время допускается питание двигателей от внешнего источника питания (с напряжением не более 30 В), как это сделать описано в следующем примечании. **Не включайте** питание интерфейсной платы, пока не будет запущена программа; до запуска программы сигналы порта могут находиться в состоянии 1.

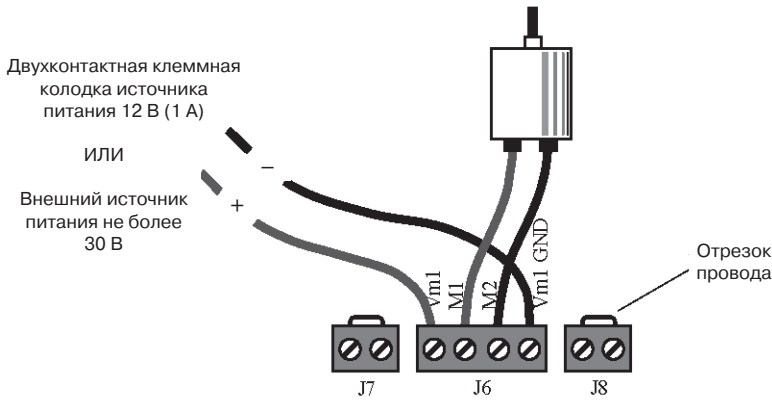


Рис. 8.21. Подключение источника питания к коллекторному двигателю

Таблица 8.9. Подключение коллекторного двигателя к мосту

Мост 1	Подключение
Vm1	Положительный полюс источника питания двигателя (+)
M1	Вывод двигателя +
M2	Вывод двигателя –
Vm1 GND	Отрицательный полюс источника питания двигателя (–)

Примечание

Две клеммы на 4-контактной клеммной колодке, обозначенные Vm1 и Vm1 GND служат для подключения питания к мосту (Vm2 и Vm2 GND для другого моста).

Источник питания интерфейсной платы может питать двигатели напряжением 12 В током до 1 А. Подключение к этому источнику осуществляется посредством соединения отрезком изолированного провода клеммы Vm1 (или, если нужно, Vm2) с источником +12 В на 2-контактной клеммной колодке (к ней подключается блок питания). Другим изолированным проводом соединяются земля моста Vm1 GND (или Vm2 GND) и земля источника питания. Самое главное, нужно соединить проводником клеммы в двух 2-контактных клеммных колодках. Эти клеммные колодки служат для подключения резисторов нужного сопротивления в случае использования схем с резистором.

При использовании внешнего источника питания нужно его положительный вывод подключить к клемме Vm1 (или Vm2, если нужно) 4-контактной клеммной колодки. Отрицательный вывод внешнего источника питания подключить к клемме Vm1 GND (или Vm2 GND). НЕЛЬЗЯ подключать внешний источник питания через 2-контактную клеммную колодку питания платы.

Определившись с подключением можно приступить реализации способов управления им, в частности крутящим моментом и скоростью. Наиболее подходящим средством для этого является широтно-импульсная модуляция (ШИМ).

8.5.2. Формирование сигнала с ШИМ

Из материала раздела 8.2.2 следует, что при использовании широтно-импульсной модуляции импульсам с большей шириной будут соответствовать большие скорость/момент. Другими словами, чем больше *скважность* импульсов, тем больше скорость. В нашем случае скважность является отношением длительности импульса напряжения на двигателе к длительности периода повторения импульсов. Это отношение обычно выражают в процентах. Чтобы управлять скоростью, нужно изменять скважность. Значит, программа должна обеспечивать возможность устанавливать нужную скважность. Обычно скважность задается точно. В этой главе скважность не будет выдерживаться точно. Скважность будет формироваться при помощи программных циклов. Компьютеры разной производительности будут генерировать импульсы с различными периодами следования, но с одинаковой скважностью.

Сигнал ШИМ можно сформировать следующим образом. Предположим, что один период состоит из 256 последовательных записей в регистр. Тогда период будет равен тому времени, которое потребуется компьютеру, чтобы выполнить 256 операций записи в регистр. В зависимости от нужной скважности, часть из этих 256 операций записи можно использовать для записи в регистр значения 0x09 (**Табл. 8.8**), заставляя вал двигателя вращаться в направлении «прямо». Оставшимися операциями можно записывать в регистр значение 0x00, отключая питание двигателя. Например, если нужна скважность 50%, то первые 128 опера-

ций записи должны выводить значение 0x09, а остальные 128 операций значение 0x00. Таким образом, происходит управление скоростью вращения. Значения от 0 до 255 (образующие 256 различных скоростей вращения) будут устанавливать скорость вращения. ШИМ сигнал показан на **Рис. 8.22**.

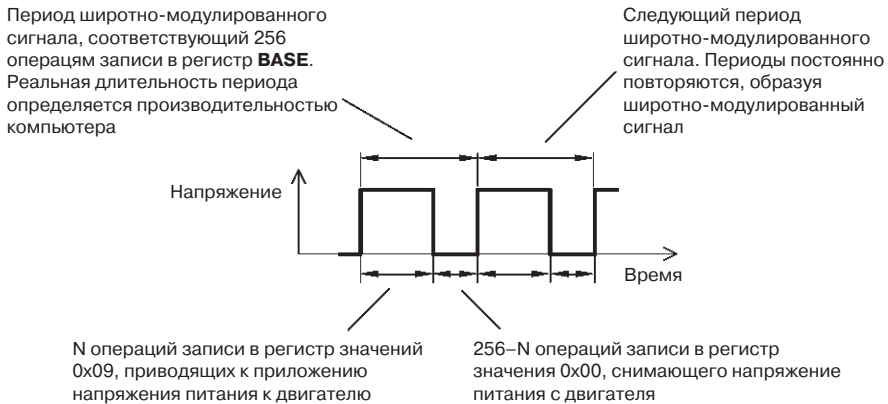


Рис. 8.22. Широтно-модулированный сигнал

Управление скоростью вращения в обратном направлении будет осуществляться похожим образом, за исключением того, что для подачи напряжения на двигатель будет использоваться значение 0x06 вместо 0x09. Правильно выражаясь, мы реализовали систему управления скоростью без *обратной связи*. В данном случае не происходит измерения скорости вращения и не производится коррекция отклонений скорости от заданной. Другими словами, нет слежения за скоростью и её поправки.

В **Листинге 8.7** приведено определение класса `DCMotor`. При сопоставлении с **Листингом 8.6** видно, что для параметра `baseaddress` указано значение по умолчанию. Это не является очевидным в определении функций в **Листинге 8.7**.

Листинг 8.7. Определение функций-членов класса `DCMotor`

```
DCMotor::DCMotor(int baseaddress) : Motor(baseaddress)
{
}

void DCMotor::Forward()
{
    int j;
    for(j = 0; j < GetSpeed(); j++)
        WritePort0(0x09);
    for( ; j < 256; j++)
        WritePort0(0x00);
}

void DCMotor::Reverse()
{
    int j;
    for(j = 0; j < GetSpeed(); j++)
```

```
        WritePort0(0x06);  
    for( ; j < 256; j++)  
        WritePort0(0x00);  
}  
  
void DCMotor::Brake()  
{  
    WritePort0(0x0C);  
}
```

Функции `Forward()` и `Reverse()` очень похожи. Они различаются только значением, записываемым в регистр в первом цикле `for`, поэтому ограничимся рассмотрением только одной функции `Forward()`.

В функции `Forward()` используются функции, выделенные жирным шрифтом ниже:

```
void DCMotor::Forward()  
{  
    int j;  
    for(j = 0; j < GetSpeed(); j++)  
        WritePort0(0x09);  
    for( ; j < 256; j++)  
        WritePort0(0x00);  
}
```

Все выделенные функции являются членами классов иерархии. Функция `GetSpeed()` унаследована от абстрактного класса `AbstractMotor`, а функция `WritePort0()` унаследована от класса `ParallelPort`.

Ранее говорилось, что скорость вращения определяется скважностью ШИМ сигнала. Скважность задаётся количеством операций записи в регистр `BASE` значений, приводящих к подаче напряжения питания на двигатель. Период ШИМ сигнала соответствует времени выполнения 256 операций записи в регистр `BASE`. В диапазоне от 0 до 255 содержится 256 различных значений, поэтому число 255 соответствует скважности 100% (максимальная скорость), а 0 соответствует скважности 0% (нет вращения).

Когда выполнение программы доходит до первого цикла `for`, то в унаследованном члене данных должно находиться допустимое значение (из диапазона от 0 до 255). В первом цикле счетчик цикла инициализируется значением 0. Затем в теле цикла вызывается функция `WritePort0()`, записывающая 0x09 в регистр `BASE`, тем самым прикладывая напряжение питания к двигателю. Потом происходит инкремент счётчика цикла выражением `j++`. Далее происходит проверка условия цикла `j < GetSpeed()`. Функция `Forward()` класса `DCMotor` не имеет *прямого* доступа к унаследованному члену данных `Speed`, так как это собственный член класса `AbstractMotor`. Вследствие этого значение `Speed` извлекается при помощи функции `GetSpeed()` класса `AbstractMotor`. Если условие цикла истинно, то тело цикла выполняется ещё раз. Например, если `Speed` равна 5, то функция `WritePort0()` будет выполняться при значениях `j`, равных 0, 1, 2, 3 и 4. Когда `j` становится равным 5, то условие цикла получается ложным и выполнение цикла `for` прекращается и начинает выполняться второй цикл `for`.

Обратите внимание на необычность второго цикла `for`, у него отсутствует выражение инициализации. Значение `j` «достаётся по наследству» от первого цикла

for, чтобы второй цикл мог выполнить оставшиеся итерации, и поэтому инициализация `j` в данном случае не нужна.

Воспользовавшись данными предыдущего примера, увидим, что на входе во второй цикл `for j` будет равно 5. Второй цикл `for` будет выполняться, пока `j` не превысит 255. При выполнении тела цикла всякий раз в регистр `BASE` выводится значение `0x00`, снимая напряжение с двигателя. Как только `j` достигнет значения 256, цикл `for` завершится, закончив полный период ШИМ. Будет выполнен цикл с 5 включающими напряжение питания записями в регистр из 256 записей, то есть скважность равна 2%.

Периоды ШИМ должны следовать один за другим, чтобы получался непрерывный ШИМ сигнал. Поэтому функцию `Forward()` нужно вызывать повторно до тех пор, пока пользователь не прервёт действие команды «вращение прямо». Это может быть реализовано, например, при помощи цикла `while` совместно с условием `!kbhit()`. Цикл `while` будет продолжаться до тех пор, пока не будет нажата кнопка на клавиатуре. Функция `kbhit()` вернёт ненулевое значение при нажатии на кнопку клавиатуры. Если перед функцией поставить оператор отрицания (`!`), то `!kbhit()` будет возвращать ненулевое значение, пока кнопка **не** нажата. Тогда цикл `while` будет выполняться до нажатия на кнопку. Выполнение одной итерации цикла `while` будет генерировать один период ШИМ сигнала, поэтому цикл `while` будет формировать ШИМ сигнал, пока не будет нажата кнопка. Цикл `while` можно реализовать вне функций-членов, например в функции `main()`.

Принцип работы функции `Reverse()` такой же, как и у функции `Forward()`. Различие только в том, что в первом цикле `for` на мост передаётся другое значение, переключая его таким образом, что полярность напряжения на выводах двигателя меняется на противоположную, и вращение происходит в направлении «назад». Широтно-модулированный сигнал формируется так же, как и в случае функции `Forward()`.

Единственный вызов функции `WritePort0()` в функции `Brake()` служит для замыкания выводов двигателя и его остановки.

В разделе 8.6 будет показано применение класса этого класса для управления коллекторным двигателем.

Класс *StepperMotor*

Класс `StepperMotor` должен удовлетворять требованиям различных типов шаговых двигателей и двум режимам их работы. В **Табл. 8.10** перечислены эти режимы и их аббревиатуры. Определение класса `StepperMotor` приведено в **Листинге 8.8**.

Таблица 8.10. Аббревиатуры типов шаговых двигателей

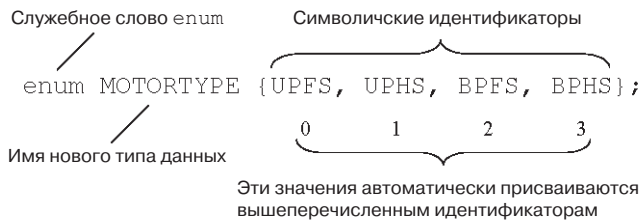
Тип шагового двигателя	Режим	Аббревиатура
Униполярный (UniPolar)	Режим полных фаз (Full-Step)	UPFS
Униполярный (UniPolar)	Режим полуфаз (Half-Step)	UPHS
Биполярный (BiPolar)	Режим полных фаз (Full-Step)	BPFS
Биполярный (BiPolar)	Режим полуфаз (Half-Step)	BPHS

Листинг 8.8. Новый тип данных `MOTORTYPE` и класс `StepperMotor`

```
enum MOTORTYPE {UPFS, UPHS, BPFS, BPHS};
class StepperMotor : public Motor
{
    private:
        MOTORTYPE MotorType;
        unsigned char Switching[8];
        int CycleIndex;
        int MaxIndex;

    public:
        StepperMotor(MOTORTYPE motortype = UPFS,
                     int baseaddress = 0x378);
        virtual void Forward();
        virtual void Reverse();
        virtual void Brake();
};
```

В начале этого листинга создан пользовательский тип данных `MOTORTYPE`. Такой тип данных является перечислимым типом, так как все его возможные значения перечисляются при его определении. Значения всегда типа `int` и называются константами перечисления. Идентификаторы констант должны быть символическими, чтобы был ясен их смысл и легче читался текст программы.

**Рис. 8.23.** Перечислимые типы данных

В объявлении на **Рис. 8.23** имя нового типа данных `MOTORTYPE`. Идентификаторам `UPFS`, `UPHS`, `BPFS` и `BPHS` автоматически присваиваются константы 0, 1, 2 и 3, соответственно. Переменную этого типа можно объявить так:

```
MOTORTYPE NewMotor;
```

Если вместо компилятора C++ используется компилятор C, то это объявление будет выглядеть следующим образом:

```
enum MOTORTYPE NewMotor;
```

Переменная `NewMotor` может иметь любое из перечисленных значений, то есть те значения, которые были перечислены между фигурными скобками. Если бы мы указали значение `UPFS`, например равным 2, то `UPHS`, `BPFS` и `BPHS` будут иметь значения 3, 4 и 5, соответственно.

```
Enum MOTORTYPE {UPFS = 2, UPHS, BPFS, BPHS};
```


Так как перечисляемые константы типа `int`, то с ними производить арифметические операции. Например:

```
MOTORTYPE NewBreed;
NewBreed = UPFS + BFFS; //Хоть и лишено всякого смысла, но допустимо!
```

Это допустимо, но в контексте нашей задачи лишено смысла. Но всё же в некоторых случаях могут оказаться полезными арифметические операции с перечислимыми типами; их нужно выполнять внимательно.

Согласно **Табл. 8.10**, перечисляемый тип данных из **Листинга 8.8** может представлять любой тип шагового двигателя, управление которым будет осуществляться посредством интерфейсной платы. Теперь вернёмся к рассмотрению определения класса `StepperMotor` в **Листинге 8.8**. Ниже приведены описания его четырёх собственных членов данных.

MOTORTYPE MotorType

Первый член данных `MotorType`. После инициализации этот член хранит тип шагового двигателя, находящегося под управлением.

unsigned char Switching[8]

Так объявляется массив `Switching`, состоящий из восьми элементов типа `unsigned char`. В программе он будет заполняться масками коммутации, в зависимости от выбранного пользователем двигателя. Значения масок коммутации станут известны далее, когда будут определены функции класса.

int CycleIndex

Член данных `CycleIndex` служит индексом для последовательного обращения к байтовым элементам массива `Switching`, обеспечивая правильную последовательность фаз вращения шагового двигателя. Для вращения вала шагового двигателя в направлении «прямо» в режиме полных фаз, последовательность значений `CycleIndex` будет 0, 1, 2, 3, 0 и так далее. Вращение в обратную сторону будет происходить при следующем порядке следования значений `CycleIndex`: 0, 3, 2, 1, 0 и так далее. Аналогично, в полуфазном режиме вращение «прямо» будет обеспечиваться последовательностью 0, 1, 2, 3, 4, 5, 6, 7, 0 и так далее. Вращение в направлении «назад» обеспечивается последовательностью значений `CycleIndex` 0, 7, 6, 5, 4, 3, 2, 1, 0 и так далее.

int MaxIndex

Этот член определяет индекс массива `Switching`, по достижении которого нужно переходить обратно на начало массива. Можно сделать вывод, что `MaxIndex` определяет максимальное значение `CycleIndex`. В режиме полных фаз нужны только четыре элемента массива `Switching`, `MaxIndex` равен 4, в то время как в полуфазном режиме используются все восемь, `MaxIndex` равен 8.

Определения функций-членов класса `StepperMotor` приведены в **Листинге 8.9**.

Листинг 8.9. Функции-члены класса StepperMotor

```
StepperMotor::StepperMotor(MOTORTYPE motortype,
    int baseaddress) : Motor(baseaddress)
{
```

```
MotorType = motortype;
CycleIndex = 0;

switch(MotorType)
{
    case UPFS : MaxIndex = 4;
        Switching[0] = 0x11;
        Switching[1] = 0x12;
        Switching[2] = 0x22;
        Switching[3] = 0x21;
        break;
    case UPHS : MaxIndex = 8;
        Switching[0] = 0x01;
        Switching[1] = 0x11;
        Switching[2] = 0x10;
        Switching[3] = 0x12;
        Switching[4] = 0x02;
        Switching[5] = 0x22;
        Switching[6] = 0x20;
        Switching[7] = 0x21;
        break;
    case BPFS : MaxIndex = 4;
        Switching[0] = 0x99;
        Switching[1] = 0x69;
        Switching[2] = 0x66;
        Switching[3] = 0x96;
        break;
    case BPHS : MaxIndex = 8;
        Switching[0] = 0x99;
        Switching[1] = 0x09;
        Switching[2] = 0x69;
        Switching[3] = 0x60;
        Switching[4] = 0x66;
        Switching[5] = 0x06;
        Switching[6] = 0x96;
        Switching[7] = 0x90;
    }
}

void StepperMotor::Forward()
{
    if(++CycleIndex == MaxIndex) CycleIndex = 0;
    WritePort0(Switching[CycleIndex]);
    delay(259-GetSpeed());
}

void StepperMotor::Reverse()
{
    if(--CycleIndex == -1) CycleIndex = MaxIndex - 1;
```

```

    WritePort0(Switching[CycleIndex]);
    delay(259-GetSpeed());
}

void StepperMotor::Brake()
{
    switch(MotorType)
    {
        case UPFS : case UPHS :
            WritePort0(0x11);
            break;
        case BPFS : case BPHS :
            WritePort0(0x99);
    }
}

```

Как обычно, в конструкторе происходит инициализация собственных членов данных класса. Если передано значение параметра `motortype`, то это значение присваивается собственному члену `MotorType`. Начальное значение `CycleIndex` всегда равно нулю. Член `MaxIndex` принимает значения 4 или 8, в зависимости от режима работы двигателя, полнофазного или полуфазного. Массив `Switching` заполняется значениями, определяемыми типом двигателя и режимом его работы, как описано ниже.

В выражении `switch` по значению `MotorType` выбирается один из вариантов заполнения массива `Switching`, соответствующих типу двигателя и режиму работы. Значения элементов массива `Switching` служат для организации переключения состояний моста в нужной последовательности, обеспечивая тем самым правильную последовательность фаз вращения двигателя. В каждой фазе задействованы оба моста и к соответствующим обмоткам двигателя приложено практически полное напряжение источника питания. Скорость/положение вала двигателя определяется длительностью/номером фазы. Вследствие этого для управления скоростью или крутящим моментом ШИМ не применяется.

Функции `Forward()` и `Reverse()` класса `StepperMotor` имеют один и тот же принцип работы, поэтому рассмотрим только функцию `Forward()`, **Рис. 8.24.**

```

void StepperMotor::Forward()
{
    if(++CycleIndex == MaxIndex)
        CycleIndex = 0;
    WritePort0(Switching[CycleIndex]);
    delay(257-GetSpeed());
}

```

CycleIndex инкрементируется и сравнивается с предельным значением. Если достигнуто предельное значение, **CycleIndex** сбрасывается в исходное состояние

Скорость задаётся путём введения изменяемой задержки между операциями записи в регистр

В порт записывается элемент массива **Switching**, по одному элементу на каждую фазу

Рис. 8.24. Принцип работы функции `Forward()`

В функции `Brake()` при помощи выражения `switch` выбирается нужное действие для торможения двигателя того или иного типа, биполярного или униполярного. Торможение двигателей четырёх различных типов выполняется двумя разными способами замыканием и размыканием нужных ключей моста. Отметим, что в униполярных двигателях нельзя выполнить торможение замыканием обмоток, вместо этого с обмоток снимается напряжение.

Подключение биполярных и униполярных двигателей к мосту показано соответственно на **Рис. 8.25** и **Рис.8.26**. Теперь определены все классы и их функции-члены. Создание иерархии классов завершено. Теперь осталось написать функцию `main()`, где можно будет воспользоваться этими классами.

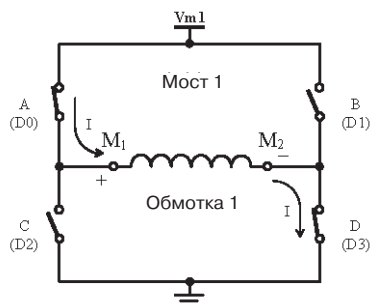


Рис. 8.25. Подключение биполярного двигателя к мосту

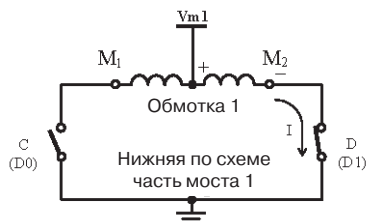
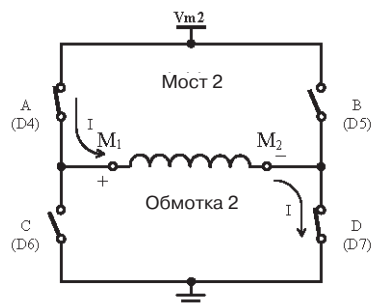
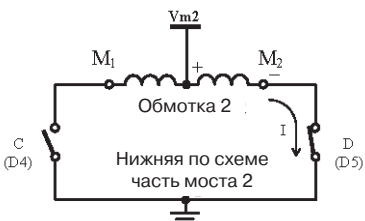


Рис. 8.26. Подключение униполярного двигателя к мосту



8.6. Виртуальные функции в приложении

Теперь уже можно приступить к написанию программы, где будут использоваться виртуальные функции. Программа будет способна управлять любым типом двигателя из поддерживаемых созданной иерархией классов, а именно:

1. Коллекторные двигатели (двигатели постоянного тока).
2. Униполярные двигатели в режиме полных фаз.
3. Униполярные шаговые двигатели полуфазном режиме.
4. Биполярные шаговые двигатели в режиме полных фаз.
5. Биполярные шаговые двигатели в режиме полуфаз.

Сначала мы разработаем ту часть программы, где происходит управление двигателем с использованием виртуальных функций. Затем добавим код, позволяющий пользователю выбрать тип двигателя.

Главное преимущество виртуальных функций в том, что происходит автоматическое связывание объекта и соответствующей функции во время выполнения программы. Можно писать программу в «общем виде». Начнём с того, что определим переменную, которая будет представлять любой объект иерархии. Для этого лучше всего подходит переменная типа `Motor`, так как этот класс является базовым для всех классов двигателей в иерархии. Указатель (то есть эта переменная) может указывать на любой из этих производных классов, как описано ниже.

C++ Указатели на базовый класс

Указатель на базовый класс может указывать как на объекты этого класса, так и на объекты производных от него классов. Если этот указатель указывает на объект производного класса и от указателя вызывается виртуальная функция, то будет вызвана соответствующая функция-член этого класса.

Поэтому можно создать указатель на класс `Motor`, как показано ниже, а затем использовать его для указания на любые классы двигателей, производных от него:

```
Motor *MotorPtr;
```

Программа будет выполнять следующие действия:

1. Вращать вал двигателя в направлении «прямо» со скоростью 150, пока не будет нажата кнопка на клавиатуре.
2. Вращать вал двигателя в направлении «прямо» со скоростью 255, пока не будет нажата кнопка на клавиатуре.
3. Вращать вал двигателя в направлении «назад» со скоростью 150, пока не будет нажата кнопка на клавиатуре.
4. Вращать вал двигателя в направлении «назад» со скоростью 255, пока не будет нажата кнопка на клавиатуре.
5. Торможение двигателя.
6. Отключать питание двигателя.

Код, реализующий эти действия, приведён в **Листинге 8.10**. Для обнаружения нажатия кнопки на клавиатуре использована функция `kbhit()` и функция `getch()`, очищающая буфер клавиатуры после нажатия на кнопку.

Листинг 8.10. Управление двигателем

```
Motor *MotorPtr;

// Здесь должен происходить выбор типа двигателя

//..... Здесь начинается управление двигателем .....
MotorPtr->SetSpeed(150);
while(!kbhit()) MotorPtr->Forward();
getch(); // Очистка буфера клавиатуры

MotorPtr->SetSpeed(255);
while(!kbhit()) MotorPtr->Forward();
```

```

getch();

MotorPtr->SetSpeed(150);
while(!kbhit()) MotorPtr->Reverse();
getch();

MotorPtr->SetSpeed(255);
while(!kbhit()) MotorPtr->Reverse();
getch();

cout << endl << "    Braking Applied!" << endl;
while(!kbhit()) MotorPtr->Brake();
getch();

MotorPtr->Off();
//..... Здесь завершается управление двигателем .....

```

Пользователю нужно предоставить список всех доступных типов двигателей, чтобы он мог выбрать нужный. Реализация этой возможности приведена в **Листинге 8.11**.

Листинг 8.11. Вывод списка типов двигателей на экран

```

int Selection;

clrscr();

cout << endl << "    MOTOR MENU";
cout << endl << "    -----" << endl;
cout << "    1  DC Motor" << endl;
cout << "    2  UPFS" << endl;
cout << "    3  UPHS" << endl;
cout << "    4  BPFS" << endl;
cout << "    5  BPHS" << endl;
cout << "    6  QUIT" << endl;
cout << endl;
cout << "    Select the MOTOR Number: ";

cin >> Selection;

```

После того, как тип двигателя стал известным, соответствующий объект нужно разместить в памяти, используя динамическое выделение памяти. В **Листинге 8.12** приведён этот фрагмент.

Листинг 8.12. Динамическое выделение памяти под объект класса двигателя

```

switch(Selection)
{
    case 1 : MotorPtr = new DCMotor;
            break;
    case 2 : MotorPtr = new StepperMotor(UPFS);

```

```

        break;
    case 3 : MotorPtr = new StepperMotor(UPHS);
        break;
    case 4 : MotorPtr = new StepperMotor(BPFS);
        break;
    case 5 : MotorPtr = new StepperMotor(BPHS);
        break;
    case 6 : return;

default: cout << endl;
        cout << "    Unspecified Motor type..."
        cout << "  PRESS a key to END Program!";
        getch();
        exit(1); // Завершение программы
}

if(MotorPtr == NULL)
{
    cout << "Memory allocation failed" << endl;
    getch();
    exit(1);
}

```

По завершении этого фрагмента указатель `MotorPtr` должен указывать на объект в памяти. Если не был указан тип двигателя или не удалось выделить память под объект, то выполнение программы завершается. После инициализации указателя можно выполнять код из **Листинга 8.10**. Полный текст функции `main()` приведён в **Листинге 8.13**.

Листинг 8.13. Функция `main()`, управляющая двигателем

```

void main()
{
    Motor *MotorPtr;
    int Selection;

    clrscr();

    cout << endl << "    MOTOR MENU";
    cout << endl << "    -----" << endl;
    cout << "    1  DC Motor" << endl;
    cout << "    2  UPFS" << endl;
    cout << "    3  UPHS" << endl;
    cout << "    4  BPFS" << endl;
    cout << "    5  BPHS" << endl;
    cout << "    6  QUIT" << endl;
    cout << endl;
    cout << "    Select the MOTOR Number: ";

    cin >> Selection;

    switch(Selection)

```

```
{
    case 1 : MotorPtr = new DCMotor;
        break;
    case 2 : MotorPtr = new StepperMotor(UPFS);
        break;
    case 3 : MotorPtr = new StepperMotor(UPHS);
        break;
    case 4 : MotorPtr = new StepperMotor(BPFS);
        break;
    case 5 : MotorPtr = new StepperMotor(BPHS);
        break;
    case 6 : return;

default: cout << endl;
    cout << "    Unspecified Motor type..."
    cout << "    PRESS a key to END Program!";
    getch();
    exit(1); // Завершение программы
}

if(MotorPtr == NULL)
{
    cout << "Memory allocation failed" << endl;
    getch();
    exit(1);
}

cout << "*****" << endl;
cout << "* CONNECT BOARD POWER SUPPLY NOW *" << endl;
cout << "*****" << endl;
cout << endl;
cout << "    After conecting power,";
cout << "    press a key to continue" << endl;
getch();
cout << "    Keypress changes Speed/Rotation (& Braking)." << endl;

//..... Здесь начинается управление двигателем .....
MotorPtr->SetSpeed(150);
while(!kbhit()) MotorPtr->Forward();
getch(); // Очистка буфера клавиатуры

MotorPtr->SetSpeed(255);
while(!kbhit()) MotorPtr->Forward();
getch();

MotorPtr->SetSpeed(150);
while(!kbhit()) MotorPtr->Reverse();
getch();

MotorPtr->SetSpeed(255);
```



```

while(!kbhit()) MotorPtr->Reverse();
getch();

cout << endl << "    Braking Applied!" << endl;
while(!kbhit()) MotorPtr->Brake();
getch();

MotorPtr->Off();
//..... Здесь завершается управление двигателем .....
// Освободим память, занимаемую объектом двигателя
delete MotorPtr;
}

```

8.6.1. Виртуальные деструкторы

Деструктор класса вызывается неявно при выполнении оператора `delete` над объектом, об этом шла речь в разделе 5.3.8. Поскольку в иерархии классов не были написаны деструкторы, то имеются только автоматические деструкторы, сгенерированные компилятором для каждого класса иерархии. Для каждого выбранного типа двигателя в выражении `switch` создаётся свой объект, **Листинг 8.13**. При завершении программы, где был создан объект двигателя, происходит освобождение памяти, занимаемой им:

```
delete MotorPtr;
```

Так как `MotorPtr` является указателем на абстрактный класс `Motor`, то предыдущее выражение вызовет автоматический деструктор класса `Motor` и будет освобождена только лишь занимаемая им память. Выражение `delete` не освободит память, занимаемую объектом класса двигателя (например, `DCMotor`), что приводит к утечке памяти. Такое явление можно продемонстрировать на примере программы с использованием упрощённых класса и функции `main()`, **Листинг 8.14**. Для простоты примера классы `Base` и `Derived` не имеют членов данных. Для визуализации вызова обоих деструкторов они содержат выражения `cout`. Если бы не было выражений `cout`, то деструкторы были бы идентичными автоматическим.

Листинг 8.14. Использование неvirtуальных деструкторов

```

#include <conio.h>
#include <iostream.h>

class Base
{
public:
    Base() {}
    ~Base()
    {
        cout << "Base type object deleted" << endl;
    }
};

class Derived : public Base

```

```

{
    public:
        Derived() {};
        ~Derived()
        {
            cout << "Derived type object deleted" << endl;
        }
};

void main()
{
    Base *BasePtr;

    BasePtr = new Derived; // BasePtr указывает на объект
                          // класса Derived
    delete BasePtr; // Уничтожение объекта
}

```

Указатель `BasePtr` согласно объявления указывает на тип `Base`. В то же время он используется для указания на динамически создаваемый объект класса `Derived`. При помощи следующего выражения предпринимается попытка уничтожения объекта класса `Derived`:

```
delete BasePtr;
```

Казалось бы, это приведёт к вызову деструктора класса `Derived`. На самом же деле после запуска программы на экране появляется сообщение:

```
Base type object deleted
```

Следовательно, деструктор класса `Derived` не вызывался, как ожидалось, чтобы уничтожить динамически созданный объект класса `Derived`. Исправим программу, сделав деструктор `~Base()` виртуальным. Изменённая программа приведена в **Листинге 8.15**.

Листинг 8.15. Использование виртуальных деструкторов

```

#include <conio.h>
#include <iostream.h>

class Base
{
    public:
        Base() {}
        virtual ~Base()
        {
            cout << "Base type object deleted" << endl;
        }
};

class Derived : public Base
{

```

```

public:
    Derived() {}
    ~Derived()
    {
        cout << "Derived type object deleted" << endl;
    }
};

void main()
{
    Base *BasePtr;

    BasePtr = new Derived; // BasePtr указывает на объект
                           // класса Derived
    delete BasePtr; // Уничтожение объекта
}

```

В Листинге 8.15 перед деструктором `~Base()` стоит служебное слово `virtual`. Тем самым обеспечивается ссылка на все виртуальные деструкторы следующего уровня иерархии, что приводит к вызову соответствующего деструктора. После запуска программы на экране можно будет увидеть следующее:

```

Derived type object deleted
Base type object deleted

```

Это говорит о том, что оператор `delete` вызывает деструкторы классов `Base` и `Derived`, надлежащим образом освобождая память, занимаемую объектом, созданного на основе этих классов. Заметим, что при инстанцировании объекта производного класса сначала вызывается конструктор базового класса, а затем конструктор производного класса. Производный класс наследует члены базового класса. Поэтому необходимо создавать виртуальные деструкторы, чтобы освобождение памяти происходило правильно.

Вернёмся теперь к программе управления двигателями. Создадим несколько деструкторов, чтобы оператор `delete` мог правильно освобождать память. В каждом классе иерархии класса `Motor` будет по одному деструктору, они будут виртуальными. Тела этих деструкторов можно оставить пустыми. Нужно всего лишь создать цепочку виртуальных деструкторов на протяжении иерархии, чтобы правильно выполнялось их позднее связывание. Исправленные определения классов приведены в Листингах 8.16–8.18.

C++ Имена виртуальных деструкторов

В иерархии классов все виртуальные функции должны иметь одинаковые сигнатуры, т. е. должны быть одинаковыми имена, количество, типы и порядок следования формальных аргументов. Тем не менее, имена виртуальных деструкторов в иерархии разные. Но, несмотря на это, позднее связывание выполняется правильно в операторе `delete`.

Листинг 8.16. Класс `AbstractMotor` с виртуальным деструктором

```
class AbstractMotor
{
    private:
        int Speed;
    public:
        AbstractMotor();
        void SetSpeed(int speed);
        int GetSpeed();
        virtual void Off() = 0;
        virtual void Forward() = 0;
        virtual void Reverse() = 0;
        virtual void Brake() = 0;
        virtual ~AbstractMotor() {}
};
```

Листинг 8.17. Класс `ParallelPort` с виртуальным деструктором

```
class ParallelPort
{
    private:
        unsigned int BaseAddress;
        unsigned char InDataPort1;

    public:
        ParallelPort();
        ParallelPort(int baseaddress);
        void WritePort0(unsigned char data);
        void WritePort2(unsigned char data);
        unsigned char ReadPort1();
        virtual ~ParallelPort() {}
};
```

Листинг 8.18. Класс `Motor` с виртуальным деструктором

```
class Motor : public AbstractMotor, public ParallelPort
{
    public:
        Motor(int baseaddress = 0x378);
        void Off();
        virtual void Forward() = 0;
        virtual void Reverse() = 0;
        virtual void Brake() = 0;
        virtual ~Motor() {}
};
```

Теперь, когда есть правильно работающие деструкторы, можно написать полностью программу, управляющую двигателями при помощи виртуальных функций и способную надлежащим образом освобождать память от объектов. Такая про-

грамма приведена в **Листинге 8.19**. Обратите внимание, что виртуальные деструкторы не объявлялись в классах `DCMotor` и `StepperMotor`, поскольку это конечные классы иерархии. Таким образом, программа будет работать как и раньше, только уже с виртуальными деструкторами в составе классов.

Примечание

Перед подключением любого типа двигателя убедитесь, что **отключено** напряжение питания интерфейсной платы. Причина этого требования в следующем.

До вызова программы управления двигателями состояние порта неизвестно. При этом возможно, что состояния сигналов порта могут находиться в такой комбинации, когда управляющие ключевые транзисторы замыкают источник питания на землю (это приведёт к выходу из строя транзисторов и, возможно, источника питания). Программа выводит на экран сообщение, разрешающее пользователю включать питание интерфейсной платы только тогда, когда сигналы порта установлены в безопасное состояние. По завершении программы сигналы порта снова устанавливаются в безопасное состояние, предотвращая тем самым повреждение транзисторов и/или источник питания.

Коллекторный двигатель подключается к интерфейсной плате согласно **Табл. 8.7** и **Табл. 8.9**. Подключение шагового двигателя изображено на **Рис. 8.25** и **Рис. 8.26**.

Если вращения вала не происходит, то в первую очередь нужно проверить подключение двигателя.

Листинг 8.19. Полный текст программы управления двигателями с использованием виртуальных функций

```

/*****
    Программа управления двигателями с использованием
    виртуальных функций.
*****/

#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

class ParallelPort
{
private:
    unsigned int BaseAddress;
    unsigned char InDataPort1;

public:
    ParallelPort();

```

```
ParallelPort(int baseaddress);
void WritePort0(unsigned char data);
void WritePort2(unsigned char data);
unsigned char ReadPort1();
virtual ~ParallelPort() {}

};

ParallelPort::ParallelPort()
{
    BaseAddress = 0x378;
    InDataPort1 = 0;
}

ParallelPort::ParallelPort(int baseaddress)
{
    BaseAddress = baseaddress;
    InDataPort1 = 0;
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress, data);
}

void ParallelPort::WritePort2(unsigned char data)
{
    outportb(BaseAddress+2, data ^ 0x0B);
}

unsigned char ParallelPort::ReadPort1()
{
    InDataPort1 = inportb(BaseAddress+1);
    // Инвертируем старший бит для компенсации внутренней
    // инверсии схемы порта принтера.
    InDataPort1 ^= 0x80;
    // Используем маску (или фильтр) для обнуления
    // неиспользуемых битов D0, D1 и D2.
    InDataPort1 &= 0xF8;
    return InDataPort1;
}

class AbstractMotor
{
private:
    int Speed;

public:
    AbstractMotor();
    void SetSpeed(int speed);
};
```

```

        int GetSpeed();
        virtual void Off() = 0;
        virtual void Forward() = 0;
        virtual void Reverse() = 0;
        virtual void Brake() = 0;
        virtual ~AbstractMotor() {}
};

AbstractMotor::AbstractMotor()
{
    Speed = 0;
}

void AbstractMotor::SetSpeed(int speed)
{
    Speed = speed;
    if (Speed > 255) Speed = 255; // Ограничение максимального значения
    if (Speed < 0) Speed = 0; // Ограничение минимального значения
}

int AbstractMotor::GetSpeed()
{
    return Speed;
}

class Motor : public AbstractMotor, public ParallelPort
{
public:
    Motor(int baseaddress = 0x378);
    void Off();
    virtual void Forward() = 0;
    virtual void Reverse() = 0;
    virtual void Brake() = 0;
    virtual ~Motor() {}
};

Motor::Motor(int baseaddress) : ParallelPort(baseaddress)
{
    Off();
}

void Motor::Off()
{
    WritePort0(0x00);
}

class DCMotor : public Motor
{
public:

```

```
        DCMotor(int baseaddress = 0x378);
        virtual void Forward();
        virtual void Reverse();
        virtual void Brake();
};

DCMotor::DCMotor(int baseaddress) : Motor(baseaddress)
{
}

void DCMotor::Forward()
{
    int j;
    for(j = 0; j < GetSpeed(); j++)
        WritePort0(0x09);
    for( ; j < 256; j++)
        WritePort0(0x00);
}

void DCMotor::Reverse()
{
    int j;
    for(j = 0; j < GetSpeed(); j++)
        WritePort0(0x06);
    for( ; j < 256; j++)
        WritePort0(0x00);
}

void DCMotor::Brake()
{
    WritePort0(0x0C);
}

enum MOTORTYPE {UPFS, UPHS, BPFS, BPHS};

class StepperMotor : public Motor
{
private:
    MOTORTYPE MotorType;
    unsigned char Switching[8];
    int CycleIndex;
    int MaxIndex;

public:
    StepperMotor(MOTORTYPE motortype = UPFS,
        int baseaddress = 0x378);
    virtual void Forward();
    virtual void Reverse();
    virtual void Brake();
};
```



```

};

StepperMotor::StepperMotor(MOTORTYPE motortype,
    int baseaddress) : Motor(baseaddress)
{
    MotorType = motortype;
    CycleIndex = 0;

    switch(MotorType)
    {
        case UPFS : MaxIndex = 4;
            Switching[0] = 0x11;
            Switching[1] = 0x12;
            Switching[2] = 0x22;
            Switching[3] = 0x21;
            break;
        case UPHS : MaxIndex = 8;
            Switching[0] = 0x01;
            Switching[1] = 0x11;
            Switching[2] = 0x10;
            Switching[3] = 0x12;
            Switching[4] = 0x02;
            Switching[5] = 0x22;
            Switching[6] = 0x20;
            Switching[7] = 0x21;
            break;
        case BPFS : MaxIndex = 4;
            Switching[0] = 0x99;
            Switching[1] = 0x69;
            Switching[2] = 0x66;
            Switching[3] = 0x96;
            break;
        case BPHS : MaxIndex = 8;
            Switching[0] = 0x99;
            Switching[1] = 0x09;
            Switching[2] = 0x69;
            Switching[3] = 0x60;
            Switching[4] = 0x66;
            Switching[5] = 0x06;
            Switching[6] = 0x96;
            Switching[7] = 0x90;
    }
}

void StepperMotor::Forward()
{
    if(++CycleIndex == MaxIndex) CycleIndex = 0;
    WritePort0(Switching[CycleIndex]);
    delay(259-GetSpeed());
}

```

```
}

void StepperMotor::Reverse()
{
    if(--CycleIndex == -1) CycleIndex = MaxIndex - 1;
    WritePort0(Switching[CycleIndex]);
    delay(259-GetSpeed());
}

void StepperMotor::Brake()
{
    switch(MotorType)
    {
        case UPFS : case UPHS :
            WritePort0(0x11);
            break;
        case BPFS : case BPHS :
            WritePort0(0x99);
    }
}

void main()
{
    Motor *MotorPtr;
    int Selection;

    clrscr();

    cout << endl << "    MOTOR MENU";
    cout << endl << "    -----" << endl;
    cout << "    1  DC Motor" << endl;
    cout << "    2  UPFS" << endl;
    cout << "    3  UPHS" << endl;
    cout << "    4  BPFS" << endl;
    cout << "    5  BPHS" << endl;
    cout << "    6  QUIT" << endl;
    cout << endl;
    cout << "    Select the MOTOR Number: ";

    cin >> Selection;

    switch(Selection)
    {
        case 1 : MotorPtr = new DCMotor;
            break;
        case 2 : MotorPtr = new StepperMotor(UPFS);
            break;
        case 3 : MotorPtr = new StepperMotor(UPHS);
            break;
    }
}
```

```

        case 4 : MotorPtr = new StepperMotor(BPFS);
                break;
        case 5 : MotorPtr = new StepperMotor(BPHS);
                break;
        case 6 : return;

default: cout << endl;
        cout << "    Unspecified Motor type..."
        cout << "  PRESS a key to END Program!";
        getch();
        exit(1); // Завершение программы
}

if(MotorPtr == NULL)
{
    cout << "Memory allocation failed" << endl;
    getch();
    exit(1);
}

cout << "*****" << endl;
cout << "** CONNECT BOARD POWER SUPPLY NOW *" << endl;
cout << "*****" << endl;
cout << endl;
cout << "    After conecting power,";
cout << "  press a key to continue" << endl;
getch();
cout << "    Keypress changes Speed/Rotation (& Braking)." << endl;

//..... Здесь начинается управление двигателем .....
MotorPtr->SetSpeed(150);
while(!kbhit()) MotorPtr->Forward();
getch(); // Очистка буфера клавиатуры

MotorPtr->SetSpeed(255);
while(!kbhit()) MotorPtr->Forward();
getch();

MotorPtr->SetSpeed(150);
while(!kbhit()) MotorPtr->Reverse();
getch();

MotorPtr->SetSpeed(255);
while(!kbhit()) MotorPtr->Reverse();
getch();

cout << endl << "    Braking Applied!" << endl;
while(!kbhit()) MotorPtr->Brake();

```

```

    getch();

    MotorPtr->Off();
    //..... Здесь завершается управление двигателем .....
    // Освободим память, занимаемую объектом двигателя
    delete MotorPtr;
}

```

Объём программы не должен вызывать удивления, так как текст программы выше функции `main()` обычно находится в заголовочных файлах или в файлах библиотек и поэтому не виден. Если бы мы поступили таким образом, то программа представляла бы собой только функцию `main()`.

Понаблюдайте за процессом остановки двигателя без применения динамического торможения, закомментировав вызов функции `Brake()` в **Листинге 8.19**. Различие особенно будет заметно, если на валу будет инерционная нагрузка.

Если бы не использовались виртуальные функции, то код управления двигателями пришлось бы включать в каждый раздел выражения `case` в функции `main()` из **Листинга 8.13**. Для каждого случая выражения `case` нужно было бы объявлять указатель на объект соответствующего типа, а функции бы связывались с объектами во время программирования (раннее связывание). Функция `main()`, являющаяся аналогом этой функции из **Листинга 8.13**, но без использования виртуальных функций, приведена в **Листинге 8.20**.

Листинг 8.20. Функция `main()` **без** использования виртуальных функций

```

void main()
{
    int Selection;
    DCMotor* DCMotorPtr;
    StepperMotor* UPFSStepperMotorPtr;
    StepperMotor* UPHSStepperMotorPtr;
    StepperMotor* BPFSSStepperMotorPtr;
    StepperMotor* BPHSStepperMotorPtr;

    clrscr();

    cout << endl << "    MOTOR MENU";
    cout << endl << "    -----" << endl;
    cout << "    1  DC Motor" << endl;
    cout << "    2  UPFS" << endl;
    cout << "    3  UPHS" << endl;
    cout << "    4  BPFSS" << endl;
    cout << "    5  BPHS" << endl;
    cout << "    6  QUIT" << endl;
    cout << endl;
    cout << "    Select the MOTOR Number: ";

    cin >> Selection;

    switch(Selection)

```

```

{
    case 1 : DCMotorPtr = new DCMotor;
        if(DCMotorPtr == NULL)
        {
            cout << "Memory allocation failed" << endl;
            exit(1);
        }
        cout << "*****" << endl;
        cout << "* CONNECT BOARD POWER SUPPLY NOW *" << endl;
        cout << "*****" << endl;
        cout << endl;
        cout << "    After conecting power,";
        cout << "    press a key to continue" << endl;
        getch();
        cout << "    Keypress changes Speed/Rotation (& Braking)."
            << endl;

        //..... Здесь начинается управление двигателем .....
        DCMotorPtr->SetSpeed(150);
        while(!kbhit()) DCMotorPtr->Forward();
        getch(); // Очистка буфера клавиатуры

        DCMotorPtr->SetSpeed(255);
        while(!kbhit()) DCMotorPtr->Forward();
        getch();

        DCMotorPtr->SetSpeed(150);
        while(!kbhit()) DCMotorPtr->Reverse();
        getch();

        DCMotorPtr->SetSpeed(255);
        while(!kbhit()) DCMotorPtr->Reverse();
        getch();

        cout << endl << "    Braking Applied!" << endl;
        while(!kbhit()) DCMotorPtr->Brake();

        DCMotorPtr->Off();
        //..... Здесь завершается управление двигателем .....
        // Освободим память, занимаемую объектом двигателя
        delete DCMotorPtr;
        break;

    case 2 : UPFSStepperMotorPtr = new StepperMotor(UPFS);
        if(UPFSStepperMotorPtr == NULL)
        {
            cout << "Memory allocation failed" << endl;
            exit(1);
        }
}

```

```

cout << "*****" << endl;
cout << "** CONNECT BOARD POWER SUPPLY NOW *" << endl;
cout << "*****" << endl;
cout << endl;
cout << "    After conecting power,";
cout << "  press a key to continue" << endl;
getch();
cout << "    Keypress changes Speed/Rotation (& Braking)."
      << endl;

//..... Здесь начинается управление двигателем .....
UPFSStepperMotorPtr->SetSpeed(150);
while(!kbhit())UPFSStepperMotorPtr->Forward();
getch(); // Очистка буфера клавиатуры

UPFSStepperMotorPtr->SetSpeed(255);
while(!kbhit())UPFSStepperMotorPtr->Forward();
getch();

UPFSStepperMotorPtr->SetSpeed(150);
while(!kbhit())UPFSStepperMotorPtr->Reverse();
getch();

UPFSStepperMotorPtr->SetSpeed(255);
while(!kbhit())UPFSStepperMotorPtr->Reverse();
getch();

cout << endl << "    Braking Applied!" << endl;
while(!kbhit())UPFSStepperMotorPtr->Brake();

UPFSStepperMotorPtr->Off();
//..... Здесь завершается управление двигателем .....
// Освободим память, занимаемую объектом двигателя
delete UPFSStepperMotorPtr;
break;

case 3 : UPFSStepperMotorPtr = new StepperMotor(UPHS) ;
if(UPFSStepperMotorPtr == NULL)
{
    cout << "Memory allocation failed" << endl;
    exit(1);
}
cout << "*****" << endl;
cout << "** CONNECT BOARD POWER SUPPLY NOW *" << endl;
cout << "*****" << endl;
cout << endl;
cout << "    After conecting power,";
cout << "  press a key to continue" << endl;
getch();

```

```

cout << "      Keypress changes Speed/Rotation (& Braking)."
      << endl;

//..... Здесь начинается управление двигателем .....
UPHSSStepperMotorPtr->SetSpeed(150);
while(!kbhit())UPHSSStepperMotorPtr->Forward();
getch(); // Очистка буфера клавиатуры

UPHSSStepperMotorPtr->SetSpeed(255);
while(!kbhit())UPHSSStepperMotorPtr->Forward();
getch();

UPHSSStepperMotorPtr->SetSpeed(150);
while(!kbhit())UPHSSStepperMotorPtr->Reverse();
getch();

UPHSSStepperMotorPtr->SetSpeed(255);
while(!kbhit())UPHSSStepperMotorPtr->Reverse();
getch();

cout << endl << "      Braking Applied!" << endl;
while(!kbhit())UPHSSStepperMotorPtr->Brake();

UPHSSStepperMotorPtr->Off();
//..... Здесь завершается управление двигателем .....
// Освободим память, занимаемую объектом двигателя
delete UPHSSStepperMotorPtr;
break;

case 4 : BPFSSStepperMotorPtr = new StepperMotor(BPFS) ;
if(BPFSSStepperMotorPtr == NULL)
{
    cout << "Memory allocation failed" << endl;
    exit(1);
}

cout << "*****" << endl;
cout << "* CONNECT BOARD POWER SUPPLY NOW *" << endl;
cout << "*****" << endl;
cout << endl;
cout << "      After conecting power,";
cout << "  press a key to continue" << endl;
getch();
cout << "      Keypress changes Speed/Rotation (& Braking)."
      << endl;

//..... Здесь начинается управление двигателем .....
BPFSSStepperMotorPtr->SetSpeed(150);
while(!kbhit())BPFSSStepperMotorPtr->Forward();
getch(); // Очистка буфера клавиатуры

BPFSSStepperMotorPtr->SetSpeed(255);

```

```

while(!kbhit())BPFSStepperMotorPtr->Forward();
getch();

BPFSStepperMotorPtr->SetSpeed(150);
while(!kbhit())BPFSStepperMotorPtr->Reverse();
getch();

BPFSStepperMotorPtr->SetSpeed(255);
while(!kbhit())BPFSStepperMotorPtr->Reverse();
getch();

cout << endl << "    Braking Applied!" << endl;
while(!kbhit())BPFSStepperMotorPtr->Brake();

BPFSStepperMotorPtr->Off();
//..... Здесь завершается управление двигателем .....
// Освободим память, занимаемую объектом двигателя
delete BPFSStepperMotorPtr;
break;
case 5 : BPHSStepperMotorPtr = new StepperMotor (BPHS) ;
if(BPHSStepperMotorPtr == NULL)
{
    cout << "Memory allocation failed" << endl;
    exit(1);
}
cout << "*****" << endl;
cout << "* CONNECT BOARD POWER SUPPLY NOW *" << endl;
cout << "*****" << endl;
cout << endl;
cout << "    After conecting power,";
cout << "  press a key to continue" << endl;
getch();
cout << "    Keypress changes Speed/Rotation (& Braking)."
    << endl;

//..... Здесь начинается управление двигателем .....
BPHSStepperMotorPtr->SetSpeed(150);
while(!kbhit())BPHSStepperMotorPtr->Forward();
getch(); // Очистка буфера клавиатуры

BPHSStepperMotorPtr->SetSpeed(255);
while(!kbhit())BPHSStepperMotorPtr->Forward();
getch();

BPHSStepperMotorPtr->SetSpeed(150);
while(!kbhit())BPHSStepperMotorPtr->Reverse();
getch();

BPHSStepperMotorPtr->SetSpeed(255);

```



```

while(!kbhit()) BPHSStepperMotorPtr->Reverse();
getch();

cout << endl << "    Braking Applied!" << endl;
while(!kbhit()) BPHSStepperMotorPtr->Brake();

BPHSStepperMotorPtr->Off();
//..... Здесь завершается управление двигателем .....
// Освободим память, занимаемую объектом двигателя
delete BPHSStepperMotorPtr;
break;
case 6 : return;

default: cout << endl;
        cout << "    Unspecified Motor type..."
        cout << "    PRESS a key to END Program!";
        getch();
        exit(1); // Завершение программы
    }
}

```

Как видно, программирование без виртуальных функций может оказаться совершенно неэффективным. Напомним, что операторы `delete` из **Листингов 8.10, 8.13** и **8.20** освобождают память, занятую объектом двигателя.

8.7. Ввод с клавиатуры

Гораздо удобнее управлять двигателем с клавиатуры. Для этого программа должна распознавать нажатия на те клавиши, которые предназначены для управления двигателем. Распознать клавиши можно несколькими способами. Самый простой способ заключается в использовании функций `getch()` и `getche()`. Другой вариант подразумевает применение функции `kbhit()`.

Как видно из названия функций `getch()` и `getche()`, они позволяют получить код символа нажатой клавиши. Функция `getche()` ещё и отображает символ клавиши на экране. Такое поведение функции называется «эхо» (echo), о чём говорит буква «e» в конце имени функции `getche()`. Но пользоваться этими функциями на практике неудобно, потому что они ждут нажатия на кнопку. Выполнение функции `getch()` приостанавливает выполнение программы до тех пор, пока не будет нажата клавиша. Такая задержка мешает формировать сигналы оперативного управления двигателем.

Функция `kbhit()` работает иначе. Она не ждёт нажатия кнопки. Она проверяет, не была ли нажата кнопка в период времени, предшествующий её вызову. Если кнопка была нажата, то функция возвращает истину, а если нажатия не было, то 0 (т. е. ложь). Поскольку она не ждёт нажатия, то программа продолжает выполняться. Нужно отметить, что `kbhit()` не очищает буфер клавиатуры, но это необходимо делать для того, чтобы функция могла зафиксировать следующее нажатие.

Встроенная библиотечная функция `bioskey()` используется для обнаружения нажатий как на обычные кнопки *ASCII набора (Приложение В)*, так и на *клавиши расширенного набора*, такие как «Ins», «Delete» и т. д. Клавиши расширенного набора кодируются двухбайтовым кодом, тогда как обычные клавиши одним байтом.

Функция `bioskey()` может работать в различных режимах, в зависимости от значения переданного ей аргумента. Она имеет один параметр, значение которого определяет, будет ли она обнаруживать нажатие на клавишу, как `kbhit()`, или считывать код нажатой клавиши. Если эта функция используется для чтения кода нажатой клавиши, то она выполняет очистку буфера.

При помощи функции `peekb()` можно прочитать значение из специальной ячейки памяти, позволяющее обнаружить нажатие кнопок «Right Shift», «Left Shift», «Ctrl», «Alt», «Caps Lock», «Scroll Lock» и т. д. Функция `peekb()` не позволяет обнаружить нажатие некоторых кнопок, например, клавиш со стрелками.

Байт, расположенный по этому адресу, представляет собой набор флагов (или, иначе говоря, битов, **Рис. 8.27**), которые означают нажатые в данный момент клавиши. Функция `peekb()` быстрее остальных и будет меньше всего мешать непрерывному выполнению программы. Поэтому использование функции `peekb()` обеспечит плавную работу двигателя. Если применять более медленные функции чтения клавиатуры, то управление двигателем может происходить неправильно.

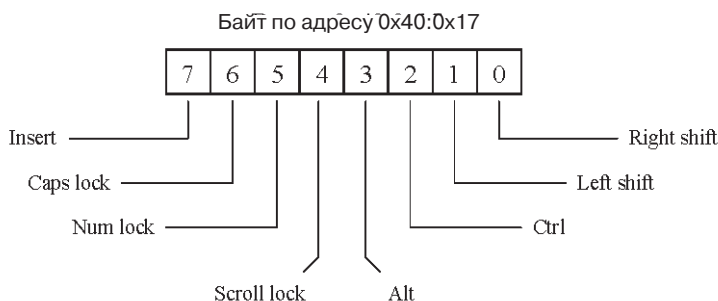


Рис. 8.27. В байте по адресу 0x40:0x17 записаны состояния клавиш

Благодаря функции `peekb()` можно управлять двигателем при помощи нажатия следующих клавиш:

- «Ctrl» для вращения в направлении «вперёд».
- «Alt» для вращения в направлении «назад».
- «Right Shift» для увеличения скорости.
- «Left Shift» для уменьшения скорости.
- Одновременное нажатие «Right Shift» и «Left Shift» для торможения двигателя.
- Отсутствие нажатия на какие-либо клавиши будет обесточивать двигатель.
- «Insert» для завершения программы.

Состояние клавиш «Right Shift», «Left Shift», «Alt» и «Insert» можно определить чтением специальной ячейки памяти функцией `peekb()`. Эта 8-битная ячейка расположена по адресу 0x40:0x17, адрес в формате *сегмент:смещение*. Когда

нажата одна из вышеперечисленных клавиш, соответствующий ей бит устанавливается в 1. Что такое *сегмент* и *смещение* описано в *Technical Reference: Personal Computer AT by IBM Corporation*.

Фрагмент программы с циклом `while`, приведённый в **Листинге 8.21**, выполняется непрерывно, пока переменная `Quit` равна 0. Этот фрагмент можно рассматривать как альтернативу фрагменту из **Листинга 8.10**.

Листинг 8.21. Управление двигателем с клавиатуры

```
Motor *MotorPtr;

int Quit = 0;
unsigned char key = 0;
int SpeedLock = 1;

while(!Quit)
{
    key = peekb(0x40,0x17); // Чтения байта состояния
                           // клавиш управления
    if(key & 0x80) // Проверка нажатия на "INSERT"
                  // (старший бит равен 1)
        Quit = 1; // Завершение программы

    else
    {
        // Если "RIGHT SHIFT" и "LEFT SHIFT"
        // не нажаты одновременно, то и
        // SpeedLock находится в неактивном состоянии

        if(!(key & 0x01) && !(key & 0x02))
            SpeedLock = 0;

        key &= 0x0F; // Оставить только биты клавиш
                    // "RIGHT SHIFT", "LEFT SHIFT", "ALT" и "CTRL"

        switch(key)
        {
            case 0x04 :
                MotorPtr->Forward();
                break;

            case 0x08 :
                MotorPtr->Reverse();
                break;

            case 0x01 :
                if(!SpeedLock)
                {
                    MotorPtr->SetSpeed(MotorPtr->GetSpeed() + 4);
```

```
        SpeedLock = 1;
    }
    break;

    case 0x02 :
    if (!SpeedLock)
    {
        MotorPtr->SetSpeed(MotorPtr->GetSpeed() - 4);
        SpeedLock = 1;
    }
    break;

    case 0x03 :
    MotorPtr->Brake();
    break;

    case 0x05 :
    if (!SpeedLock)
    {
        MotorPtr->SetSpeed(MotorPtr->GetSpeed() + 4);
        SpeedLock = 1;
    }
    MotorPtr->Forward();
    break;

    case 0x06 :
    if (!SpeedLock)
    {
        MotorPtr->SetSpeed(MotorPtr->GetSpeed() - 4);
        SpeedLock = 1;
    }
    MotorPtr->Forward();
    break;

    case 0x09 :
    if (!SpeedLock)
    {
        MotorPtr->SetSpeed(MotorPtr->GetSpeed() + 4);
        SpeedLock = 1;
    }
    MotorPtr->Reverse();
    break;

    case 0x0B :
    if (!SpeedLock)
    {
        MotorPtr->SetSpeed(MotorPtr->GetSpeed() - 4);
        SpeedLock = 1;
    }
}
```

```

        MotorPtr->Reverse();
        break;

        case 0x00 :
            MotorPtr->Off();
    }
}
delete MotorPtr;

```

В цикле `while` функция `peekb()` считывает содержимое ячейки памяти по адресу `0x40:0x17`. Полученное значение сохраняется в переменной `key`. В первую очередь проверяется, не было ли нажатия на клавишу «Insert», означающую завершение программы. Нажатие обнаруживается путём выполнения операции AND над значениями `key` и `0x80`. Если нажата клавиша «Insert», то получится ненулевое значение и `Quit` будет присвоено значение 1. В результате цикл `while` будет прекращён, а программа завершится штатным способом.

Если не было обнаружено нажатие «Insert», то программа продолжит выполняться. Теперь нужно узнать состояние четырёх младших битов, **Рис. 8.27**. Эти биты соответствуют четырём другим кнопкам управления двигателем, а клавиши «Caps Lock», «Num Lock» и «Scroll Lock» в программе не используются. Отфильтровав биты неиспользуемых клавиш (обнулив их), получим уникальные комбинации битов, соответствующие комбинациям клавиш управления. Обнуление битов происходит путём выполнения операции AND над значениями `key` и `0x0F`. Результат операции сохраняется в `key`, а затем проверяется в выражении `switch`. Все возможные управляющие воздействия перечислены в выражении `switch`, где происходит выбор необходимых действий. В **Листинге 8.22** приведён вариант **Листинга 8.19**, где реализовано управление с клавиатуры.

В программе задействован механизм «фиксации скорости», который использует переменную `SpeedLock` и предназначен для изменения скорости вращения вала двигателя. Такой механизм обеспечивает ступенчатое изменение скорости на небольшую величину при каждом нажатии соответствующей клавиши управления. Повторные считывания нажатой клавиши в цикле не должны приводить к дальнейшему изменению скорости благодаря механизму фиксации скорости. Без этого механизма при удержании клавиши изменения скорости будет происходить ненужное изменение скорости в ходе выполнения цикла `while`.

Скорость может меняться только при однократном нажатии на кнопки скорости («Left Shift» или «Right Shift»). При нажатии любой из этих кнопок переменная `SpeedSet` переходит в активное состояние. Как только `SpeedLock` станет активной (т. е. примет ненулевое значение), дальнейшее изменение скорости станет невозможным до тех пор, пока обе кнопки («Left Shift» и «Right Shift») не будут отпущены. Отпускание этих кнопок «разблокирует» механизм фиксации скорости, сбросив `SpeedLock` в неактивное состояние (т. е. `!SpeedLock` теперь будет истинным) и изменение скорости будет снова разрешено.

Фиксация скорости включена во все ветви изменения скорости выражения `switch`. Для выполнения выражений изменения скорости `SpeedLock` должна быть неактивной (`SpeedLock = 0`). `SpeedLock` обнуляется всякий раз при отпускании клавиш «Left Shift» и «Right Shift». При нажатии любой из этих клавиш проис-

ходит выбор соответствующего варианта изменения скорости внутри выражения `if`. Изменение скорости происходит путём увеличения или уменьшения текущей скорости `Speed` на 4. Повторное выполнение этого действия внутри выражения `if` блокируется установкой `SpeedLock` в активное состояние. Блокировка будет продолжаться до тех пор, пока не кнопки скорости не будут отпущены.

ВНИМАНИЕ!

Приступайте к выполнению нижеследующей программы только после подключения двигателя и выполнив процедуру включения питания согласно примечанию на стр. 220. Если двигатель не работает, то в первую очередь проверьте подключение двигателя.

Листинг 8.22. Полный текст программы управления двигателем с клавиатуры

```

/*****
Программа управления двигателем с использованием
виртуальных функций и клавиатуры
*****/

#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

class ParallelPort
{
private:
    unsigned int BaseAddress;
    unsigned char InDataPort1;

public:
    ParallelPort();
    ParallelPort(int baseaddress);
    void WritePort0(unsigned char data);
    void WritePort2(unsigned char data);
    unsigned char ReadPort1();
    virtual ~ParallelPort() {}
};

ParallelPort::ParallelPort()
{
    BaseAddress = 0x378;
    InDataPort1 = 0;
}

ParallelPort::ParallelPort(int baseaddress)

```

```
{
    BaseAddress = baseaddress;
    InDataPort1 = 0;
}

void ParallelPort::WritePort0(unsigned char data)
{
    outportb(BaseAddress,data);
}

void ParallelPort::WritePort2(unsigned char data)
{
    outportb(BaseAddress+2,data ^ 0x0B);
}

unsigned char ParallelPort::ReadPort1()
{
    InDataPort1 = inportb(BaseAddress+1);
    // Инвертируем старший бит для компенсации внутренней
    // инверсии схемы порта принтера.
    InDataPort1 ^= 0x80;
    // Используем маску (или фильтр) для обнуления
    // неиспользуемых битов D0, D1 и D2.
    InDataPort1 &= 0xF8;
    return InDataPort1;}

class AbstractMotor
{
private:
    int Speed;
public:
    AbstractMotor();
    void SetSpeed(int speed);
    int GetSpeed();
    virtual void Off() = 0;
    virtual void Forward() = 0;
    virtual void Reverse() = 0;
    virtual void Brake() = 0;
    virtual ~AbstractMotor() {}
};

AbstractMotor::AbstractMotor()
{
    Speed = 0;
}

void AbstractMotor::SetSpeed(int speed)
{
```

```
    Speed = speed;
    if (Speed > 255) Speed = 255; // Ограничение максимального значения
    if (Speed < 0) Speed = 0; // Ограничение минимального значения
}

int AbstractMotor::GetSpeed()
{
    return Speed;
}

class Motor : public AbstractMotor, public ParallelPort
{
public:
    Motor(int baseaddress = 0x378);
    void Off();
    virtual void Forward() = 0;
    virtual void Reverse() = 0;
    virtual void Brake() = 0;
    virtual ~Motor() {}
};

Motor::Motor(int baseaddress) : ParallelPort(baseaddress)
{
    Off();
}

void Motor::Off()
{
    WritePort0(0x00);
}

class DCMotor : public Motor
{
public:
    DCMotor(int baseaddress = 0x378);
    virtual void Forward();
    virtual void Reverse();
    virtual void Brake();
};

DCMotor::DCMotor(int baseaddress) : Motor(baseaddress)
{
}

void DCMotor::Forward()
{
    int j;
    for(j = 0; j < GetSpeed(); j++)
        WritePort0(0x09);
}
```



```

    for( ; j < 256; j++)
        WritePort0(0x00);
}

void DCMotor::Reverse()
{
    int j;
    for(j = 0; j < GetSpeed(); j++)
        WritePort0(0x06);
    for( ; j < 256; j++)
        WritePort0(0x00);
}

void DCMotor::Brake()
{
    WritePort0(0x0C);
}

enum MOTORTYPE {UPFS, UPHS, BPFS, BPHS};

class StepperMotor : public Motor
{
private:
    MOTORTYPE MotorType;
    unsigned char Switching[8];
    int CycleIndex;
    int MaxIndex;

public:
    StepperMotor(MOTORTYPE motortype = UPFS,
        int baseaddress = 0x378);
    virtual void Forward();
    virtual void Reverse();
    virtual void Brake();
};

StepperMotor::StepperMotor(MOTORTYPE motortype,
    int baseaddress) : Motor(baseaddress)
{
    MotorType = motortype;
    CycleIndex = 0;

    switch(MotorType)
    {
    case UPFS : MaxIndex = 4;
        Switching[0] = 0x11;
        Switching[1] = 0x12;
        Switching[2] = 0x22;
        Switching[3] = 0x21;
    }
}

```

```

        break;
    case UPHS : MaxIndex = 8;
        Switching[0] = 0x01;
        Switching[1] = 0x11;
        Switching[2] = 0x10;
        Switching[3] = 0x12;
        Switching[4] = 0x02;
        Switching[5] = 0x22;
        Switching[6] = 0x20;
        Switching[7] = 0x21;
        break;
    case BPFS : MaxIndex = 4;
        Switching[0] = 0x99;
        Switching[1] = 0x69;
        Switching[2] = 0x66;
        Switching[3] = 0x96;
        break;
    case BPHS : MaxIndex = 8;
        Switching[0] = 0x99;
        Switching[1] = 0x09;
        Switching[2] = 0x69;
        Switching[3] = 0x60;
        Switching[4] = 0x66;
        Switching[5] = 0x06;
        Switching[6] = 0x96;
        Switching[7] = 0x90;
    }
}

void StepperMotor::Forward()
{
    if(++CycleIndex == MaxIndex) CycleIndex = 0;
    WritePort0(Switching[CycleIndex]);
    delay(259-GetSpeed());
}

void StepperMotor::Reverse()
{
    if(--CycleIndex == -1) CycleIndex = MaxIndex - 1;
    WritePort0(Switching[CycleIndex]);
    delay(259-GetSpeed());
}

void StepperMotor::Brake()
{
    switch(MotorType)
    {
        case UPFS : case UPHS :
            WritePort0(0x11);

```

```

        break;
    case BPFS : case BPHS :
        WritePort0(0x99);
    }
}

void main()
{
    int Quit = 0;
    unsigned char key = 0;
    int SpeedLock = 1;
    int Selection;

    clrscr();

    cout << endl << "    MOTOR MENU";
    cout << endl << "    -----" << endl;
    cout << "    1  DC Motor" << endl;
    cout << "    2  UPFS" << endl;
    cout << "    3  UPHS" << endl;
    cout << "    4  BPFS" << endl;
    cout << "    5  BPHS" << endl;
    cout << "    6  QUIT" << endl;
    cout << endl;
    cout << "    Select the MOTOR Number: ";

    cin >> Selection;

    switch(Selection)
    {
        case 1 : MotorPtr = new DCMotor;
            break;
        case 2 : MotorPtr = new StepperMotor(UPFS);
            break;
        case 3 : MotorPtr = new StepperMotor(UPHS);
            break;
        case 4 : MotorPtr = new StepperMotor(BPFS);
            break;
        case 5 : MotorPtr = new StepperMotor(BPHS);
            break;
        case 6 : return;

    default: cout << endl;
        cout << "    Unspecified Motor type..."
        cout << "    PRESS a key to END Program!";
        getch();
        exit(1); // Завершение программы
    }

    if(MotorPtr == NULL)

```

```
{
    cout << "Memory allocation failed" << endl;
    getch();
    exit(1);
}

cout << "*****" << endl;
cout << "* CONNECT BOARD POWER SUPPLY NOW *" << endl;
cout << "*****" << endl;
cout << endl;
cout << "    After connecting power,";
cout << "    press a key to continue" << endl;
getch();

while(!Quit)
{
    key = peekb(0x40,0x17); // Чтения байта состояния
                           // клавиш управления
    if(key & 0x80) // Проверка нажатия на "INSERT"
                  // (старший бит равен 1)
        Quit = 1; // Завершение программы

    else
    {
        // Если "RIGHT SHIFT" и "LEFT SHIFT"
        // не нажаты одновременно, то и
        // SpeedLock находится в неактивном состоянии

        if(!(key & 0x01) && !(key & 0x02))
            SpeedLock = 0;

        key &= 0x0F; // Оставить только биты клавиш
                    // "RIGHT SHIFT", "LEFT SHIFT", "ALT" и "CTRL"

        switch(key)
        {
            case 0x04 :
                MotorPtr->Forward();
                break;

            case 0x08 :
                MotorPtr->Reverse();
                break;

            case 0x01 :
                if(!SpeedLock)
                {
                    MotorPtr->SetSpeed(MotorPtr->GetSpeed() + 4);
                    SpeedLock = 1;
                }
            }
        }
    }
}
```

```
    }  
    break;  
  
    case 0x02 :  
    if(!SpeedLock)  
    {  
        MotorPtr->SetSpeed(MotorPtr->GetSpeed() - 4);  
        SpeedLock = 1;  
    }  
    break;  
  
    case 0x03 :  
    MotorPtr->Brake();  
    break;  
  
    case 0x05 :  
    if(!SpeedLock)  
    {  
        MotorPtr->SetSpeed(MotorPtr->GetSpeed() + 4);  
        SpeedLock = 1;  
    }  
    MotorPtr->Forward();  
    break;  
  
    case 0x06 :  
    if(!SpeedLock)  
    {  
        MotorPtr->SetSpeed(MotorPtr->GetSpeed() - 4);  
        SpeedLock = 1;  
    }  
    MotorPtr->Forward();  
    break;  
  
    case 0x09 :  
    if(!SpeedLock)  
    {  
        MotorPtr->SetSpeed(MotorPtr->GetSpeed() + 4);  
        SpeedLock = 1;  
    }  
    MotorPtr->Reverse();  
    break;  
  
    case 0x0B :  
    if(!SpeedLock)  
    {  
        MotorPtr->SetSpeed(MotorPtr->GetSpeed() - 4);  
        SpeedLock = 1;  
    }  
    MotorPtr->Reverse();
```

```
        break;

        case 0x00 :
            MotorPtr->Off();
    }
}

delete MotorPtr;
}
```

В программе управления двигателями, где используются виртуальные функции, можно написать код управления, общий для всех типов двигателей. Такой подход позволяет связать нужную функцию с объектом, который выбирается во время выполнения (позднее связывание). После того, как выбран тип двигателя, происходит динамическое выделение памяти под объект двигателя и осуществляется управление двигателем с использованием позднего связывания. Если не пользоваться виртуальными функциями, то пришлось бы писать очень большой и сложный код, пример которого приведён в **Листинге 8.20**.

Программа управления двигателями с виртуальными функциями не нуждается в специальном коде для каждого типа двигателя. Код программы не зависит от конкретного типа двигателя и будет работать даже с другими типами двигателей, которые можно добавить в будущем. Такие возможности являются основным преимуществом использования виртуальных функций и это можно рассматривать как главное достоинство объектно-ориентированного программирования.

8.8. Заключение

В этой главе были рассмотрены конструкция и работа коллекторных и шаговых двигателей. Также были описаны различные методы управления ими.

Была разработана иерархия классов для всех типов двигателей, обсуждавшихся в начале главы. Затем были объяснены основные понятия абстрактных классов и чистых виртуальных функций. Было рассказано о множественном наследовании в иерархии классов. Была показана необходимость создания виртуальных деструкторов в иерархии классов. В отличие от конструкторов, деструкторы могут быть виртуальными. Такие деструкторы нужны для освобождения динамически выделенной под объекты памяти, когда они станут ненужными в программе.

Разработана программа для управления любым типом двигателей из иерархии и реализована в функции `main()` для демонстрации принципов и преимуществ позднего связывания. Для более удобного управления двигателем добавлена возможность управления с клавиатуры.

8.9. Литература

1. Bergsman, P., *Controlling The World With Your PC*, HighText Publications, San Diego, 1994.
2. Stiffler, K., *Design with Microprocessors for Mechanical Engineers*, McGrawHill, 1992.

3. PARKER HANFINN CORP, *Positioning Control Systems and Drives*, 1992-1993.
4. Lafore, R. *Object Oriented Programming in MICROSOFT C++*, Waite Group Press, 1992.
5. Wang, P. S., *C++ with Object Oriented Programming*, PWS Publishing, 1994.
6. Borland, *Borland C++ version 5, User's Guide*, Borland International, 1996.
7. Borland, *Borland C++ version 5, Programmers Guide*, Borland International, 1996.
8. Barton, J. J. And L. R. Nackman, *Scientific and Engineering C++ – An introduction with Advanced Techniques and Examples*, Addison Wesley, 1994.
9. Stevens, A. *Teach Yourself C++ Fifth Edition*, MIS Press, 1997.
10. Schildt, H. *Teach Yourself C++ Third Edition*, Osborne/McGraw-Hill, 1998.

9 Методы программирования

Содержание главы:

- Что подразумевает разработка программы?
- Методы эффективного программирования.
- Модульное программирование.
- Заголовочные файлы.
- Файлы функций.
- Файлы проекта и make-файлы.

9.1. Введение

На данный момент изучено объектно-ориентированное программирование в плане написания эффективных программ, где основной упор делается на написание исходного кода. В этой главе будет показано, как планировать программу при помощи псевдокода, формировать структуру программы и писать программу так, чтобы свести к минимуму опечатки. Также будет рассмотрен способ создания модульных программ.

Модульная программа создаётся путём разбиения большого файла программы на несколько модулей по функциональному признаку, которые затем размещаются в отдельных файлах. Такой подход значительно облегчает обслуживание программ. Более того, облегчается модифицирование и отладка программ. Будет изучено создание многофайловых программ из нескольких исходных файлов, на основе которых в дальнейшем будет генерироваться конечный исполняемый файл.

9.2. Методы эффективного программирования

В этой главе будет использоваться слово *кодирование*, под которым следует понимать процесс написания выражений программы. Нельзя приступать к кодированию, пока не будет разработан детальный план программы. Такой план представляет собой обобщённое описание и называется *псевдо-кодом*. При написании объектно-ориентированной программы в первую очередь создаются классы объектов и иерархия этих классов. В результате применения объектно-ориентированного подхода получаются хорошо структурированные программы. Разработка программы должна происходить в несколько контролируемых этапов. На каждом этапе программа (или часть программы) может быть откомпилирована и проверена на ошибки.

Большинство текстовых редакторов могут копировать и вставлять фрагменты текста. Можно минимизировать количество ошибок в тексте программы, пользуясь операциями копирования/вставки. Такие ошибки (опечатки) составляют большую часть всех ошибок компиляции.

Псевдо-код

Применение псевдокода для написания основы программы может помочь в её разработке. В следующем примере показано, как создаётся псевдо-код и применяется в формировании исходного кода.

Описание задачи: имеется кран, перемещающий груз из точки **A** в точку **B**. Известно, что кран управляется тремя коллекторными двигателями: один перемещает груз вверх/вниз, другой по оси **x**, а третий по оси **y**.

Тогда псевдо-код будет таким:

- Ввод координат точек **A** и **B**.
- Перемещение крана в точку **A**.
- Поднять груз.
- Переместить кран в точку **B**.
- Опустить нагрузку.
- Конец.

Этот псевдо-код можно преобразовать в программу на C++. В следующем примере программы приведена реализация функции `main()` с псевдокодом:

```
void main()
{
    Crane OurCrane; // Создание объекта Crane.
    Point a, b; // Создание двух объектов точек

    cout << "Enter coordinates of point A ";
    cin >> a;

    OurCrane.MoveToPoint(a);
    OurCrane.LiftLoad();
    OurCrane.MoveToPoint(b);
    OurCrane.LowerLoad();
}
```

В функции `main()` используются объекты типа `Point`. Также имеется объект типа `Crane` и предполагается, что класс `Crane` обладает функциями-членами `MoveToPoint()`, `LiftLoad()` и `LowerLoad()`. Для работы необходимы две точки и три двигателя; один двигатель перемещает груз вверх/вниз, второй двигатель выполняет перемещение груза по оси **x**, третий по оси **y**.

Класс `Point` мог бы быть таким:

```
class Point
{
    private:
        int X;
        int Y;

    public:
```

```
Point();  
Point(int x, int y);  
void SetX(int x);  
void SetY(int y);  
int GetX();  
int GetY();  
}
```

А класс `Crane` может быть таким:

```
Class Crane  
{  
    private:  
        Point A, B;  
        DCMotor LiftMotor, Xmotor, Ymotor;  
  
    public:  
        Crane();  
        void SetPointA(Point a);  
        void SetpointB(Point b);  
        void MoveToPoint();  
        void LiftLoad();  
        void LowerLoad();  
};
```

Все функции-члены класса `Crane`, осуществляющие перемещение, должны использовать функции `Forward()`, `Reverse()` и `Brake()` класса `DCMotor`.

Как было показано, структурное программирование начинается с написания псевдо-кода, образующим структуру программы. Каждый из этих структурных элементов представляет собой выражение программы в функции `main()`. Для точного указания необходимых действий использованы функции класса. В следующем разделе описан хороший подход к написанию класса `AbstractMotor` и его функций-членов.

Создание функций в определении класса

При кодировании классов и их функций-членов нужно поддерживаться несколькими основными принципами. Вообще, члены класса могут находиться где угодно в пределах определения класса (между пары фигурных скобок) с различными атрибутами доступа. Тем не менее, разделение данных и функций на две группы может облегчить кодирование.

В классах часто определяются конструкторы, выполняющие инициализацию членов данных. Если члены данных сгруппированы вместе, то их можно скопировать одним блоком в конструктор и инициализировать. Они могут быть скопированы также в определения других функций. Определения функций давать не обязательно, а исходный код уже можно компилировать, чтобы убедиться в отсутствии синтаксических ошибок.

Обратимся к определению класса `AbstractMotor` из **Листинга 8.1** главы 8, который воспроизведён в **Листинге 9.1**.

Листинг 9.1. Определение класса `AbstractMotor`

```
class AbstractMotor
{
    private:
        int Speed;

    public:
        AbstractMotor();
        void SetSpeed(int speed);
        int GetSpeed();
        virtual void Off() = 0;
        virtual void Forward() = 0;
        virtual void Reverse() = 0;
        virtual void Brake() = 0;
};
```

В этом определении данные и функции разделены на две группы. Это даёт возможность скопировать набор не чистых виртуальных функций (которые подлежат определению) ниже за пределами класса, **Листинг 9.2**. Чистые виртуальные функции копировать не нужно, поскольку они не требуют определения. Полезно группировать чистые виртуальные функции вместе, предпочтительно в конце объявления функций-членов. Так нужно для удобства копирования функций-членов, которые нужно определить, целыми группами.

Листинг 9.2. Копирование функций, подлежащих определению

```
class AbstractMotor
{
    private:
        int Speed;

    public:
        AbstractMotor();
        void SetSpeed(int speed);
        int GetSpeed();
        virtual void Off() = 0;
        virtual void Forward() = 0;
        virtual void Reverse() = 0;
        virtual void Brake() = 0;
};
AbstractMotor();
void SetSpeed(int speed);
int GetSpeed();
```

В таком виде файл нельзя компилировать, потому что он не завершён и, соответственно, нарушен синтаксис. Его нужно исправить, оформив функции в соответствии требованиям синтаксиса. Для этого каждой только что скопированной функции нужно добавить оператор области видимости (`::`). В случае функции `AbstractMotor()` из **Листинга 9.2** должно получиться следующее:

```
AbstractMotor::AbstractMotor()
```

У каждой функции на конце нужно удалить точку с запятой и добавить пару фигурных скобок, обозначающих начало и конец каждой функции-члена. Изменения **Листинга 9.2** показаны в **Листинге 9.3**.

Листинг 9.3. Заготовки функций-членов класса `AbstractMotor`

```
class AbstractMotor
{
    private:
        int Speed;

    public:
        AbstractMotor();
        void SetSpeed(int speed);
        int GetSpeed();
        virtual void Off() = 0;
        virtual void Forward() = 0;
        virtual void Reverse() = 0;
        virtual void Brake() = 0;
};

AbstractMotor()
{
}

void SetSpeed(int speed)
{
}

int GetSpeed()
{
}
```

Теперь можно компилировать файл. Хотя тела функций и не содержат никаких выражений, синтаксис программы всё же правильный. Заметим, что связи программы не могут быть отредактированы (или разрешены; или слинкованы), пока не будет функции `main()`. Но при компиляции объектный код генерироваться всё-таки будет. Является полезным компилировать программу на ранних стадиях, что позволяет обнаружить нарушения синтаксиса.

Заготовки функций

Определения функций с пустыми телами (**Листинг 9.3**) можно назвать заготовками функций. Тела этих функций можно определить на этой ранней стадии программирования. Но они оставлены пустыми, чтобы упростить задачу формирования хорошей структуры программы и правильного синтаксиса. Может оказаться полезным написать даже всю программу с заготовками функций, чтобы проверить её работоспособность и структуру.

Заполнение тела конструктора

Следующим этапом является кодирование тел функций-членов, начиная, скажем, с определения конструктора. Как упоминалось выше, обычной задачей конструкторов является инициализация членов данных класса. Это можно сделать, копируя весь набор членов данных в конструкторы класса. В вышеприведённом примере есть только один член данных. В **Листинге 9.4** показан конструктор, куда скопирован член данных. Функция ещё не является законченной и не выполняет возложенных на неё задач.

Листинг 9.4. Копирование объявления члена данных в тело конструктора

```
AbstractMotor::AbstractMotor()  
{  
    int Speed;  
}
```

В **Листинге 9.4** нужно исправить единственное выражение, как показано в **Листинге 9.5**. Как только это будет сделано, конструктор сможет инициализировать этот член данных и работать в соответствии требованиям.

Листинг 9.5. Конструктор с исправленной синтаксической ошибкой

```
AbstractMotor::AbstractMotor()  
{  
    int Speed = 0; // инициализируется  
                  // скопированный член данных  
}
```

Правильное обращение с круглыми, квадратными и фигурными скобками

Пары круглых `()`, квадратных `[]` и фигурных `{ }` скобок широко используются в программировании на C++. При описании всех трёх типов скобок будем называть их просто скобками. В тексте программы иногда забывают поставить закрывающую скобку в пару к открывающей. Люди, пользуясь своим интеллектом, могут найти и исправить ошибку, но для компилятора эта задача слишком сложная. Поэтому программисту приходится самому исправлять ошибку. Неправильное использование скобок обычно происходит в тех случаях, когда программист теряет контроль над парами скобок, и при компиляции возникают ошибки. По мере прибавления выражений в программе иногда бывает очень сложно найти то место, где нужно поставить скобку. Задача ещё более усложняется при вложенных парах скобок.

Проблема парности скобок решается путём одновременной постановки пар скобок. Способ, приведённый ниже, позволяет легко реализовать определение класса:

Первый шаг:

```
class AbstractMotor  
{  
};
```

Второй шаг:

```
class AbstractMotor
{
    private:
        int Speed;

    public:
        AbstractMotor();
        void SetSpeed(int speed);
        int GetSpeed();
        virtual void Off() = 0;
        virtual void Forward() = 0;
        virtual void Reverse() = 0;
        virtual void Brake() = 0;
};
```

Далее показан способ определения функции:

Первый шаг:

```
void SetSpeed();
```

Второй шаг:

```
void SetSpeed(int speed)
{
}
```

Третий шаг:

```
void SetSpeed(int speed)
{
    int Speed;
}
```

Четвёртый шаг:

```
void SetSpeed(int speed)
{
    Speed = 0;
}
```

Вложенные блоки

В следующем примере показано, как формируются вложенные выражения if:

Первый шаг:

```
if()
{
}
else
{
}
```

Второй шаг:

```

if()
{
if()
{
}
else
{
}
}
else
{
}
}

```

Уровень вложенности обозначается отступом. Тогда каркас выражения `if` будет выглядеть следующим образом:

```

if()
{
    if()
    {
    }
    else
    {
    }
}
else
{
}
}

```

Тот же принцип поможет избежать ошибок в условии выражения `if`:

```

if()
{
    if(( ) && ( ))
    {
    }
    else
    {
    }
}
else
{
}
}

```

В качестве условных выражений можно взять, например, такие:

```

if(b > 0)
{
    if((a != 0) && (b/a > n))

```

```
{  
}  
else  
{  
}  
}  
else  
{  
}
```

9.3. Модульные программы

В программах, созданных в предыдущих главах, весь текст программы находился в одном файле. Если для простых программ это приемлемо, то с ростом их размера сложности это становится непрактичным. В больших программах весь код разбит на отдельные модули, хранящиеся в разных файлах. Поскольку в таком случае нужно компилировать больше одного файла, то такая программа называется *многофайловой*.

Есть несколько причин разработки многофайловых программ. В случае объектно-ориентированного программирования в каждом файле может содержаться код одного класса. В таком случае для распространения программного обеспечения какого-либо объекта нужен только один файл с его кодом. Разработчики программ обычно предоставляют пользователям модули объектного кода (пользователи не могут их прочитать) вместе с такими файлами, которые позволяют разрешить связи модуля и программы и, таким образом, пользоваться объектным кодом. Такой подход предотвращает противозаконное использование объектного кода.

Если в программе используется какой-либо объект, то в её текст нужно включить только один файл, содержащий код этого объекта. Объём текста программы при этом уменьшается. Если бы в этом файле находилось несколько классов, то пришлось бы компилировать ненужные функции и компилирование программы происходило бы медленнее.

9.3.1. Разбиение программы на модули

Процесс создания многофайловой программы будет показан на примере программы управления двигателями, разработанной в главе 8 (иерархия классов приведена на **Рис. 9.1**). Каждый класс представлен двумя файлами. Определение класса размещается в заголовочном файле, а определение функций-членов этого класса в файле функций, **Рис. 9.2**. Оставшаяся часть программы находится в функции `main()` (но так бывает не всегда).

Компилятор должен найти определение класса в заголовочном файле, а определение функций в файле функций. Файл функций (содержащий сходный код, *.cpp) компилируется разработчиком в файл .obj или .lib. Пользующийся этими файлами программист уже не может их прочитать, и поэтому программное обеспечение разработчика защищено от неавторизованного копирования и незаконного использования.

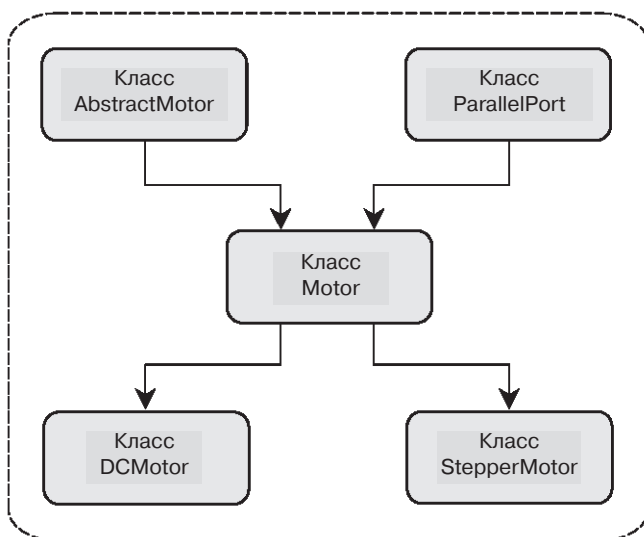


Рис. 9.1. Иерархия классов двигателей (глава 8)

Объектно-ориентированная и обычная программа имеют отличия. В первом случае функции являются функциями-членами, а во втором случае все функции не являются членами классов. В объектно-ориентированной программе эти два типа функций могут использоваться одновременно. В заголовке такой программы могут находиться определение класса и объявления некоторых функций, не являющихся членами класса.

Если в программе на С/С++ используются функции, не являющиеся членами класса, то обычно объявления таких функций размещают в заголовочном файле какого-нибудь класса. Для линковки файл функций должен быть преобразован в `.obj` или `.lib`.

9.3.2. Сборка многофайловой программы

Как говорилось выше, чтобы использовать какой-либо класс в программе, нужно подключить соответствующий заголовочный файл перед первым использованием класса. Компилятор сначала прочитает определение класса (из первого заголовочного файла), а затем правильность реализации класса будет проверяться на протяжении оставшейся части программы. Программа будет правильно откомпилирована только в том случае, если функции вызываются в соответствии их объявлениям в заголовочных файлах. На **Рис. 9.3** показано как путём подключения соответствующих заголовочных файлов в пользовательской программе действуют классы `Motor`, `DCMotor` и `StepperMotor`.

На **Рис. 9.4** показано включение заголовочного файла класса `DCMotor` в файл с определениями функций этого класса. Такие действия гарантируют соответствие определений функций их объявлениям в классе `DCMotor` (который произведён от класса `Motor` и нужным образом переопределён), определённом в файле `dcmotor.h`.

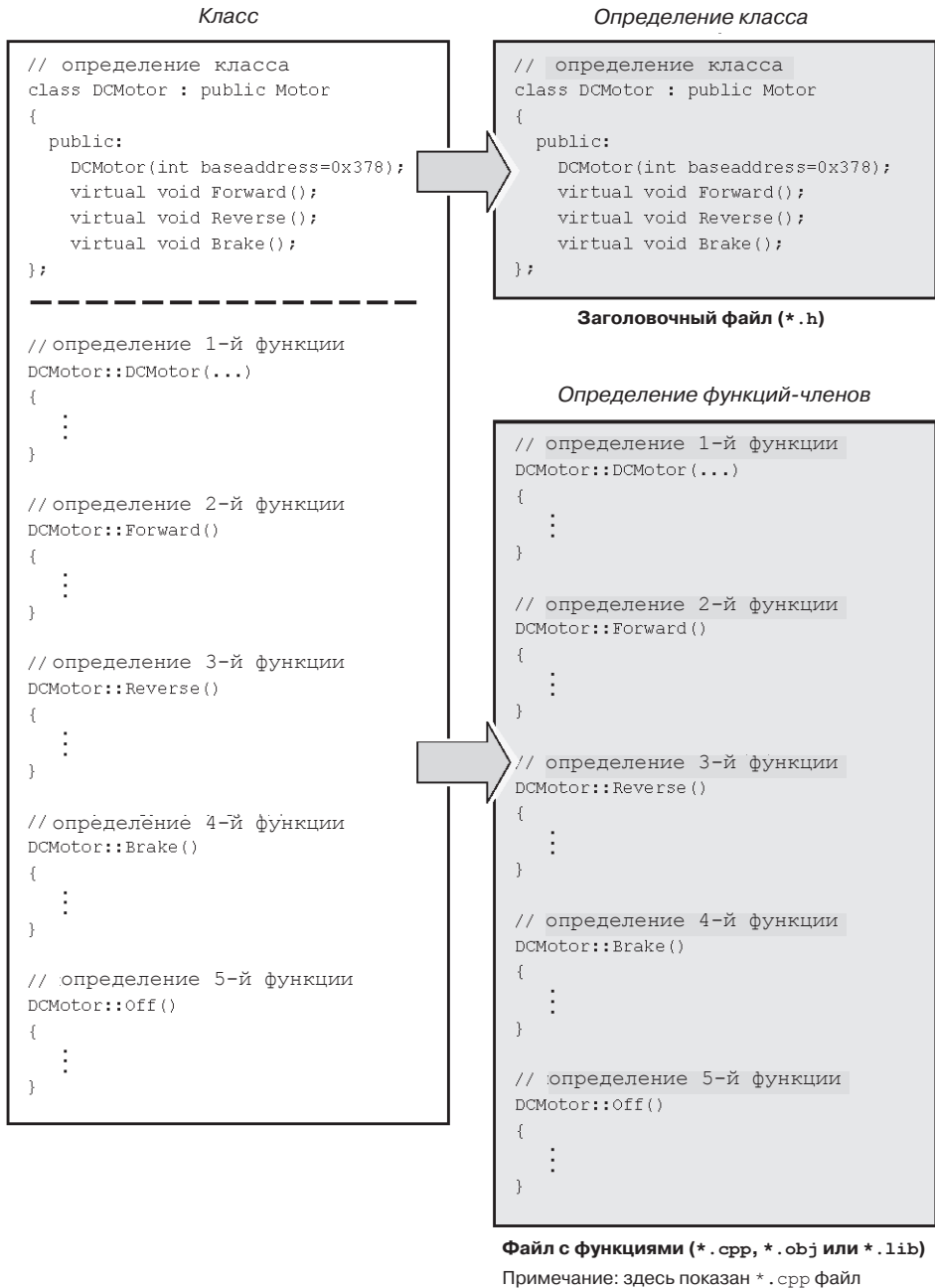


Рис. 9.2. Разделение кода на заголовочный файл и файл с функциями

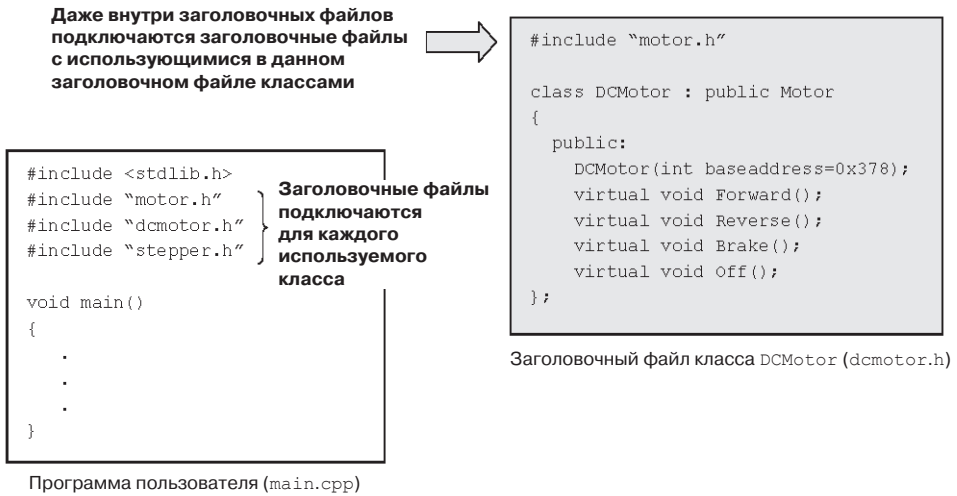


Рис. 9.3. Применение директив включения файла

Директивы включения

Имена файлов, которые подключаются из include-каталога пакета C/C++, заключаются в угловые скобки (< и >):

```
#include <stdlib.h> // подключение стандартных библиотечных функций C
```

Заголовочные файлы из того же каталога или любого другого, указанного в пути файла, заключаются в двойные кавычки " и ":

```
#include "motor.h" // Подключение класса Motor.
```

Правильное применение угловых скобок и двойных кавычек имеет важное значение, поскольку позволяет компилятору находить нужные файлы на этапах компилирования и разрешения связей.

Предотвращение многократного включения заголовочных файлов

Каждый заголовочный файл включается в программу только один раз. Если заголовочный файл подключить более одного раза, то компилятор это воспримет как ошибку и выдаст сообщение об ошибке. Например, при многократном подключении компилятор может обнаружить переопределение класса, уже определённого во время первого подключения заголовочного файла; такое переопределение является ошибкой.

Включение заголовочных файлов для каждого необходимого класса может привести к многократному включению файлов. Например, если в программе используются классы DCMotor и StepperMotor, то нужно включить файлы dcmotor.h и stepper.h. Оба этих файла подключают файл motor.h. Таким образом, файл motor.h оказывается подключенным дважды. Есть механизм, позволяющий включать файлы каждого класса и в то же время предотвращающий многократное подключение заголовочного файла. Следующая процедура заключается в использовании флага состояния:

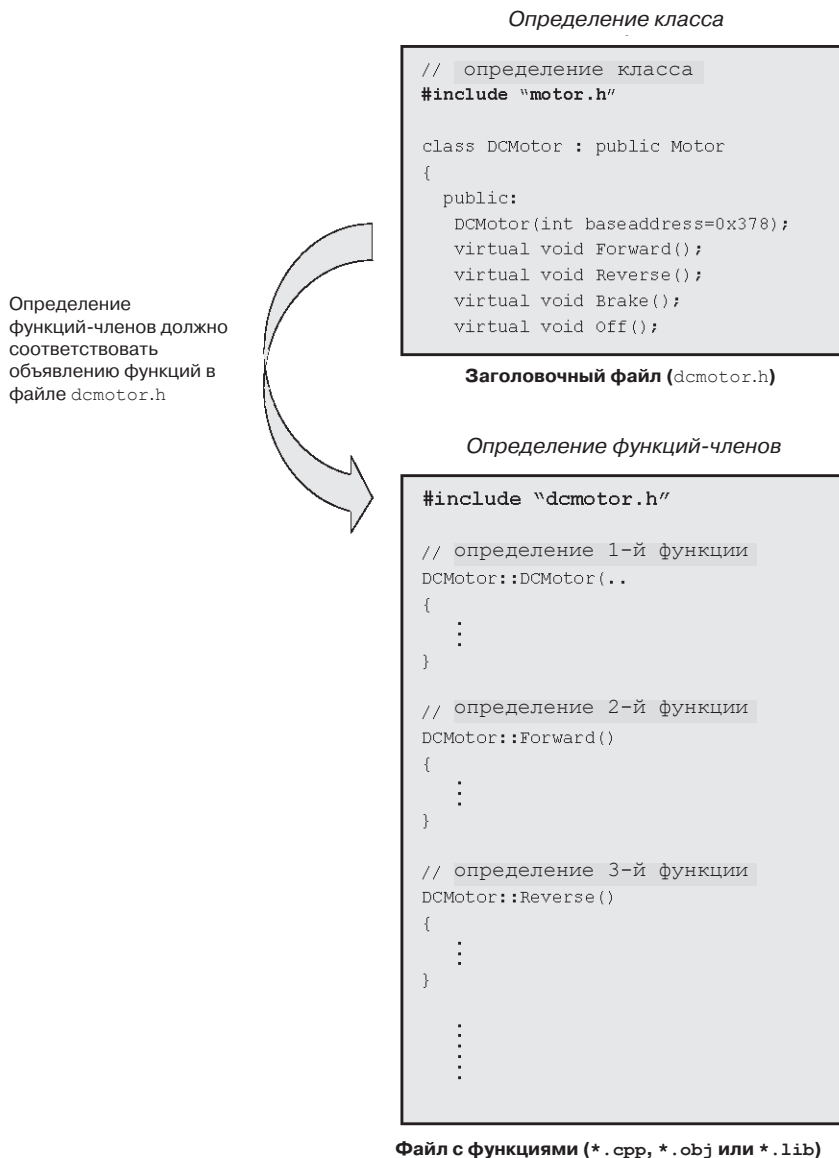


Рис. 9.4. Включение заголовочного файла класса в файл функций этого класса

1. До подключения файлов препроцессором флаг должен быть пассивном состоянии (т. е. файл ещё не включен).
2. Когда какой-либо заголовочный файл в первый раз, этот флаг проверяется и становится ясно, что этот файл ранее не подключался. Флаг переводится в активное состояние, означая, что файл уже подключен.

3. При последующих попытках подключения этого заголовочного файла тестирование флага выявит, что он уже был ранее подключен и отменит чтение файла препроцессором, исключив тем самым многократное подключение.

Заграждающие директивы заголовочных файлов

Заграждающие директивы являются директивами препроцессора. Они играют роль только что описанного флага состояния и не позволяют подключить один и тот же заголовочный файл более одного раза.

В следующем примере показано, как заграждающие директивы могут быть применены в заголовочном файле класса `AbstractMotor`.

Листинг 9.6. Заголовочный файл `AbsMotor.h`

```
#ifndef AbsmotorH
#define AbsmotorH

class AbstractMotor
{
    private:
        int Speed;

    public:
        AbstractMotor();
        void SetSpeed(int speed);
        int GetSpeed();
        virtual void Off() = 0;
        virtual void Forward() = 0;
        virtual void Reverse() = 0;
        virtual void Brake() = 0;
};

#endif
```

Первая директива препроцессора `#ifndef` означает «если не определено». Она похожа на выражение `if`, чьё тело начинается с `#ifndef`, а заканчивается в строке с `#endif`. Поэтому выражение:

```
#ifndef AbsmotorH
```

будет воспринято препроцессором в виде «если `AbsmotorH` не определён». Тело этого выражения будет рассмотрено только в том случае, если не было определён идентификатор `AbsmotorH`.

Идентификатор `AdsmotorH` не определялся в ходе обработки файлов программы. Поэтому, когда препроцессор встретит выражение `#ifndef AbsmotorH`, он выполнит его тело. Первой строкой в теле этого выражения является выражение:

```
#define AbsmotorH
```

Директива `#define` служит для определения идентификатора `AbsmotorH`. Идентификатор должен быть уникальным и не использоваться для обозначения

заголовочных файлов других классов. Неправильные имена идентификаторов могут привести к труднообнаруживаемым ошибкам в программе. Поскольку имена файлов в файловой системе компьютера уникальны, то лучшим выходом будет использовать их в качестве имён этих идентификаторов. Согласно такому принципу выбран идентификатор `AbsmotorH`, соответствующий заголовочному файлу `absmotor.h`. После того, как будут подключены строки тела выражения `#ifndef`, компилятор определит класс `AbstractMotor` и будет определён идентификатор `AbsmotorH`. Когда этот файл будет подключен ещё раз, условие выражения `#ifndef AbsmotorH` будет ложным и тело выражения будет проигнорировано, предотвратив тем самым повторное подключение.

9.4. Пример программы управления двигателями

В этом разделе будет рассмотрен процесс создания многофайловой программы на примере разработанной в главе 8 программы управления двигателями. Начнём с создания программных модулей для каждого класса программы. Каждый модуль будет представлен заголовочным файлом и файлом определения функций.

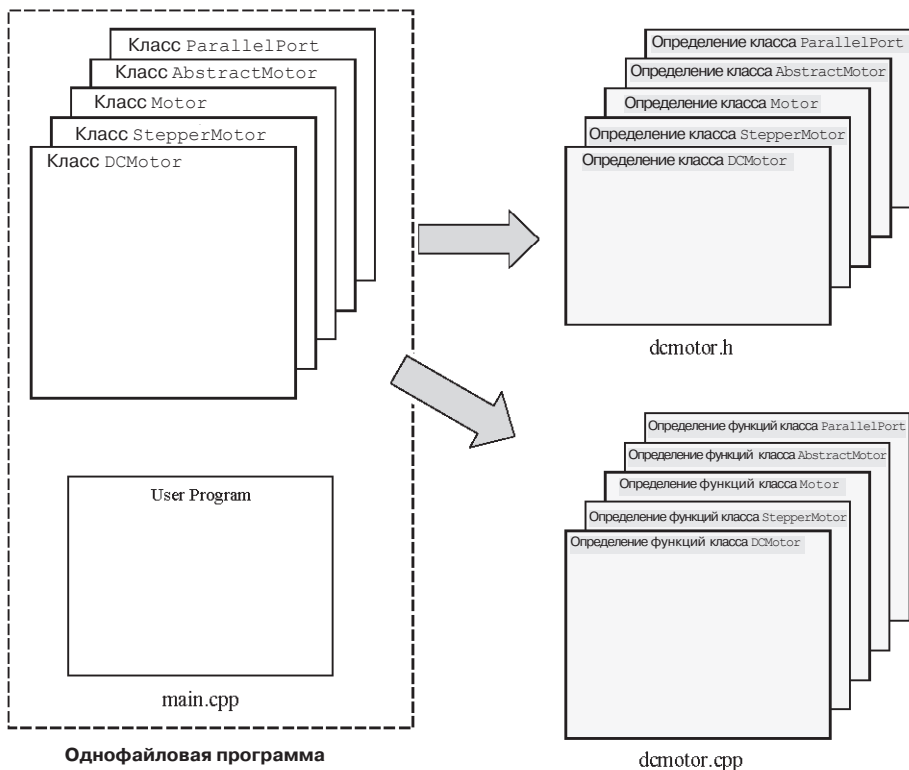


Рис. 9.5. Формирование многофайловой программы

В левой части **Рис. 9.5** изображена однофайловая программа с её главной функцией `main()` и классами. Справа изображены заголовочные файлы классов и соответствующие им файлы с определениями функций. Программа из главы 8 (**Листинг 8.19**) представлена следующим образом. Каждый раздел кода программы скопирован в файл с соответствующим названием и расширением (заголовочный файл `*.h` и файл определения функций `*.cpp`). Файлы сохранены в том же каталоге, где находится исходный файл программы, чтобы сократить время их поиска.

Листинг 9.7. Разбиение программы управления двигателями

absmotor.h (заголовочный файл)

```
-----
#ifndef AbsmotorH
#define AbsmotorH

class AbstractMotor
{
    private:
        int Speed;

    public:
        AbstractMotor();
        void SetSpeed(int speed);
        int GetSpeed();
        virtual void Off() = 0;
        virtual void Forward() = 0;
        virtual void Reverse() = 0;
        virtual void Brake() = 0;
        virtual ~AbstractMotor() {}
};

#endif
-----
```

absmotor.cpp (файл определения функций)

```
-----
#include "absmotor.h"

AbstractMotor::AbstractMotor()
{
    Speed = 0;
}

void AbstractMotor::SetSpeed(int speed)
{
    Speed = speed;
    if (Speed > 255) Speed = 255; // Ограничение максимального значения
    if (Speed < 0) Speed = 0; // Ограничение минимального значения
}

int AbstractMotor::GetSpeed()
```

```
{
    return Speed;
}
```

 pport.h (заголовочный файл)

```
#ifndef PportH
```

```
#define PportH
```

```
class ParallelPort
```

```
{
    private:
        unsigned int BaseAddress;
        unsigned char InDataPort1;

    public:
        ParallelPort();
        ParallelPort(int baseaddress);
        void WritePort0(unsigned char data);
        void WritePort2(unsigned char data);
        unsigned char ReadPort1();
        virtual ~ParallelPort() {}
};
```

```
#endif
```

 pport.cpp (файл определения функций)

```
#include <dos.h>
```

```
#include "pport.h"
```

```
ParallelPort::ParallelPort()
```

```
{
    BaseAddress = 0x378;
    InDataPort1 = 0;
}
```

```
ParallelPort::ParallelPort(int baseaddress)
```

```
{
    BaseAddress = baseaddress;
    InDataPort1 = 0;
}
```

```
void ParallelPort::WritePort0(unsigned char data)
```

```
{
    outportb(BaseAddress, data);
}
```

```
void ParallelPort::WritePort2(unsigned char data)
```

```
{
```



```

    outportb(BaseAddress+2,data ^ 0x0B);
}

unsigned char ParallelPort::ReadPort1()
{
    InDataPort1 = inportb(BaseAddress+1);
    // Инвертируем старший бит для компенсации внутренней
    // инверсии схемы порта принтера.
    InDataPort1 ^= 0x80;
    // Используем маску (или фильтр) для обнуления
    // неиспользуемых битов D0, D1 и D2.
    InDataPort1 &= 0xF8;
    return InDataPort1;}
}

```

motor.h (заголовочный файл)

```

#ifndef MotorH
#define MotorH

#include "absmotor.h"
#include "pport.h"

class Motor : public AbstractMotor, public ParallelPort
{
    public:
        Motor(int baseaddress = 0x378);
        void Off();
        virtual void Forward() = 0;
        virtual void Reverse() = 0;
        virtual void Brake() = 0;
        virtual ~Motor() {}
};

#endif

```

motor.cpp (файл определения функций)

```

#include "motor.h"

Motor::Motor(int baseaddress) : ParallelPort(baseaddress)
{
    Off();
}

void Motor::Off()
{
    WritePort0(0x00);
}

```

dcmotor.h (заголовочный файл)

```
-----
#ifndef DcmotorH
#define DcmotorH

#include "motor.h"

class DCMotor : public Motor
{
    public:
        DCMotor(int baseaddress = 0x378);
        virtual void Forward();
        virtual void Reverse();
        virtual void Brake();
};

#endif
-----
```

dcmotor.cpp (файл определения функций)

```
-----
#include "dcmotor.h"

DCMotor::DCMotor(int baseaddress) : Motor(baseaddress)
{
}

void DCMotor::Forward()
{
    int j;
    for(j = 0; j < GetSpeed(); j++)
        WritePort0(0x09);
    for( ; j < 256; j++)
        WritePort0(0x00);
}

void DCMotor::Reverse()
{
    int j;
    for(j = 0; j < GetSpeed(); j++)
        WritePort0(0x06);
    for( ; j < 256; j++)
        WritePort0(0x00);
}

void DCMotor::Brake()
{
    WritePort0(0x0C);
}
-----
```

stepper.h (заголовочный файл)

```
#ifndef StepperH
#define StepperH

#include "motor.h"

enum MOTORTYPE {UPFS, UPHS, BPFS, BPHS};

class StepperMotor : public Motor
{
    private:
        MOTORTYPE MotorType;
        unsigned char Switching[8];
        int CycleIndex;
        int MaxIndex;

    public:
        StepperMotor(MOTORTYPE motortype = UPFS,
                     int baseaddress = 0x378);
        virtual void Forward();
        virtual void Reverse();
        virtual void Brake();
};

#endif
```

stepper.cpp (файл определения функций)

```
#include <dos.h>

#include "stepper.h"

StepperMotor::StepperMotor(MOTORTYPE motortype,
                           int baseaddress) : Motor(baseaddress)
{
    MotorType = motortype;
    CycleIndex = 0;

    switch(MotorType)
    {
        case UPFS : MaxIndex = 4;
                    Switching[0] = 0x11;
                    Switching[1] = 0x12;
                    Switching[2] = 0x22;
                    Switching[3] = 0x21;
                    break;
        case UPHS : MaxIndex = 8;
                    Switching[0] = 0x01;
                    Switching[1] = 0x11;
```

```

        Switching[2] = 0x10;
        Switching[3] = 0x12;
        Switching[4] = 0x02;
        Switching[5] = 0x22;
        Switching[6] = 0x20;
        Switching[7] = 0x21;
        break;
    case BPFS : MaxIndex = 4;
        Switching[0] = 0x99;
        Switching[1] = 0x69;
        Switching[2] = 0x66;
        Switching[3] = 0x96;
        break;
    case BPHS : MaxIndex = 8;
        Switching[0] = 0x99;
        Switching[1] = 0x09;
        Switching[2] = 0x69;
        Switching[3] = 0x60;
        Switching[4] = 0x66;
        Switching[5] = 0x06;
        Switching[6] = 0x96;
        Switching[7] = 0x90;
    }
}

void StepperMotor::Forward()
{
    if(++CycleIndex == MaxIndex) CycleIndex = 0;
    WritePort0(Switching[CycleIndex]);
    delay(259-GetSpeed());
}

void StepperMotor::Reverse()
{
    if(--CycleIndex == -1) CycleIndex = MaxIndex - 1;
    WritePort0(Switching[CycleIndex]);
    delay(259-GetSpeed());
}

void StepperMotor::Brake()
{
    switch(MotorType)
    {
        case UPFS : case UPHS :
            WritePort0(0x11);
            break;
        case BPFS : case BPHS :
            WritePort0(0x99);
    }
}
}-----

```

main.cpp (программа пользователя)

```

-----
/*****
Программа управления двигателями с использованием
виртуальных функций (глава 8)
*****/

#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

#include "motor.h"
#include "dcmotor.h"
#include "stepper.h"

void main()
{
    Motor *MotorPtr;
    int Selection;

    clrscr();

    cout << endl << "    MOTOR MENU";
    cout << endl << "    -----" << endl;
    cout << "    1  DC Motor" << endl;
    cout << "    2  UPFS" << endl;
    cout << "    3  UPHS" << endl;
    cout << "    4  BPFS" << endl;
    cout << "    5  BPHS" << endl;
    cout << "    6  QUIT" << endl;
    cout << endl;
    cout << "    Select the MOTOR Number: ";

    cin >> Selection;

    switch(Selection)
    {
        case 1 : MotorPtr = new DCMotor;
                break;
        case 2 : MotorPtr = new StepperMotor(UPFS);
                break;
        case 3 : MotorPtr = new StepperMotor(UPHS);
                break;
        case 4 : MotorPtr = new StepperMotor(BPFS);
                break;
        case 5 : MotorPtr = new StepperMotor(BPHS);
                break;
    }
}

```

```

        case 6 : return;

default: cout << endl;
        cout << "    Unspecified Motor type..."
        cout << "    PRESS a key to END Program!";
        getch();
        exit(1); // Завершение программы
    }

    if(MotorPtr == NULL)
    {
        cout << "Memory allocation failed" << endl;
        getch();
        exit(1);
    }

    cout << "*****" << endl;
    cout << "* CONNECT BOARD POWER SUPPLY NOW *" << endl;
    cout << "*****" << endl;
    cout << endl;
    cout << "    After connecting power,";
    cout << "    press a key to continue" << endl;
    getch();
    cout << "    Keypress changes Speed/Rotation (& Braking)." << endl;

    //..... Здесь начинается управление двигателем .....
    MotorPtr->SetSpeed(150);
    while(!kbhit()) MotorPtr->Forward();
    getch(); // Очистка буфера клавиатуры

    MotorPtr->SetSpeed(255);
    while(!kbhit()) MotorPtr->Forward();
    getch();

    MotorPtr->SetSpeed(150);
    while(!kbhit()) MotorPtr->Reverse();
    getch();

    MotorPtr->SetSpeed(255);
    while(!kbhit()) MotorPtr->Reverse();
    getch();

    cout << endl << "    Braking Applied!" << endl;
    while(!kbhit()) MotorPtr->Brake();
    getch();

    MotorPtr->Off();
    //..... Здесь завершается управление двигателем .....
    // Освободим память, занимаемую объектом двигателя
    delete MotorPtr;
}
-----

```

Файл главной функции

При разработке функции `main()` программист использует объекты иерархии классов. Программисту нужно знать только интерфейсы этих объектов, т. е. публичные члены классов. Таким образом, можно пользоваться функциями-членами в главной функции в соответствии с их определениями в классах. Пользуясь этими классами, программисту не нужно знать подробности реализации функций-членов. С другой стороны, компилятор должен располагать полной информацией о классе и его базовых классах, если они есть. Мы выполняем это требование путём подключения соответствующих заголовочных файлов.

В файлах `dcmotor.h` и `stepper.h` подключается только `motor.h`. В файле `motor.h`, в свою очередь, подключаются файлы `absmotor.h` и `pport.h`. Поэтому, когда компилятор подключает файл `dcmotor.h`, то файлы `absmotor.h`, `pport.h` и `motor.h` к этому времени являются уже просмотренными. Эти файлы содержат определения всех необходимых базовых классов для определения классов `DCMotor` и `StepperMotor`.

В функции `main()` используются объекты классов `Motor`, `DCMotor` и `StepperMotor`. Поэтому в функции `main()` должны быть подключены три соответствующих заголовочных файла. Имена файлов указаны в двойных кавычках, что заставляет компилятор искать их в текущем каталоге и каталогах, указанных в путях поиска. Файлы, имена которых указаны в угловых скобках (`<` и `>`), созданы не нами. Они находятся в каталоге `include` пакета `C/C++`.

Функция `main()` является идеальным местом проверки заграждающих директив заголовочных файлов. Заголовочные файлы `dcmotor.h` и `stepper.h` оба подключают `motor.h`. Поэтому в программе этот файл оказывается подключенным трижды: один раз явно в функции `main()`, и два раза неявно из файлов `dcmotor.h` и `stepper.h`. Но это не означает, что класс `Motor` будет определяться три раза. Первый просмотр файла `motor.h` определит класс и заграждающий идентификатор `MotorH`. Все остальные попытки подключения `motor.h` будут пресекаться препроцессором, потому что идентификатор `MotorH` уже определён. Поэтому в случае последующих двух неявных подключениях `motor.h` в файлах `dcmotor.h` и `stepper.h` повторного подключения не произойдёт. Из этого не следует, что файлы `dcmotor.h` и `stepper.h` окажутся неработоспособными. Они будут пользоваться уже известным компилятору определением класса `Motor` (из первого подключения файла `motor.h`), а значит и определяют классы `DCMotor` и `StepperMotor`.

Создание библиотек

В процессе разрешения связей для формирования исполняемого файла используются файлы библиотек. Библиотечные файлы образованы объединением нескольких объектных файлов в один, как правило, с расширением `.lib`. В пакетах `C/C++` есть утилиты для создания библиотечных файлов. Библиотечный файл будет содержать откомпилированные определения всех функций, имеющих в объектных файлах. Файл библиотеки является двоичным (бинарным) и не может быть прочитан программистом.

Файлы проектов и make-файлы

Процесс компиляции и разрешения связей многофайловой программы более сложен, чем в случае однофайловой программы. Но этот процесс можно легко

автоматизировать двумя способами. В первом случае создаётся *файл проекта*, в котором перечислены все файлы, необходимые для создания исполняемого файла. В файл проекта включаются все исходные файлы, библиотечные файлы и объектные файлы. Следует отметить, что заголовочные файлы не включаются в файл проекта. Препроцессор подключит заголовочные файлы во время компилирования исходных файлов.

В другом случае создают так называемый *make-файл*. Этот файл обрабатывается утилитой *make*, вызывающей компилятор и линкер в соответствии с содержащимися в make-файле командами.

Файлы проекта

Создадим файл проекта, например, с именем `drive.*` для компиляции программы управления двигателями. Расширение файла проекта определяется используемым компилятором. Например, для Inprise™ Borland C++ для DOS это будет `.prj`, а для Microsoft™ Visual C++ `.dsp`. Теперь есть три разных варианта обработки файлов проекта:

Вариант 1:

```
main.cpp
absmotor.cpp
pport.cpp
motor.cpp
dcmotor.cpp
stepper.cpp
```

После обработки файла проекта все файлы `.cpp` будут откомпилированы в соответствующие файлы `.obj`. Затем эти `.obj` файлы будут слинкованы с системными файлами `.obj` и/или `.lib` в конечный исполняемый файл.

Если все файлы, соответствующие классам, уже имеются в виде `.obj` файлов, то файл проекта будет выглядеть следующим образом:

Вариант 2:

```
main.cpp
absmotor.obj
pport.obj
motor.obj
dcmotor.obj
stepper.obj
```

Файл `.cpp` откомпилируется в `.obj` файл. Затем все `.obj` файлы будут слинкованы с системными файлами `.obj` и/или `.lib` в `.exe` файл.

Файлы классов иерархии можно объединить в один `.lib` файл. Предположим, что был создан библиотечный файл `motors.lib`. Тогда файл проекта будет таким:

Вариант 3:

```
main.cpp
motors.lib
```

В этом случае `.exe` файл будет образован путём компиляции `.cpp` файла и линкировании его с `motors.lib`, а также другими системными `.obj` и/или `.lib` файлами.

Make-файлы

Другим средством автоматизации компилирования и линкирования является создание make-файла, содержащего последовательность команд компилирования и разрешения связей, необходимых для создания исполняемого файла. В make-файле можно создать команды, выполняющие необходимые действия, например, только компилирование, только линкирование, компилирование и линкирование и т. п.

Ниже в **Листинге 9.8** приведён make-файл, собирающий файл `drive.exe` программы управления двигателем. Заменяя слова `CC` вызовами компилятора `C++` получим работающий make-файл. Имена файлов должны соответствовать реальным именам файлов. Например, если компилятор вместо файла `pport.obj` генерирует `pport.o`, то все расширения `.obj` нужно заменить на `.o`.

В make-файле описываются *зависимости файлов* и командные строки компилятора. Зависимости должны начинаться с крайнего левого столбца, как показано в следующем фрагменте из **Листинга 9.8**:

```
drive: driv1.obj pport.obj dcmotor.obj stepper.obj motor.obj absmotor.obj
```

Это выражение говорит утилите `make`, что если хотя бы один из перечисленных после двоеточия файлов будет изменён, то исполняемый файл `drive.exe` нужно собирать заново.

Листинг 9.8. Пример make-файла для сборки `drive.exe`

```
# make-файл программы управления двигателями
drive: driv1.obj pport.obj dcmotor.obj stepper.obj
      motor.obj absmotor.obj
      CC -edrive driv1.obj pport.obj dcmotor.obj
        stepper.obj motor.obj absmotor.obj
driv1.obj : driv1.cpp motor.h stepper.h dcmotor.h
motor.h : pport.h absmotor.h
stepper.h : motor.h
dcmotor.h : motor.h
      CC -c driv1.cpp
pport.obj : pport.cpp pport.h
      CC -c pport.cpp
dcmotor.obj : dcmotor.cpp dcmotor.h
      CC -c dcmotor.cpp
stepper.obj : stepper.cpp stepper.h
      CC -c stepper.cpp
motor.obj : motor.cpp motor.h
      CC -c motor.cpp
absmotor.obj : absmotor.cpp absmotor.h
      CC -c absmotor.cpp
```

Командные строки компилятора должны начинаться после символа табуляции, а не крайнего левого столбца:

```
CC -edrive driv1.obj pport.obj dcmotor.obj
      stepper.obj motor.obj absmotor.obj
```

Эти команды инструктируют утилиту `make`, как создать файл `drive.exe`. Ключ `-e` задаёт имя исполняемого файла, в данном случае `drive.exe`. Остальные выражения в `make`-файле имеют аналогичный смысл. Ключ `-c` означает, что требуется только компилирование файла.

Утилита `make` по умолчанию обрабатывает файл с именем `makefile`. Если файл из **Листинга 9.8** назвать `makefile`, то при помощи команды `make` можно собирать файл `drive.exe`.

9.5. Заключение

На первых этапах планирования программы её работа должна быть описана в виде псевдо-кода. Такое описание затем уточняется в ходе дальнейшей работы над программой путём введения необходимых объектов, которые затем определяются. Эти объекты могут быть представлены иерархией классов, что особенно эффективно, если задействован механизм виртуальных функций.

При вводе текста программы можно допустить опечатки. В таком случае компилятор обнаруживает ошибки и информирует об этом программиста. Вероятность появления ошибок можно уменьшить, воспользовавшись средствами копирования/вставки, имеющихся в современных редакторах. Операции копирования/вставки хорошо сочетаются с правильным оформлением текста классов. Таким образом, упрощается определение функций и уменьшается вероятность возникновения ошибок. Распространённой ошибкой также является неправильная расстановка вложенных скобок. Были продемонстрированы несколько способов, помогающих избежать этих проблем. Отступы нужны для обозначения уровня вложенности, облегчают чтение текста программы, а поэтому снижают риск появления таких ошибок в исходном тексте.

Модульный подход подразумевает создание заголовочного файла и файла определения функций для каждого класса. При распространении классов или использовании их в программах, нужно распространять или использовать только лишь соответствующие модули. Так как модульные программы состоят из множества файлов, то формирование исполняемого файла оказывается более сложной задачей, чем в случае однофайловой программы. Исполняемый файл модульной программы собирается при помощи либо файла проекта, либо утилиты `make`, автоматизирующих этот процесс.

9.6. Литература

1. Meyer, B., *Object Oriented Software Construction*, Prentice Hall, 1988.
2. Staugaard A. C. (Jr), *Structured and Object Oriented Techniques*, Prentice Hall, 1997.
3. Lafore, R. *Object Oriented Programming in MICROSOFT C++*, Waite Group Press, 1992.
4. Wang, P. S., *C++ with Object Oriented Programming*, PWS Publishing, 1994.
5. Winston, P. H., *On to C++*, Addison Wesley, 1994.

10 Измерение напряжения и температуры

Содержание главы:

- Преобразование напряжения в частоту (ПНЧ) при помощи генератора, управляемого напряжением (ГУН).
- Восприятие температуры при помощи термисторов.
- Класс для ГУН.
- Подсчёт количества импульсов.
- Программирование графики.
- Программы измерения напряжения и температуры.

10.1. Введение

В этой главе описываются средства преобразования аналогового напряжения в последовательность импульсов, частота которых пропорциональна приложенному напряжению. Такой способ даёт отличную возможность измерения аналогового сигнала при помощи одного цифрового входа. Для этого будет использовано устройство, известное как генератор, управляемый напряжением, ГУН (*Voltage-Controlled Oscillator, VCO*), выполняя тем самым «аналого-цифровое преобразование».

Будет разработан код, измеряющий период следования импульсов для определения входного напряжения. Также будет рассмотрена работа термочувствительного резистора, и с его помощью будет измеряться температура. В этой главе дано введение в программирование графики на примере разработки программы, отображающей импульсы на экране.

10.2. Преобразование напряжения в последовательность импульсов

Одним из простейших аналого-цифровых преобразователей является *преобразователь напряжение-частота* (ПНЧ). Преобразователь напряжение-частота генерирует импульсы, частота которых пропорциональна входному напряжению. Особым типом ПНЧ является *генератор, управляемый напряжением* (ГУН), генерирующий либо синусоидальный сигнал, либо прямоугольные импульсы.

Как видно из **Рис. 10.1**, работа типового ПНЧ основана на заряде конденсатора C током, пропорциональным входному напряжению.

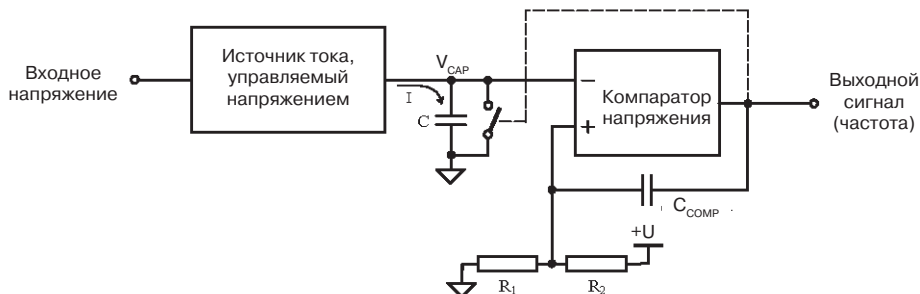


Рис. 10.1. Типовая схема преобразователя напряжение-частота

Выход компаратора, имеющий на выходе в исходном состоянии положительное напряжение, меняет состояние на противоположное всякий раз, когда напряжение на конденсаторе C , приложенное к инвертирующему входу, превысит напряжение на прямом (неинвертирующем) входе. Когда это происходит, на выходе компаратора появляется отрицательное напряжение и замыкается цепь разряда конденсатора C . При этом происходит разряд конденсатора C , а к RC -цепочке, подключенной к прямому входу компаратора, прикладывается отрицательное напряжение.

Затем напряжение на неинвертирующем входе начинает возрастать, со временем достигая «нуля» на разряженном конденсаторе. Когда это произойдет, на выходе компаратора появится положительное напряжение. Цепь разряда конденсатора размыкается, позволяя току заряжать его до тех пор, пока напряжение на нём не превысит напряжение на прямом входе компаратора. На этом цикл преобразования напряжение-частота завершается. Этот процесс непрерывно повторяется, формируя выходные импульсы. При увеличении входного напряжения зарядный ток конденсатора C возрастает, а время достижения порогового напряжения уменьшается, приводя к увеличению частоты выходного сигнала.

На интерфейсной плате ГУН реализован на КМОП микросхеме фазовой автоподстройки частоты 4046. Схемы фазовой автоподстройки частоты имеют множество применений, но в данном случае это просто ГУН. Заметим, что выходной сигнал линеен только в диапазоне входного напряжения 1.5...3.5 В. Диапазон выходной частоты задаётся двумя резисторами и конденсатором.

10.3. Измерение температуры

Есть много типов электрических сенсоров, чувствительных к изменению температуры. Это *термисторы*, *термопары*, *термочувствительные конденсаторы*, *полупроводниковые диоды* и *кварцевые кристаллы*.

Термисторы являются наиболее распространёнными датчиками температуры и только этот тип сенсоров будет рассмотрен здесь. Они представляют собой обочные резисторы с очень большим *температурным коэффициентом сопротивления* (ТКС), обычно этот коэффициент отрицательный (*Negative Temperature Coefficient, NTC*). Отрицательный ТКС означает, что сопротивление с ростом температуры термистора уменьшается. Сопротивление от температуры изменяется

по экспоненциальному закону, что осложняет работу с ними. Но зато они дешёвы, обладают высокой чувствительностью и имеют небольшие размеры.

Простейшим способом измерения температуры термистором, является его подключение по схеме делителя напряжения, **Рис. 10.2**.

Выходное напряжение этой схемы с NTC-термистором с ростом температуры будет уменьшаться, как видно из **Рис. 10.3**. Форму кривой можно приблизить к прямой линии, если сопротивление резистора смещения выбрать равным сопротивлению термистора на середине температурного диапазона. Если схеме, снимающей напряжение с делителя требуется значительный входной ток (в нашем случае это ГУН), то может потребоваться буферирование выходного сигнала. Нужно помнить, что для достижения корпусом термистора температуры предмета, к которому он приложен, или окружающей среды, нужно некоторое время. Также температура корпуса может увеличиваться, если через термистор протекает слишком большой ток.

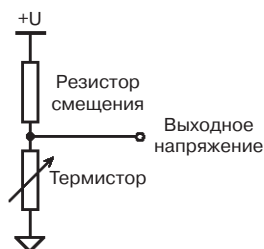


Рис. 10.2. Схема делителя напряжения с термистором

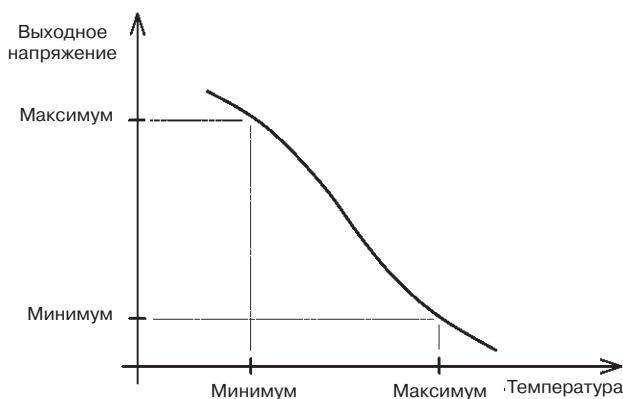


Рис. 10.3. Типичная зависимость напряжения от температуры NTC-термистора в схеме делителя напряжения

10.4. Класс ГУН

Выходное напряжение делителя напряжения с термистором подаётся на вход ГУН. Это напряжение определяет частоту выходного сигнала (а значит и его период). Чем больше входное напряжение ГУН, тем больше будет частота его выходного сигнала. Если разработать метод измерения периода следования импульсов (т. е. время полного цикла), то можно будет вычислить частоту. Как и предполагалось, выходной сигнал ГУН будет подаваться на параллельный порт компьютера.

Можно создать новый класс, который можно будет использовать в других программах, использующих ГУН. Такой объект должен обрабатывать принимаемый

параллельным портом сигнал таким образом, чтобы измерить его период посредством опроса его уровня. Под уровнем сигнала подразумеваются ВЫСОКИЙ или НИЗКИЙ логические уровни сигнала в любой момент времени. Новый объект можно создать на базе класса `ParallelPort`. Определение нового класса `VCO` приведено в **Листинге 10.1**.

Листинг 10.1. Определение класса `VCO` в файле `vco.h`

```
#ifndef VcoH
#define VcoH

#include "pport.h"

enum BITNUMBER {Bit7 = 0x80, Bit6 = 0x40,
                Bit5 = 0x20, Bit4 = 0x10,
                Bit3 = 0x08};

class VCO : public ParallelPort
{
private:
    long int Period;
    BITNUMBER Bit;

public:
    VCO(int baseaddress = 0x378, BITNUMBER bit = Bit3);
    long int MeasurePeriod();
    long int GetPeriod();
    int SignalLevel();
    virtual ~VCO() {};
};

#endif
```

В классе `VCO` определены два члена данных и пять функций-членов. В члене данных `Period` будет храниться измеренный период. Чтение сигнала ГУН со входа параллельного порта будет осуществляться через регистр `BASE+1`. Сигнал ГУН может подаваться на любую из входных линий регистра `BASE+1`, с 3-й по 7-ю. Чтобы пользователь мог выбирать линию подключения ГУН, в конструктор введён параметр, позволяющий указать номер бита (следовательно, и входную линию порта). Для этого определён перечислимый тип `BITNUMBER`, идентификаторам которого присвоены целые значения. Смысл значений идентификаторов станет ясен, когда будут определяться функции-члены. Константа `Bit3` выбрана в качестве значения по умолчанию, которое будет определять линию подключения ГУН.

Перед тем, как определить функции-члены класса `VCO`, нужно выяснить способ измерения периода импульсов. Следить за логическим сигналом выхода ГУН можно путём опроса регистра `BASE+1` и отфильтровывания ненужных битов. Начало импульса можно зафиксировать, обнаружив изменение сигнала от НИЗКОГО уровня к ВЫСОКОМУ или наоборот. После обнаружения начала

импульса можно приступать к измерению периода импульса. Сигнал должен претерпеть ещё два изменения в периоде, **Рис. 10.4**.

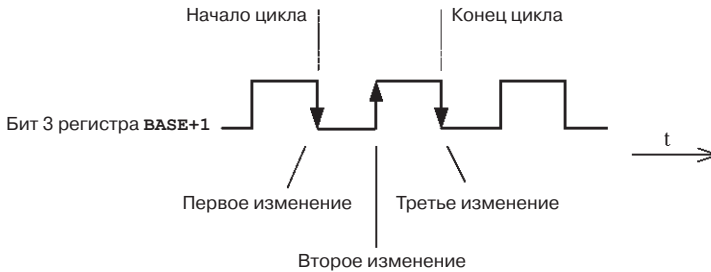


Рис. 10.4. Обнаружение полного цикла

Поскольку уровень сигнала очень важен, то сначала определим функцию-член `SignalLevel()`.

Листинг 10.2. Функция-член `SignalLevel()`

```
int VCO::SignalLevel()
{
    if(ReadPort1() & Bit)
        return 1;
    else
        return 0;
}
```

В любой момент времени уровень сигнала будет либо **ВЫСОКИЙ**, либо **НИЗКИЙ**. Обозначить эти уровни можно либо единицей, либо нулём, соответственно. Функция `SignalLevel()` из **Листинга 10.2** возвращает 0 или 1, в зависимости от уровня сигнала ГУН.

Сначала сигнал считывается из регистра `BASE+1` унаследованной функцией `ReadPort1()`. Операция AND над считанным байтом и маской `Bit` отфильтровывает все биты, кроме одного, который используется для чтения сигнала ГУН. Если сигнал **ВЫСОКИЙ**, то AND над прочитанным байтом и `Bit` будет равен `Bit`, т. е. ненулевому значению, условие выражения `if` будет истинным и будет возвращена 1. Если уровень будет **НИЗКИЙ**, то результат операции AND будет нулевым и функция вернёт 0.

Изменение сигнала можно отследить путём непрерывного опроса порта, проверяя, не произошло ли изменения бита используемой линии. Измерение периода начинается после обнаружения изменения сигнала и продолжается до тех пор, пока не будет обнаружено ещё два изменения сигнала. Идеальным средством измерения периода импульсов было применение средств реального времени, однако они ещё не обсуждались (глава 13). Вместо этого воспользуемся подсчётом количества итераций цикла считывания порта в течении одного периода. Если известно время выполнения операций чтения порта и фильтрации битов, то можно вычислить фактическую длительность периода. Тем не менее, время выполнения

будет разным для разных компьютеров, а также на него будут оказывать влияние системные события.

Для измерения периода нужно выполнить следующие шаги (подразумевается, что сигнал подключен к биту 3):

1. Обнулить счётчик.
2. Считывать в цикле уровень сигнала, пока не будет обнаружено изменение состояния бита 3.
3. Считывать в цикле уровень сигнала, пока не будет обнаружено второе изменение состояния бита 3.
4. Считывать в цикле уровень сигнала, пока не будет обнаружено третье изменение состояния бита 3.
5. Вернуть значение счётчика в качестве длительности периода.

Эти шаги (т. е. псевдо-код) можно использовать для определения функции-члена `MeasurePeriod()`. Рассмотрим этот псевдо-код. Шаги 2, 3 и 4 очень похожи. Начнём с кодирования шага 2. Тогда можно будет полученный код использовать для шагов 3 и 4, а после каждого чтения порта будем инкрементировать счётчик.

Шаг 2 псевдокода можно реализовать, как показано в **Листинге 10.3**. В функции `SignalLevel()` используется унаследованная от класса `ParallelPort` функция `ReadPort1()`.

Листинг 10.3. Обнаружение изменения сигнала ГУН

```
VCO Vco;
unsigned char Signal;

Signal = Vco.SignalLevel();
while(Signal == Vco.SignalLevel());
```

Сначала функцией `SignalLevel()` считывается текущий уровень сигнала и запоминается в переменной `Signal`. Затем запускается цикл `while`, осуществляющий повторяющиеся считывания уровня сигнала путём вызова функции `SignalLevel()` и сравнивающий полученное значение с сохранённым уровнем сигнала. Если они равны, то цикл `while` продолжает выполняться. Тело цикла пустое. Как только будет обнаружено изменение уровня сигнала, условие цикла станет ложным и цикл `while` завершится.

Определения всех функций-членов класса `VCO` приведены в **Листинге 10.4**. Функция `MeasurePeriod()` измеряет период сигнала и возвращает полученное значение. Функция `GetPeriod()` только лишь считывает и возвращает значение собственного члена данных `Period`. На этом создание класса `VCO` завершено. В следующем разделе мы научимся измерять напряжение с помощью объекта класса `VCO`.

Листинг 10.4. Определения функций-членов класса `VCO` в файле `vco.cpp`

```
#include "vco.h"

VCO::VCO(int baseaddress, BITNUMBER bit)
    : ParallelPort(baseaddress)
{
```



```
    Bit = bit;
    Period = 0;
}

long int VCO::MeasurePeriod()
{
    unsigned char Signal;
    Period = 0;

    Signal = SignalLevel();
    while(Signal == SignalLevel()); // пустое тело цикла

    Signal = SignalLevel();
    while(Signal == SignalLevel())
        Period++;

    Signal = SignalLevel();
    while(Signal == SignalLevel())
        Period++;

    return Period;
}

long int VCO::GetPeriod()
{
    return Period;
}

int VCO::SignalLevel()
{
    if(ReadPort1() & Bit)
        return 1;
    else
        return 0;
}
```

10.5. Измерение напряжения с помощью ГУН

Как говорилось в разделе 10.2, генератор, управляемый напряжением (ГУН) является электронной схемой, которая генерирует прямоугольные импульсы, частота которых пропорциональна входному напряжению. Для проверки его работы нужно подать на его вход несколько разных уровней напряжения. Проще всего это сделать, подключив вход ГУН к выходу потенциометра, размещённого на интерфейсной плате. Напряжение потенциометра будем измерять вольтметром, чтобы можно было оценить соответствие между входным напряжением и выходным сигналом ГУН. Другой способ не требует использования вольтметра, потому что вместо потенциометра подключается ЦАП, устанавливающий на входе ГУН из-

вестный уровень сигнала. Управлять выходным напряжением ЦАП можно записью в него чисел, как это делалось в главе 6.

Для простоты использования программы в функции `main()` задействовано следующее управление с клавиатуры:

- Нажатие стрелки «ВВЕРХ» увеличивает выходное напряжение ЦАП.
- Нажатие стрелки «ВНИЗ» уменьшает выходное напряжение ЦАП.
- Нажатие «Alt+X» завершает выполнение программы.

Фрагмент-заготовка кода, реализующего эти действия, приведён в **Листинге 10.5**.

Листинг 10.5. Реализация управления с клавиатуры

```
int Quit = 0, key;

while(!Quit)
{
    // Здесь нужно вставить код измерения
    // и отображения периода импульсов

    if(bioskey(1) != 0) // проверка, не было ли нажатия кнопки
    {
        key = bioskey(0); // считывание кода кнопки

        switch(key)
        {
            /* Alt-X */ case 0x2d00 : Quit = 1;
                               break;
            /* ВВЕРХ */ case 0x4800 : // увеличить выходное напряжение ЦАП;
                               break;
            /* ВНИЗ */ case 0x5000 : // уменьшить выходное напряжение ЦАП;
            }
        }
    }
}
```

В дальнейшем в цикле `while` будет находится код, измеряющий и отображающий значение периода. Этот код не показан, на его месте пока комментарий. Далее в цикл `while` следует выражение `if`, содержащее выражение `switch`. Истинная ветвь выражения `if` будет выполняться в том случае, если условие этого выражения будет истинным, т. е. когда будет нажата какая-нибудь кнопка. Функция `bioskey()` с аргументом 1 вернёт значение, равное «истине», если была нажата кнопка. Если не было нажатия на кнопку, то будет возвращено значение «ложь». Функция `bioskey()` *не ждёт* нажатия на кнопку. Это значит, что если кнопка не нажималась, то тело выражения `if` будет пропущено, и будут выполняться следующие выражения цикла `while` (например, измерение и отображение периода импульсов).

Если произошло нажатие кнопки, то будет выполняться тело выражения `if`. В начале истинной ветви определяется, какая именно кнопка была нажата. Для этого можно воспользоваться функцией `bioskey()` с аргументом 0. Теперь функция `bioskey()` вернёт код нажатой кнопки. В выражениях вариантов `switch` перечислены все возможные значения кнопок: стрелка «ВВЕРХ», стрелка «ВНИЗ»

и «Alt+X». Выражение `switch` выбирает соответствующий нажатым кнопкам вариант и выполняет его. Если не было найдено подходящего варианта, то никаких действий не предпринимается. В случае нажатия «Alt+X» переменной `Quit` присваивается 1. Тогда условие цикла `while` становится ложным и выполнение цикла прекращается. Код обработки нажатия стрелок «ВВЕРХ» и «ВНИЗ» пока не определён, на его месте находится комментарий.

Функция `main()`, управляющая выходом ЦАП и измеряющая период ГУН, приведена в **Листинге 10.6**. На месте комментариев **Листинга 10.5** теперь находится код программы. Заметим, что программа работает и с ГУН, и с ЦАП. Поэтому в начале функции `main()` инстанцируются объекты `Dac` и `Vco` классов `DAC` и `VCO`, соответственно.

Листинг 10.6. Измерение периода импульсов ГУН
(ЦАП задаёт входной сигнал ГУН) в файле `period.cpp`

```
#include <bios.h>
#include <conio.h>

#include "dac.h"
#include "vco.h"

void main()
{
    DAC Dac;
    VCO Vco;
    int Quit = 0, key;
    unsigned char DACbyte;

    clrscr();
    Dac.SendData(0); // нулевое начальное значение

    while(!Quit);
    {
        gotoxy(10,10);
        cprintf("The pulse period is %10lu\\a",
               Vco.MeasurePeriod()/1000);

        if(bioskey(1) != 0)
        {
            DACbyte = DAC.GetLastOutput();
            key = bioskey(0);
            switch(key)
            {
                /* Alt-X */ case 0x2d00 : Quit = 1;
                               break;
                /* ВВЕРХ */ case 0x4800 :
                               if(DACbyte > 247) // ограничение максимального
                                   DACbyte = 247; // значения
                               Dac.SendData(DACbyte + 8);
            }
        }
    }
}
```

```

        break;
/* ВНИЗ */ case 0x5000 :
        if(DACbyte < 8) // ограничение минимального
            DACbyte = 8; // значения
        Dac.SendData(DACbyte - 8);
    }
}
}
}

```

Функция `gotoxy()` позиционирует курсор положение с экранными координатами (10,10). Первое число в скобках соответствует координате **x**, она отсчитывается с левой стороны экрана. Второе число соответствует координате **y**, её начало вверху экрана. Система экранных координат приведена на **Рис. 10.5**. Поэтому измеренный период импульсов будет отображаться, начиная с позиции (10,10). Функция `cprintf()` аналогична функции `printf()`, рассмотренной ранее, только она не преобразовывает символ новой строки (`\n`) в комбинацию символов новой строки и возврата каретки (`\n\r`). Это сделано специально для вывода на экран. В нашем случае позиционирование курсора осуществляется функцией `gotoxy()`, и поэтому нет необходимости в переводе каретки или переходе на новую строку.

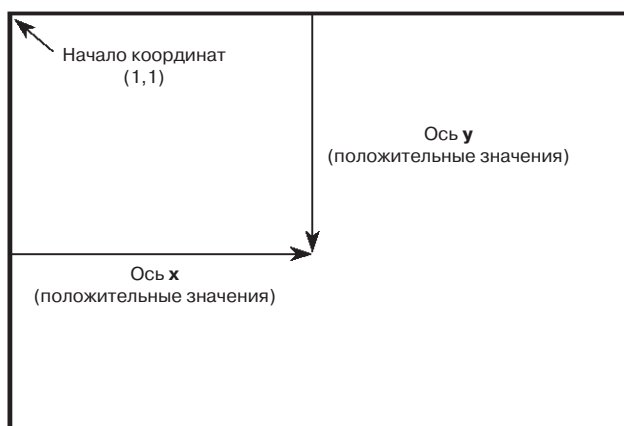


Рис. 10.5. Экранные координаты в текстовом режиме

Измеренное значение является значением периода сигнала в условных единицах. Это значение возвращает функция `MeasurePeriod()` объекта `Vco`. Обращение к функции `MeasurePeriod()` происходит во время вызова функции `cprintf()`. Функция `cprintf()` печатает на экране измеренное значение, делённое на 1000.

Предыдущее значение, выведенное в ЦАП, возвращается функцией `GetLastOutput()` класса `DAC`. Это значение используется в выражении `switch`, чтобы обеспечить значение ЦАП в пределах допустимого диапазона 0...255 при каждом нажатии кнопок со стрелками «ВВЕРХ» или «ВНИЗ». Обработка кнопок со стрелками «ВВЕРХ» и «ВНИЗ» выполняется с использованием функции `SendData()` класса `DAC`. В зависимости от нажатой кнопки, в ЦАП записывается либо увеличенное, либо уменьшенное на 8 значение. В функции `SendData()` в

собственном члене данных LastOutput класса DAC запоминается только что выведенное ЦАП значение.

Чтобы собрать исполняемую программу по фрагменту из **Листинга 10.6** необходимы три программных модуля. Это модули ParallelPort, VCO и DAC. Сборка исполняемой программы осуществляется путём компиляции и линкирования всех модулей при помощи файла проекта или make-файла. Модуль ГУН состоит из заголовочного файла (**Листинг 10.1**) и файла определений функций (**Листинг 10.4**). Заголовочный файл класса ParallelPort и файл определения его функций были созданы в разделе 9.4 и повторены в **Листинге 10.7** и **Листинге 10.8**, соответственно.

Листинг 10.7. Заголовочный файл pport.h для класса ParallelPort

```
#ifndef PportH
#define PportH

class ParallelPort
{
    private:
        unsigned int BaseAddress;
        unsigned char InDataPort1;

    public:
        ParallelPort();
        ParallelPort(int baseaddress);
        void WritePort0(unsigned char data);
        void WritePort2(unsigned char data);
        unsigned char ReadPort1();
        virtual ~ParallelPort() {}
};

#endif
```

Листинг 10.8. Файл определения функций pport.cpp для класса ParallelPort

```
#include <dos.h>
#include "pport.h"

ParallelPort::ParallelPort()
{
    BaseAddress = 0x378;
    InDataPort1 = 0;
}

ParallelPort::ParallelPort(int baseaddress)
{
    BaseAddress = baseaddress;
    InDataPort1 = 0;
}

void ParallelPort::WritePort0(unsigned char data)
```

```

{
    outportb(BaseAddress,data);
}

void ParallelPort::WritePort2(unsigned char data)
{
    outportb(BaseAddress+2,data ^ 0x0B);
}

unsigned char ParallelPort::ReadPort1()
{
    InDataPort1 = inportb(BaseAddress+1);
    // Инвертируем старший бит для компенсации внутренней
    // инверсии схемы порта принтера.
    InDataPort1 ^= 0x80;
    // Используем маску (или фильтр) для обнуления
    // неиспользуемых битов D0, D1 и D2.
    InDataPort1 &= 0xF8;
    return InDataPort1;}
}

```

Пока ещё не были созданы заголовочный файл и файл определения функций для модуля ЦАП. Эти файлы приведены в **Листинге 10.9** и **Листинге 10.10**, соответственно.

Листинг 10.9. Заголовочный файл `dac.h` для класса DAC

```

#ifndef Dach
#define Dach

#include "pport.h"

class DAC : public ParallelPort
{
private:
    unsigned char LastOutput;

public:
    DAC();
    DAC(int baseaddress);
    void SendData(unsigned char data);
    unsigned char GetLastOutput();
    ~DAC() {};
};

#endif

```

Листинг 10.10. Файл определения функций `dac.cpp` для класса DAC

```

#include "dac.h"

DAC::DAC()

```

```

{
    LastOutput = 0;
}

DAC::DAC(int baseaddress) : ParallelPort(baseaddress)
{
    LastOutput = 0;
}

void DAC::SendData(unsigned char data)
{
    ParallelPort::WritePort0(data);
    LastOutput = data;
}

unsigned char DAC::GetLastOutput()
{
    return LastOutput;
}

```

Сборка исполняемого файла		
Требуемые файлы	Листинг	Содержимое файла проекта
pport.cpp	Листинг 10.8	pport.cpp
pport.h	Листинг 10.7	
vco.cpp	Листинг 10.4	vco.cpp
vco.h	Листинг 10.1	
dac.cpp	Листинг 10.10	dac.cpp
dac.h	Листинг 10.9	
period.cpp	Листинг 10.6	period.cpp

В вышеприведённой таблице перечислены все необходимые файлы для сборки исполняемого файла программы, все они должны находиться в одном каталоге. Создайте файл проекта в выбранной среде разработки и внесите в него файлы, перечисленные в упомянутой таблице в столбце «Содержимое файла проекта». Тогда компилятор и линкер создадут исполняемый файл. Таблицы такого рода будут встречаться и далее, по мере появления программ, состоящих из нескольких модулей.

Перед запуском программы интерфейсной плате нужно выполнить соединения согласно **Табл. 10.1**, **Табл. 10.2** и **Табл. 10.3**. В этих таблицах перечислены все необходимые соединения для работы с ГУН и ЦАП. Выход ЦАП нужно переключить в униполярный режим, установив джампер *LINK1* на интерфейсной плате. Не забудьте подключить батарею напряжением 9 В, обеспечивающую работу ЦАП, к её клеммной колодке (*J14*). Следует помнить, что передаточная характеристика ГУН линейна (линейность порядка 1%) только в том случае, если вход-

ное напряжение находится в диапазоне 2.2...2.8 В. Хотя линейность ГУН и ухудшается за пределами этого диапазона, тем не менее, даже в этом случае можно им пользоваться, если снять характеристику генератора во всем диапазоне входных напряжений.

Таблица 10.1. Подключение ЦАП

Регистр BASE (Буфер U13)	DAC0800 (U8)
D0	D0 (12)
D1	D1 (11)
D2	D2 (10)
D3	D3 (9)
D4	D4 (8)
D5	D5 (7)
D6	D6 (6)
D7	D7 (5)

Таблица 10.2. Подключение входа ГУН

LM358 (U10B)	ГУН (4046, U4)
VDAC (7)	VIN (9)

Таблица 10.3. Подключение выхода ГУН

ГУН (4046, U4)	Регистр BASE+1 (Буфер U6)
VCO OUTPUT (4)	D3

Примечание

Если схема окажется неработоспособной, то в первую очередь нужно проверить напряжение батареи 9 В. Её напряжение в подключенном состоянии должно быть не менее 7 В.

10.6. Работа с графикой – отображение прямоугольных импульсов

В разделе 10.5 была создана программа, измеряющая период прямоугольных импульсов на выходе ГУН и отображающая результат на экране. В этом разделе воспользуемся графическими средствами для вывода картинки, тогда у пользователя появится возможность наблюдать форму сигнала.

Аналоговое напряжение на входе ГУН очень удобно формировать при помощи имеющегося на печатной плате потенциометра. Перемещая рукоятку потенциометра можно изменять напряжение на его выходе в диапазоне 0...5 В, а значит и частоту выходного сигнала ГУН. В **Табл. 10.4** и **Табл. 10.5** приведены соединения, необходимые для подключения выхода потенциометра ко входу ГУН.

Таблица 10.4. Подключение входа ГУН

Потенциометр (POT1)	ГУН (4046, U4)
OUTPUT	VIN (9)

Таблица 10.5. Подключение выхода ГУН

ГУН (4046, U4)	Регистр BASE+1 (Буфер U6)
OUTPUT	VIN (9)

10.6.1. Работа с экраном

Разрабатываемая программа будет отрисовывать форму выходного сигнала ГУН в заданной области экрана. Сигнал на экране будет выглядеть как в осциллографе. Сигнал должен отображаться в режиме реального времени, то есть изменение сигнала на экране должно происходить одновременно с изменением выходного сигнала ГУН.

Существует множество библиотечных функций работы с графикой, ими можно воспользоваться в этой программе. При помощи этих функций можно выбрать графический драйвер, соответствующий ему видеорежим, разрешение по осям **x** и **y**, и т. п. Экран представляет собой упорядоченный набор пикселей, свечением каждого из которых можно управлять независимо от остальных. Часто возникает необходимость узнать количество пикселей экрана до выбора его размерности, так как разные форматы экрана состоят из разного количества пикселей.

В программах, работающих с графикой в DOS, сначала нужно переключить систему в графический режим, которому, в свою очередь, нужен графический *драйвер*. Драйвер является модулем исполняемого кода, выполняющего вывод графики на экран. Драйверы могут работать в различных режимах, которые определяются размерностями экрана в пикселах по осям **x** и **y**, а также используемой палитрой цветов. В программе сначала нужно выбрать подходящий драйвер, а затем определить размерность экрана в пикселах по осям **x** и **y**. Зная эти величины, можно вычислить координаты области экрана, называемой *портом просмотра* (*viewport*), размещающейся в центре экрана, где и будет отображаться осциллограмма сигнала. На **Рис. 10.6** приведены система экранных координат и вычисления координат осциллограммы.

Размер порта просмотра по оси **x** будет составлять $\frac{1}{2}$ горизонтального размера экрана, а по оси **y** 150 пикселей. Максимальный и минимальный уровни осциллограммы расположены на расстоянии 50 пикселей от верхней и нижней границ

порта просмотра, соответственно. Функции `getmaxx()` и `getmaxy()` позволяют выяснить разрешение экрана по осям **x** и **y**. Конфигурация порта просмотра теперь определена (начало его системы координат находится в его левом верхнем углу).

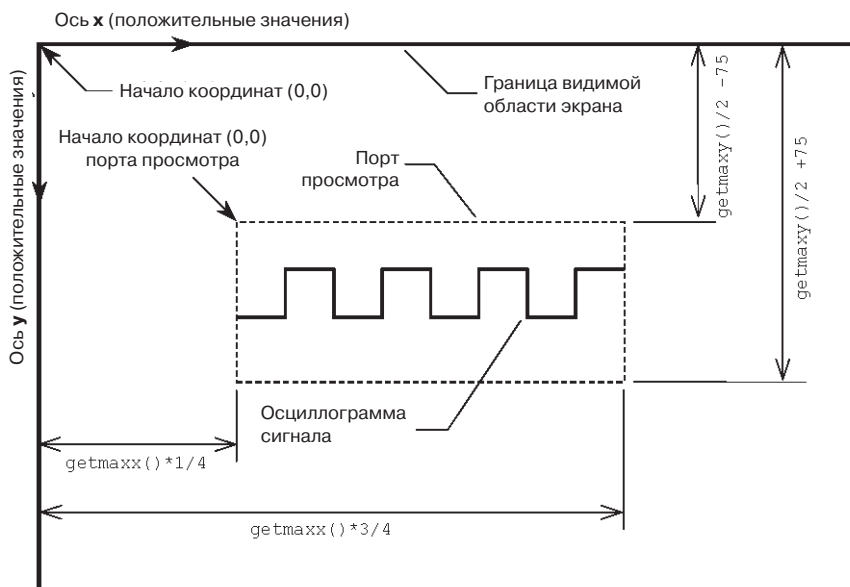


Рис. 10.6. Формат отображения сигнала ГУН

Сигнал отрисовывается в виде линии, состоящей из последовательности точек, следующим образом (помните, что ось **y** направлена вниз). После чтения данных из входного регистра координата **y** (по оси **y**) текущей точки в системе координат порта просмотра устанавливается равной 50 пикселям, если был считан ВЫСОКИЙ уровень, и 100 пикселям, если был считан НИЗКИЙ уровень. Первому значению сигнала, считанного с ГУН, будет соответствовать точка с координатой **x**, равной 0 пикселей по оси **x** в системе координат порта просмотра, следующему $x=1$, затем $x=2$ и т. д. По достижении границы по оси **x**, нужно снова начать отрисовывание осциллограммы с координаты $x=0$, предварительно стерев ранее выведенные точки. Рисование продолжается до тех пор, пока не будет обнаружено нажатие на клавишу, после этого программа завершается.

Программа должна выполнить следующие действия:

1. Переключить систему в графический режим и выбрать видеорежим.
2. Выяснить разрешение экрана по осям **x** и **y**.
3. Создать порт просмотра.
4. Выполнить цикл `while` с условием `!kbhit()`. Завершить программу, если была нажата клавиша.
5. Прочитать уровень сигнала ГУН из входного регистра.
6. Вывести на экран точку в соответствии уровнем сигнала (ВЫСОКИЙ или НИЗКИЙ) и инкрементировать координату **x**.

7. Если x не выходит за установленную границу, то вернуться к действию 4. В противном случае присвоить x исходное значение (начало новой осциллограммы), очистить порт просмотра и вернуться к действию 4.

Программная реализация вышеперечисленных действий приведена в **Листинге 10.11**. Графический вывод представляет собой базовую функциональность программы и может быть усовершенствован в ходе дальнейшей модернизации.

Листинг 10.11. Графическое отображение выходного сигнала ГУН в файле `trace.cpp`

```

/*****
Частота выходного сигнала ГУН меняется в
соответствии с изменениями его входного напряжения.
Входное напряжение ГУН задаётся потенциометром, (POT1)
расположенным на интерфейсной плате. Программа
считывает выходной сигнал ГУН и отображает его
осциллограмму на экране.
*****/

#include <graphics.h>
#include <stdlib.h>
#include <iostream.h>
#include <conio.h>
#include <dos.h>

#include "vco.h"

void main()
{
    VCO Vco;
    int i = 0; // определяет значение координаты по оси x
    int SignalLevel;
    int Driver = DETECT, GraphicsMode, ErrorCode;
    int X, Y;

    // переключение в графический режим
    initgraph(&Driver, &GraphicsMode, "");

    // не возникло ли ошибок ?
    ErrorCode = graphresult();
    if(ErrorCode != grOk)
    {
        cout << "Graphics error: ";
        << grapherrormsg(ErrorCode) << endl;
        cout << "Press any key to halt:" << endl;
        getch();
        exit(1);
    }
}

```

```

}

X = getmaxx();
Y = getmaxy();
rectangle(X/4-1, Y/2-76, X*3/4+1, Y/2+76); // рамка
setviewport(X/4, Y/2-75, X/4*3, Y/2+75, 1;

while(!kbhit())
{
    SignalLevel == Vco.SignalLevel();
    if(SignalLevel == 0) // НИЗКИЙ уровень
        lineto(i, 100);
    else // ВЫСОКИЙ уровень
        lineto(i, 50);

    i++;
    delay(2);

    if(i > X/2) // достигнута половина ширины экрана, т.е.
    {
        // ширина порта просмотра
        i = 0;
        while(Vco.SignalLevel()); // ждём НИЗКИЙ уровень

        // ждём, пока опять не появится ВЫСОКИЙ уровень
        while(!Vco.SignalLevel());
        clearviewport();
    }
}
}

```

Сборка исполняемого файла		
Требуемые файлы	Листинг	Содержимое файла проекта
pport.cpp	Листинг 10.8	pport.cpp
pport.h	Листинг 10.7	
vco.cpp	Листинг 10.4	vco.cpp
vco.h	Листинг 10.1	
trace.cpp	Листинг 10.11	trace.cpp

Файл `graphics.h` предоставляет возможность использования графических функций, например, `initgraph()`, `grapherror()`, `moveto()`, `lineto()` и других, а также констант `DETECT` и `grOk`. Как и в предыдущей программе, создаётся объект `Vco` класса `VCO`. В функции `main()` объявлены несколько переменных типа `int`. В переменных `x` и `y` запоминается разрешение экрана по осям `x` и `y`, соответственно. В дальнейшем они будут иметь другой смысл. Переменная `i` определяет координату пикселей по оси `x` в системе координат порта просмотра. По достижении границы порта просмотра она сбрасывается в 0, чтобы начать отрисовку новой осциллограммы.

Смысл переменных `Driver` и `GraphicsMode` диктуется функцией `initgraph()`. Первый параметр функции `initgraph()` должен указывать тип графического драйвера. Таким драйвером может быть `Colour Graphics Adapter (CGA)`, `Enhanced Graphics Adapter (EGA)`, `Video Graphics Array (VGA)` и другие. Значение `Driver`, равное 1, соответствует CGA, 2 соответствует EGA. Описание этих констант можно найти в документации на функцию `initgraph()`. Каждый из драйверов поддерживает несколько видеорежимов, характеризующихся разрешением экрана (количеством пикселей) и цветовой палитрой. Например, `GraphicsMode`, равная 2, соответствует разрешению 640×480 и 16 цветам. Если `GraphicsMode` равно 1, то будет разрешение 640×320 и 16 цветов. Если же `Driver` будет равно `DETECT` (определённое как 0), то будет выбран драйвер, поддерживаемый видеокартой, и видео режим с максимальным разрешением. В этом случае не нужно указывать значение `GraphicsMode`. Третий параметр должен быть строкой, определяющей путь к графическому драйверу, в данном случае это путь к файлу `EGAVGA.BGI`. Если графический драйвер лежит в текущем каталоге (где находится исполняемый файл программы), то этот параметр может быть пустой строкой. Обратите внимание, что в вызове `initgraph()` перед первыми двумя аргументами стоит знак `&`. Его необходимость вызвана тем, что эти аргументы являются указателями (т. е. являются адресами).

Успешное выполнение функции `initgraph()` проверяется путём вызова функции `graphresult()` и запоминания возвращённого значения в переменной `ErrorCode`. Если `ErrorCode` не равна константе `grOk`, значит, произошла ошибка. Сообщение, соответствующее значению `ErrorCode`, можно получить при помощи функции `grapherrormsg()`. Истинная ветвь выражения `if` выводит на экран сообщение об ошибке, после чего вызывается функция завершения программы `exit()`. Если ошибки не произошло, то программа перейдёт к следующему этапу: определению разрешения экрана по осям `x` и `y`. Периметр порта просмотра очерчивается прямоугольником, а затем создаётся и сам порт. Как говорилось ранее, порт просмотра является областью экрана, где будет отображаться осциллограмма. После создания порта просмотра начало системы координат перемещается в его левый верхний угол.

В цикле `while` с условием выполнения `!kbhit()` происходит непрерывное рисование осциллограммы на экране. Функция `lineto()` работает в системе координат порта просмотра. Рисование новой осциллограммы начинается при нулевом значении `i`. Проводится отрезок между текущей точкой и новой точкой, вертикальное положение которой определяется уровнем сигнала. Затем инкрементируется значение `i` для проведения нового отрезка. Когда переменная `i` достигает предела по оси `x` (равного ширине порта просмотра $x/2$), происходит её обнуление. Оставшийся код нужен для синхронизации осциллограммы таким образом, чтобы следующая осциллограмма начиналась с НИЗКОГО уровня. После этого выведенная осциллограмма стирается.

Следует отметить, что отображаемые импульсы могут быть шире, чем на самом деле, так как компьютер тратит некоторое время на обработку прерываний.

10.7. Измерение температуры

Измерение температуры в данном случае производится косвенно по выходному напряжению делителя напряжения с термистором. Если в делителе используется

термистор с отрицательным температурным коэффициентом сопротивления, то выходное напряжение с увеличением температуры будет нелинейно уменьшаться (**Рис. 10.3**). Зависимость напряжения от температуры для упрощения программирования будем считать линейной. Программа будет работать с той же схемой, которая была описана ранее в разделе 10.6 (только вместо выхода РОТ задействован выход сигнала термистора, VTH). Типичная зависимость периода ГУН от температуры термистора приведена на **Рис. 10.7**.

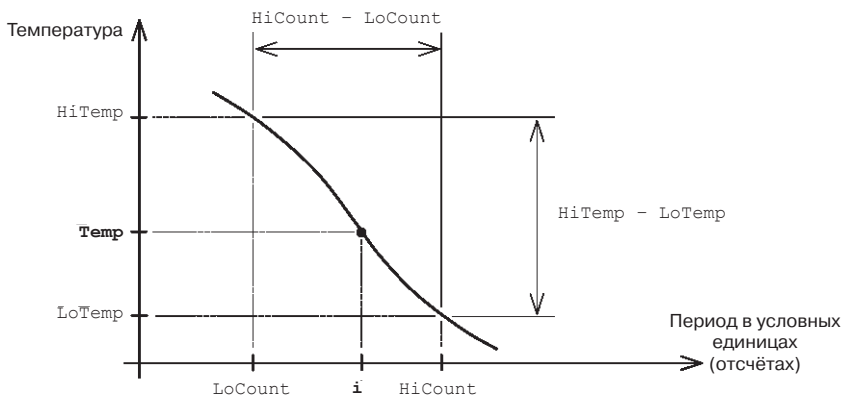


Рис. 10.7. Типичная передаточная характеристика системы термистор-ГУН (применительно к калибровке)

10.7.1. Калибровка термистора

Программа должна измерять период ГУН и преобразовывать полученное значение в температуру. Прежде всего, нужно откалибровать термистор. Калибровка производится путём измерения периода ГУН, когда термистор контактирует со средой с известной температурой, такой как, лёд с водой, тело человека или кипящая вода. Тогда можно будет определить формулу или таблицу значений для экстраполяции/интерполяции температуры по периоду ГУН (в пределах линейной части передаточной характеристики термистор-период ГУН). Нужно учитывать, что выходной сигнал термистора может существенно отклоняться от диапазона линейности ГУН (1.5...3.5 В). Если сигнал на входе ГУН будет вне диапазона линейности, то ГУН будет искажать сигнал температуры. Но даже и в этом случае можно измерять температуру при помощи откалиброванной системы термистор-ГУН, правда, с меньшей точностью.

Калибровочное выражение можно получить следующим образом (**Рис 10.7**). В программу из **Листинга 10.6** можно добавить несколько выражений, чтобы можно было вводить значения меньшей и большей температур калибровки. Измерив соответствующие им периоды ГУН можно константы калибровочного выражения. Если температуры калибровки не были введены (HiTemp и LoTemp), то будет отображаться период ГУН, как это было в программе из **Листинга 10.6**. После правильно выполненной калибровки будет отображаться температура. Для этого программу нужно дополнить. Факт введения значений обоих температур кали-

бровки можно зафиксировать при помощи *флагов*. Если установлены оба флага, то это значит, что были введены значения обеих температур и можно вычислить константы калибровочного выражения. Тогда можно отображать значение температуры, а не периода ГУН. По возможности будем использовать те же клавиши управления, что в **Листинге 10.6**: «Alt+X» для завершения программы, стрелка «ВВЕРХ» для ввода большей температуры и стрелка «ВНИЗ» для меньшей.

Действия программы можно перечислить в следующем порядке:

1. Инициализировать счётчик нулевым значением.
2. Циклически проверять уровень сигнала, пока не будет обнаружено его изменение.
3. Циклически проверять уровень сигнала, пока опять не будет обнаружено его изменение, инкрементируя значение счётчика после каждой проверки.
4. Циклически проверять уровень сигнала, пока не будет обнаружено третье изменение уровня сигнала, инкрементируя значение счётчика после каждой проверки.
5. Проверить, введены ли значения калибровочных температур (установлены ли флаги). Если оба флага установлены, то вычислить значение температуры, в противном случае отобразить значение периода.
6. Проверить, не было ли нажатия кнопок клавиатуры. Если кнопки не нажимались, то перейти к шагу 1.
7. Если нажимались кнопки «Alt+X», то завершить программу.
8. Если было нажатие на стрелку «ВВЕРХ», ввести значение большей температуры.
9. Если было нажатие на стрелку «ВНИЗ», ввести значение меньшей температуры.

Некоторые действия можно описать более подробно:

- 8.1. Вывести пользователю просьбу ввести значение большей температуры и запомнить введённое значение.
- 8.2. Запомнить значение периода.
- 8.3. Установить флаг ввода большего значения, если оно было введено.
- 9.1. Вывести пользователю просьбу ввести значение меньшей температуры и запомнить введённое значение.
- 9.2. Запомнить значение периода.
- 9.3. Установить флаг ввода меньшего значения, если оно было введено.

Программная реализация этих действий приведена в **Листинге 10.12**.

Листинг 10.12. Измерение температуры при помощи термистора и ГУН
в файле temp.cpp

```

/*****
Программа измеряет длительность периода ГУН, входное
напряжение которого задаётся выходным сигналом
термистора, расположенного на интерфейсной плате.
Также реализована возможность калибровки термистора

```

и программа может отображать значение температуры.

```

*****/

#include <iostream.h>
#include <bios.h>
#include <conio.h>
#include "vco.h"

void main()
{
    VCO Vco;
    int Quit = 0, HiFlag = 0, LoFlag = 0;
    int key = 0;
    float HiTemp, LoTemp, Temp;
    long int HiCount, LoCount;

    clrscr();
    while(!Quit)
    {
        Vco.MeasurePeriod();
        clrscr();
        gotoxy(10,10);

        if((HiFlag == 1) && (LoFlag == 1))
        {
            Temp = LoTemp + (HiTemp - LoTemp) *
                (Vco.GetPeriod() - LoCount)/(HiCount - LoCount);
            cprintf("The temperature is: %7.1 lf (deg)\a",Temp);
        }
        else
            cprintf("The pulse period is: 10lu\a",
                Vco.GetPeriod()/1000);

        if(bioskey(1) != 0)
        {
            key = bioskey(0);
            switch(key)
            {
                /* Alt-X */      case 0x2d00 : Quit = 1;
                                break;

                /* BBEPX */     case 0x4800 : gotoxy(10,5);
                                cout << "Enter Upper
                                    Calibration Temp: ";
                                cin >> HiTemp;
                                HiCount = Vco.GetPeriod();
                                HiFlag = 1;
                                break;

                /* ВНИЗ */      case 0x5000 : gotoxy(10,6);

```



```
        cout << "Enter Lower  
            Calibration Temp: ";  
        cin >> LoTemp;  
        LoCount = Vco.GetPeriod();  
        LoFlag = 1;  
    }  
}  
}
```

Сборка исполняемого файла		
Требуемые файлы	Листинг	Содержимое файла проекта
pport.cpp	Листинг 10.8	pport.cpp
pport.h	Листинг 10.7	
vco.cpp	Листинг 10.4	vco.cpp
vco.h	Листинг 10.1	
temp.cpp	Листинг 10.12	temp.cpp

Переменная `HiFlag` служит флагом (признаком) ввода большей калибровочной температуры, аналогичным образом переменная `LoFlag` означает факт ввода меньшей калибровочной температуры. Несмотря на то, что переменные `HiFlag` и `LoFlag` типа `int`, они будут иметь значения либо 1, либо 0. В переменных `HiTemp` и `LoTemp` хранятся значения верхней и нижней температур, соответственно, введенные во время калибровки. Значения периода сигнала (исчисленные в отсчётах) при большей и меньшей температурах калибровки запоминаются в переменных `HiCount` и `LoCount`, соответственно. Вычисленное значение температуры сохраняется в переменной `Temp`.

В первом выражении `if` функции `main()` выясняется, введены ли значения большей и меньшей температур калибровки путём проверки состояния флагов `HiFlag` и `LoFlag`. Если оба флага установлены, то по известному калибровочному выражению вычисляется значение температуры и выводится на экран.

Обратите внимание на важность правильной записи математических операций в формуле вычисления значения переменной `Temp`:

```
Temp = LoTemp + (HiTemp - LoTemp) *  
        (Vco.GetPeriod() - LoCount)/(HiCount - LoCount);
```

Значения `Vco.GetPeriod()`, `HiCount` и `LoCount` типа `long int`, тогда как `Temp`, `LoTemp` и `HiTemp` типа `float`. Если выражение в нижней строке заключить в круглые скобки, то его результат компилятор приведёт к типу `long int` (в данном случае это неправильно, выражение должно быть типа `float`). Аналогичным образом, изменение порядка математических операций может неявно заставить компилятор менять типы членов выражения, что приведёт к изменению результата выражения.

Если же значения этих температур не были введены, то выражение не вычисляется и вместо него на экран выводится длительность периода ГУН. В выражении `switch` проверяются нажатия на кнопки стрелок «ВВЕРХ», «ВНИЗ» и комбинации «Alt+X». Если была нажата стрелка «ВВЕРХ», то выводится предложение ввести значение большей температуры, запоминаемое в переменной `HiTemp`. Значение, полученное от вызова функции `Vco.GetPeriod()` (возвращающей значение члена данных `Period`), сохраняется в переменной `HiCount`. Затем устанавливается флаг `HiFlag`. Похожая последовательность действий выполняется и при нажатии стрелки «ВНИЗ».

Помните: термистору нужно некоторое время, чтобы его температура сравнялась с температурой окружающей среды. Поэтому нужно выждать достаточное время перед нажатием стрелок «ВВЕРХ» и «ВНИЗ» во время калибровки на большей и меньшей температурах, соответственно. После калибровки можно проверить правильность работы программы. При помещении термистора в среду с эталонной температурой, использованной при калибровке, на экране должна отображаться примерно равная ей температура.

10.8. Заключение

В этой главе был описан принцип работы генератора, управляемого напряжением (ГУН). На выходе ГУН генерируются прямоугольные импульсы, частота которых пропорциональна входному напряжению. Зная зависимость частоты от напряжения (или периода, как в рассмотренном случае), можно измерять напряжение. Таким образом, ГУН можно использовать в качестве недорогого варианта аналого-цифрового преобразователя.

На базе класса `ParallelPort` был создан новый класс `VCO`. Был рассмотрен способ программного измерения периода входного цифрового сигнала путём непрерывной проверки его уровня и инкрементирования счётчика, значение которого и является периодом. Создана программа, выводящая осциллограмму входного сигнала при помощи графических средств. Затем была разработана программа измерения температуры, использующая термистор и ГУН.

10.9. Литература

1. Bentley, J., *Principles of Measurement Systems*, Second edition, Longman Scientific & Technical, Essex, 1988.
2. Horowitz, P. And Hill, W., *The art of electronics*, Cambridge University Press, Cambridge, 1989.
3. NS CMOS, *CMOS Logic Databook*, National Semiconductor Corporation, 1988.
4. Webb, R. E., *Electronics for Scientists*, Ellis Horwood, New York, 1990.
5. Wobshall, D., *Circuit Design for Electronic Instrumentation*, McGraw-Hill, 1987.
6. Lafore, R. *Object Oriented Programming in MICROSOFT C++*, Waite Group Press, 1992.
7. Wang, P. S., *C++ with Object Oriented Programming*, PWS Publishing, 1994.
8. Winston, P. H., *On to C++*, Addison Wesley, 1994.

11 Аналого-цифровое преобразование

Содержание главы:

- Сведения об аналого-цифровом преобразовании (АЦП).
- Типы АЦП.
- Выборка сигналов.
- Класс АЦП.
- Измерение напряжения и температуры при помощи АЦП.

11.1. Введение

В этой главе изучаются принципы работы аналого-цифровых преобразователей и собственно *аналого-цифрового преобразования*. Затем будут обсуждаться ограничения, накладываемые на выборку и преобразование сигнала.

Такие физические величины, как температура, давление, скорость потока (расход газа/жидкости) и расстояние измеряются при помощи датчиков. Выходным сигналом аналоговых датчиков обычно служат ток, напряжение или заряд, находящиеся в некоторой зависимости от измеряемой величины, математическое выражение этой зависимости может быть получено после проведения калибровки. Процедура калибровки была описана в предыдущей главе. Для перевода такого сигнала в цифровую величину, которую можно ввести в вычислительное устройство (компьютер), применяют аналого-цифровые преобразователи (АЦП). Схемы предварительной обработки сигнала преобразуют его в напряжение, которое затем преобразовывается при помощи АЦП в числовые значения.

Выходные данные АЦП на интерфейсной плате считываются при помощи программы, эта же программа осуществляет управление аналого-цифровым преобразователем. Для этого на базе класса `ParallelPort` создаётся новый класс, который и реализует эту функциональность. Созданный класс будет использоваться в программах, измеряющих напряжение.

11.2. Аналого-цифровое преобразование

Аналого-цифровое преобразование это процесс выборки мгновенного значения сигнала и преобразование его в целочисленное значение, пропорциональное величине преобразуемого сигнала. Сигнал обычно представляет собой напряжение. Аналого-цифровое преобразование находит широкое применение в различных системах, от кодирования речевого сигнала в телекоммуникационных системах

до систем управления и сбора данных. На **Рис. 11.1** показана схема типового АЦП (8-битного). Преобразование начинается после активации входного сигнала «Запуск преобразования». По завершении этого процесса аналого-цифровой преобразователь изменит состояние своего выхода «Преобразование завершено». По этому сигналу обслуживающее АЦП устройство уведомляется о том, что преобразование завершено и можно считывать результат преобразования.



Рис. 11.1. Блок-схема 8-битного АЦП

Время, которое проходит от запуска до появления цифрового значения на выходе АЦП, называется *временем преобразования*. Сигнал «Преобразование завершено» может не использоваться в тех системах, где время промежутка времени между запуском преобразования и чтением результата больше, чем время преобразования.

Аналоговый сигнал характеризуется бесконечным набором значений сигнала. Значения сигнала преобразуются в целые числа, диапазон которых ограничен. Такое преобразование называется *квантованием*. Например, на **Рис. 11.2** и в **Табл. 11.1** приведён 3-битный АЦП, преобразующий входное напряжение (скажем от 0 до 3.5 В) в числа от 0 до 7. В этом примере весь диапазон входного напряжения поделён (или квантован) на семь интервалов.

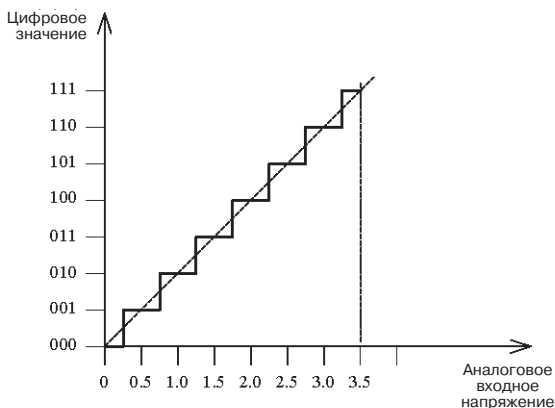


Рис. 11.2. Идеальное аналого-цифровое преобразование

Таблица 11.1. Квантование аналогового напряжения в 3-битный код

Квантованное аналоговое входное напряжение	Выходной 3-битный код АЦП	Десятичное значение
0.0 В	000	(0)
0.5 В	001	(1)
1.0 В	010	(2)
1.5 В	011	(3)
2.0 В	100	(4)
2.5 В	101	(5)
3.0 В	110	(6)
3.5 В	111	(7)

Цифровой код АЦП является правильным и не содержит ошибки в том случае, когда входной сигнал точно равен одному из квантированных уровней входного напряжения. Тем не менее, АЦП присуща так называемая *погрешность квантования*, возникающая, когда входное напряжение не равно квантованному уровню входного напряжения. Обратимся, например, к **Рис. 11.2**; погрешность квантования равна нулю, когда напряжение равно 0 В, 0.5 В, 1.0 В и так далее. Погрешность квантования максимальна (равна половине одного шага квантования), когда входное напряжение лежит в середине между двумя соседними уровнями квантования; в предыдущем примере она составляет 0.25 В.

Возможен также другой тип ошибки в аналого-цифровом преобразовании, *монотонная погрешность*. Если входной сигнал преобразователя увеличивается на величину, кратную шагу квантования, то и выходной код должен измениться на соответствующее число. Если этого не происходит, значит имеет место монотонная погрешность, уменьшающая разрешающую способность АЦП.

Выходной цифровой код АЦП может иметь последовательный или параллельный формат. Цифровой выход преобразователя, показанного на **Рис. 11.1**, является параллельным и состоит из восьми выходных цифровых линий. Преобразователи с параллельным выходом, как правило, имеют большую скорость работы по сравнению с преобразователями с последовательным выходом, но они требуют подключения большего количества сигнальных линий. С другой стороны, АЦП с последовательным выходом имеют более сложный интерфейс. В некоторых преобразователях имеется встроенный аналоговый мультиплексор, позволяющий обрабатывать несколько входных сигналов (скорость работы при этом упадёт).

В начале процесса преобразования входное напряжение считывается входными цепями преобразователя. Для нормальной работы АЦП необходимо, чтобы выходной импеданс внешней сигнальной цепи должен быть достаточно низким по сравнению с входным импедансом АЦП. Преобразователь должен работать с ограниченным по величине входным сигналом, поэтому перед подключением внешней схемы к АЦП нужно принять меры к ограничению этого напряжения.

11.3. Методы преобразования

Электронные схемы АЦП реализуют различные методы аналого-цифрового преобразования, включая преобразователи типа напряжение-частота: *интегрирующие* АЦП, АЦП *двойного интегрирования*, АЦП *последовательного приближения* и *флэш*-АЦП. В некоторых преобразователях для достижения характеристик, присущих отдельным типам АЦП, используется комбинация этих типов. Например, АЦП последовательного приближения реализуется на базе быстродействующей технологии флэш (flash), получается недорогой и очень быстрый преобразователь. Метод преобразования напряжение-частота рассматривался ранее в главе 10 и обычно не используется из-за сравнительно невысокой скорости преобразования. Остальные типы АЦП используются часто и будут рассмотрены далее.

Интегрирующий АЦП

Такой преобразователь состоит из источника постоянного тока, который заряжает конденсатор, компаратор напряжения и счётчик с источником тактирующих импульсов и логикой управления, **Рис. 11.3**.

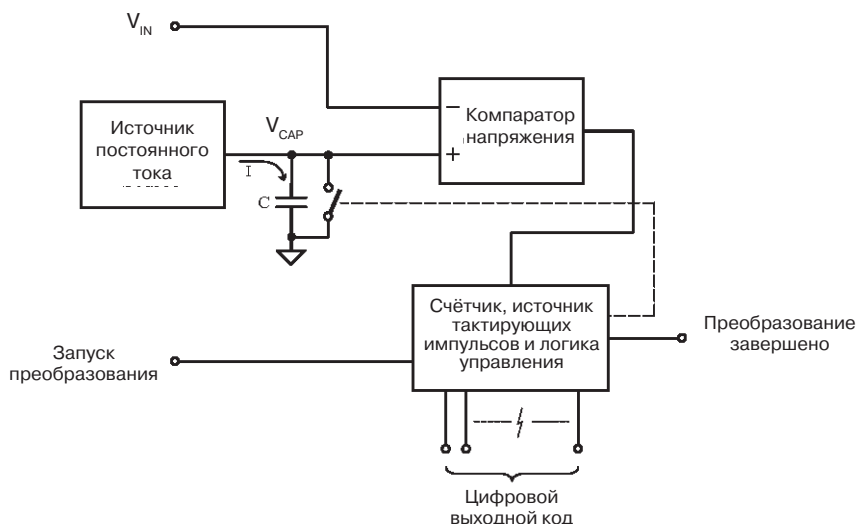


Рис. 11.3. Упрощённая модель интегрирующего АЦП

Преобразование начинается после того, как сигнал «Запуск преобразования» будет приведён в активное состояние и происходит в два этапа:

1. *Инициализация.* Логика управления замыкает ключ цепи разряда конденсатора C , разряжая конденсатор. Затем обнуляется счётчик, логика управления размыкает ключ разряда конденсатора и начинается счёт.
2. *Интегрирование.* Источник постоянного тока образует на конденсаторе возрастающее напряжение, приложенное к положительному входу компаратора (этот процесс и называется интегрированием). Когда это возрастающее

напряжение достигнет величины напряжения, приложенного к отрицательному входу компаратора, выход компаратора переключится из НИЗКОГО состояния в ВЫСОКОЕ. Изменение выходного сигнала компаратора приводит к остановке счёта и завершению преобразования. Затем выход «Преобразование завершено» переводится в активное состояние, сигнализируя о готовности данных АЦП. На выходе счётчика теперь присутствует цифровое представление входного сигнала. Для преобразования большего по величине сигнала требуется больше времени, чтобы достичь этого уровня, что приведёт к большему цифровому выходному значению.

После активации сигнала «Запуск преобразования» вышеописанные действия произойдут в той же последовательности.

Скорость преобразования интегрирующих АЦП невелика, но при этом они обеспечивают хорошую точность преобразования, определяемую долговременной стабильностью тактовых импульсов счётчика и качеством конденсатора, остаточная поляризация диэлектрика у которого должна быть как можно меньше. После разряда конденсатора, диэлектрик которого обладает высокой остаточной поляризацией, на нём сохранится остаточный заряд. Такой заряд будет служить источником погрешности. Конденсаторы с очень малой остаточной поляризацией диэлектрика после разряда будут иметь очень незначительное напряжение на своих выводах. Другим преимуществом интегрирующих АЦП является усреднение шумов сигнала во время преобразования.

АЦП двойного интегрирования

Преобразователь двойного интегрирования по принципу действия похож на интегрирующий АЦП, только в процессе преобразования интегрирование производится дважды, что существенно повышает точность. На **Рис. 11.4** приведена блок-схема такого преобразователя.

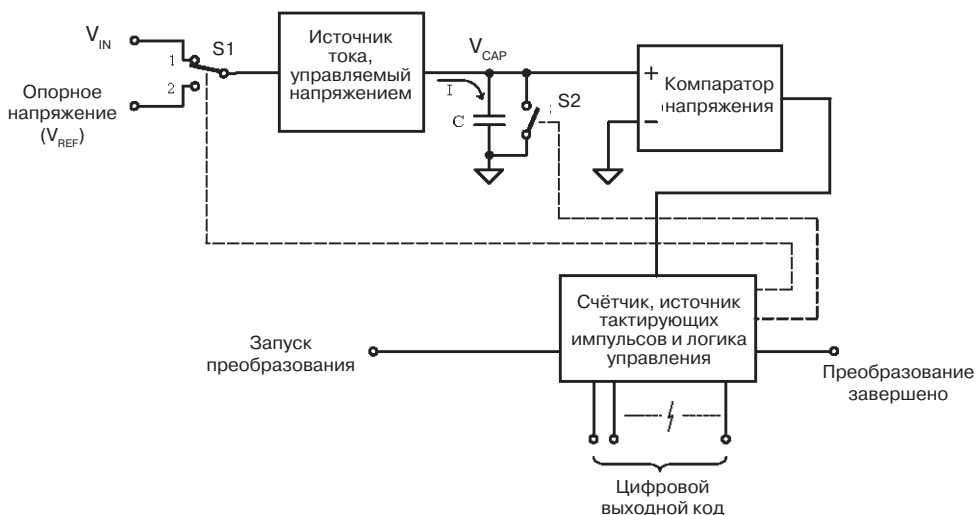


Рис. 11.4. Упрощённая модель АЦП двойного интегрирования

После активации сигнала «Запуск преобразования» начинается процесс преобразования, проходящий в три этапа:

1. *Инициализация.* Замыкается ключ $S2$, разряжая интегрирующий конденсатор C .
2. *Интегрирование «ВВЕРХ»* напряжения V_{IN} . В начале этого этапа счётчик обнуляется, а переключатель $S1$ устанавливается в положение 1, подавая напряжение V_{IN} на вход источника тока, управляемого напряжением. Размыкается ключ $S2$, позволяя току, пропорциональному входному напряжению V_{IN} заряжать интегрирующий конденсатор C . На конденсаторе при этом формируется линейно возрастающее напряжение, обозначенное буквой A на **Рис. 11.5**. Нарастание ограничивается по времени, обычно нарастание продолжается до тех пор, пока счётчик не досчитает до максимума, чтобы обеспечить наибольшую точность преобразования. По истечении этого периода счётчик обнуляется и начинается последний этап преобразования.
3. *Интегрирование «ВНИЗ»* при помощи опорного напряжения. Переключатель $S1$ переводится в положение 2, подавая на вход источника тока, управляемого напряжением отрицательное высокостабильное опорное напряжение. Возникающий при этом ток, имеющий всегда одну и ту же отрицательную величину на этом этапе, разряжает интегрирующий конденсатор C . Разряд продолжается до тех пор, пока напряжение на неинвертирующем входе компаратора не упадёт до потенциала земли, к которой подключен инвертирующий вход компаратора. Когда это поройдётся, выходной сигнал компаратора изменится на противоположный и остановит счётчик. Значение, достигнутое счётчиком, будет представлять собой преобразованное значение входного сигнала.

На **Рис. 11.5** показаны графики напряжений, формирующихся в течении двух этапов интегрирования. Линии C и D , расположенные снизу, получены при более низком входном напряжении V_{IN} . Линии B и D имеют одинаковый наклон, так как образованы одним и тем же по величине током (определяемым V_{REF}).

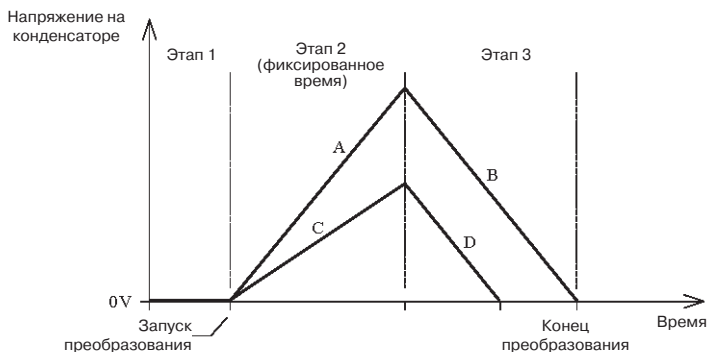


Рис. 11.5. Упрощённая модель АЦП двойного интегрирования

По сравнению с интегрирующим АЦП, двойное интегрирование даёт большую точность преобразования, определяемую, главным образом, стабильностью опор-

ного напряжения. В отличие от интегрирующего АЦП, метод двойного интегрирования не зависит от долговременной стабильности источника тактовых импульсов, поскольку в этапах 2 и 3 используется один и тот же источник тактовых импульсов. Двойное интегрирование тоже устойчиво к шумам и тоже требует качественного конденсатора с очень маленькой остаточной поляризацией диэлектрика. Такой тип преобразователя сравнительно медленный, но в тоже время очень точный, разрешающая способность может достигать 18 бит. Другие преобразователи не могут достичь такой точности без существенного увеличения их стоимости, и поэтому АЦП двойного интегрирования широко применяются в инструментальных средствах, например точных цифровых мультиметрах.

АЦП последовательного приближения

Такие АЦП очень распространены из-за относительно высокой скорости преобразования, хорошей точности и небольшой стоимости. На **Рис. 11.6** показана блок-схема такого преобразователя.

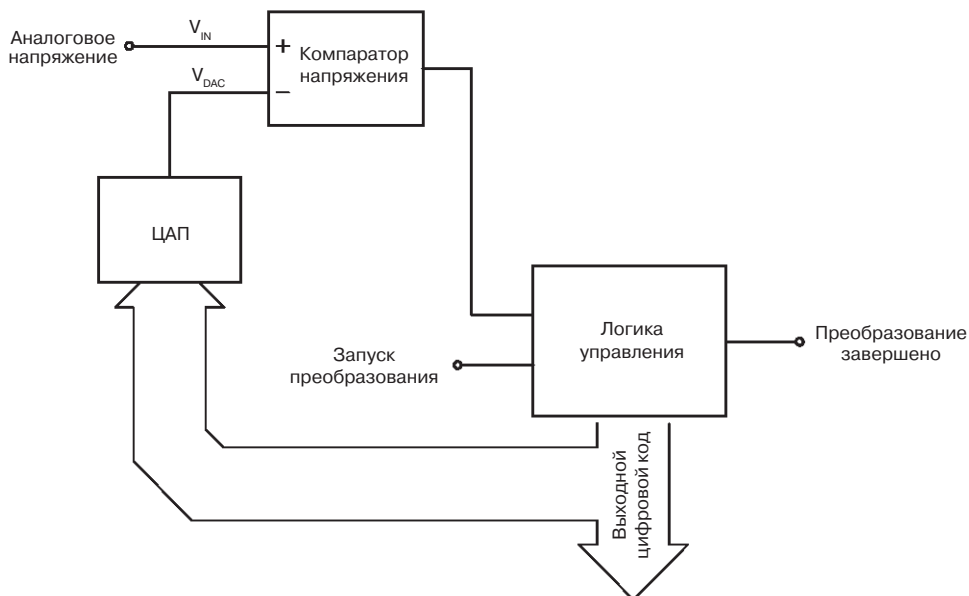


Рис. 11.6. Упрощённая модель АЦП последовательного приближения

Преобразование начинается непосредственно после активации сигнала «Запуск преобразования» и происходит следующим образом:

1. *Инициализация.* Сбрасываются в исходное состояние все выходные сигналы логики контроля.
2. *Процесс последовательного приближения.* Выходной код формируется в ходе проведения ряда сравнений напряжения, когда входное аналоговое напряжение сравнивается с аналоговым представлением выходного кода, этот код формируется в процессе цикла последовательного приближения.

Происходит сравнение аналогового представления каждого бита с аналоговым входным напряжением, начиная с наиболее значимого бита. Например, в случае 8-битного преобразователя тестовым кодом было бы значение 1000 0000, представляющее собой середину диапазона входного сигнала. Этот код поступает на вход цифро-аналогового преобразователя, формирующего аналоговое напряжение V_{DAC} . Напряжение V_{DAC} сравнивается с сигналом V_{IN} при помощи компаратора. Если V_{IN} больше чем V_{DAC} , на выходе компаратора будет ВЫСОКИЙ уровень и логика управления оставит этот сравниваемый бит и на выходе появится значение 1000 0000. Если V_{IN} окажется меньше, на выходе компаратора будет НИЗКИЙ уровень и логика управления обнулит этот бит, выходное значение будет тогда 0000 0000.

Следующее сравнение будет происходить с участием соседнего младшего бита. При этом этот бит добавится к коду, полученному на предыдущем шаге. Если считать, что в первом сравнении V_{IN} оказалось больше, то получим значение 1100 0000. Код передаётся в ЦАП, формируется новое значение V_{DAC} . Аналогичным образом обрабатываются оставшиеся биты. По завершении сравнения всех битов активируется сигнал «Преобразование завершено». Такой тип преобразователя характеризуется постоянным временем преобразования и высокой скоростью преобразования.

На интерфейсной плате установлен АЦП последовательного приближения ADC0804. Имеется несколько сигналов управления этой микросхемой, один из них $/RD$ ($/$ означает НИЗКИЙ активный уровень) и служит для чтения выходных данных. Другой сигнал $/CS$, используемый совместно с сигналом $/RD$, служит для перевода цифровых выходов в *третье состояние*. Когда выход микросхемы находится в третьем состоянии, то его выходное сопротивление становится очень высоким. Другие два состояния являются логическими уровнями ВЫСОКИЙ и НИЗКИЙ. Третье состояние выходов ADC0804 нужно для построения микропроцессорных систем, где шина данных общая для многих устройств. В таких системах требуется, чтобы в определённые моменты времени АЦП «отключалось» бы от шины данных, чтобы ей могли воспользоваться другие устройства.

Запуск преобразования и чтение результата происходит согласно **Рис. 11.7**. Сигнал $/CS$ следует удерживать в НИЗКОМ состоянии в течение всего цикла преобразования. Во многих устройствах он всегда может быть низким (когда нет необходимости в третьем состоянии выхода).

Внимание!

Входное напряжение АЦП не должно превышать +5 В и падать ниже 0 В. Если входное напряжение выйдет за пределы этого диапазона, АЦП выйдет из строя.

Также следует обратить внимание на то, что сигнал завершения преобразования $/INTR$ переходит в активное состояние (НИЗКИЙ уровень) лишь на короткое время. При чтении этого сигнала при помощи компьютера нужно принимать во внимание, что программа работает с параллельным портом сравнительно медленно и может «не заметить» изменение этого сигнала. Обычно этот сигнал *защёлкивается* специальной схемой. Защёлкивание подразумевает регистрацию

(запоминание) какого-либо события, при этом состояние выходного сигнала схемы-защёлки служит признаком факта события. На интерфейсной плате не предусмотрена схема-защёлка. В программах после запуска преобразования сигнал $/INTR$ опрашиваться не будет. Вместо этого программа будет ждать некоторое время, длительность ожидания будет превышать время преобразования, и только затем считывать результат преобразования.

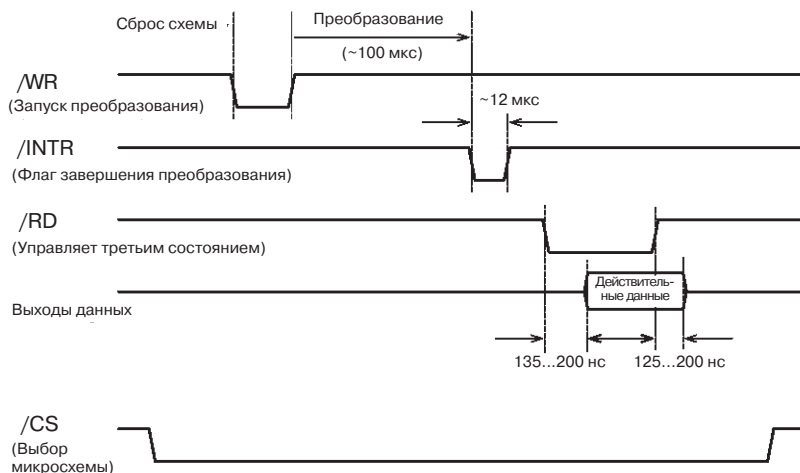


Рис. 11.7. Временная диаграмма работы ADC0804

В схеме преобразователя ADC0804 шины земли аналоговой и цифровой частей разделены для снижения шумов в аналоговой части. Раздельные земли нужно соединить вместе по схеме из **Приложения А, Рис. А.26**.

Флэш-АЦП

Преобразователи этого типа самые быстрые из аналого-цифровых преобразователей, но вместе с тем и самые дорогие. Флэш-АЦП применяются в таких устройствах и задачах, как цифровая обработка сигналов, обработка видеосигнала и других задачах обработки сигналов, в цифровых осциллографах. Таким преобразователям не нужен сигнал «Запуск преобразования», преобразование выполняется непрерывно, как показано на **Рис. 11.8**.

Для простоты выбран 2-битный АЦП, в котором диапазон входного напряжения квантуется на четыре уровня. Например, если диапазон входного сигнала 0...3 В, тогда уровни квантования будут 0, 1, 2 и 3 В. Напряжение V_{REF} должно составлять 4 В. Входное напряжение сравнивается с четырьмя уровнями квантования при помощи трёх компараторов. Четвёртый уровень 0 В не требует компаратора.

Первый компаратор, на отрицательном входе которого напряжение V_1 , выполняет проверку входного сигнала на превышение 1 В. Если V_{IN} превышает V_1 , то на выходе компаратора будет **ВЫСОКИЙ** логический уровень, поступающий на вход кодирующей логики. В противном случае на выходе компаратора будет **НИЗКИЙ** логический уровень. Остальные два компаратора имеют пороговые напряжения 2 и 3 В. Кодирующая логика преобразует выходные сигналы компара-

торов в соответствующий n -битный код, в данном случае $n = 2$. Такое преобразование происходит непрерывно и происходит очень быстро, скорость работы определяется временем срабатывания компараторов и задержкой в кодирующей логике.

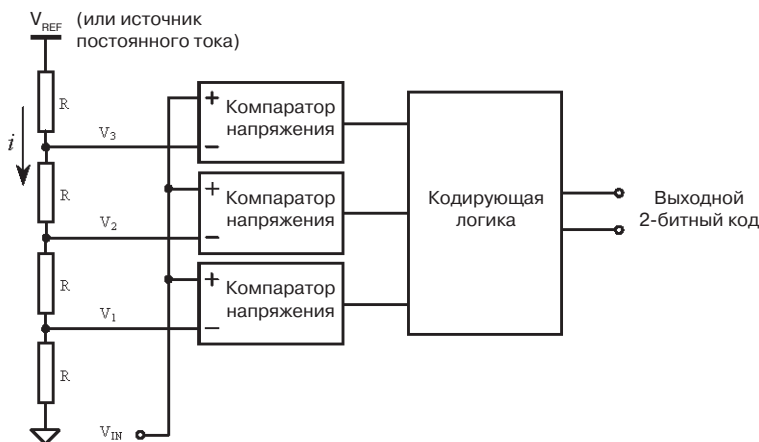


Рис. 11.8. Упрощённая структура 2-битного АЦП

Другим важным отличием флэш-АЦП является крайне малое *время захвата*. Время захвата является периодом времени, затрачиваемого на «чтение» аналогового сигнала во время преобразования. В случае флэш-АЦП время захвата определяется временем защёлкивания данных (запоминания в кодирующей логике) и время кодирования при этом не учитывается. Благодаря этой особенности такие преобразователи идеально подходят для устройств с быстроменяющимися сигналами и им не нужны схемы выборки/хранения, речь про которые пойдёт далее в этой главе.

По мере увеличения разрешающей способности количество компараторов резко возрастает. Для n -битного компаратора требуется $2^n - 1$ компараторов. Преобразователи, разрядность которых 8...10 бит уже довольно дороги и заметно больше по размеру. Один из способов наращивания разрядности заключается в каскадном включении преобразователей. Например, из четырёх 6-битных модулей можно собрать 8-битный преобразователь.

11.4. Измерение напряжения при помощи АЦП

Перед тем как приступить к измерению меняющегося сигнала при помощи АЦП, полезно разобраться с базовыми понятиями обработки сигналов, такими как *скорость нарастания*, *выборка и хранение*, *ошибка дискретизации* и *эквивалентная выборка*. Рассмотрим периодический сигнал, Рис. 11.9.

Представим, что этот сигнал преобразуется в цифровую форму 8-битным АЦП, время преобразования которого 100 мкс (ADC0804 на интерфейсной плате). Во время начала преобразования входное напряжения либо возрастает, либо убывает, в зависимости от момента запуска преобразования. Разберём случай, когда входное напряжение возрастает от 0 В со скоростью 5 В за 0.5 с (10 В/с). Зная, что

скорость нарастания напряжения равна 10 В/с, получим, что в течение 100 мкс входное напряжение возрастёт на 1 мВ.

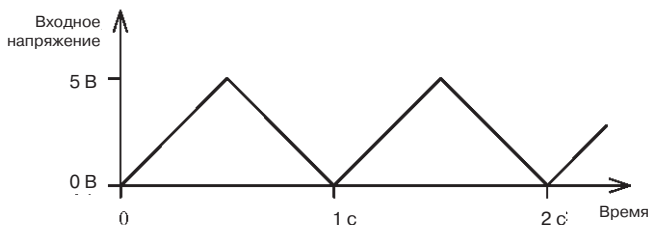


Рис. 11.9. Треугольный сигнал

В 8-битном АЦП диапазон входного напряжения разбивается на 256 уровней. Для достижения максимальной точности преобразования, примем диапазон входного сигнала равным 5 В. Тогда разность двух соседних уровней будет составлять $5 \text{ В} / 256 = 19 \text{ мВ}$. Точность преобразования не будет теряться, если в течение преобразования входное напряжение не будет изменяться более чем на половину разности между двумя соседними уровнями квантования. Значит, напряжение не должно изменяться более чем на 9.5 мВ за 100 мкс, соответственно, скорость нарастания не должна превышать 95 В/с. В данном случае сигнал изменяется очень медленно и потерь точности не будет, поскольку сигнал изменяется на 1 мВ (со скоростью 10 В/с) в течение преобразования, в то время как преобразователь может обрабатывать сигнал, изменяющийся со скоростью 9.5 мВ за время преобразования, то есть 95 В/с, без потери точности.

Если период треугольного сигнала принять равным 0.1 с, то скорость нарастания сигнала будет равна $5 \text{ В} / 0.05 \text{ с} (100 \text{ В/с})$. В этом случае преобразователь будет с трудом отслеживать его, так как максимальная точность достигается при изменении сигнала не быстрее, чем 95 В/с. Как можно видеть, период сигнала 0.1 с или частота 10 Гц являются предельными для такого АЦП (а значит и для ADC0804). Треугольный сигнал редко нужно оцифровывать, в отличие от синусоидального. Оцифровка синусоидального сигнала осуществляется следующим образом.

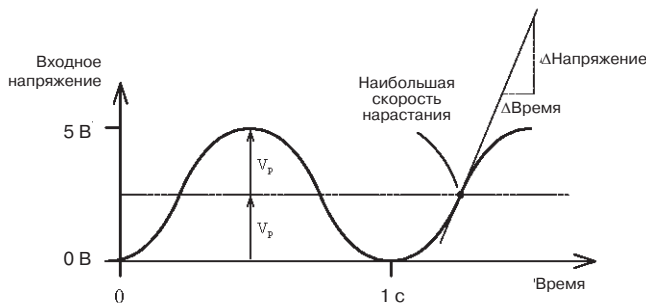


Рис. 11.10. Синусоидальный сигнал

Синусоидальный сигнал можно представить так:

$$v(t) = V_p \sin \omega t + V_p,$$

где V_p является амплитудой сигнала, ω – круговая частота, рад/с; $\omega = 2\pi f$, где f – частота сигнала в Герцах.

Скорость нарастания сигнала можно выразить так:

$$\frac{dv}{dt} = \omega V_p \cos \omega t \quad (\text{максимум при } \cos \omega t = 1)$$

$$\begin{aligned} \text{Максимальная скорость нарастания} &= \omega V_p \\ &= 2\pi f V_p \end{aligned}$$

Входной сигнал АЦП оказывает влияние на цифровой результат в течение всего цикла преобразования длительностью T_c . Поэтому для достижения наибольшей точности изменение напряжения за это время не должно превышать половины шага квантования (или половину наименее значащего бита). Шаг квантования равен отношению диапазона входного сигнала к количеству уровней квантования (разрешающей способности) АЦП. В случае n -битного АЦП количество уровней квантования будет равно 2^n , тогда половина шага квантования будет равна:

$$\frac{1}{2} \text{ шага квантования} = \frac{1}{2} \frac{\text{Диапазон входного напряжения}}{2^n}$$

Максимальная скорость нарастания входного сигнала, при которой не будет происходить потери точности:

$$\text{Максимальная допустимая скорость нарастания} = \frac{\text{Диапазон входного сигнала} / 2^{n+1}}{T_c}$$

Для достижения оптимального соотношения точности работы и максимальной частоты преобразуемого сигнала следует выбирать максимально возможный диапазон входного сигнала.

Из выражений для скорости нарастания сигнала и максимальной допустимой скорости нарастания АЦП можно вывести неравенство:

Скорости нарастания сигнала < Допустимая скорость нарастания АЦП

$$\begin{aligned} 2\pi f V_p &< \frac{\text{Диапазон входного сигнала} / 2^{n+1}}{T_c} \\ f &< \frac{\text{Диапазон входного сигнала}}{\pi V_p T_c 2^{n+1}} \end{aligned}$$

Для повышения точности преобразования и повышения максимальной частоты сигнала часто применяется так называемая *схема выборки/хранения*, которая включается между источником сигнала и входом АЦП. Поскольку у многих АЦП нет встроенной схемы выборки/хранения, то часто применяют внешние схемы выборки/хранения.

Выборка и хранение

Схемы выборки/хранения «запоминают» мгновенное значение изменяющегося напряжения и АЦП выполняет преобразование неизменного во времени сигнала. Схема выборки хранения управляется сигналом «Запомнить уровень сигнала». Запоминание происходит в два приёма, как показано на **Рис. 11.11** и **Рис. 11.12**.

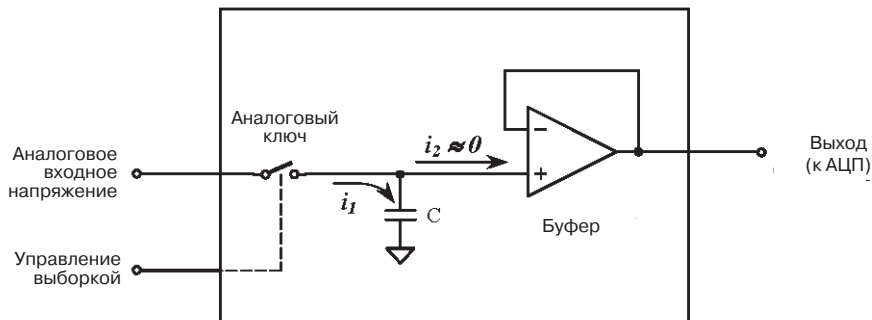


Рис. 11.11. Упрощённая схема выборки/хранения

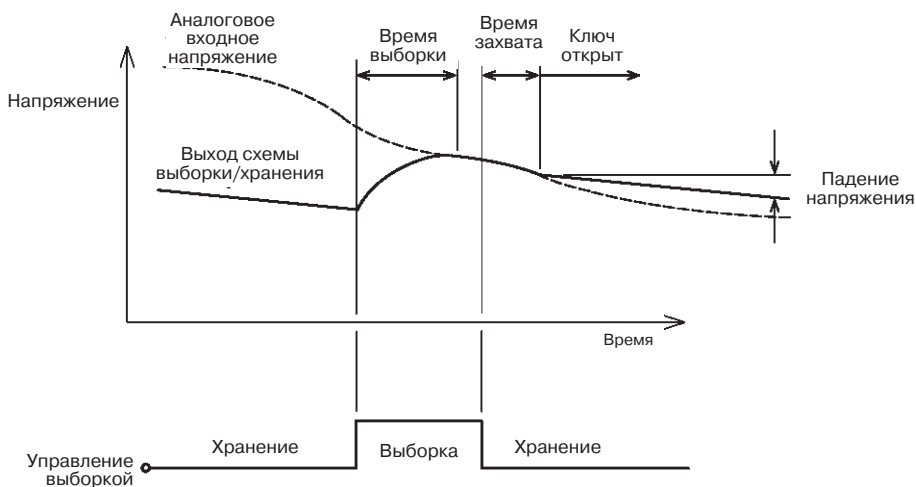


Рис. 11.12. Временная диаграмма работы схемы выборки-хранения

1. *Выборка входного сигнала.* Как только поступит сигнал «Выборка», аналоговый ключ замыкается и конденсатор заряжается до напряжения входного сигнала. Интервал времени, в течение которого конденсатор заряжается до уровня входного напряжения (с заданной точностью), называется *временем выборки*.
2. *Хранение уровня.* После прохождения сигнала «Выборка» схема переходит в режим хранения уровня сигнала, подаваемого на вход АЦП. Аналоговый ключ, к сожалению, размыкается не мгновенно, вследствие этого выходной сигнал схемы зависит от входного напряжения в течение этого промежутка времени, при этом возникает искажение выборки и АЦП может сформировать неверный код. Аналого-цифровое преобразование выполняется в течение периода хранения.

В идеальном случае выходное напряжение схемы выборки/хранения в режиме хранения остаётся неизменным на протяжении всего процесса аналого-цифрового преобразования. В реальных схемах выходное напряжение уменьшается с течением времени, что обусловлено утечкой заряда конденсатора в режиме хранения через подключенные к нему цепи и вследствие потерь в самом конденсаторе.

Ошибка дискретизации

Набор значений выборок периодического сигнала зависит от частоты выборки, **Рис. 11.13**, **Рис. 11.14** и **Рис. 11.15**. Рассмотрим случай синусоидального сигнала. Восстановленный из выборок сигнал будет похож на оригинальный сигнал, показанный на **Рис. 11.13**, если период выборки будет меньше половины периода сигнала. Под ошибкой дискретизации подразумевается различие частот оригинального и восстановленного сигналов. Сложность в том, что искажённый сигнал имеет синусоидальную форму и ту же амплитуду, как и оригинальный сигнал и оригинальный сигнал по полученным выборкам воспроизводится неправильно.

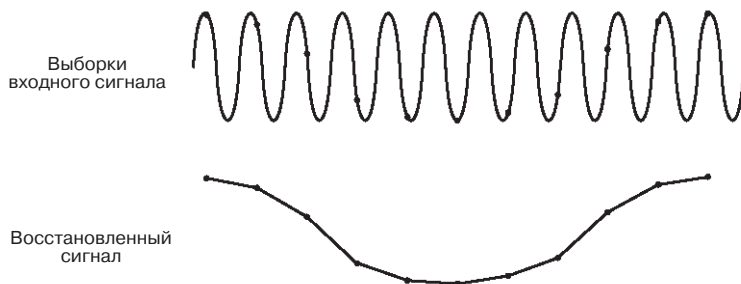


Рис. 11.13. Неправильное восстановление сигнала, частота выборки слишком мала

Если частота выборки равна удвоенной частоте входного сигнала, то восстановленный сигнал будет пилообразным, амплитуда этого сигнала будет зависеть от начальной фазы выборки.

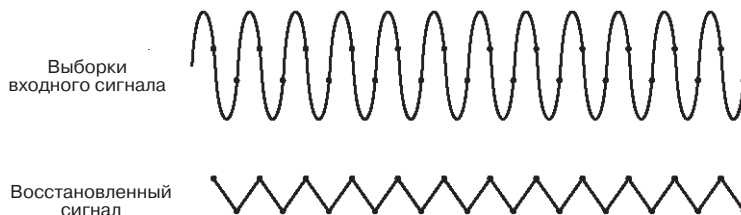


Рис. 11.14. Восстановленный сигнал, когда в течение периода сигнала делается две выборки

С увеличением частоты дискретизации увеличивается точность воспроизведения оригинального сигнала, **Рис. 11.15**.

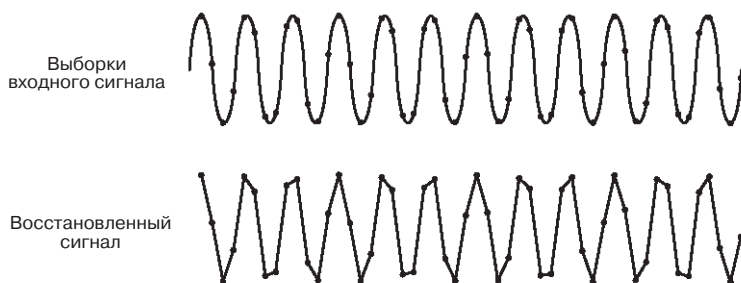


Рис. 11.15. Восстановленный сигнал, в течение периода сигнала в среднем делается 5 выборки

Прямая и эквивалентные выборки

Рассмотренные ранее выборки сигналов были прямыми, когда сигнал восстанавливается по последовательности выборок оригинального сигнала. Периодические сигналы высокой частоты можно оцифровывать и затем восстанавливать при помощи метода *эквивалентной выборки*. В этом случае сигнал восстанавливается по сохранённым в памяти наборам выборок. Восстановление оригинального сигнала показано на **Рис. 11.16**. Такой метод часто используется в цифровых осциллографах. При включении режима высокочастотного периодического сигнала формируется искусственная последовательность выборок, частота которых значительно превышает скорость работы АЦП осциллографа.

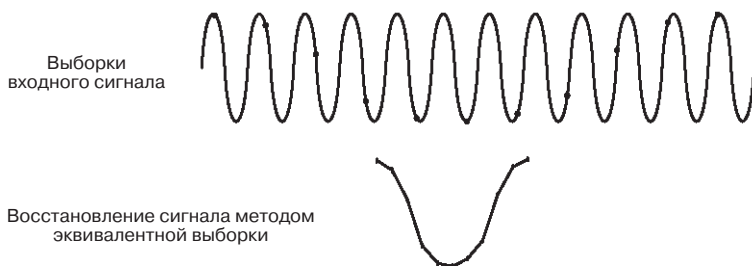


Рис. 11.16. Эквивалентная выборка

Метод эквивалентной выборки непригоден для непериодических сигналов, что иллюстрирует **Рис. 11.17**. В таких случаях выборка должна происходить и запоминаться на достаточно высокой скорости, чтобы оцифровать сигнал с приемлемой точностью. В связи с этим в цифровых осциллографах стремятся обеспечить соответствующие скорость выборки и объём памяти, что, в свою очередь, отражается на стоимости цифровых осциллографов.

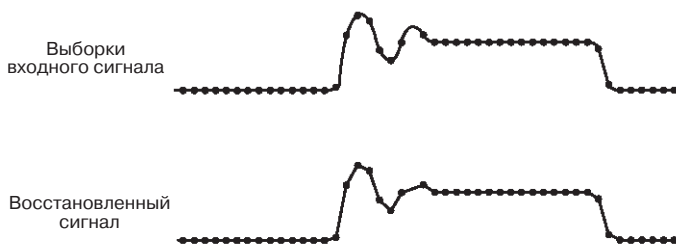


Рис. 11.17. Прямая выборка неперiodического сигнала

11.5. Класс АЦП

В предыдущих главах были рассмотрены вопросы создания классов для таких устройств, как параллельный порт, ЦАП, электродвигатели и ГУН. Таким же образом можно создать класс АЦП. Аналого-цифровое преобразование заключается в представлении аналогового входного сигнала в виде целых чисел, которые можно прочесть через параллельный порт компьютера.

Процесс преобразования для большинства АЦП происходит в следующей последовательности:

1. Запуск АЦП.
2. Ожидание завершения преобразования.
3. Чтение результата преобразования.

На входе некоторых АЦП имеется аналоговый мультиплексор, позволяющий переключать вход АЦП между несколькими аналоговыми входами устройства. В таком случае перед вышеперечисленными действиями нужно выбрать требуемый аналоговый вход. На интерфейсной плате установлен АЦП без мультиплексора, поэтому нет необходимости в выборе канала.

В классе АЦП нужно реализовать все вышеперечисленные операции. Схема интерфейсной платы предусматривает взаимодействие АЦП с параллельным портом компьютера, поэтому класс ADC лучше всего создать на базе класса `ParallelPort`.

В классе ADC должен быть только один собственный член данных, в котором будет храниться результат аналого-цифрового преобразования. Помимо конструкторов в классе должна быть функция, выполняющая аналого-цифровое преобразование и записывающая результат в собственный член данных. Также должна быть функция для чтения этого члена данных. Определение класса ADC приведено в **Листинге 11.1**.

Листинг 11.1. Заголовочный файл `adc.h` для класса ADC

```
#ifndef AdcH
#define AdcH

#include "pport.h"

class ADC : public ParallelPort
```

```
{
    private:
        unsigned char ADCValue;

    public:
        ADC(int baseaddress = 0x378);
        unsigned char ADConvert();
        unsigned char GetADCValue();
};
#endif
```

Из трёх функций наиболее важна `ADConvert()`, поэтому её рассмотрим в первую очередь. Перед тем как писать код этой функции, нужно определить, что именно требуется от параллельного порта и как его использовать.

На **Рис. 11.18** показано условное изображение ADC0804, слева показаны входы, а справа выходы. Обозначения выводов и их описания приведены в **Табл. 11.2**. Те же обозначения использованы на принципиальной схеме и нанесены на интерфейсной плате рядом с микросхемой АЦП.

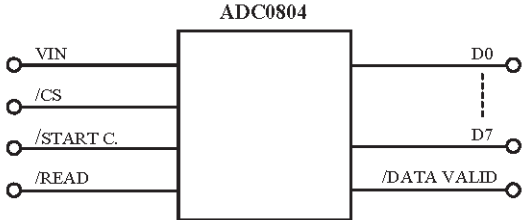


Рис. 11.18. Условное обозначение ADC0804

Таблица 11.2. Выводы микросхемы АЦП

Обозначение вывода	Вход	Выход	Назначение
VIN	•		Аналоговый вход
/CS	•		Выбор микросхемы (активирует микросхему)
/START C.	•		Запуск преобразования
/READ	•		Разрешить чтение данных
/DATA VALID		•	Готовность данных
D0–D7		•	Цифровые выходы результата преобразования
Примечание: / означает активный уровень НИЗКИЙ.			

Входы и выходы АЦП обозначены точками в соответствующих колонках **Табл. 11.2**. Теперь нужно разобраться, как работать с этими сигнала при помощи параллельного порта.

Работа с сигналами

Входной аналоговый сигнал АЦП подаётся на вывод `VIN`. В качестве входного сигнала можно воспользоваться выходными сигналами потенциометра, термистора или ЦАП в униполярном режиме (джампер установлен в положение *LINK1*).

Входы АЦП работают следующим образом. Сигнал «*выбор микросхемы*» (/CS) разрешает работу микросхемы, активный уровень НИЗКИЙ. Для чтения состояния цифровых выходов сигнал «*чтение*» (/READ) переводят в состояние НИЗКИЙ. В системах с общей шиной данных сигнал «выбор микросхемы» формируется схемой *декодирования адреса*. На интерфейсной плате нет общей шины данных, поэтому этот сигнал всегда НИЗКИЙ, то есть подключен к земле, а в параллельном порте не нужно выделять для него отдельную линию.

Компьютер должен запускать преобразование сигналом «*запуск преобразования*», подаваемым на вход /START С.. Преобразование запускается импульсом НИЗКОГО уровня сигнала. Такой импульс можно сгенерировать установкой НИЗКОГО уровня сигнала, который изначально имел ВЫСОКИЙ уровень, с последующим возвратом в исходное состояние.

По завершении преобразования на выходе «готовность данных» появится короткий импульс НИЗКОГО уровня, сигнализирующий о готовности результата преобразования. Так как этот импульс очень короткий, то можно пропустить момент готовности данных. На практике этот импульс обычно регистрируют при помощи схемы-защёлки, предотвращающей возможность его пропуска. В целях упрощения схемы на интерфейсной плате не предусмотрена схема-защёлка. Поэтому лучшим вариантом будет выдержать необходимую паузу после запуска преобразования перед чтением результата. При этом отпадает необходимость в использовании ещё одной линии параллельного порта для опроса сигнала /DATA VALID.

Подключение АЦП к параллельному порту

Теперь определим линии параллельного порта, которые будут подключены к выводам АЦП. Выходной сигнал ЦАП может служить в качестве входного сигнала для АЦП и это будет учтено при выборе линий порта. Входы и выходы АЦП и ЦАП перечислены в **Табл. 11.3**.

Таблица 11.3. Входы и выходы микросхем ЦАП и АЦП

Цифровые входы ЦАП	АЦП	
	Цифровые входы	Цифровые выходы
D0	/CS	D0
D1	/RD	D1
D2	/START С.	D2
D3		D3
D4		D4
D5		D5
D6		D6
D7		D7
		/DATA VALID

Цифровые входы ЦАП: восемь выходных линий параллельного порта нужно подключить к цифровым входам ЦАП. Таким образом, биты данных регистра BASE (D0...D7) будут входными данными ЦАП (D0...D7).

Цифровые входы АЦП: управлять входом /START С. можно при помощи бита **D0** регистра **BASE+2**. Сигналы /CS и /READ подключены к земле, поэтому цифровые выходы АЦП всегда в активном состоянии. Мы можем так поступить, поскольку АЦП не подключен к общей шине данных.

Цифровые выходы АЦП: программа должна считывать восемь выходных сигналов АЦП по параллельному порту (сигнал /DATA VALID не используется). Параллельный порт располагает пятью входными сигналами (**D3...D7**), доступными в регистре **BASE+1**. Сигналы регистра **BASE+2** не используются в качестве входных, так как они оказываются неработоспособными на высоких скоростях передачи данных.

На интерфейсной плате предусмотрен четырёхканальный мультиплексор, изображённый на **Рис. 11.19**, способствующий передаче данных посредством порта. Воспользовавшись этим устройством мы получим возможность передавать по четырём линиям порта восемь бит данных путём разбиения байта на две четырёхбитные группы. Сначала мультиплексором выделяется первые четыре бита и передаются в порт, затем выделяются оставшиеся четыре бита и также передаются в порт. При этом нужен дополнительный сигнал порта для управления выбором нужной группы битов. Выходные данные АЦП представляют собой один байт, поэтому младшую тетраду (четырёхбитную группу **D0...D3**) будем выделять НИЗКИМ уровнем, а старшую (**D4...D7**) ВЫСОКИМ.

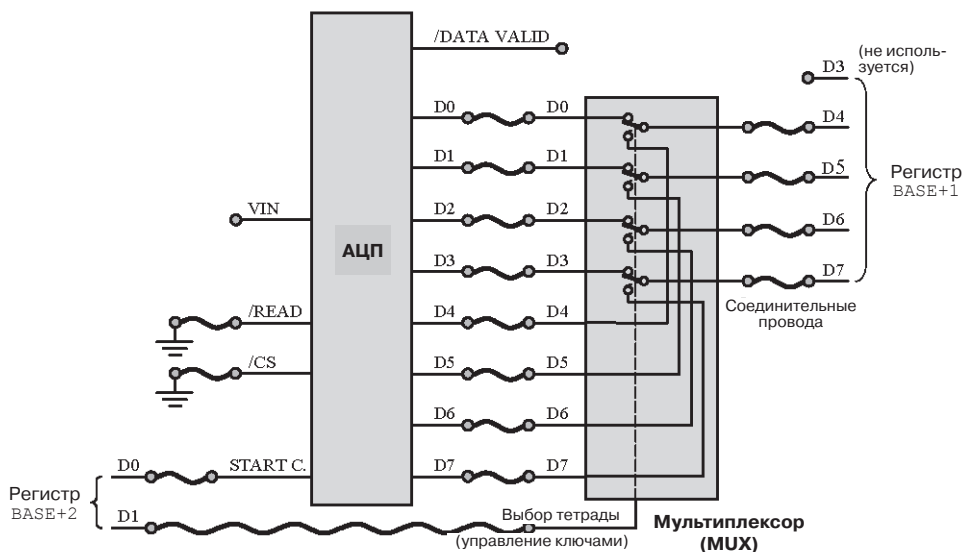


Рис. 11.19. Подключение АЦП с использованием мультиплексора

Теперь, когда появилась возможность передавать восемь бит только по четырём линиям, распределим подключения оставшихся сигналов параллельного порта. Четыре входных линии регистра **BASE+1** **D4...D7** будут принимать четыре бита от мультиплексора, передающего байт в виде последовательности двух тетрад.

Предусмотрим возможность чтения сигнала АЦП /DATA VALID, для этого одноимённый выход АЦП подключим к линии **D3** регистра **BASE+1**. При этом в

программу нужно будет внести дополнительные инструкции, осуществляющие чтение этого бита.

Цифровой вход мультиплексора: управлять выбором тетрады будем выходным сигналом **D1** регистра BASE+2.

Все подключения параллельного порта к АЦП, ЦАП и мультиплексору сведены в **Табл. 11.4**. В этой таблице не отображено подключение мультиплексора к АЦП, эти соединения показаны на **Рис. 11.19**.

Таблица 11.4. Подключение параллельного порта к АЦП, ЦАП и MUX

Регистр BASE	Регистр BASE+1	Регистр BASE+2
D0 – DAC, D0 D1 – DAC, D1 D2 – DAC, D2 D3 – DAC, D3 D4 – DAC, D4 D5 – DAC, D5 D6 – DAC, D6 D7 – DAC, D7	D3 – ADC, /DATA VALID D4 – MUX, D4 D5 – MUX, D5 D6 – MUX, D6 D7 – MUX, D7	D0 – ADC, /START C. D1 – MUX, Select
<i>Примечание:</i> 1. Сигналы АЦП CS и READ подключаются к земле. 2. Сигнал АЦП DATA VALID не используется. 3. ЦАП работает в униполярном режиме, для этого в позицию LINK1 устанавливается джампер.		

Теперь можно приступить к кодированию функции ADConvert(). В **Листинге 11.2** показано определение этой функции.

Листинг 11.2. Функция-член ADCConvert()

```
unsigned char ADC::ADCConvert()
{
    // Объявление переменных, где будут
    // храниться принятые тетрады
    unsigned char LowNibble, HighNibble;

    // Формирование импульса запуска АЦП
    WritePort2(0x01); // ВЫСОКИЙ уровень сигнала /START C.
    WritePort2(0x00); // НИЗКИЙ уровень сигнала /START C.
    WritePort2(0x01); // ВЫСОКИЙ уровень сигнала /START C.

    // Установить ВЫСОКИЙ уровень сигнал выбора тетрады (D1),
    // сигнал /START C. остаётся в ВЫСОКОМ состоянии.
    // Эти действия выполняются дольше, чем преобразование,
    // поэтому нет необходимости контролировать сигнал /DATA VALID.
    WritePort2(0x03); // 0000 0011

    // На этот момент преобразование уже завершено.
    // Прочитать старшую тетраду и обнулить младшую.
```

```

HighNibble = ReadPort1() & 0xF0;

// Установить НИЗКИЙ уровень сигнала выбора тетрады (D1).
WritePort2(0x01); // 0000 0001

// Прочитать младшую тетраду, обнулив при этом младшую,
// переместить биты на их место из старшей тетрады.
LowNibble = (ReadPort1() >> 4) & 0x0F;

// Сформировать байт из двух тетрад.
ADCValue = HighNibble + LowNibble;

return ADCValue;
}

```

Фрагменты кода из **Листинга 11.2**, выделенные жирным шрифтом, требуют пояснения. Известно, что при чтении регистра `BASE+1` данные содержатся только в битах **D4...D7**. Восьмибитные данные АЦП передаются в два приёма, сначала считывается и запоминается старшая тетрада, затем считывается и запоминается младшая тетрада. Восьмибитное значение результата АЦП получается путем сложения этих двух тетрад, которое затем сохраняется в `ADCValue`.

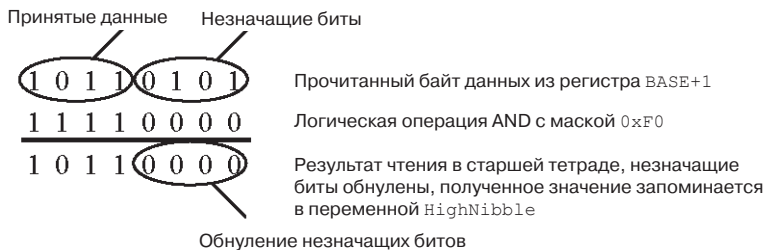


Рис. 11.20. Чтение старшей тетрады и обнуление незначащих битов

Унаследованная функция `ReadPort1()` считывает байт данных из регистра `BASE+1`. В принятом байте данных младшая тетрада содержит незначащие биты, значение которых ничем не определено и они не несут никакой информации. Незначащие биты младшей тетрады обнуляются при помощи логической операции AND с маской `0xF0`. Операция обнуления показана на **Рис. 11.20**.

После чтения младшей тетрады, которая фактически находится в старшей тетраде принятого байта, нужно сдвинуть биты принятого байта на четыре битовые позиции вправо. Теперь, когда принятая тетрада заняла своё истинное положение в байте, нужно выполнить логическую операцию AND с маской `0x0F`, обнулив тем самым незначащие биты старшей тетрады. Операции над младшей тетрадой данных показаны на **Рис. 11.21**.

Теперь есть байт (`unsigned char`) `LowNibble`, в младшей тетраде которого содержатся данные, а старшая тетрада обнулена. Аналогично в байте `HighNibble` старшая тетрада занята данными, а младшая содержит нули. Теперь нужно объединить эти две тетрады, и в результате такого объединения получится исходный байт результата АЦП, который сохраним в `ADCValue`. На **Рис. 11.22** показано, как получается значение `ADCValue`.

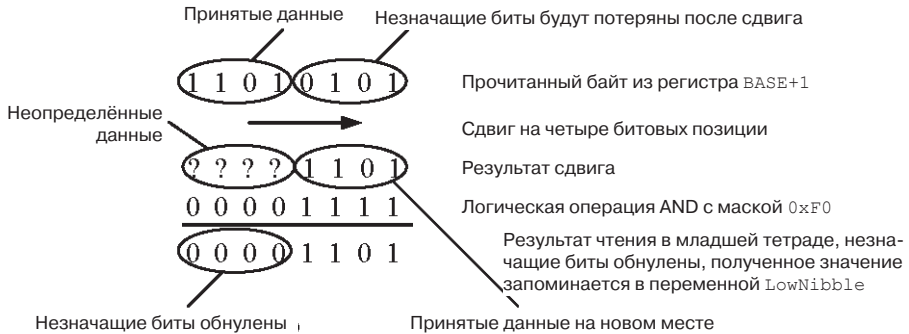


Рис. 11.21. Чтение младшей тетрады и обнуление незначимых битов

0 0 0 0 1 1 0 1	LowNibble
<u>1 0 1 1 0 0 0 0</u>	HighNibble
1 0 1 1 1 1 0 1	ADCValue

Рис. 11.22. Сложение старшей и младшей тетрад для получения результата АЦП

Функцию из **Листинга 11.2** можно переписать более рациональным образом, **Листинг 11.3**. Все операции с данными можно осуществлять в одной переменной `ADCValue`, при этом отпадает необходимость использования переменных `LowNibble` и `HighNibble`.

Листинг 11.3. Более рациональный вариант функции `ADConvert()`

```
unsigned char ADC::ADConvert()
{
    // Формирование импульса запуска АЦП
    WritePort2(0x01); // ВЫСОКИЙ уровень сигнала /START С.
    WritePort2(0x00); // НИЗКИЙ уровень сигнала /START С.
    WritePort2(0x01); // ВЫСОКИЙ уровень сигнала /START С.

    // Установить ВЫСОКИЙ уровень сигнала выбора тетрады (D1),
    // сигнал /START С. остаётся в ВЫСОКОМ состоянии.
    // Эти действия выполняются дольше, чем преобразование,
    // поэтому нет необходимости контролировать сигнал /DATA VALID.
    WritePort2(0x03); // 0000 0011

    // На этот момент преобразование уже завершено.
    // Прочитать старшую тетраду и обнулить младшую.
    ADCValue = ReadPort1() & 0xF0;

    // Установить НИЗКИЙ уровень сигнала выбора тетрады (D1).
    WritePort2(0x01); // 0000 0001

    // Прочитать младшую тетраду и восстановить исходное
```



```
// 8-битное число.
    ADCValue += (ReadPort1() >> 4) & 0x0F;

    return ADCValue;
}
```

Определение класса ADC завершается определением функций-членов, **Листинг 11.4**. Внешние по отношению к этому классу функции смогут обращаться к собственному члену данных ADCValue посредством функции-члена GetADCValue().

Листинг 11.4. Определение функций класса ADC в файле adc.cpp

```
#include "adc.h"

ADC::ADC(int baseaddress) : ParallelPort(baseaddress)
{
    ADCValue = 0;
}

unsigned char ADC::ADConvert()
{
    WritePort2(0x01); // Запуск преобразования
    WritePort2(0x00);
    WritePort2(0x01);

    WritePort2(0x03); // Разрешить чтение старшей тетрады
    ADCValue = ReadPort1() & 0xF0;

    WritePort2(0x01); // Разрешить чтение младшей тетрады
    ADCValue += (ReadPort1() >> 4) & 0x0F; // Восстановление
                                         // исходных данных

    return ADCValue;
}

unsigned char ADC::GetADCValue()
{
    return ADCValue;
}
```

11.6. Измерение напряжения при помощи АЦП

Ранее уже рассматривалось измерение напряжения при помощи ГУН в главе 10. Функция MeasurePeriod() возвращала число, пропорциональное периоду ГУН, которое затем использовалось для вычисления напряжения на входе ГУН. Можно воспользоваться этой программой, заменив класс VCO на класс ADC. В таком случае число, пропорциональное входному напряжению, будет возвращать функция ADConvert(). Входное напряжение ГУН в предыдущей программе задавалось при

помощи класса DAC. В новой программе входное напряжение АЦП (на вход VIN) также будет подаваться от ЦАП.

В **Листинге 11.5** приведена модифицированная функция `main()` из **Листинга 10.6**, где вместо ГУН используется АЦП.

Листинг 11.5. Измерение напряжения при помощи АЦП в файле `voltage.cpp`

```
#include <conio.h>
#include <bios.h>

#include "dac.h"
#include "adc.h"

void main()
{
    DAC Dac;
    ADC Adc;
    int Quit = 0, key;
    unsigned char DACbyte;

    clrscr();
    Dac.SendData(0); // Обнулить ЦАП

    while(!Quit)
    {
        gotoxy(10, 10);
        cprintf("The ADC output is: %3u",
                (int) Adc.ADConvert());

        if(bioskey(1) != 0)
        {
            DACbyte = Dac.GetLastOutput();
            key = bioskey(0);
            switch(key)
            {
                /* Alt-X */ case 0x2d00:
                            Quit = 1;
                            Dac.SendData(0); // Обнулить ЦАП
                            break;

                /* BBEFX */ case 0x4800:
                            if(DACbyte > 247) // ограничение
                                DACbyte = 247; // верхнего уровня
                            Dac.SendData(DACbyte + 8);
                            break;

                /* ВНИЗ */ case 0x5000:
                            if(DACbyte < 8) // ограничение
                                DACbyte = 8; // нижнего уровня
            }
        }
    }
}
```

```
        Dac.SendData(DACbyte + 8);  
    }  
}  
}
```

Были изменены несколько выражений функции `main()`. Обработка кнопок клавиатуры выполняется как и в **Листинге 10.6** программы работы с ГУН. В изменённой функции добавлены выражения, ограничивающие максимальное и минимальное значения, выводимые в ЦАП.

Сборка исполняемого файла		
Требуемые файлы	Листинг	Содержимое файла проекта
pport.cpp	Листинг 10.8	pport.cpp
pport.h	Листинг 10.7	
dac.cpp	Листинг 10.10	Dac.cpp
dac.h	Листинг 10.9	
adc.cpp	Листинг 11.4	adc.cpp
adc.h	Листинг 11.1	
voltage.cpp	Листинг 11.5	voltage.cpp

Для компиляции функции `main()` (**Листинг 11.5**) необходимо полное определение класса `DAC`. Класс `USO` в этой программе больше не используется, поэтому нет необходимости определения этого класса. С другой стороны, наличие этого класса не повлияет на работу программы. После компиляции и сборки программа будет выводить на экран целое число, пропорциональное входному напряжению АЦП, подаваемому на вход `VIN`.

Можно немного изменить программу, чтобы вместо целого числа выводилось напряжение. На выходе АЦП формируется 8-битное число, принимающее значения из диапазона 0...255, включительно, то есть 256 разных значений. Весь диапазон входного сигнала квантуется АЦП на 256 уровней напряжения. Таким образом диапазон 5 В делится на 256 интервалов. Значение 0 на выходе АЦП тогда соответствует 0 В на входе `VIN`, а значение *i* соответствует напряжению

$$\text{Входное напряжение} = \frac{\text{Диапазон входного сигнала}}{\text{Количество уровней квантования}} \times i$$

С точки зрения компилятора выражение `5/256` является операцией деления целого числа на целое число, частное тоже будет целого типа и равно 0. Поэтому это выражение в программе нужно записать так:

```
5.0/256.0 * i;
```

Теперь компилятор преобразует эти числа в числа с плавающей точкой, а результат отношения будет ненулевым числом с плавающей точкой. Вызов функции `cprintf()` в **Листинге 11.5** нужно изменить с учётом этих изменений:

```
cprintf("The ADC output is: %5.2f (V)",
        5.0/256.0 * Adc.ADConvert());
```

Теперь программа будет отображать на экране значение напряжения, подаваемого на вход АЦП (VIN). Необходимые подключения на плате приведены на **Рис. 11.19**, в **Табл. 11.4** и **Табл. 11.5**.

Таблица 11.5. Подключение некоторых выводов АЦП

	ADC0804 (U8)
VDAC (вывод 7, U10B)	VIN
Земля	/READ
Земля	/CS

Примечание

Нужно убедиться, что выход ЦАП переключен в униполярный режим (0...+5 В) при помощи джампера в позиции *LINK1*. Затем подключить батарею 9 В **до** подключения выхода ЦАП ко входу АЦП.

11.7. Измерение температуры при помощи АЦП

В предыдущей главе была разработана программа, измеряющая температуру при помощи ГУН и термистора. В этой программе (**Листинг 10.12**) нужно сделать незначительные изменения, чтобы она могла измерять температуру с использованием АЦП. Модифицированная программа приведена в **Листинге 11.6**. При этом передаточная характеристика термистора будет точно интерпретироваться в полном диапазоне 0...+5 В, поскольку линейность АЦП очень высока во всём диапазоне входного напряжения. Аналогичная линейность ГУН приходится лишь на диапазон +2.2...+2.8 В.

Листинг 11.6. Измерение температуры при помощи АЦП, файл temp.cpp

```
/*
Эта программа предназначена для измерения температуры
при помощи АЦП и термистора, который расположен на
интерфейсной плате и выходное напряжение которого
подаётся на вход АЦП. Выходные данные АЦП будут
пропорциональными выходному сигналу цепи термистора,
а следовательно и температуре корпуса термистора.
Программа осуществляет калибровку термистора по двум
значениям температуры. Калибровочное выражение затем
будет линейно преобразовывать сигнал термистора
в значение температуры. После калибровки программа
будет показывать значение температуры
*/
```

```

*****/

#include <bios.h>
#include <conio.h>
#include <iostream.h>

#include "adc.h"

void main()
{
    ADC Adc;
    int Quit = 0, HiFlag = 0, LoFlag = 0;
    int key = 0;
    float HiTemp, LoTemp, Temp;
    long int HiCount, LoCount;

    clrscr();

    while(!Quit)
    {
        Adc.ADConvert();

        gotoxy(10, 10);
        if((HiFlag == 1) && (LoFlag == 1))
        {
            Temp = LoTemp + (HiTemp - LoTemp) *
                (Adc.GetADCValue() - LoCount) /
                (HiCount - LoCount);
            cprintf("The temperature is: %6.1f (deg)",
                Temp);
        }
        else
            cprintf("The ADC value is: %3u",
                (int) Adc.GetADCValue());

        if(bioskey(1) != 0)
        {
            key = bioskey(0);

            switch(key)
            {
                case 0x2d00 : /* Alt-X */
                    Quit = 1;
                    break;

                case 0x4800 : /* BBEPX */
                    gotoxy(10, 5);
                    count << "Enter upper";
                    Calibration Temp.
            }
        }
    }
}

```

```

        cin >> HiTemp;
        HiCount = Adc.GetADCValue();
        HiFlag = 1;
        break;

    case 0x5000 : /* ВНИЗ */
        gotoxy(10, 6);
        cout << "Enter Lower
                Calibration Temp.";
        cin >> LoTemp;
        LoCount = Adc.GetADCValue();
        LoFlag = 1;
    }
}
}
}

```

Сборка исполняемого файла		
Требуемые файлы	Листинг	Содержимое файла проекта
pport.cpp	Листинг 10.8	pport.cpp adc.cpp Temp.cpp
pport.h	Листинг 10.7	
adc.cpp	Листинг 11.4	
adc.h	Листинг 11.1	
temp.cpp	Листинг 11.6	

Теперь нужно немного изменить схему подключения устройств. Сейчас ЦАП не формирует входного напряжения, вместо него используется выходной сигнал цепи термистора. Сигнал температуры с термистора подаётся на вход АЦП (VIN), **Табл. 11.6**. Остальные подключения АЦП и мультиплексора приведены в **Табл. 11.19**.

Таблица 11.6. Подключение термистора к АЦП

Термистор	ADC0804 (U8)
VTH	VIN /READ (подключен к земле) /CS (подключен к земле)

Оказалось, что очень просто переделать программу, работавшую с ГУН, для работы с АЦП. При создании классов предусматривалась возможность такой замены устройств в дальнейших версиях программы. Приведённые примеры иллюстрируют простоту сопровождения и модернизации объектно-ориентированных программ.

11.8. Заключение

В этой главе были описаны устройство и работа аналого-цифровых преобразователей. Были рассмотрены наиболее распространённые типы АЦП и принципы их работы. Было также рассказано о том, какую важную роль играет схема выборки/хранения и как возникает ошибка дискретизации при недостаточно высокой частоте дискретизации (выборки).

Приобретённые знания об объектно-ориентированном программировании были использованы для создания программ, работающих с АЦП через параллельный порт компьютера. Аналогично классу `VCO`, созданному в главе 10, был разработан класс `ADC`. Объектно-ориентированный подход при создании программ позволил перейти от ГУН к использованию АЦП, сделав лишь незначительные изменения в программе.

11.9. Литература

1. Fluke, *The ABC's of Oscilloscopes*, Fluke Corporation, 1997.
2. Horowitz, P. And Hill, W., *The Art of Electronics*, Cambridge University Press, Cambridge, 1989.
3. Loveday, G., *Microprocessor Sourcebook*, Pitman Publishing Limited, London, 1986.
4. NS *DATA CONVERSION/ACQUISITION Databook*, National Semiconductor Corporation, 1984.
5. Stiffler, K., *Design with Microprocessors for Mechanical Engineers*, McGraw-Hill, 1992.
6. Webb, R. E., *Electronics for Scientists*, Ellis Horwood, New York, 1990.
7. Wobshall, D., *Circuit Design for Electronic Instrumentation*, McGraw-Hill, 1987.
8. Van Gilluwe, F., *The Undocumented PC*, Addison Wesley, 1994.
9. Winston, P. H., *On to C++*, Addison Wesley, 1994.

12 Сбор данных с использованием перегрузки операторов

Содержание главы:

- Передача параметров по значению и по ссылке.
- Возврат значения по ссылке.
- Перегрузка операторов.
- Конструктор копии и оператор присвоения.
- Файловый ввод/вывод.
- Дружественные функции.
- Транзитные объекты.
- Сбор данных при помощи АЦП.

12.1. Введение

Некоторые программы из этой книги можно улучшить, уменьшив объём используемой ими памяти и увеличив скорость их работы, задействовав механизмы *передачи параметров по ссылке и возврата результата по ссылке*. Программные конструкции тоже можно упростить, воспользовавшись *перегрузкой операторов и дружественными функциями*.

В этой главе будут созданы программы сбора данных с использованием перегрузки операторов, в которых будут реализованы вышеупомянутые особенности. Сбор данных подразумевает преобразование сигналов в цифровую форму, например с помощью АЦП. Затем данные обрабатываются и/или записываются в запоминающее устройство, например на винчестер, или, в некоторых случаях выводятся на устройство отображения информации, например, на экран компьютера или принтер.

12.2. Перегрузка операторов

Когда имеет место перегрузка оператора, то выполняемое им действие зависит от типа его аргументов. Например, в случае оператора деления (/) могут получиться два разных результата, приведённых ниже. В первом случае получается целое частное, а во втором с плавающей точкой:

```
5/2;      // результат равен 2
5.0/2.0 // результат равен 2.5
```

Аналогично, двойная угловая скобка может выполнять различные операции:


```
int y = 200;
cout << y; // 200 отправлено на стандартное устройство вывода
y << 1;    // выполняется битовый сдвиг влево на 1 позицию
```

Действия оператора определяются типом его операнда. В выражении `cout << y` объект `cout` является экземпляром класса `ostream`, а `y` объектом типа `int`. Оператор `<<` выполняет все необходимые действия для вывода значения `y` на экран. В выражении `y << 1` оба операнда типа `int`, значит, в этом случае происходит сдвиг влево.

Операторы, перечисленные в Табл. 12.1, не могут быть перегружены.

Таблица 12.1. Операторы, которые нельзя перегружать

.	?:	::	.*	sizeof
---	----	----	----	--------

Перегрузку операторов рассмотрим на фрагменте программы, где двойная правая угловая скобка (`<<`) и двойная левая угловая скобка (`>>`) перегружены для выполнения следующих действий:

1. Выполнение аналого-цифрового преобразования при помощи объекта `Adc` класса `ADC` и сохранение полученных данных в переменной `value` типа `unsigned char`. Операция должна записываться так:
`Adc >> value;`
2. Выполнение аналого-цифрового преобразования и пересылка результата преобразования на стандартное устройство вывода (экран). Объекты `cout` и `Adc` являются экземплярами классов `ostream` и `ADC`, соответственно. Записываться эта операция будет так:

```
cout << Adc;
```

Используя подобные выражения можно упростить программы сбора данных, использующих АЦП и где нужно записывать данные в файл или отображать на экране.

Операторы можно перегрузить двумя разными способами, особым образом определяя функции перегрузки операторов:

1. Как функцию-член класса.
2. Как обычную функцию вне класса.

Вышеприведённые способы перегрузки операторов будут рассматриваться в следующих разделах. Операторы могут быть унарными (как, например, `++` в выражении `++i`), или бинарными (`+` в выражении `x+y`). Унарные операторы, перечисленные в Табл. 12.2, выполняют действия над объектом класса `ObjectX`. Бинарные операторы работают с двумя объектами, один объект класса `ObjectX`, другой `ObjectY`. В данном примере перегружается символ `@`.

Таблица 12.2. Заголовки функций перегрузки операторов

Унарный оператор в качестве функции-члена	<code>ObjectX::operator@()</code>
Унарный оператор в качестве обычной функции	<code>operator@(ObjectX x)</code>
Бинарный оператор в качестве функции-члена	<code>ObjectX::operator@(ObjectY y)</code>
Бинарный оператор в качестве обычной функции	<code>operator@(ObjectX x, Object y)</code>

Перегруженные выше операторы используются следующим образом. В случае унарного оператора, операнд должен находиться справа от оператора. Например, если *x* является объектом класса `ObjectX`, оператор будет использоваться следующим образом:

```
@x;
```

Операнды бинарного оператора находятся по обе стороны от него. Если *x* является экземпляром класса `ObjectX`, а *y* класса `ObjectY`, то оператор будет использоваться так:

```
x @ y;
```

Операторы, перегруженные как функции-члены и как обычные функции, используются одинаковым образом. Прежде чем приступить к изучению перегрузки операторов, следует разобраться с такими понятиями языка C++, как передача данных по ссылке, по значению и копирование объектов конструктором копий.

12.2.1. Передача параметров функции по значению

В разработанных ранее программах широко используются функции, принимающие параметры в качестве входных данных. Во время вызова функции на место этих параметров подставляются копии значений фактических аргументов (те значения, которые используются в вызываемой функции). Переменные-копии значений создаются в момент вызова функции и уничтожаются, когда функция завершает свою работу. В результате такой замены изменение значения переменной-копии не отражается на значении исходной переменной.

Передачу параметров функции можно продемонстрировать на следующем примере, где происходит суммирование возрастающей последовательности из *n* чисел, начинающейся с нуля.

```
#include <iostream.h>

// ВНИМАНИЕ! Результат этой функции недоступен вне функции
void FindSum(int sum, int n)
{
    for(int j = 0; j < n; j++)
        sum = sum + j;
}

void main()
{
    int Sum = 0;
    int n = 10;

    FindSum(Sum, n); // Вызов функции FindSum()
    cout << "The sum of " << n << " integers is " << Sum
         << endl;
}
```

Программа выведет на экран следующее:

```
The sum of 10 integers is 0
```

При вызове функции создаются переменные-копии `Sum` и `n`. Как и следовало ожидать, `Sum` меняет своё значение внутри этой функции. Когда выполнение функции завершается, эта переменная уничтожается, а значит и полученный результат. В результате этого объявленная в функции `main()` переменная `Sum` сохраняет своё начальное значение, то есть 0.

Такой способ передачи параметров называется *передачей по значению*, когда создаётся переменная-копия. Метод обладает двумя недостатками:

1. Затрачивается лишнее время на создание копии.
2. Расходуется оперативная память.

Если параметр занимает большой объем памяти, то это приведёт к затратам времени на создание копии этих данных и выделению дополнительной памяти для их размещения.

12.2.2. Передача параметров функции по ссылке

Другой способ передачи параметров называется *передачей по ссылке*. Передача по ссылке даёт возможность функции менять значение переданной ей переменной. Рассмотренный в разделе 12.2.1 фрагмент программы приведён ниже, претерпев незначительные, на первый взгляд, изменения. В этом примере функция меняет значение переменной `Sum` посредством параметра `sum`. Функция использует саму переменную `Sum` для возврата результата, а не её копию.

```
#include <iostream.h>

void FindSum(int& sum, int n) // поменялся заголовок функции
{
    for(int j = 0; j < n; j++)
        sum = sum + j;
}

void main()
{
    int Sum = 0;
    int n = 10;

    FindSum(Sum, n);
    cout << "The sum of " << n << " integers is " << Sum
         << endl;
}
```

Программа напечатает на экране:

```
The sum of 10 integers is 45
```

Изменения программы выделены жирным шрифтом. Параметр `sum` объявлен не как `int`, а как *ссылка на int*, обозначаемая `int&`. Поэтому при вызове этой функции не создаётся переменной-копии, функция выполняет операции над самой переменной из окружения вызова, то есть переменной `Sum`, объявленной в функции `main()`.

Передача параметров по ссылке оказывается более эффективным и с точки зрения времени выполнения программы и с точки зрения расхода памяти. Функция может возвращать результат с помощью ссылок и как обычно, в возвращаемом значении.

Служебное слово *const* и параметры-ссылки

При объявлении параметров, передаваемых по ссылке, можно указывать служебное слово `const`, чтобы предотвратить изменение значения переменной. То же относится и к объявлению переменных, передаваемых по значению. В обоих случаях значение переменной не может быть изменено внутри функции.

В предыдущем примере нельзя использовать заголовок:

```
Void FindSum(const int& sum, int n)
```

В противном случае не будет возможности менять значение параметра `sum`, передаваемого по ссылке, то есть возвращать результат функции.

12.2.3. Выбор способа передачи параметров

Передача параметров по значению предотвращает изменение исходных значений передаваемых параметров из контекста вызова. При передаче объектов по значению есть одна существенная особенность, показанная в следующем примере.

Рассмотрим простую иерархию классов и программу, **Листинг 12.1**.

Листинг 12.1. Проблемы при передаче параметров по значению

```
// Результат выполнения программы ОШИБОЧЕН!
#include <iostream.h>

class Base
{
    private:
        int BaseClassData;

    public:
        Base(int baseclassdata)
        {
            BaseClassData = baseclassdata;
        }

        virtual int GetClassData() const // Константная функция
        {
            return BaseClassData;
        }
};

class Derived : public Base
{
    private:
```

```

        int DerivedClassData;

    public:
        Derived(int derivedclassdata,
                int baseclassdata) : Base(baseclassdata)
        {
            DerivedClassData = derivedclassdata;
        }

        int GetClassData() const // Константная функция
        {
            return DerivedClassData;
        }
};

int GetData(const Base baseObject) // Передача по значению
{
    return baseObject.GetClassData();
}

void main()
{
    Base* BasePtr;
    int ClassData;

    BasePtr = new Base(100);
    ClassData = GetData(*BasePtr);
    cout << "Base class data " << ClassData << endl;
    delete BasePtr;

    BasePtr = new Derived(200, 100);
    ClassData = GetData(*BasePtr);
    cout << "Derived class data " << ClassData << endl;
    delete BasePtr;
}

```

Обратите внимание, что обе функции `GetClassData()` из **Листинга 12.1** объявлены со служебным словом `const`:

```

int GetClassData() const
{
    return DerivedClassData;
}

```

Такие функции называются *константными функциями*. Такие функции не могут менять данные-члены своего класса.

Теперь разберём функцию `GetData()`:

```

int GetData(const Base baseObject) // Передача по значению
{

```

```
    return baseObject.GetClassData();  
}
```

Параметр `baseObject` функции `GetData()`, не принадлежащей классу, передаётся по значению как объект со свойством `const`. Следовательно, этот объект не может быть изменён функцией `GetData()`. Поэтому выражение `baseObject.GetClassData()` не должно изменять `baseObject`. Изменению препятствует тот факт, что функция `GetClassData()` объявлена как константная. В следующей программе, когда параметры будут передаваться по ссылке, они будут передаваться как объекты со свойством `const`, чтобы предотвратить их изменение функцией. При этом программы будут очень простыми, что позволит сконцентрировать внимание на передаче параметров по ссылке и по значению.

Функция `GetData()` предназначена для чтения члена данных класса. Если `baseObject` класса `Base`, то функция вернёт значение `BaseClassData`, а если класса `Derived`, то будет возвращено значение `DerivedClassData`.

В этой программе есть два разных вызова функции `GetData()`. Рассмотрим первый случай:

```
Base* BasePtr = new Base(100);  
int ClassData = GetData(*BasePtr);
```

При этом предполагается, что в теле функции будет вызвана функция-член `GetClassData()`, принадлежащая классу `Base`. Так и происходит, возвращается значение 100 члена `BaseClassData` и присваивается переменной `ClassData`. Теперь обратимся к следующему вызову:

```
Base* BasePtr = new Derived(200, 100);  
int ClassData = GetData(*BasePtr);
```

Аналогично, предполагается, что внутри функции `GetData()` произойдёт вызов функции-члена `GetClassData()`, принадлежащей классу `Derived`. Переменной `ClassData` должно быть присвоено значение 200. На самом же деле этого не происходит. Программа приводит к неожиданному результату, присваивая переменной `ClassData` значение 100. Следует отметить, что параметр передаётся путём разыменования указателя. Известно, что указатель на объекты базового класса может указывать и на объекты производных классов. Если бы не указатель, то не было бы возможности передать объект класса `Derived` в качестве фактического аргумента параметра класса `Base`. При попытке передачи объекта производного класса вместо базового компилятор сообщил бы о несоответствии типов.

Во втором случае, приведённом выше, при вызове функции `GetData()` создаётся **копия** объекта `Base`, а не `Derived`, поскольку `GetData()` принимает параметры по значению. Получается, что производный класс недоступен в функции `GetData()`, доступны лишь элементы базового класса, унаследованные в производном классе. Такая ситуация типична для случаев, когда тип фактического аргумента не соответствует типу параметра. Компилятор не может обнаружить такие несоответствия, поскольку они происходят только во время выполнения. Программа выводит на экран следующую информацию, которая позволяет обнаружить ошибку:

```
Base class data 100  
Derived class data 100
```

Известно, что класс `Derived` не содержит значение 100. Программа должна сначала записать, а затем прочитать значение 200. Проблему можно решить, передавая параметр `baseObject` функции `GetData()` по ссылке. При передаче по ссылке копия фактического аргумента не создаётся и в функции `GetData()` будет доступен весь объект класса `Derived`, функция вернёт правильное значение 200. Правильный вариант программы приведён в **Листинге 12.2**.

Листинг 12.2. Исправленная программа из Листинга 12.1

```
// Результат выполнения программы правилен
#include <iostream.h>

class Base
{
    private:
        int BaseClassData;

    public:
        Base(int baseclassdata)
        {
            BaseClassData = baseclassdata;
        }

        virtual int GetClassData() const
        {
            return BaseClassData;
        }
};

class Derived : public Base
{
    private:
        int DerivedClassData;

    public:
        Derived(int derivedclassdata,
                int baseclassdata) : Base(baseclassdata)
        {
            DerivedClassData = derivedclassdata;
        }

        int GetClassData() const // Константная функция
        {
            return DerivedClassData;
        }
};

int GetData(const Base& baseObject) // Передача по ссылке
{
```

```
        return baseObject.GetClassData();
    }

void main()
{
    Base* BasePtr;
    int ClassData;

    BasePtr = new Base(100);
    ClassData = GetData(*BasePtr);
    cout << "Base class data " << ClassData << endl;
    delete BasePtr;

    BasePtr = new Derived(200, 100);
    ClassData = GetData(*BasePtr);
    cout << "Derived class data " << ClassData << endl;
    delete BasePtr;
}
```

Программа выводит на экран следующее:

```
Base class data 100
Derived class data 200
```

Из этого примера видно, что если есть необходимость передать в функцию объект, то это лучше всего сделать по ссылке. Как показано в **Листинге 12.2** перед параметром можно указать служебное слово `const` для предотвращения изменения объекта внутри функции.

12.2.4. Конструктор копии

В предыдущих главах экземпляры классов создавались при помощи автоматически созданных и написанных программистом конструкторов. Конструктор копии это особый конструктор, предназначенный для создания копии объекта. Если разработчик класса не создал свой собственный конструктор копии, то компилятор сгенерирует конструктор копии автоматически (конструктор копии по умолчанию). Такой конструктор создаёт копию объекта, последовательно копируя все члены одного объекта в другой в трёх различных случаях:

1. Когда нужно создать копию объекта при передаче объекта в качестве параметра функции по значению.
2. Когда объект возвращается как результат по значению и нужно создать его копию.
3. Когда объект создаётся и инициализируется с использованием другого объекта, переданного в конструктор в виде параметра.

Ранее созданный конструктором по умолчанию объект также можно инициализировать оператором присвоения. Оператор присвоения должен выполнять те же действия, что и конструктор копии. Если разработчик не перегрузил этот оператор, то компилятор сможет выполнять эти действия над объектами. Обсуждение перегрузки оператора присвоения отложим, пока не разберёмся основами перегрузки операторов.

Приведём примеры создания экземпляров класса при помощи разных конструкторов:

```
DCMotor Motor1;           // конструктор по умолчанию
DCMotor Motor2(Motor1);   // конструктор копии
DCMotor Motor3;           // конструктор по умолчанию
Motor3 = Motor1;          // оператор присвоения
```

Создадим программу, которая работает с массивами, используя объект `IntArray`. Она поможет усвоить понятие конструктора копии. Определение класса `IntArray` приведено в **Листинге 12.3**.

Листинг 12.3. Заголовочный файл `intarray.h` с определением класса массива целых чисел

```
#ifndef IntarrayH
#define IntarrayH

class IntArray
{
    private:
        int NumInts;
        int* ArrayPointer;

    public:
        IntArray();           // Конструктор по умолчанию
        IntArray(int numints); // Конструктор
        ~IntArray();          // Деструктор
        void EnterArray();     // Прочие функции-члены
        void PrintArray();
};
#endif
```

Класс `IntArray` представляет собой массив целых чисел заданной длины. Член `NumInts` содержит количество элементов, а указатель `ArrayPointer` указывает на участок памяти, где был динамически размещён этот массив. Деструктор `~IntArray()` освобождает память, которая была занята массивом. Функция `EnterArray()` обеспечивает пользователю возможность ввести значения массива с клавиатуры. Функция `PrintArray()` служит для вывода содержимого массива на экран.

В конструкторы `IntArray` и конструктор по умолчанию инициализируют члены данных `NumInts` и `ArrayPointer`. Если конструктор вызывается с параметром, то происходит динамическое выделение памяти для размещения массива:

```
IntArray::IntArray() // Конструктор по умолчанию
{
    NumInts = 0;
    ArrayPointer = NULL;
}

IntArray::IntArray(int numints) // Конструктор
```

```
{
    if(numints <= 0)
    {
        NumInts = 0;
        ArrayPointer = NULL;
    }
    else
    {
        NumInts = numints;
        ArrayPointer = new int[NumInts];
    }
}
```

Видно, что если параметр `numints` равен 0 или отрицателен, то `NumInts` обнуляется, а указателю `ArrayPointer` присваивается предопределённая константа `NULL`, означающая, что указатель ни на что не указывает.

Предположим, что в функции `main()` создаются два объекта `A` и `B` класса `IntArray`. Затем нужно передать объекты `A` и `B` по значению в функцию `AddArrays()`, выполняющей сложение объекта `A` класса `IntArray` с объектом `B` класса `IntArray`:

```
void AddArrays(IntArray a, IntArray b) // Передача по значению
{
    // вывод результата суммирования на экран
}

void main()
{
    IntArray A(5);
    IntArray B(5);

    AddArrays(A, B);
    .
    .
    .
}
```

Когда объекты `A` и `B` передаются в качестве фактических аргументов по значению в функцию `AddArrays()` должны создаваться их копии. Для этого будет вызван сгенерированный компилятором конструктор копий, схематичное изображение процесса создания копий показан на **Рис. 12.1**.

Автоматический конструктор копий (тот, который был создан компилятором) копирует объекты путём копирования всех их членов. С другой стороны, динамически выделенная память не копируется, поэтому указатели оригинального объекта и объекта-копии указывают на одну и ту же область памяти.

При завершении работы функции `AddArrays()` вызывается деструктор `~IntArray()`. Деструктор освободит память, занятую копиями объектов `A` и `B`, а вместе с ними и их динамически созданные массивы. Результат выполнения деструктора показан на **Рис. 12.2**.

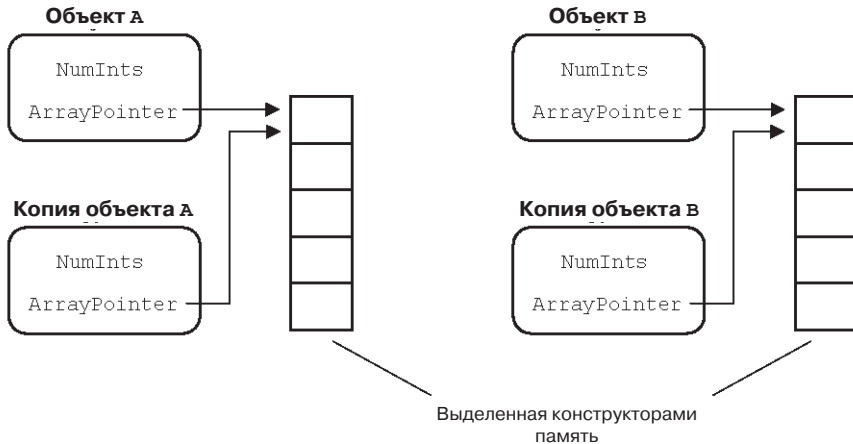


Рис. 12.1. Копирование объектов А и В конструктором копий, сгенерированным компилятором



Рис. 12.2. Результат удаления копий объектов А и В

Данные объектов А и В, созданных в функции `main()` теперь потеряны. Проблему, созданную автоматическим конструктором копий, можно решить, написав другой конструктор копий, способный копировать не только члены класса, но и те данные, на которые они указывают. Наличие такого конструктора исправит ситуацию, проиллюстрированную на **Рис. 12.1** и данные будут выглядеть следующим образом, **Рис. 12.3**.

Теперь после вызова деструктора для уничтожения копий, исходные данные не повреждаются, **Рис. 12.4**.

Листинг 12.4. Определение конструктора копий в файле `intarray.h`

```
#ifndef IntarrayH
#define IntarrayH

class IntArray
{
    private:
        int NumInts;
        int* ArrayPointer;

    public:
```

```

IntArray();
IntArray(int numints);
IntArray(const IntArray& intArray);
~IntArray();
void EnterArray();
void PrintArray();
}
#endif

```

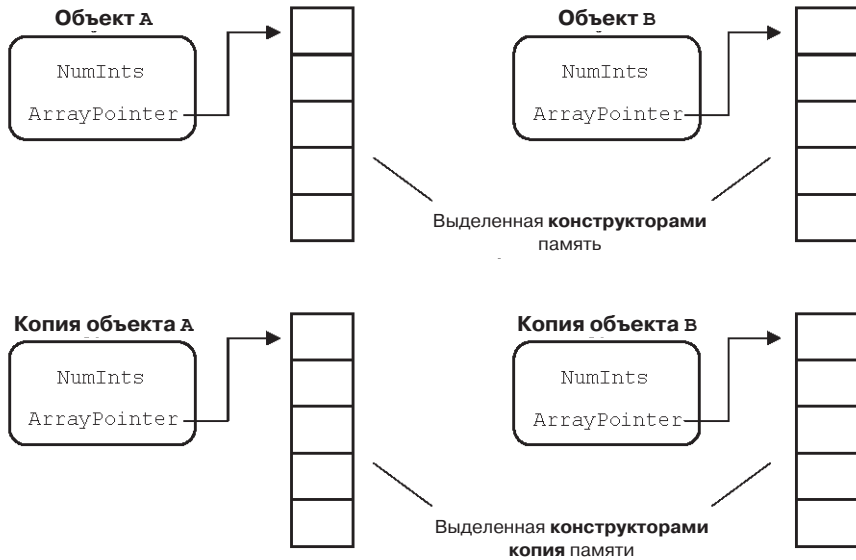


Рис. 12.3. Копии объектов А и В сделаны конструктором копий, написанным программистом

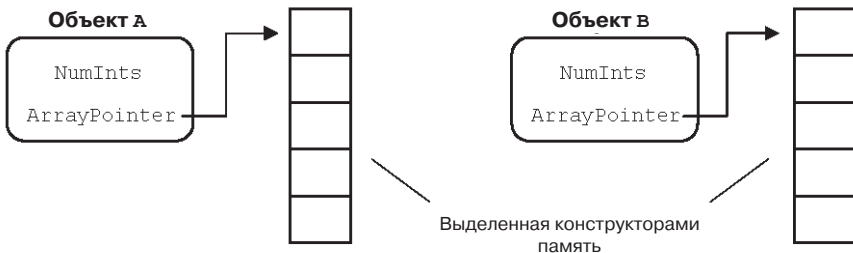


Рис. 12.4. Копии объектов А и В сделаны конструктором копий, написанным программистом

Заголовок конструктора копии:

```
IntArray(const IntArray& intArray);
```

Вспомним, что конструкторы не возвращают значений. Конструктору копии должен передаваться по ссылке аргумент, тип которого соответствует классу, которому он принадлежит. Если параметр передать по значению, то при вызове такого конструктора, в свою очередь, будет вызван конструктор копии значения параметра, затем снова вызов конструктора копии и так далее. Поэтому параметр конструктора копии *должен* передаваться по ссылке. Передаваемый по ссылке параметр защищён от случайного изменения служебным словом `const`. Таким образом, переданный параметр не будет изменяться.

Определение функций класса приведено в **Листинге 12.5**.

Листинг 12.5. Определение функций класса `IntArray` в файле `intarray.h`

```
#include <iostream.h>
#include "intarray.h"

IntArray::IntArray()
{
    NumInts = 0;
    ArrayPointer = NULL;
}

IntArray::IntArray(int numints)
{
    if(numints <= 0)
    {
        NumInts = 0;
        ArrayPointer = NULL;
    }
    else
    {
        NumInts = numints;
        ArrayPointer = new int[NumInts];
    }
}

IntArray::IntArray(const IntArray& intArray)
{
    NumInts = intArray.NumInts;
    ArrayPointer = new int[NumInts];
    for(int i = 0; i < NumInts; i++)
        *(ArrayPointer + i) = *(intArray.ArrayPointer + i)
}

IntArray::~IntArray()
{
    if(ArrayPointer != NULL)
    {
        delete ArrayPointer;
        ArrayPointer = NULL;
    }
}
```

```

}

void IntArray::EnterArray()
{
    cout << "Enter " << NumInts << " integer values."
    << endl;
    for(int i = 0; i < NumInts; i++)
        cin >> *(ArrayPointer + i);
}

void IntArray::PrintArray()
{
    for(int i = 0; i < NumInts; i++)
        cout << *(ArrayPointer + i) << '\t';
    cout << endl;
}

```

В функцию `main()` можно переписать с явными вызовами конструктора копии и деструктора, уничтожающего объекты, созданные конструктором копии. Программа в файле `copycnst.cpp`, **Листинг 12.6**, показывает, что вызов деструктора объекта-копии *не приводит* к уничтожению оригинального объекта.

Листинг 12.6. Программа тестирования пользовательского конструктора копии, файл `copycnst.cpp`

```

#include <iostream.h>
#include <conio.h>

#include "intarray.h"

void main()
{
    IntArray A(5);
    A.EnterArray();

    cout << "A "; A.PrintArray(); // Вывод массива A на экран
    getch();
    IntArray B(A); // Вызов конструктора копии
    cout << "B "; B.PrintArray(); // Вывод массива B на экран
    B.~IntArray(); // Уничтожение объекта B
    // Печать массива A для проверки содержимого
    cout << "A "; A.PrintArray();
}

```

Сборка исполняемого файла		
Требуемые файлы	Листинг	Содержимое файла проекта
<code>copycnst.cpp</code>	Листинг 12.6	<code>copycnst.cpp</code>
<code>intarray.cpp</code>	Листинг 12.5	<code>intarray.cpp</code>
<code>intarray.h</code>	Листинг 12.4	

12.2.5. Перегрузка операторов при помощи функций-членов

В этом разделе будет перегружен оператор `>>` для класса `ADC`, он будет выполнять аналого-цифровое преобразование и сохранять результат в переменной. Для этого в определение класса нужно включить новую функцию, которая и будет перегружать оператор. Воспользовавшись **Табл. 12.2** напомним новый заголовочный файл определения класса `ADC`, приведённое в **Листинге 12.7**.

Параметр функции перегрузки оператора передаётся по ссылке, чтобы функция могла менять переданное ей значение в контексте вызова, то есть прочитанный результат АЦП.

Листинг 12.7. Перегрузка оператора при помощи функции-члена в файле `adc.h`

```
#ifndef AdcH
#define AdcH
#include "pport.h"

class ADC : public ParallelPort
{
    private:
        unsigned char ADCValue;

    public:
        ADC(int baseaddress = 0x378);
        unsigned char ADConvert();
        unsigned char GetADCValue();
        void operator>>(unsigned char& value);
}
#endif
```

В соответствии с этими изменениями файл `adc.cpp` (**Листинг 11.4**) приобретает вид, представленный в **Листинге 12.8**.

Листинг 12.8. Файл определения функций класса `ADC`, определённого в **Листинге 12.7**

```
#include "adc.h"

ADC::ADC(int baseaddress) : ParallelPort(baseaddress)
{
    ADCValue = 0;
}

unsigned char ADC::ADConvert()
{
    WritePort2(0x01);
    WritePort2(0x00);
    WritePort2(0x01);

    WritePort2(0x03);
```

```

    ADCValue = ReadPort1() & 0xF0;

    WritePort2(0x01);
    ADCValue += (ReadPort1() >> 4) & 0x0F;

    return ADCValue;
}

unsigned char ADC::GetADCValue()
{
    return ADCValue;
}

void ADC::operator>>(unsigned char& value)
{
    ADConvert();
    value = ADCValue;
}

```

При перегрузке оператора `>>` использован только один параметр, хотя он работает с двумя операндами. На место этого параметра подставляется правый операнд оператора `>>` при его использовании. Левым операндом этого оператора является объект класса `ADC` (функция `main()`), **Листинг 12.9**. Также в теле функции перегрузки использованы выражения:

```

ADConvert();
value = ADCValue;

```

вместо

```

value = ADConvert();

```

Функция перегрузки оператора нарочно написана как можно более проще, потому что далее будет написана ещё одна программа, где оператор будет перегружен при помощи обычной функции (не являющейся членом класса). В функции `main()` (**Листинг 12.9**) приведён пример использования оператора, перегруженного функцией-членом.

Листинг 12.9. Использование перегруженного оператора `>>` в функции `main()`, файл `ovld.cpp`

```

#include <bios.h>
#include <conio.h>

#include "adc.h"

void main()
{
    ADC Adc;
    int Quit = 0, key;

```



```
unsigned char Value;

clrscr();

while(!Quit)
{
    gotoxy(10, 10);

    Adc >> Value;
    printf("The ADC output is %10d\\a", (int) Value);

    if(bioskey(1) != 0);
    {
        key = bioskey(0);

        if(key == 0x2d00) Quit = 1; // Alt-X
    }
}
```

Сборка исполняемого файла		
Требуемые файлы	Листинг	Содержимое файла проекта
pport.cpp	Листинг 10.8	pport.cpp
pport.h	Листинг 10.7	
adc.cpp	Листинг 12.8	adc.cpp
adc.h	Листинг 12.7	
ovld.cpp	Листинг 12.9	ovld.cpp

Для проверки работоспособности программы ко входу АЦП нужно подклю-
чить выход потенциометра.

12.2.6. Перегрузка операторов при помощи обычных функций

Перегрузка оператора теперь будет осуществляться функцией, не принадлежа-
щей классу. Определение класса ADC в файле adc.h для такого случая приведено в
Листинге 12.10.

Листинг 12.10. Перегрузка оператора внешней функцией, файл adc.h

```
#ifndef AdcH
#define AdcH

#include "pport.h"

class ADC : public ParallelPort
{
```

```

private:
    unsigned char ADCValue;

public:
    ADC(int baseaddress = 0x378);
    unsigned char ADConvert();
    unsigned char GetADCValue();
};

// Объявление внешней функции
void operator>>(ADC adc, unsigned char& value);
#endif

```

В **Листинге 12.11** приведены определения функций класса (файл `adc.cpp`), определённого выше.

Листинг 12.11. Определения функций класса `ADC`, определённого в **Листинге 12.10**.

```

#include "adc.h"

ADC::ADC(int baseaddress) : ParalellPort(baseaddress)
{
    ADCValue = 0;
}

unsigned char ADC::ADConvert()
{
    WritePort2(0x01);
    WritePort2(0x00);
    WritePort2(0x01);

    WritePort2(0x03);
    ADCValue = ReadPort1() & 0xF0;

    WritePort2(0x01);
    ADCValue += (ReadPort1() >> 4) & 0x0F;

    return ADCValue;
}

unsigned char ADC::GetADCValue()
{
    return ADCValue;
}

void operator>>(ADC adc, unsigned char& value)
{
    adc.ADConvert();
    value = adc.GetADCValue();
}

```

Сборка исполняемого файла		
Требуемые файлы	Листинг	Содержимое файла проекта
pport.cpp	Листинг 10.8	pport.cpp
pport.h	Листинг 10.7	
adc.cpp	Листинг 12.11	adc.cpp
adc.h	Листинг 12.10	
ovld.cpp	Листинг 12.9	ovld.cpp

Теперь функция перегрузки объявлена с двумя параметрами, в отличие от случая, рассмотренного в разделе 12.2.5. При использовании оператора >> первый параметр соответствует левому операнду, а второй – правому. Первый параметр теперь нужно явно указывать, так как функция не принадлежит классу. Теперь нельзя воспользоваться следующим выражением, поскольку из этой функции нет доступа к собственному члену данных ADCValue:

```
value = adc.ADCValue;
```

Вместо этого нужно вызывать функцию-член GetADCValue(). Теперь можно собрать программу, используя ранее использовавшийся файл main.cpp (**Листинг 12.9**), в котором подключается заголовочный файл adc.h с определением класса, и подключив на этапе разрешения связей (линковки) файл adc.cpp с определением функций (**Листинг 12.11**).

12.2.7. Дружественные связи

Обычные функции можно объявить дружественными какому-либо классу, что позволяет избежать сложностей, связанных с доступом к собственным членам данных класса. Дружественная функция объявляется служебным словом `friend` и имеет полный доступ ко всем членам того класса, в котором она объявлена. Такие функции можно рассматривать как функции-члены классы, к которым можно обращаться без использования операторов доступа (`.` или `->`). Аккуратное использование дружественных функций может упростить такие задачи, как потоковый ввод/вывод, который рассмотрим в нескольких словах. Пример дружественных функций можно привести, незначительно изменив заголовочный файл из **Листинга 12.10**, тогда он будет выглядеть, как в **Листинге 12.12**.

Листинг 12.12. Использование дружественных функций, файл `adc.h`

```
#ifndef AdcH
#define AdcH

#include "pport.h"

class ADC : public ParallelPort
{
private:
    unsigned char ADCValue;
```

```

public:
    ADC(int baseaddress = 0x378);
    unsigned char ADConvert();
    unsigned char GetADCValue();
    friend void operator>>(ADC adc, unsigned char& value)
};
#endif

```

Так как функция перегрузки дружественна классу ADC, ей доступен собственный член данных ADCValue. Таким образом, вместо

```
value = adc.GetADCValue();
```

можно записать:

```
value = adc.ADCValue;
```

В этом случае не происходит вызова функции и скорость выполнения программы увеличивается. Файл определения функций в **Листинге 12.13**.

Листинг 12.13. Определение функций в файле adc.cpp в соответствии с заголовочным файлом из **Листинга 12.12**

```

#include "adc.h"

ADC::ADC(int baseaddress) : ParallelPort(baseaddress)
{
    ADCValue = 0;
}

unsigned char ADC::ADConvert()
{
    WritePort2(0x01);
    WritePort2(0x00);
    WritePort2(0x01);

    WritePort2(0x03);
    ADCValue = ReadPort1() & 0xF0;

    WritePort2(0x01);
    ADCValue += (ReadPort1() >> 4) & 0x0F;

    return ADCValue;
}

unsigned char ADC::GetADCValue()
{
    return ADCValue;
}

void operator>>(ADC adc, unsigned char& value)

```

```

{
    adc.ADConvert();
    value = adc.ADCValue;
}

```

Сборка исполняемого файла		
Требуемые файлы	Листинг	Содержимое файла проекта
pport.cpp	Листинг 10.8	pport.cpp
pport.h	Листинг 10.7	
adc.cpp	Листинг 12.13	adc.cpp
adc.h	Листинг 12.12	
ovld.cpp	Листинг 12.9	ovld.cpp

В предыдущих разделах перегружался оператор для выполнения аналого-цифрового преобразования и сохранения результата в переменной. Для этой цели использовался оператор `>>`. Далее будет перегружен оператор `<<` для вывода результата АЦП на экран или в файл. Перед тем как продолжить нужно познакомиться с основами *потоков ввода/вывода* и *транзитных объектов*.

12.2.8. Потоки ввода/вывода

Потоки ввода/вывода можно описать как перенос одного или нескольких объектов из одного места в другое. Примером таких потоков могут служить данные, пересылаемые клавиатурой, выводимые на экран, записываемые в файл или читаемые из него и так далее. В C++ есть множество классов и функций, предназначенных для работы с потоками ввода/вывода. Потоки вывода можно создавать конструктором `ostream`, а потоки ввода конструктором `istream`.

Классы `ostream` и `istream` входят в состав стандартной библиотеки C++ и позволяют осуществлять операции записи и чтения для множества объектов, включая файлы. Воспользоваться этими классами станет можно после включения в программу заголовочного файла `fstream.h`, так же, как включался файл `iostream.h`, где объявлялись классы `cin` и `cout`.

Пример простой программы с использованием файлового ввода/вывода приведён в **Листинге 12.14**.

Листинг 12.14. Файловый ввод/вывод

```

#include <fstream.h>

main()
{
    int Data;

    ifstream is("infile.dat");
    ofstream os("outfile.dat");

    while(is)

```

```
{
    is >> Data; // Число из файла в переменную Data
    if(!is.fail())
        os << '\t' << Data; // Данные в файл
}
os.close();
is.close();
return 0;
}
```

Выражения

```
ifstream is("infile.dat");
ofstream os("outfile.dat");
```

создают объекты `is` и `os`, вызывая конструкторы `ifstream()` и `ofstream()`, соответственно. Каждый конструктор открывает файл, имя которого передаётся конструктору в виде параметра. Конструкторы также создают буфер, необходимый для передачи данных между файлом и памятью.

Предположим, что файл `infile.dat` содержит числа:

```
10
20
30
40
50
```

Числа могут разделяться пробелами, знаками табуляции, символами перевода строки или перевода каретки. Переменная `Data` служит для временного хранения прочитанного из файла `infile.dat` числа.

В файле `outfile.dat` в результате выполнения программы получится следующее:

```
10    20    30    40    50
```

В файл `outfile.dat` попадут только те данные, при чтении которых из файла `infile.dat` не возникло ошибки. Ошибку чтения файла можно обнаружить при помощи функции-члена `fail()` класса `ifstream`. По достижении конца файла объект `is` класса `ifstream` будет возвращать 0 и цикл `while` завершится.

12.2.9. Транзитные объекты

Транзитные объекты, с одной стороны, передаются функции по ссылке, и возвращаются той же функцией по ссылке, с другой стороны. Здесь будут рассматриваться возвращаемые по ссылке значения. Во-первых, выясним, для чего нужны транзитные объекты. Разберём случай с тремя операциями потокового вывода целых чисел `a`, `b` и `c`:

```
cout << a;
cout << b;
cout << c;
```

Те же три операции можно записать в виде одного выражения:

```
cout << a << b << c;
```

Последовательность выполнения операций можно показать при помощи скобок:

```
((cout << a) << b) << c);
```

После вывода числа *a* оставшаяся часть выражения может быть представлена в виде:

```
((cout) << b) << c);
```

Таким образом, последней будет выполнена операция:

```
(cout << c);
```

В рассмотренном примере перегруженный оператор `<<` работает с двумя операндами: `cout` и целым числом. Перегруженный оператор возвращает объект `cout`. Поэтому выражение `(cout << a)` можно заменить на `cout`. В рассматриваемом случае `cout` передаётся в функцию, а затем возвращается этой функцией. Поэтому `cout` является транзитным объектом в функции перегрузки оператора `<<`. Транзитные объекты не должны «пропадать» во время этого процесса, копирования объектов не происходит. Таким образом, параметры должны и передаваться, и возвращаться по ссылке.

Транзитные объекты особенно подходят для выполнения «цепочечных» операций. Ниже представлен другой пример с целыми числами *a*, *b* и *c*:

```
a = b = c = 3;
```

И в этом случае порядок выполнения операций можно показать скобками:

```
(a = (b = (c = 3))) ;
```

После выполнения первого присвоения выражение приобретает вид:

```
(a = (b = (c))) ;
```

В операции `c = 3`, переменная *c* сначала передаётся функции, а затем она же возвращается той же функцией, поэтому `c = 3` заменяется на *c*.

Для улучшения представления о транзитных функциях можно привести пример из повседневной жизни. Предположим, что имеется нитка (*string*) и набор бусинок (*bead*). Необходимо нанизать все бусинки, одну за другой, нанизать на нитку. Напишем функцию, принимающую в качестве входных параметров нитку и бусинку. Функция будет нанизывать бусинку на нитку и возвращать *ту же самую* нитку с добавленной бусинкой.

В определении функции укажем, что она возвращает значение по ссылке, добавив `&` после типа возвращаемого значения. В определении эта деталь выделена жирным шрифтом:

```
String& Add_A_Bead(String& string, Bead bead)
{
    string = string + bead;
    return string;
}
```

Функция `Add_A_Bead()` имеет два параметра: `string` (нитка) типа `String` и `bead` (бусинка) типа `Bead`. Выражение `string = string + bead` символизирует нанизывание бусинки на нитку. Можно нанизать n бусинок на *одну и ту же* нитку, выполнив эту функцию n раз.

Если результат возвращать не по ссылке, то после каждого нанизывания бусинки будет создаваться **новая копия** нитки с увеличенным количеством бусинок на ней. Если такая функция будет выполняться n раз, то будут создаваться и разрушаться n копий нитки, первая нитка будет с одной бусинкой, вторая с двумя и так далее, до нитки n , которой будет нанизано n бусинок: какая трата времени и памяти! Наконец, работа происходит не с данной ниткой, а с её копией.

12.2.10. Оператор присвоения

Если разработчик не создаёт функцию, перегружающую оператор присвоения, то компилятор сам создаёт перегруженный оператор присвоения. Такой оператор работает подобно конструктору копии. Оператор присвоения используется для копирования членов одного **существующего** объекта в соответствующие члены другого **существующего** объекта. Перегруженный компилятором оператор присвоения обладает тем же недостатком, что и автоматический конструктор копии: он не копирует блоки памяти, на которые указывают члены-указатели. Разработчик должен самостоятельно перегрузить оператор присвоения, чтобы присвоение происходило правильно.

Заголовок функции перегрузки оператора присвоения будет выглядеть так:

```
IntArray& operator=(const IntArray& intarray);
```

Здесь параметр передаётся так же, как и в случае конструктора копии, но возвращаемое значение является ссылкой на объект класса `IntArray`. Такое требование позволяет выполнять «цепочечное» присвоение, как в данном примере:

```
int a, b, c;  
a = b = c = 3;
```

Класс `IntArray` и связанные с ним исходные файлы рассматривались в разделе 12.2.4 и далее используются при перегрузке оператора присвоения. Класс с перегруженным оператором приведён в **Листинге 12.15** (заголовочный файл) и **Листинге 12.16** (определение функций).

Листинг 12.15. Заголовочный файл `intarray.h` с перегруженным оператором присвоения

```
#ifndef IntarrayH  
#define IntarrayH  
  
class IntArray  
{  
    private:  
        int NumInts;  
        int* ArrayPointer;  
  
    public:
```



```

IntArray();
IntArray(int numints);
IntArray(const IntArray& IntArray);
IntArray& operator=(const IntArray& intArray);
~IntArray();
void EnterArray();
void PrintArray();
}
#endif

```

Листинг 12.16. Файл определений функций `intarray.cpp` класса `IntArray` с перегруженным оператором присвоения

```

#include <iostream.h>
#include "intarray.h"

IntArray::IntArray()
{
    NumInts = 0;
    ArrayPointer = NULL;
}

IntArray::IntArray(int numints)
{
    if(numints <= 0)
    {
        NumInts = 0;
        ArrayPointer = NULL;
    }
    else
    {
        NumInts = numints;
        ArrayPointer = new int[NumInts];
    }
}

IntArray::IntArray(const IntArray& intArray)
{
    NumInts = intArray.NumInts;
    ArrayPointer = new int[NumInts];
    for(int i = 0; i < NumInts; i++)
        *(ArrayPointer + i) = *(intArray.ArrayPointer + i)
}

IntArray& IntArray::operator=(const IntArray& intArray)
{
    if(this != &intArray)
    {
        if(ArrayPointer != NULL)

```

```

        delete ArrayPointer; // Освободить занятую память
        NumInts = intArray.NumInts;
        ArrayPointer = new int[NumInts];
        for(int i = 0; i < NumInts; i++)
            *(ArrayPointer + i) = *(intArray.ArrayPointer + i);
    }
    return *this;
}

IntArray::~IntArray()
{
    if(ArrayPointer != NULL)
    {
        delete ArrayPointer;
        ArrayPointer = NULL;
    }
}

void IntArray::EnterArray()
{
    cout << "Enter " << NumInts << " integer values." << endl;
    for(int i = 0; i < NumInts; i++)
        cin >> *(ArrayPointer + i);
}

void IntArray::PrintArray()
{
    for(int i = 0; i < NumInts; i++)
        cout << *(ArrayPointer + i) << '\t';
    cout << endl;
}

```

В данном случае в функции перегрузки оператора присвоения был использован указатель `this`, указывающий на тот экземпляр класса (объект), от имени которого эта функция была вызвана. В условии `if` проверяется, не копируется ли объект сам в себя. Если это так, то нет никакого смысла в данной операции копирования. В противном случае проверяется, что `ArrayPointer` действительно указывает на ранее размещённый в памяти массив. Если это верно, то занятая этим массивом память освобождается оператором `delete`. Удаление массива необходимо для того, чтобы избежать утечки памяти: если этого не сделать, то в памяти будут находиться данные, к которым невозможно обратиться, и в то же время этот участок памяти будет недоступным для размещения других данных, поскольку операционная система считает его занятым. Далее выполняются те же действия, что и в конструкторе копии. Наконец, происходит возврат объекта, чтобы могла работать «цепочка» операторов присвоения. Возвращается выражение `*this`, где `this` является указателем на объект, а `*this` это сам объект.

В **Листинге 12.17** приведена программа, демонстрирующая использование конструктора копии и оператора присвоения класса `IntArray`.

Листинг 12.17. Пример использования оператора присвоения
в файле asgnopr.cpp

```
#include <iostream.h>
#include <conio.h>

#include "intarray.h"

void main()
{
// Вызов конструктора
    IntArray A(5);
    A.EnterArray();

// Вывод массива A
    cout << "A "; A.PrintArray();
    getch();

// Вызов конструктора копии
    IntArray B(A);

// Вывод массива B
    cout << "B "; B.PrintArray();
    getch();

// Вызов автоматического конструктора
    IntArray C;

// Использование оператора присвоения
    C = A;

// Вывод массива C
    cout << "C "; C.PrintArray();
}
```

Сборка исполняемого файла		
Требуемые файлы	Листинг	Содержимое файла проекта
asgnopr.cpp	Листинг 12.17	asgnopr.cpp
intarray.cpp	Листинг 12.16	intarray.cpp
intarray.h	Листинг 12.15	

12.3. Сбор данных

В этом разделе будет создана программа сбора данных с использованием перегрузки операторов, где будут реализованы ранее рассмотренные программные конструкции. Аналого-цифровое преобразование будет выполняться функцией

перегрузки оператора <<, который будет работать с объектом класса и потоком cout. Оператор будет выводить результат АЦП в поток вывода cout. Для этого нужно внести изменения в заголовочный файл adc.h, добавив в него функцию перегрузки оператора. Такую функцию можно написать без обращения к собственным членам данных класса ADC. Функция будет внешней по отношению к классу и её не нужно объявлять как дружественную этому классу. Новый заголовочный файл приведён в **Листинге 12.18**.

Листинг 12.18. Перегрузка оператора << в файле adc.h

```
#ifndef AdcH
#define AdcH

#include <iostream.h>
#include "pport.h"

class ADC : public ParallelPort
{
    private:
        unsigned char ADCValue;

    public:
        ADC(int baseaddress = 0x378);
        unsigned char ADConvert();
        unsigned char GetADCValue();
        friend void operator>>(ADC adc, unsigned char& value);
};

// объявление внешней функции
ostream& operator<<(ostream& os, ADC adc);
#endif
```

Можно видеть, что нижеприведённая функция очень похожа на ранее рассматривавшуюся функцию Add_A_Bead(), раздел 12.2.9:

```
ostream& operator<<(ostream& os, ADC adc);
```

Функция перегрузки оператора работает аналогичным образом, за исключением того, что она вызывается посредством оператора <<. Файл с определениями функций приведён в **Листинге 12.19**.

Листинг 12.19. Определение функций в файле adc.cpp, объявленных в заголовочном файле из **Листинга 12.18**

```
#include <iostream.h>
#include "adc.h"

ADC::ADC(int baseaddress) : ParallelPort(baseaddress)
{
    ADCValue = 0;
}

unsigned char ADC::ADConvert()
```

```

{
    WritePort2(0x01);
    WritePort2(0x00);
    WritePort2(0x01);

    WritePort2(0x03);
    ADCValue = ReadPort1() & 0xF0;

    WritePort2(0x01);
    ADCValue += (ReadPort1() >> 4) & 0x0F;

    return ADCValue;
}

unsigned char ADC::GetADCValue()
{
    return ADCValue;
}

void operator>>(ADC adc, unsigned char& value)
{
    adc.ADConvert();
    value = adc.ADCValue;
}

ostream& operator<<(ostream& os, ADC adc)
{
    os << " " << (int) adc.ADConvert();

    return os;
}

```

Следующий пример элегантного кода демонстрирует преимущества перегрузки операторов.

```
cout << Adc << Adc << Adc;
```

Каждая операция << над объектом `Adc` выполняет аналого-цифровое преобразование и выводит результат в стандартное устройство вывода. Таким образом, выполнение вышеприведённого кода приведёт к выводу на экран трёх значений АЦП.

Порядок выполнения операций показан ниже при помощи скобок:

```
((cout << Adc) << Adc) << Adc);
```

В **Листинге 12.20** приведена функция `main()` с которой можно поэкспериментировать (файл `dataacq.cpp`). В этой программе выполняется три аналого-цифровых преобразования раз в одну секунду, результат выводится на экран. В целях уменьшения влияния шума в качестве значения выборки брать среднее значение нескольких выборок. Нужно учитывать, что вывод на экран выполняется относительно долго, что несколько снижает частоту выборок.

Листинг 12.20. Программа с использованием оператора << в файле dataacq.cpp

```

#include <conio.h>
#include <bios.h>
#include <doc.h>

#include "adc.h"

void main()
{
    ADC Adc;
    int Quit = 0;

    clrscr();

    while(!Quit)
    {
        cout << endl << Adc << Adc << Adc;

        if(bioskey(1) != 0)
            if(bios(0) == 0x2d00) Quit = 1; /* Alt-X */
        delay(1000);
    }
}

```

Сборка исполняемого файла		
Требуемые файлы	Листинг	Содержимое файла проекта
pport.cpp	Листинг 10.8	pport.cpp
pport.h	Листинг 10.7	
adc.cpp	Листинг 12.19	adc.cpp
adc.h	Листинг 12.18	
dataacq.cpp	Листинг 12.20	dataacq.cpp

Перегружать операторы можно и в других программах. Например, в классе DCMotor, разработанном в главе 8, можно перегрузить оператор ++, который увеличивал бы Speed на величину шага приращения скорости. Аналогично, в этом же классе можно перегрузить оператор -- для такого же уменьшения скорости в Speed. Такой оператор мог бы использоваться так:

```

DCMotor Motor1;
Motor1++; // увеличить скорость

```

В классе DAC можно перегрузить оператор << чтобы можно было посылать целые числа в ЦАП:

```

DAC Dac;
Dac << N;

```

12.4. Заключение

Передавать параметры функции и возвращать её результат можно разными способами. В большинстве функций передаются только копии аргументов из контекста вызова. Вместо копий можно передавать ссылку на параметр. При передаче параметра по ссылке функция работает с оригинальным аргументом из контекста вызова. Такой способ передачи более быстрый и не требует выделения дополнительной памяти. Вследствие этих особенностей передавать объекты лучше всего по ссылке.

Аналогично, большинство функций возвращают только копию результата, полученного в функции. Если же результат возвращается по ссылке, то возвращается сам объект результата, а не его копия. Такая возможность может быть полезной в перегруженных операторах, особенно когда они используются для «цепочечных» операций.

Дружественные функции класса имеют неограниченный доступ ко всем его членам. Хотя они и не являются функциями-членами класса, права доступа у них те же. Дружественные функции отличаются от функций-членов тем, что вызываются без оператора доступа (`.` или `->`).

12.5. Литература

1. Winston, P. H., *On to C++*, Addison Wesley, 1994.
2. Johnsonbaugh, R and Martin Kalin, *Object-Oriented Programming in C++*, Prentice Hall, 1995.
3. Staugaard A. C. (Jr), *Structured and Object Oriented Techniques*, Prentice Hall, 1997.
4. Lafore, R. *Object Oriented Programming in MICROSOFT C++*, Waite Group Press, 1992.
5. Wang, P. S., *C++ with Object Oriented Programming*, PWS Publishing, 1994.

13 Таймер персонального компьютера

Содержание главы:

- Что такое таймер?
- Устройство таймера персонального компьютера.
- Конфигурирование таймера.
- Измерение времени.
- Измерение скорости реакции.
- Рисование с привязкой ко времени.
- Оцифровка сигнала с метками времени.

13.1. Введение

В ранее созданных программах, в которых так или иначе было задействовано время, не использовалось реальное время. Обратимся к программе формирования сигнала ШИМ, глава 8. Тогда ШИМ сигнал формировался при помощи задержек в цикле, фактическая длительность которых была неизвестной и зависела от скорости работы компьютера. В тех случаях, когда нужно точное измерение времени, используется аппаратный таймер. Такой таймер есть и в персональном компьютере, предназначенный для измерения отрезков времени. Таймер работает непрерывно и независимо от процессора, что обеспечивает точность измерения времени, а освободившиеся при этом вычислительные ресурсы компьютера могут быть использованы для других нужд.

На самом деле таймер компьютера состоит из трёх таймеров. В более современных компьютерах уже пять независимых таймеров. Мы будем рассматривать только самый распространённый вариант, то есть компьютер с тремя таймерами. Таймеры могут работать в двух основных режимах: измерение интервала времени и подсчёт количества событий. Для работы всех таймеров необходим *тактовый сигнал*, то есть непрерывная последовательность импульсов, частота которых известна и высокостабильна. Наличие стабильной тактовой частоты позволяет писать программы с привязкой к реальному времени.

13.2. Устройство таймера персонального компьютера

Большинство компьютеров, особенно старых моделей, оснащены *программируемым таймером 8254*, состоящим из трёх таймеров: таймера 0, таймера 1 и тайме-

ра 3, **Рис. 13.1.** Таймеры могут работать в нескольких режимах, которые задаются сигналом разрешения тактирования (*gate signal*) и содержимым регистра управления (*control register*), речь о них пойдет в разделе 13.2.2. Из этих режимов будут рассмотрены формирование одиночного импульса (интервала времени), генерация прямоугольных импульсов и генерация коротких импульсов с заданным периодом, раздел 13.2.3. Все три таймера тактируются одним тактовым сигналом, только у таймера 2 есть дополнительный сигнал разрешения тактирования, которым можно управлять из программы.

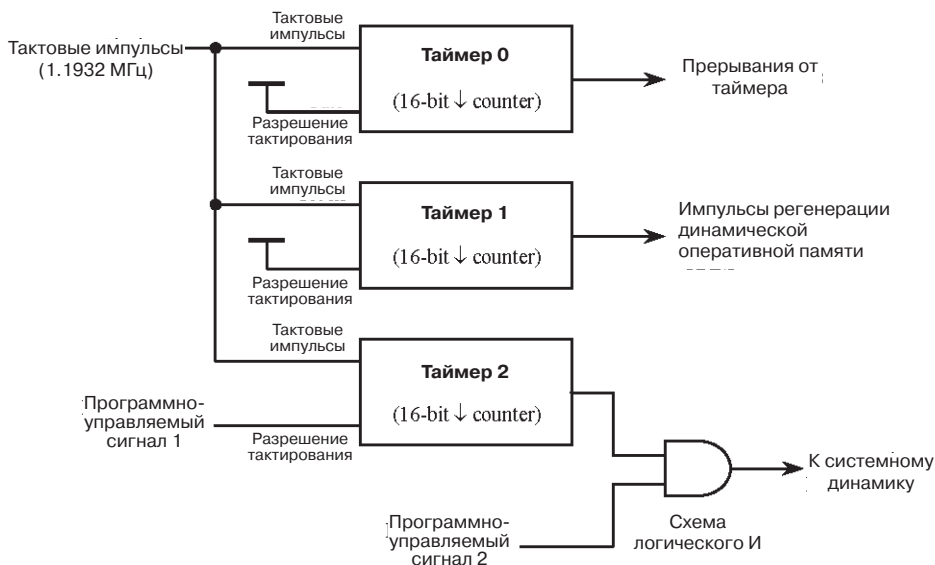


Рис. 13.1. Таймеры персонального компьютера

В основе всех таймеров лежит 16-битный *счётчик*. Счётчик можно считать особой ячейкой памяти, содержимое которой инкрементируется каждый раз по приходе тактирующего импульса. В персональном компьютере поступающие тактовые импульсы *уменьшают* (декрементируют) значение счётчика. Когда значение счётчика достигает нуля, то в большинстве режимов происходит изменение выходного сигнала таймера.

Так как все три таймера работают от одного источника тактовых импульсов, то их нельзя использовать в качестве счётчиков. Счётчик подразумевает использование таймера/счётчика для подсчёта входных импульсов, обычно приходящих в произвольные моменты времени. Тактовая частота таймеров не зависит от скорости работы компьютера и равна 1.1932 МГц. Поэтому измерение времени на всех персональных компьютерах выполняется одинаково.

Каждый таймер персонального компьютера выполняет определённые задачи, рассмотренные ниже.

Таймер 0

Таймер 0 служит для формирования так называемого *прерывания от таймера*. Прерывание от таймера периодически принуждает центральный процессор

компьютера выполнять специальную подпрограмму, обновляющую системное время. Такое обновление происходит каждые 54.9 миллисекунды. Сигнал разрешения тактирования всегда находится в ВЫСОКОМ состоянии и, соответственно, недоступен для программы. Поскольку сигналы тактирования и разрешения тактирования программно-неуправляемы, в этом таймере можно задавать только значение периода и режим работы. Текущее значение счётчика таймера 0 можно получить при помощи регистра-защёлки. Также можно считывать выходной сигнал таймера 0 из его *регистра состояния*.

Примечание

Прерывание это сигнал, формируемый программой или электронными схемами, который служит для переключения центрального процессора на выполнение требуемых действий (специальной подпрограммы). В зависимости от приоритета прерывания оно или будет обслужено процессором немедленно, или отложено для последующей обработки. Прерывание от таймера 0 имеет наивысший приоритет и процессор сразу же обрабатывает его.

Таймер 1

Таймер 1 генерирует периодический сигнал, который запускает схему регенерации динамической оперативной памяти. Сигналы тактирования и разрешения тактирования этого таймера также программно-недоступны. Как и в таймере 0, можно читать текущее значение счётчика таймера 1 из регистра-защёлки. Выходной сигнал таймера 1 можно прочесть в его *регистре состояния*.

Таймер 2

Таймер 2 служит для работы с системным динамиком. Выходным сигналом таймера можно управлять из программы посредством схемы логического И, подключенной к выходу таймера и один из входов которой программно-управляем, **Рис. 13.1.** Сигнал тактирования подаётся на таймер 2 всегда, а сигналом разрешения тактирования можно управлять из программы. Текущее значения счётчика таймера 2 можно прочесть из его регистра-защёлки, выходной сигнал из *регистра состояния* таймера 2.

13.2.1. Конфигурирование счётчиков

Как упоминалось выше, все три счётчика декрементируются тактовым сигналом. Загруженное в счётчики значение определяет длительность периода времени, который они отсчитывают. Таким образом, запись небольшого числа в счётчик означает, что значение счётчика достигнет нуля за короткий промежуток времени. Значение каждого счётчика можно читать и записывать в любой момент времени.

Значение счётчика постоянно изменяется в процессе его работы и поэтому нужно моментально «фотографировать» его значение и запоминать его. Как раз для этого и предназначен *регистр-защёлка*. Для каждого таймера предусмотрены входные и выходные регистры-защёлки, позволяющие записывать (загружать) и считывать его текущее значение в произвольные моменты времени, соответствен-

Выбор счётчика

Выбор счётчика осуществляется битами 6 (**SC0**) и 7 (**SC1**) регистра управления:

SC1	SC0	Счётчик
0	0	Счётчик 0
0	1	Счётчик 1
1	0	Счётчик 2

Выбор байта (байтов) счётчика для чтения/записи

Биты 4 (**RL0**) и 5 (**RL1**) служат для выбора старшего (MSB), младшего (LSB) или обоих байтов счётчика для чтения/записи:

Требуемое действие	RL1	RL0
Защёлкивание значения счётчика	0	0
Чтение/запись LSB	0	1
Чтение/запись MSB	1	0
Чтение/запись LSB, затем MSB	1	1
Примечание: MSB – старший байт, LSB – младший байт.		

Если, например, нужно записать LSB (младший байт) в счётчик, то **RL1** и **RL0** нужно установить в 0 и 1, соответственно. Затем в регистр-защёлку соответствующего таймера (например, 0x40 для таймера 0) записывается значение LSB.

Если нужно записать MSB (старший байт) счётчика, то **RL1** и **RL0** нужно установить в 1 и 0, соответственно. Затем значение MSB записывается в регистр-защёлку соответствующего таймера (0x40 в случае таймера 0).

Для записи 16-битного числа в счётчик нужно установить в 1 оба бита **RL1** и **RL0**. Затем последовательно выполняется две операции записи в регистр-защёлку соответствующего таймера (0x40 в случае таймера 0), сначала LSB (младший байт), а затем MSB (старший байт).

Формат числа в счётчике (двоичное или BCD)

Формат числа в счётчике может быть двоичным (используемый чаще всего) или BCD (**B**inary **C**oded **D**ecimal, двоично-десятичный формат). Формат BCD рассматриваться не будет, поэтому этот бит будет всегда обнуляться.

BCD	Формат числа
0	Двоичный
1	Двоично-десятичный (BCD)

Режим работы таймера

Режимы работы таймеров изучаются в следующем разделе. Режим работы таймера задаётся битами регистра управления, как показано в таблице приведённой ниже.

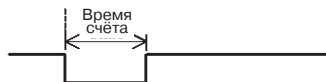
Биты выбора режима			Режим
M2	M1	M0	
0	0	0	Режим 0
0	0	1	Режим 1
0	1	0	Режим 2
0	1	1	Режим 3
1	0	0	Режим 4
1	0	1	Режим 5
1	1	0	Режим 2
1	1	1	Режим 3

13.2.3. Режимы работы таймеров

Таймеры могут работать в одном из шести режимов. Сигнал разрешения таймеров 0 и 1 всегда ВЫСОКИЙ, поэтому эти два таймера могут работать не во всех из шести возможных режимов. Управление сигналом разрешения тактированием возможно только у таймера 2 и он может работать во всех режимах.

Режим 0: одиночный импульс

В этом режиме таймер генерирует одиночный НИЗКИЙ импульс в течение заданного в счётчике количества тактовых импульсов. Каждый тактовый импульс декрементирует (уменьшает на единицу) значение счётчика, при этом сигнал разрешения тактирования должен быть ВЫСОКИМ. По достижении счётчиком значения 0 происходит возврат выходного сигнала таймера из НИЗКОГО состояния в исходное ВЫСОКОЕ. Максимальная длительность импульса достигается при загрузке в счётчик значения 65535. При этом длительность импульса будет примерно равна 54.9 мс.



В этом режиме могут работать все три таймера. Формирование одиночного импульса начинается с выбора режима работы таймера с последующей записью значения счётчика в регистр-защёлку. Декрементирование счётчика начнётся сразу же после записи значения счётчика. Если сигнал разрешения НИЗКИЙ, то счёт будет приостановлен, пока этот сигнал не станет ВЫСОКИМ. У таймеров 0 и 1 сигнал разрешения тактирования всегда ВЫСОКИЙ, в отличие от таймера 2, у которого этот сигнал выведен в регистр 0x61. В Табл. 13.3 приведено назначение битов регистра 0x61, которые касаются работы таймера 2.

Таблица 13.3. Назначение битов регистра 0x61
(управление выходом таймера 2 и системным динамиком)

Бит 7	Бит 6	Бит 5	Бит 4	Бит 3	Бит 2	Бит 1	Бит 0
х	х	Выход таймера 2	х	х	х	Включение системного динамика	Разрешение тактирования таймера 2

Режим 1: перезапускаемый одиночный импульс

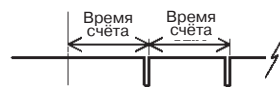
В этом режиме происходит формирование НИЗКОГО импульса, запускаемого сигналом разрешения тактирования (в этом режиме может работать только таймер 2). Длительность импульса задаётся значением счётчика. Для запуска декрементирования счётчика сигнал разрешения тактирования кратковременно переводят в ВЫСОКОЕ состояние, а затем обратно в исходное низкое. При этом выходной сигнал таймера переходит в НИЗКОЕ состояние. По достижении счётчиком значения 0 выход таймера возвращается обратно в ВЫСОКОЕ состояние.



В конце счёта происходит автоматическая перезагрузка счётчика заданным значением. Когда поступит очередной ВЫСОКИЙ импульс сигнала разрешения тактирования, запускается формирование нового импульса (перезапуск). Перезапуск таймера может быть осуществлён и во время работы счётчика. В этом случае генерирование нового импульса начнётся непосредственно после импульса перезапуска. Можно также загрузить новое значение счётчика во время счёта, это не повлияет на длительность текущего выходного импульса. Новое значение счётчика будет загружено после запуска очередного импульса.

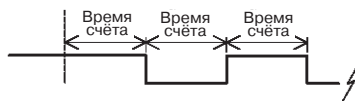
Режим 2: последовательность коротких импульсов

Режим используется для генерирования последовательности коротких НИЗКИХ импульсов. Когда значение счётчика достигает 1, выход таймера переходит в НИЗКОЕ состояние в течение одного тактового импульса, пока счётчик таймера не обнулится и процесс генерации импульсов не начнётся сначала. В этом режиме могут работать все три таймера. В таймере 2 счёт можно остановить, подав НИЗКИЙ уровень сигнала разрешения тактирования. Режим генерации импульсов, в основном, используется для периодических аппаратных прерываний, так как широкий ВЫСОКИЙ уровень таких импульсов легко обнаружить программно, тогда как короткий НИЗКИЙ уровень может быть потерян.



Режим 3: последовательность прямоугольных импульсов

На выходе таймера генерируются прямоугольные импульсы (меандр), длительности ВЫСОКОГО и НИЗКОГО уровней которых равны. Каждый раз при обнулении счётчика таймера, его выход меняет состояние на противоположное, а в счётчик записывается исходное значение счёта из регистра-защёлки. При этом сигнал разрешения тактирования должен быть ВЫСОКИМ, а значение счётчика *уменьшается на два* каждым тактовым импульсом. В этом режиме могут работать все три таймера. Прерывания таймера 0 генерируются в этом режиме.



Режим 4: программно запускаемый строб

Обратный отсчёт начинается сразу же после записи значения в счётчик. При обнулении счётчика на выходе таймера появляется короткий НИЗКИЙ импульс, длительностью в один период тактового сигнала (ко-



роткий непериодический импульс и называется *стробом*). Счётчик выключается до тех пор, пока в него не будет записано значение счёта. Сигнал разрешения тактирования должен быть **ВЫСОКИМ**.

Режим 5: аппаратно запускаемый строб

Режим очень похож на программно запускаемый строб в режиме 4, только запуск происходит не программно, а аппаратно. Обратный отсчёт запускается коротким **ВЫСОКИМ** импульсом сигнала разрешения тактирования. При обнулении счётчика на выходе таймера формируется короткий **НИЗКИЙ** импульс длительностью в один период тактового сигнала. Работа счётчика приостанавливается до следующего импульса запуска. В этом режиме может работать только таймер 2.



13.2.4. Чтение данных таймера

С таймера может быть считано значение счёта, выходной сигнал, режим чтения/записи данных счётчика, режим работы и режим счёта. Путём записи двух нижеприведённых значений в регистр управления можно извлекать очень важные данные таймера.

Вариант 1: защёлкивание значений счёта одного или нескольких таймеров

Под защёлкиванием значения счёта подразумевается запоминание текущего значения счётчика какого-либо таймера в выходном регистре-защёлке этого таймера. Защёлкивание данных одного или нескольких таймеров выполняется путём записи нижеописанного значения в регистр управления по адресу 0x43:

Данные в регистре управления по адресу 0x43

Бит 7	Бит 6	Бит 5	Бит 4	Бит 3	Бит 2	Бит 1	Бит 0
1	1	0	1	СТ2	СТ1	СТ0	0

Выбор таймера, значение счётчика которого должно быть защёлкнуто, осуществляется путём установки соответствующего бита **СТ0**, **СТ1** и **СТ2**. Защёлкнутые данные затем можно прочитать из соответствующих регистров-защёлок. Адреса этих регистров приведены в **Табл. 13.1**. Полное 16-битное значение счётчика считывается путём последовательного чтения двух байтов. В первой операции чтения считывается младший байт, во второй старший байт.

Вариант 2: состояние таймеров

Информацию о состоянии какого-либо таймера можно получить из соответствующего регистра данных. Запросить эти данные можно посредством записи в регистр управления (регистр 0x43) значения, приведённого ниже. Выбор нужного таймера и таймеров осуществляется путём установки в 1 соответствующих битов **СТ0**, **СТ1** и **СТ2** регистра управления. Байт состояния нужного таймера можно прочитать в одном из соответствующих регистров, адреса которых приведены в **Табл. 13.1**. Полученные данные расшифровываются согласно **Рис. 13.2**.

Данные в регистре управления по адресу 0x43

Бит 7	Бит 6	Бит 5	Бит 4	Бит 3	Бит 2	Бит 1	Бит 0
1	1	1	0	СТ2	СТ1	СТ0	0

7	6	5	4	3	2	1	0		
								0	В счётчике двоичное число.
								1	В счётчике двоично-десятичное число.
				0	0	0			Установлен режим 0.
				0	0	1			Установлен режим 1.
				0	1	0			Установлен режим 2.
				0	1	1			Установлен режим 3.
				1	0	0			Установлен режим 4.
				1	0	1			Установлен режим 5.
	0	0							не имеет смысла.
	0	1							Режим чтения/записи данных – только старший байт.
	1	0							Режим чтения/записи данных – только младший байт.
	1	1							Режим чтения/записи данных – сначала младший байт, затем старший.
	0								Загрузка нового значения в счётчик завершена.
	1								Выполняется загрузка нового значения в счётчик.
0									На выходе таймера НИЗКИЙ сигнал.
1									На выходе таймера ВЫСОКИЙ сигнал.

Рис. 13.2. Смысл битов байта состояния счётчика

13.3. Работа с таймером

В последующих разделах будут созданы программы, использующие таймер для точного измерения времени. Начнём с создания класса, представляющего таймер персонального компьютера. Так как схема таймера работает независимо от схемы параллельного порта, то при создании нового класса таймера можно использовать ранее созданные классы.

В программах будут измеряться отрезки времени, скорость реакции человека, а также будет формироваться развёртка во времени сигнала ГУН при выводе его на экран (глава 10). Наконец, будет написана программа, которая оцифровывает сигнал с интерфейсной платы и сохраняет полученные данные для дальнейшей обработки.

Ранее говорилось, что все три таймера персонального компьютера подключены к единому тактовому сигналу частотой 1.1932 МГц. В счётчики таймеров можно загрузить максимальное значение 65535, при этом длительность обратного от-

счёта таймера до нуля будет 54.9 мс. Для измерения более длительных отрезков времени нужно подсчитывать и количество обнулений счётчика таймера. Каждое обнуление таймера 0 называется *тиком*. В специальной области памяти BIOS запоминается количество тиков, произошедших начиная с полуночи. Аббревиатура BIOS расшифровывается как **Basic Input/Output System**, то есть базовая система ввода/вывода. Базовая система ввода/вывода представляет собой особое программное обеспечение, основное назначение которого в выполнении примитивных операций ввода/вывода. Как и большинство других операционных систем, операционная система компьютера страдает от недостаточной производительности компьютера и возникающих вследствие этого задержках при обработке внешних событий, при этом некоторые события возникают в аппаратуре компьютера. Вследствие этих задержек не всегда успевает обновиться счётчик тиков таймера, поэтому при измерении времени нельзя полагаться на достоверность счётчика тиков таймера в BIOS. Нужно разработать такой способ подсчёта тиков, чтобы была возможность точного измерения отрезков времени, длительность которых превышает один тик, используя количество прошедших тиков и текущее значение счётчика таймера. Для измерения времени будет использован таймер 0, режим работы которого наиболее подходит для этого.

13.3.1. Чтение текущего значения таймера 0 и количества тиков

Получить текущее количество прошедших тактовых импульсов таймера можно, прочитав количество тиков и текущее значение счётчика таймера 0. В разделе 13.2.4 описано как прочитать (защёлкнуть) текущее значение счётчика таймера 0 при помощи специальной команды. Защёлкнутые таким образом данные затем считываются из регистра данных таймера 0.

Перед началом работы с таймерами важно отметить, что при написании программы не будут повсеместно использоваться методы процедурного программирования (то есть вызовы функции). При этом программа будет выполняться быстро, что обеспечит приемлемую точность данных, считанных с таймера. Если бы из функций-членов класса `PCTimer` вызывались бы функции, то на эти вызовы затрачивалось бы дополнительное время.

13.4. Класс `PCTimer`

Создадим, класс позволяющий измерять длительность интервалов времени, а также осуществлять задержку выполнения программы. Для выполнения этих действий класс должен обладать следующими возможностями:

- Задавать точку «начала» времени (это не подразумевает обнуление счётчика в таймере компьютера).
- Формировать задержку заданной длительности.
- Считывать текущее значение времени.

Определение класса, в котором реализованы вышеперечисленные возможности, приведено в **Листинге 13.1**.

Листинг 13.1. Определение класса PCTimer в файле pctimer.h

```
#ifndef PctimerH
#define Pctimer

class PCTimer
{
    private:
        unsigned int InitCount;
        unsigned long TickCount;
        unsigned int LastCount;

    public:
        PCTimer();
        void ResetTimer();
        void Delay(const double& milliseconds);
        double ReadTimer();
        void UpdateTicks();
};
#endif
```

Установка начала отсчёта времени

При измерении времени наличие точки начала отсчёта времени является необходимым условием. Член данных `InitCount` служит для хранения значения счётчика таймера, то есть количество периодов тактового сигнала, остающееся до конца очередного тика. Значение этого члена как раз и представляет собой начало отсчёта времени, **Рис. 13.4**. Член `InitCount` инициализируется в момент создания объекта класса `PCTimer`, кроме того, его можно инициализировать в любой момент времени при помощи функции-члена `ResetTimer()`.

Функцию `ResetTimer()` нужно использовать для запоминания «начала» времени. В этой функции происходит считывание текущего значения таймера 0 и сохранение полученного значения в членах данных `InitCount` и `LastCount`. Член `LastCount` служит для хранения последнего считанного значения из таймера, чтобы другие функции класса могли обнаружить следующий тик, как описано ниже. Количество произошедших тиков сохраняется в члене данных `TickCount`, этот член данных можно обнулять.

Подсчёт количества тиков

Как говорилось ранее, для определения прошедшего времени нужно знать количество тиков, прошедших с момента начала отсчёта времени. Нужно постоянно следить за счётom таймера 0, чтобы избежать «потери» тиков. В трёх функциях `Delay()`, `ReadTimer()` и `UpdateTicks()` осуществляется слежение за счётom таймера и запоминание произошедших тиков. На **Рис. 13.3** приведено три различных случая слежения за тиками, пояснённых ниже.

Сначала происходит считывание значения счётчика таймера, которое сохраняется в члене `LastCount`. Спустя некоторое время это значение считывается опять, чтобы проверить, не произошёл ли тик, а считанное значение сохраняется в переменной `Count`. В случае **a** две операции чтения происходят в течении одного цикла

счёта, тика не происходит. Значение переменной Count не превышает величину в LastCount. В случае **б** между двумя считываниями начинается новый цикл обратного отсчёта. Таким образом, произошёл тик, Count больше чем LastCount. В случае **с** Count тоже не превышает LastCount, и если поступить аналогично случаю **а**, то будет принято неверное решение об отсутствии тика, поскольку две операции чтения сделаны с интервалом, превышающим время обратного отсчёта.

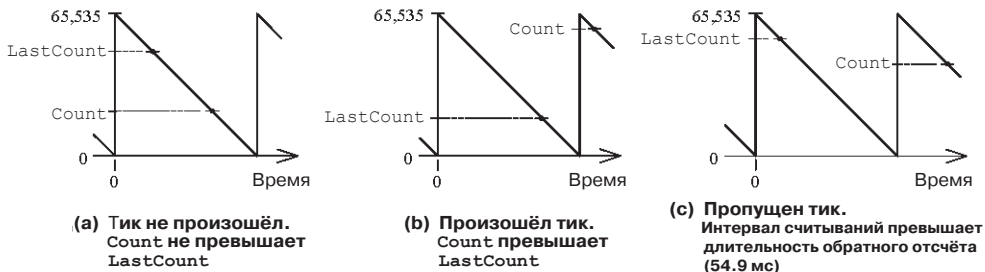


Рис. 13.3. Слежение за тиками

Подсчёт времени

В момент начала отсчёта времени происходит защёлкивание текущего значения счётчика, затем считывается из регистра и сохраняется в InitCount. Считанное значение представляет собой точку начала отсчёта времени, а также количество периодов тактового сигнала, оставшееся до завершения тика, **Рис. 13.4**.

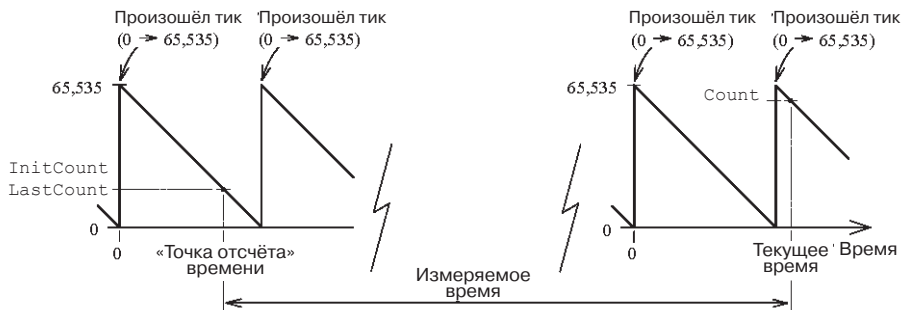


Рис. 13.4. Подсчёт времени

По завершении измерения времени происходит считывание счётчика таймера и полученное значение сохраняется в переменной Count. Количество периодов тактовой частоты таймера, прошедших с момента начала тика можно вычислить как разность $65535 - \text{Count}$. Количество произошедших тиков в измеряемом интервале времени хранится в переменной TickCount. Тогда полное количество периодов тактовой частоты таймера может быть найдено по формуле:

$$\text{TotalCounts} = \text{InitCount} + (\text{TickCount} - 1) * 65535 + (65535 - \text{Count});$$

Формулу можно упростить:

$$\text{TotalCounts} = \text{InitCount} + \text{TickCount} * 65535 - \text{Count};$$

Полученное количество периодов тактовой частоты можно легко преобразовать в значение времени, так как тактовая частота таймера известна.

Назначение и принцип работы функций класса описывается ниже.

Функция `PCTimer()` из **Листинга 13.2** является конструктором класса `PCTimer`. Конструктор размещает в памяти члены класса и считывает счётчик таймера в качестве начала отсчёта времени. Начало отсчёта времени считывается в конструкторе посредством вызова функции `ResetTimer()`.

Листинг 13.2. Конструктор класса PCTimer

```
PCTimer::PCTimer()  
{  
    ResetTimer();  
}
```

Функция `ReadTimer()` подсчитывает число периодов тактовой частоты, прошедших с момента начала отсчёта времени, функция `ReadTimer()` вызывается при этом постоянно. Подсчёт количества прошедших периодов тактовой частоты описан выше. Наконец, подсчитанное количество периодов преобразуется в значение времени в миллисекундах и возвращается как значение типа `double`. В функции `ReadTimer()` не вызываются другие функции, чтобы не возникало дополнительной задержки. Весь необходимый код сосредоточен в теле этой функции.

Функция `Delay()` формирует задержку выполнения программы, длительность задержки передаётся в качестве параметра функции, в миллисекундах. Когда эта функция вызывается, она считывает текущее значение счётчика таймера, которое является началом отсчёта задержки. Затем к этому значению прибавляется количество периодов тактовой частоты таймера, соответствующее длительности требуемой задержки. Потом выполняется непрерывное считывание значения счётчика таймера до тех пор, пока количество прошедших периодов тактовой частоты таймера не станет равным требуемому. На этом задержка заканчивается.

Функция `UpdateTicks()` осуществляет проверку тика быстрее, чем это делает функция `ReadTimer()`. При этом используются тот же метод, что и в других функциях класса `PCTimer`. Для правильного функционирования класса `PCTimer` необходимо, чтобы функция `UpdateTicks()` выполнялась по меньшей мере один раз в течение обратного отсчёта таймера, желательно каждые 40 мс. На этом разработка класса `PCTimer` завершена. Определение функций класса приведено в **Листинге 13.3** (файл `pctimer.cpp`).

Листинг 13.3. Определение функций класса PCTimer в файле pctimer.cpp

```
#include <dos.h>  
  
#include "pctimer.h"  
  
PCTimer::PCTimer()  
{  
    ResetTimer();  
}  
  
void PCTimer::ResetTimer()
```

```
{
// Защёлкивание значения счётчика таймера 0
    outportb(0x43, 0xD2);

// Считывание защёлкнутого значения
    LastCount = InitCount = inportb(0x40) + inportb(0x40)*256;
    TickCount = 0; // Обнуление
}

double PCTimer::ReadTimer()
{
    double Time;
    unsigned long TotalCounts;
    unsigned int Count;

// Защёлкнуть значение счётчика таймера 0
    outportb(0x43, 0xD2);

// Считывание защёлкнутого значения
    Count = inportb(0x40) + inportb(0x40)*256;

    if(Count > LastCount)
        TickCount++;

    LastCount = Count;

    TotalCounts = ((long) InitCount + TickCount*65535L
                  - (long) Count);
    Time = (TotalCounts/1.1932)/1000.0;
    return Time; // Время в миллисекундах
    // return TotalCounts;
}

void PCTimer::Delay(const double& milliseconds)
{
    unsigned int Count;
    long StartCount, DelayCount, EndCount, TotalCount;

// Защёлкнуть значение счётчика таймера 0
    outportb(0x43, 0xD2);

// Прочитать защёлкнутое значение
    Count = inportb(0x40) + inportb(0x40)*256;

    if(Count > LastCount)
        TickCount++;

    LastCount = Count;
    StartCount = ((long) InitCount + TickCount*65535L
```

```

        - (long) Count);
    DelayCount = (long) (milliseconds*1.1932*1000);
    EndCount = StartCount + DelayCount;

// Выполнять цикл в течение всей задержки
do
{
    // Защёлкнуть значение счётчика таймера 0
    outportb(0x43, 0xD2);

    // Прочитать защёлкнутое значение
    Count = inportb(0x40) + inportb(0x40)*256;

    if(Count > LastCount)
        TickCount++;

    LastCount = Count;
    TotalCount = ((long) InitCount + TickCount*65535L
                  - (long) Count);
}
while(TotalCount < EndCount);
}

void PCTimer::UpdateTicks()
{
    unsigned int Count;

// Защёлкивание значения счётчика таймера 0
    outportb(0x43, 0xD2);

// Прочитать защёлкнутое значение
    Count = inportb(0x40) + inportb(0x40)*256;

    if(Count > LastCount)
        TickCount++;

    LastCount = Count;
}

```

Ранее говорилось, что все функции класса вызываются при работающем таймере. Класс работал бы лучше, если бы были выключены прерывания, но отключение прерываний не выполнялось, чтобы не усложнять программу. Прерывания могут вызывать небольшие задержки выполнения программы и происходят в произвольные моменты времени.

Например, если при чтении значения таймера в функции `ReadTimer()` происходит прерывание от таймера, то выполнение функции `ReadTimer()` будет приостановлено до тех пор, пока не будет обработано прерывание от таймера. Такая задержка приведёт к тому, что функция `ReadTimer()` вернёт большее значение, чем должно было быть на момент её вызова. Влияние прерываний продемонстрировано в одной из программ с использованием функции-члена класса `PCTimer`.

Примечание

В то время, как тики таймера подсчитываются автоматически в функции `Delay()`, то функции `ReadTimer()` и `UpdateTicks()` для подсчёта тиков должны вызываться непрерывно. Поэтому, при измерении интервалов времени необходимо, чтобы функции `ReadTimer()` и `UpdateTicks()` вызывались по меньшей мере один раз за время обратного отсчёта таймера (54.9 мс), чтобы правильно считать тики таймера.

13.5. Измерение времени

Класс `PCTimer` будет использоваться для измерения времени. Ранее говорилось, что на измерение времени оказывает влияние обработка прерываний. Следующая программа позволит наблюдать задержки, возникающие вследствие обработки различных прерываний. Функция `disable()` выключает все прерывания (что приводит к прекращению функционирования большинства устройств компьютера, в том числе и клавиатуры). Поэтому прерывания следует выключать на возможно более короткое время! Функция `enable()` возобновляет обработку всех прерываний, что позволяет работать всем устройствам компьютера. Программа измерения времени приведена в **Листинге 13.4**.

Листинг 13.4. Программа измерения времени, файл `time.cpp`

```
#include <iomanip.h>
#include <math.h>
#include <iostream.h>
#include <conio.h>
#include <dos.h>

#include "pctimer.h"

main()
{
    PCTimer T;
    double TimeValue[1000];
    int i;

    // disable();
    for(i = 0; i < 1000; i++)
    {
        for(int j = 0; j < 50; j++)
            sin(j);
        TimeValue[i] = T.ReadTimer();
    }
    enable();

    for(i = 1; i < 1000; i++)
```

```

{
    cout << i << '\t';
    cout << setprecision(3) << TimeValue[i] << '\t';
    cout << setprecision(3) << (TimeValue[i] -
                                TimeValue[i - 1]) << '\t';
    cout << endl;

    if(i % 20 == 0)
    {
        cout << "Press a key for more ...";
        getch();
        cout << endl;
    }
}

return 0;
}

```

Сборка исполняемого файла		
Требуемые файлы	Листинг	Содержимое файла проекта
pctimer.cpp	Листинг 13.3	pctimer.cpp
pctimer.h	Листинг 13.1	
time.cpp	Листинг 13.4	time.cpp

Программа измеряет время выполнения каждой итерации цикла `for`, который выполняется 1000 раз. Тело этого цикла представляет собой вложенный цикл `for`, который 50 раз вычисляет функцию `sin()` с единственной целью потратить время. Меняя количество итераций вложенного цикла, можно менять время выполнения тела основного цикла `for`. Функция `ReadTimer()` вызывается в каждой итерации основного цикла `for`, читает значение таймера и сохраняет полученное значение в массиве. Следует отметить, что здесь нет необходимости в вызове функции `UpdateTicks()`, поскольку `ReadTimer()` осуществляет подсчёт тиков таймера, а также итерация основного цикла `for` выполняется быстрее, чем обратный отсчёт таймера.

На экран выводятся затраченное время и разница между двумя последовательно считанными значениями времени построчно, по 20 строк за раз. Прерывания приводят к увеличению времени, затраченного на вычисление функции `sin()`. Поэтому, если прерывания включены (то есть обрабатываются), то время выполнения вычисления будет каждый раз разным, и это будет видно по ненулевой разнице времени выполнения. Если прерывания выключить, то вычисления будут происходить за одно и то же время. Таким образом, произвольные значения времени в третьей колонке вызваны обработкой прерываний. Программу можно выполнить дважды, первый раз с выключенными прерываниями, а второй раз со включенными, и сравнить результаты. Полученные данные наглядно демонстрируют влияние прерываний на измерение времени.

13.6. Измерение скорости реакции человека

В этом разделе создадим программу, измеряющую время человеческой реакции. Программа будет зажигать светодиоды на интерфейсной плате после произвольной задержки. Испытуемый человек, увидев загоревшиеся светодиоды, должен нажать на кнопку, расположенную на интерфейсной плате. Время, прошедшее с момента включения светодиодов до нажатия на кнопку и будет временем реакции. Программа измерения времени реакции приведена в **Листинге 13.5**.

Листинг 13.5. Программа измерения времени реакции, файл `reflex.cpp`

```
#include <iomanip.h>
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>

#include "pport.h"
#include "pctimer.h"

main()
{
    double ReflexTime;
    ParallelPort PPort;
    PCTimer T;

    // Выключение светодиодов
    PPort.WritePort0(0);
    // Длинный гудок
    cout << "\a\a\a\a\a";

    // Задержка от 1.5 до 5 с
    T.Delay(1500 + rand()%3500);

    // Зажечь все 8 светодиодов
    PPort.WritePort0(255);

    // Сброс таймера
    T.ResetTimer();

    // Ждать нажатия кнопки
    while((PPort.ReadPort1() & 0x80) == 0)
        T.UpdateTicks();

    // Чтение таймера
    ReflexTime = T.ReadTimer();

    // Выключение светодиодов
    PPort.WritePort0(0);

    cout << "Your reflex time is ";
```

```

cout << setprecision(3) << ReflexTime;
cout << " ms." << endl;
getch();

return 0;
}

```

Сборка исполняемого файла		
Требуемые файлы	Листинг	Содержимое файла проекта
pport.cpp	Листинг 10.8	pport.cpp
pport.h	Листинг 10.7	
pctimer.cpp	Листинг 13.3	pctimer.cpp
pctimer.h	Листинг 13.1	
reflex.cpp	Листинг 13.5	reflex.cpp

После запуска программа выключает все светодиоды и формирует длительный звуковой сигнал системным динамиком. Затем происходит задержка, длительность которой произвольно выбирается из интервала 1.5...5 с, после чего загораются все светодиоды, а таймер начинает отсчитывать время. Подопытный человек спустя некоторое время реагирует на загоревшиеся светодиоды, нажимая на кнопку. Программа обнаруживает нажатие на кнопку и вызывает функцию `ReadTimer()`, которая вычисляет время, прошедшее с момента сброса таймера до нажатия на кнопку. Светодиоды затем гасятся, а время реакции выводится на экран.

Сигналы регистра `BASE` на интерфейсной плате нужно подключить к драйверу светодиодов в соответствии с **Табл. 13.4**. Кнопка подключается ко входу регистра `BASE+1`, **Табл. 13.5**.

Таблица 13.4. Подключение светодиодов

Сигналы регистра BASE (буфер U13)	Номер вывода ULN2803A (U3)
D0	1
D1	2
D2	3
D3	4
D4	5
D4	6
D6	7
D7	8

Таблица 13.5. Подключение кнопки

Кнопка	Сигнал регистра BASE+1 (буфер U6)
OUT	D3

13.7. Формирование развёртки во времени

В главе 10 была разработана программа, отображающая на экране осциллограмму сигнала ГУН (**Листинг 10.11**), при этом горизонтальная развёртка (по оси времени) выполнялась с использованием задержек в виде циклов, без привязки к реальному времени. В той программе после чтения сигнала ГУН просто прорисовывался отрезок осциллограммы и инкрементировалось значение *i*. Поэтому каждое значение *i* означало «новое значение сигнала». Имеющиеся теперь возможности работы в реальном времени можно применить для точной развёртки сигнала по оси времени. В **Листинге 13.6** приведена изменённая версия оригинальной программы, изменения выделены жирным шрифтом. Новая программа осуществляет горизонтальную развёртку сигнала в реальном времени.

В новой программе переменная *i* по-прежнему означает позицию сигнала по горизонтали, но имеет другое значение. Теперь *i* выражает положение сигнала в 10 мс интервалах времени. В этом случае главный цикл программы выполняется непрерывно, каждые 10 мс инкрементируя *i*. Поэтому каждые 10 мс сигнал будет «продвигаться» по горизонтали на один пиксел. Таким образом, разрешение по горизонтали будет составлять 10 мс. Вследствие этого изменения сигнала, происходящие чаще чем раз в 10 мс будут отображаться неправильно, например сплошной последовательностью вертикальных линий. Для коррекции осциллограммы в этом случае необходимо уменьшить величину задержки между чтениями сигнала.

Листинг 13.6. Осциллограмма в реальном масштабе времени, файл `timebase.cpp`

```

/*****
Частота выходного сигнала ГУН будет меняться вместе
с входным сигналом. Сигнал на вход ГУН подаётся с
потенциометра POT1 интерфейсной платы. Программа
считывает сигнал с выхода ГУН. Осциллограмма сигнала
ГУН отображается на экране.
*****/

#include <graphics.h>
#include <stdlib.h>
#include <iostream.h>
#include <conio.h>
#include <dos.h>

#include "vco.h"
#include "pctimer.h"

void main()
```

```

{
    VCO Vco;
    PCTimer T;
    int i = 0; // управляет развёрткой по горизонтали
    int SignalLevel;
    int Driver = DETECT, GraphicsMode, ErrorCode;
    int X, Y;

    // Включение графического видеорежима
    initgraph(&Driver, &GraphicsMode, "");

    // Не возникло ли ошибок
    ErrorCode = graphresult();
    if(ErrorCode != grOk)
    {
        cout << "Graphics error: "
              << grapherrormsg(ErrorCode) << endl;
        cout << "Press any key to halt:" << endl;
        getch();
        exit(1);
    }

    X = getmaxx();
    Y = getmaxy();
    rectangle(X/4 - 1, Y/2 - 76, X*3/4 + 1, Y/2 + 76); // рамка
    setviewport(X/4, Y/2 - 75, X/4*3, Y/2+75, 1);

    T.ResetTimer();
    while(!kbhit())
    {
        SignalLevel = Vco.SignalLevel();

        if(SignalLevel == 0) // НИЗКИЙ уровень сигнала
            lineto(i, 100);
        else // ВЫСОКИЙ уровень сигнала
            lineto(i, 50);

        if(T.ReadTimer() > 10)
        {
            T.ResetTimer();
            i++;
        }

        if(i > X/2) // достигнут конец порта просмотра
        {
            i = 0;
            while(Vco.SignalLevel()); // ожидание НИЗКОГО уровня
            while(!Vco.SignalLevel()); // ожидание ВЫСОКОГО уровня
            clearviewport();
        }
    }
}

```

```

        moveto(0, 50);
        T.ResetTimer();
    }
}

```

Сборка исполняемого файла		
Требуемые файлы	Листинг	Содержимое файла проекта
pport.cpp	Листинг 10.8	pport.cpp
pport.h	Листинг 10.7	
vco.cpp	Листинг 10.4	vco.cpp
vco.h	Листинг 10.1	
pctimer.cpp	Листинг 13.3	pctimer.cpp
pctimer.h	Листинг 13.1	
timebase.h	Листинг 13.6	timebase.cpp

Перед началом рисования осциллограммы задаётся «начало отсчёта» времени посредством функции `ResetTimer()`. Также эта функция вызывается каждые 10 мс, иницилируя отсчёт нового интервала времени длительностью 10 мс:

```

if(T.ReadTimer() > 10)
{
    T.ResetTimer();
    i++;
}

```

Таким образом, счётчик `i` инкрементируется каждые 10 мс, что приводит к выводу на экран очередного пиксела осциллограммы. Пока 10 мс не истекли, `i` остаётся неизменным и вывод пиксела выполняется в старую позицию.

Когда осциллограмма достигнет границы порта просмотра (то есть станет равной половине ширины экрана, $x/2$), в теле соответствующего выражения `if` происходит подготовка к выводу новой осциллограммы. При этом `i` обнуляется, а затем выполняется ожидание перехода сигнала ГУН от НИЗКОГО уровня к ВЫСОКОМУ с помощью следующих выражений:

```

while(Vco.SignalLevel()); // ожидание НИЗКОГО уровня
while(!Vco.SignalLevel()); // ожидание ВЫСОКОГО уровня

```

Ожидание такого перехода сигнала ГУН необходимо для того, чтобы осциллограмма всегда начиналась с перехода сигнала от НИЗКОГО уровня к ВЫСОКОМУ. Как только обнаруживается такой переход сигнала, порт просмотра очищается от прежнего содержимого, текущая горизонтальная координата вывода пиксела переносится в крайнюю левую позицию порта просмотра, а таймер сбрасывается для отсчёта новых 10 мс. Следует отметить, что обработка прерываний может приводить к увеличению длительности некоторых импульсов из-за затрат времени на обслуживание прерываний.

Теперь, когда есть программа вывода осциллограммы в реальном масштабе времени, можно считывать значения сигнала с известными метками времени. Сигнал в этом случае будет генерироваться при помощи ГУН и схемы заряда/разряда интерфейсной платы, затем будет происходить его оцифровка в АЦП и запись полученных значений в файл вместе с метками времени значений сигнала.

13.8. Сбор данных с метками времени

Процесс сбора данных можно продемонстрировать путём оцифровывания выходного сигнала схемы заряда/разряда, которая управляется цифровым сигналом. Каждой выборке сигнала в программе сбора данных можно поставить в соответствие метку времени, определённую при помощи объекта класса `PCTimer`.

В этом разделе будут разработаны две программы. Первая программа будет выполнять сбор данных с метками времени и сохранять эти данные в файл на диске. Вторая программа предназначена для чтения и обработки данных из файла с целью определения периода сигнала, полученного с выхода схемы заряда/разряда.

13.8.1. Схема заряда/разряда

Схемой заряда/разряда можно управлять любым цифровым сигналом, в том числе и выходным сигналом параллельного порта компьютера. В данном случае для простоты схемой заряда/разряда будет управлять выходной цифровой сигнал ГУН, **Рис. 13.5**. Конденсатор схемы заряда/разряда будет заряжаться при НИЗКОМ уровне сигнала ГУН и разряжаться при ВЫСОКОМ. Выходной аналоговый сигнал схемы заряда/разряда (**Рис. 13.6**) подаётся на вход аналого-цифрового преобразователя. Программа будет делать выборки на протяжении нескольких периодов этого сигнала и вычислять метку времени каждой выборки. Полученные данные будут записываться в файл.

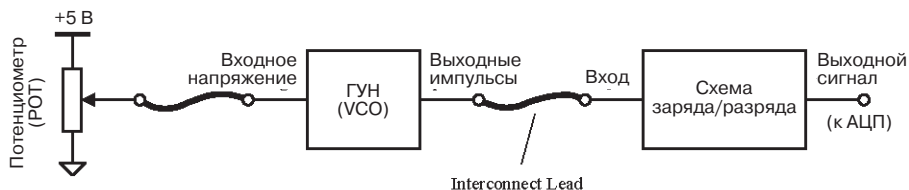


Рис. 13.5. Подключение ГУН к схеме заряда/разряда

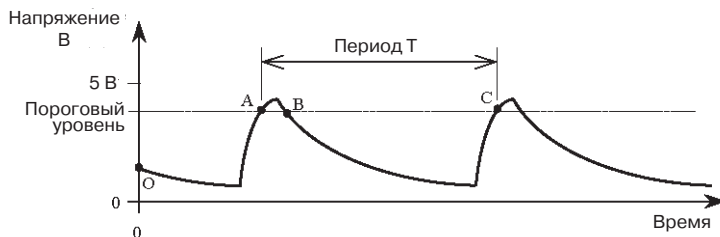


Рис. 13.6. Сигнал на выходе схемы заряда/разряда

13.8.2. Сбор данных с метками времени

В этом разделе будут разработаны две программы. Первая программа (файл `TimeStmp.cpp`) будет оцифровывать сигнал при помощи АЦП и определять метки времени для полученных данных. Данные будут попарно (значение сигнал+метка времени) записываться в файл на диске. В программе будут использованы классы `ADC` и `PCTimer`. Класс `VCO` здесь не используется, поскольку он используется только в качестве генератора импульсов для схемы заряда/разряда и с программой не взаимодействует.

Вторая программа (файл `Period.cpp`) считывает данные из файла на диске и определяет с их помощью период T входного сигнала. Ранее созданные классы в этой программе не используются.

Первая программа – `TimeStmp.cpp`

Программа выполняет следующие действия:

1. Запускает измерение времени.
2. Выполняет цикл `while` в течение заданного времени (рекомендуется 5 секунд):
 - Прочитать значение времени и сохранить его в массиве.
 - Прочитать результат АЦП и сохранить его в массиве.
 - Выждать паузу (рекомендуется 10 мс).
3. Записывает данные в файл.

Программа сбора данных в реальном масштабе времени приведена в **Листинге 13.7**.

Листинг 13.7. Программа сбора данных с метками времени в файле `timestmp.cpp`

```
#include <iomanip.h>
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <conio.h>

#include "adc.h"
#include "pctimer.h"

void main()
{
    ADC Adc;
    PCTimer T;
    double Time[500];
    unsigned char Data[500];
    double TempTime;
    double Duration = 5000; // Длительность сбора данных - 5000 мс
    int i = 0;
    const int SamplingInterval = 10; // в миллисекундах

    T.ResetTimer();
```

```

do
{
    TempTime = T.ReadTimer();
    if(TempTime > i*SamplingInterval)
    {
        Data[i] = Adc.ADConvert();
        Time[i++] = TempTime;
    }
}
while(T.ReadTimer() < Duration);

// Создать и открыть файл, затем записать в него данные
ofstream os("timestamp.dat");

for(int j = 0; j < i; j++)
{
    os << setprecision(3) << Time[j] << '\t';
    os << setprecision(3) << (double) Data[j] << endl;
}
os.close(); // закрыть файл
}

```

Сборка исполняемого файла		
Требуемые файлы	Листинг	Содержимое файла проекта
pport.cpp	Листинг 10.8	pport.cpp
pport.h	Листинг 10.7	
adc.cpp	Листинг 10.4	adc.cpp
adc.h	Листинг 10.1	
pctimer.cpp	Листинг 13.3	pctimer.cpp
pctimer.h	Листинг 13.1	
timestamp.cpp	Листинг 13.7	timestamp.cpp

В программе использованы классы ADC и PCTimer. Класс ADC служит для чтения данных АЦП. При помощи класса PCTimer определяется момент времени выборки сигнала. Создаются два экземпляра этих классов: объект Adc класса ADC и объект T класса PCTimer.

Сбор данных выполняется в течении 5 секунд с периодом выборки в 10 мс, тогда всего получается 500 выборок. Для хранения значений выборок и меток времени создаются два массива Data[] и Time[], соответственно, каждый массив состоит из 500 элементов, а переменная i служит индексом доступа к элементам массива. В переменной Duration хранится длительность сбора данных, 5000, в миллисекундах, то есть 5 секунд. Переменная SamplingInterval содержит значение периода выборок, 10 мс. Поскольку период выборки в процессе сбора данных остаётся неизменным, то его значение объявлено как const. Прочитанное из таймера компьютера время сохраняется в переменной TempTime.

В начале тела цикла `do-while` происходит запуск измерения времени объектом `t` класса `PCTimer`. В цикле выполняется чтение результата АЦП с интервалом в 10 мс. В условии `if` происходит сравнение текущего времени с периодом выборки. Когда текущее время становится больше периода выборки, то выполняется чтение результата АЦП, а полученные данные и метка времени сохраняются в массивах `Data[]` и `Time[]`, соответственно. Цикл `while` завершается по истечении времени, заданного в `Duration`.

Накопленные в массивах `Data[]` и `Time[]` данные затем записываются в файл `timestmp.dat`. Файл создаётся в конструкторе класса `ofstream` при создании объекта `os`. Имя файла `timestmp.dat` передаётся конструктору класса `ofstream` в качестве параметра. В цикле `for` переменная `j` целого типа инициализируется значением 0 и используется для последовательного вывода накопленных данных в файл до тех пор, пока значение `j` не станет равно количеству накопленных данных в предыдущем цикле `do-while`. По завершении записи данных файл объекта `os` класса `ofstream` закрывается посредством функции `close()` этого класса. В результате выполнения этой программы создаётся файл `timestmp.dat`, где в двух колонках хранятся данные, разделённые символом табуляции. В первой колонке находятся метки времени, а во второй значения выборок сигнала со схемы заряда/разряда.

Вторая программа – *Period.cpp*

Программа считывает данные из файла и сохраняет их в памяти компьютера. Считанные данные затем обрабатываются для определения периода входного сигнала.

Выполняются следующие действия:

1. Чтение данных из файла и сохранение их в памяти.
2. Просмотр всех считанных значений для обнаружения двух точек сигнала:
 - Искать значение из второй колонки, которое меньше порогового значения (точка О).
 - Искать дальше, пока не будет обнаружено значение, превышающее пороговое (точка А) и запомнить соответствующую ему метку времени.
 - Искать дальше, пока не будет обнаружено значение меньше порогового (точка В).
 - Искать дальше, пока не будет обнаружено значение, превышающее пороговое (точка С) и запомнить соответствующую ему метку времени.На этом поиск завершён.

3. Вычислить период сигнала, равный разности двух меток времени С и А.

Программа, реализующая эти действия, приведена в **Листинге 13.8**. В этой программе не используются ранее созданные классы.

Листинг 13.8. Программа вычисления периода сигнала в файле `period.cpp`

```
#include <iostream.h>
#include <fstream.h>

void main()
{
```

```
double *TimePtr; // Указатель на метку времени
double *DataPtr; // Указатель на результат АЦП
double MaxData = 0; // Наибольшее значение результата АЦП
int NumData = 0; // Количество пар данных в файле
int i = 0;
int Case = 1; // Шаг поиска, изначально равен 1
int Quit = 0; // Если Quit == 0, то продолжать работу
unsigned char Threshold;
double TimeA, TimeC;

// Создать объект is класса ifstream
ifstream is("tmddata.dat");

// Последовательное чтение пар данных из файла
// и определение максимального значения результата АЦП
while(is)
{
    is >> *TimePtr >> *DataPtr;
    if(!is.fail())
    {
        if(*DataPtr > MaxData)
            MaxData = *DataPtr;
        NumData++;
    }
}

// Закрывать файл
is.close();

// На основании MaxData вычислить пороговое значение
Threshold = MaxData - 5;

// Выделить память для данных АЦП и меток времени
TimePtr = new double[NumData];
DataPtr = new double[NumData];

// Снова открыть файл, чтение при этом опять начнётся с начала файла
is.open("tmddata.dat");

// Читать данные из файла и заполнять ими соответствующие массивы
while(is)
{
    is >> *(TimePtr + i) >> *(DataPtr + i);
    if(!is.fail())
        i++;
}

// Последовательный перебор всех элементов
// массива, на который указывает DataPtr
```

```

for(int j = 0; j < i; j++)
{
    switch(Case)
    {
        // Поиск результата АЦП, который меньше порогового значения
        case 1: if(*(DataPtr + j) < Threshold)
                Case = 2;
                else
                break;

        // Поиск результата АЦП, который больше порогового значения,
        // запомнить метку времени
        case 2: if(*(DataPtr + j) > Threshold)
                {
                    TimeA = *(TimePtr + j);
                    Case = 3;
                }
                else
                break;

        // Поиск результата АЦП, который меньше порогового значения
        case 3: if(*(DataPtr + j) < Threshold)
                Case = 4;
                else
                break;

        // Поиск результата АЦП, который больше порогового значения,
        // запомнить метку времени, установить флаг выхода
        case 4: if(*(DataPtr + j) > Threshold)
                {
                    TimeC = *(TimePtr + j);
                    Quit = 1;
                }
    }
    if(Quit)
        break;
}

// Освобождение ранее выделенной памяти
delete TimePtr;
delete DataPtr;

// Вывести на экран значение периода
cout << "The VCO signal period is ";
cout << TimeC - TimeA;
cout << " ms." << endl;
}

```

Сначала происходит последовательное чтение данных из файла, при этом определяется количество записей (пар данных результат АЦП+метка времени) в фай-

ле и найденное значение запоминается в переменной `NumData`. Наряду с этими действиями выполняется поиск максимального значения результата АЦП, найденное значение запоминается в переменной `MaxData`. Пороговое значение для вычисления периода определяется как уменьшенное на 5 максимальное значение результата АЦП, сохранённое в переменной `MaxData`. Программа выделяет память для хранения данных из файла, при этом указатели `DataPtr` и `TimePtr` указывают на соответствующие размещённые в памяти массивы. Файл закрывается, а затем опять открывается, чтобы чтение было возобновлено с начала файла и можно было заполнить созданные массивы данных.

В оставшейся части программы происходит последовательный перебор прочитанных из файла значений, которые хранятся теперь в массивах. Сначала ищется результат АЦП (из второй колонки), который меньше порогового значения. На **Рис. 13.6** это значение находится в точке *О*. Поиск продолжается, начиная с этого элемента. Теперь ищется результат АЦП, превышающий пороговое значение, на **Рис. 13.6** это значение находится в точке *А*. При обнаружении такого значения запоминается соответствующая ему метка времени. Затем поиск возобновляется, начиная с этого значения, и ищется результат АЦП, меньший порогового значения, это значение соответствует точке *В* на **Рис. 13.6**. Метку времени этой точки запоминать не нужно. Поиск продолжается от найденного значения, пока не будет обнаружен результат АЦП, превышающий пороговое значение, точка *С* на **Рис. 13.6**, при этом для найденного значения запоминается соответствующая ему метка времени. После этого выставляется флаг `Quit`, означающий, что просмотр данных можно на этом завершить. Тогда период сигнала будет равен разности сохранённых меток времени в точках *А* и *С*.

В переменной `Case` отслеживается текущая точка сигнала при переборе значений сигнала. Если `Case` равна 1, то в массиве `DataPtr` ищется первое значение сигнала, меньшее порогового. После обнаружения такого значения, `Case` присваивается значение 2, что приводит к поиску следующего значения сигнала. Переменная `Quit` играет роль флага. Изначально она равна 0, что означает «не нужно завершать работу». Значение `Quit` проверяется в каждой итерации, и если оно станет ненулевым, то выполнение цикла будет прервано выражением `break`. Как только будет обнаружена последняя точка *С*, `Quit` присваивается значение 1 и выполнение цикла поиска прекращается. После этого на экране печатается разность найденных меток времени, которая представляет собой период сигнала.

13.9. Заключение

В этой главе был изучен таймер компьютера, рассмотрены принцип его работы и способы использования. Был создан класс `PSTimer`, позволяющий измерять длительные интервалы времени. Функции-члены класса позволяют запустить измерение времени, прочитать точное значение измеренного времени, а также формировать задержки выполнения программы заданной длительности.

Класс `PSTimer` работает при включенных прерываниях. Вследствие этого время, затрачиваемое на обслуживание прерываний, приводит к незначительным искажениям в сторону увеличения при измерении времени. Влияние прерываний было продемонстрировано специальной программой, выполняющей последовательность операций равной длительности, сначала со включенными прерывания-

ми, а затем с выключенными. Также были разработаны программы: для измерения скорости реакции человека, для вывода на экран осциллограммы сигнала в реальном масштабе времени, а также для оцифровки сигнала схемы заряда/разряда с метками времени.

13.10. Литература

1. Van Gilluwe, F., *The Undocumented PC*, Addison Wesley, 1994.
2. IBM, *Technical Reference – Personal Computer AT*, IBM Corporation, 1985.
3. Auslander D. M. and Tham, C. H., *Real-Time Software for Control*, Prentice Hall, 1990.
4. Intel, *M8254 Programmable Interval Timer – Data Sheet*, Intel Corporation, 1986.

А ПРИЛОЖЕНИЕ

Электронное оборудование

Принципиальная схема

Интерфейсная плата состоит из нескольких независимых узлов, что позволяет пользователю использовать их в различных проектах книги. Кроме того, есть возможность построения на базе этих устройств широкого спектра других устройств. Всем узлам интерфейсной платы необходим источник питания, многие узлы требуют подключения к параллельному порту компьютера. В связи с этим на плате имеется блок питания и интерфейс параллельного порта.

В первую очередь нужно собрать и наладить *источник питания*. Если возникнут какие-либо неполадки, то ошибку придётся искать не по всей плате, а только в блоке питания. Затем нужно собрать и наладить *интерфейс параллельного порта*, прежде чем приступить к сборке других узлов платы.

ВАЖНО!

Перед установкой и пайкой компонентов на печатную плату необходимо убедиться в отсутствии дефектов на ней. В следующем разделе описано, как искать эти дефекты и устранять их. Остальные разделы посвящены установке компонентов, пайке и наладке устройств.

Печатная плата

Печатную плату следует осмотреть для выявления замыканий или обрывов проводников, которые могли возникнуть в процессе её изготовления или хранения. В случае обнаружения таких дефектов их нужно устранить при помощи острого ножа или паяльника, как описано ниже. В месте разрыва проводника зачищаются от изолирующего покрытия края разрыва, чтобы показалась чистая медь. На это место нужно припаять кусочек медного провода. Замыкания между проводниками устраняются при помощи острого ножа.

При помощи мультиметра нужно убедиться в отсутствии замыкания между проводниками следующих цепей:

- Между +5 В и землёй.
- Между +9 В и землёй.
- Между нестабилизированными 12 В и землёй.
- Между -8 В и землёй.

Удобнее всего проверять эти цепи на контактных площадках, подключенных к этим проводникам, **Рис. А.1**. Сопротивление между тестируемыми цепями не должно быть меньше нескольких мегаом. В противном случае в цепи есть замыкание, которое нужно найти либо визуально, либо путём разрезания проводника на отдельные участки, позволяющие выявить место замыкания при помощи мультиметра.

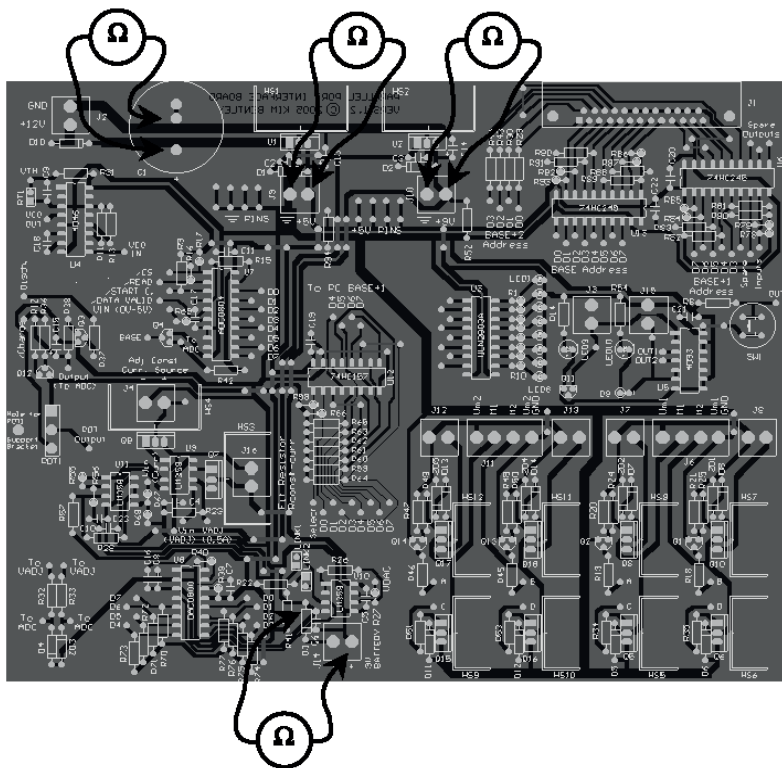


Рис. А.1. Проверка цепей источника питания на короткое замыкание

Сборка

Инструменты: кусачки и маленькие плоскогубцы с длинными губками.

Лучше всего собирать плату постепенно, в порядке следования проектов книги. Наладка платы в этом случае будет проще. Если же плата должна быть собрана сразу, то следующие инструкции помогут это сделать быстрее и проще.

На печатную плату сначала устанавливаются и припаиваются низкие компоненты, а затем более высокие. Тогда последовательность установки элементов будет следующей:

1. Диоды и резисторы поверхностного монтажа.
2. Панельки микросхем и микросхемы.

3. Светодиоды.
4. Выводы печатной платы.
5. Небольшие конденсаторы.
6. Резисторы, устанавливаемые вертикально.
7. Клеммные колодки и остльные высокие компоненты.

После установки компонента на печатную плату нужно слегка разогнуть его выводы, чтобы он удерживался в печатной плате. Когда все элементы одной группы будут установлены на плату, её переворачивают и кладут на плоскую поверхность, прижав тем самым корпуса компонентов к поверхности печатной платы. Теперь печатная плата готова к пайке.

Проще всего устанавливать компоненты небольшими группами, припаявая их, а затем откусывая торчащие над поверхностью платы выводы как можно ближе к месту пайки при помощи кусачек. При этом не возникнет ситуация, когда торчащие выводы множества компонентов будут препятствовать пайке выводов других компонентов.

При установке многих компонентов важно соблюдать их ориентацию на печатной плате. Такими компонентами являются микросхемы, электролитические конденсаторы, светодиоды, диоды и транзисторы. На **Рис. А.2** показано, как обозначается первый вывод у микросхем, что позволяет правильно ориентировать микросхему на печатной плате. На корпусе микросхем наносят специальный знак (ключ) на той стороне микросхемы, где находится её первый вывод.

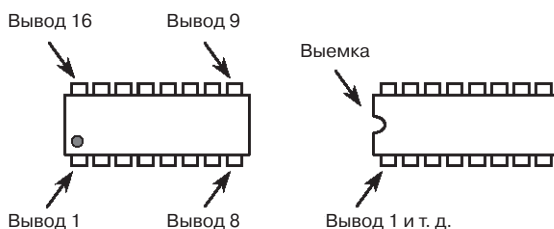


Рис. А.2. Обозначение первого вывода микросхемы (вид сверху)

Если микросхемы устанавливаются в панельки (рекомендуется такой способ установки), то панельки нужно припаять так, чтобы ключ панельки совпадал с ключом контура микросхемы на плате. Тогда при помощи ключа на панельке можно будет правильно установить в неё микросхему.

Выводы прочих компонентов специальным образом маркированы, что позволяет определить их полярность. На электролитических конденсаторах наносят знаки – или +. Для определения полярности светодиода нужно посмотреть его прозрачный корпус на просвет, **Рис. А.3**. Обычно меньший контакт является анодом, а другой катодом.

На корпусах диодов наносят кольцо рядом с выводом катода. Транзисторы устанавливаются на плату в соответствии с нанесёнными на её поверхность контурами корпусов транзисторов, указывающими их правильную ориентацию. Кроме того, микросхемы КМОП могут быть выведены

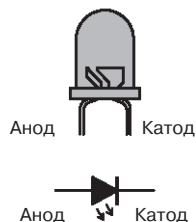


Рис. А.3. Определение полярности светодиода

из строя зарядами статического электричества, поэтому при их установке нужно соблюдать меры предосторожности.

Перед установкой радиаторов охлаждения на транзисторы и стабилизаторы напряжения нужно нанести на металлическую поверхность этих компонентов тонкий слой теплопроводящей пасты для улучшения отвода тепла.

Пайка

Инструменты и материалы: паяльник (не менее 15 Вт) и оловянно-свинцовый припой в виде проволоки диаметром 0.7...1.0 мм, 60% олова, 40% свинца, с флюсом внутри.

Ручная пайка представляет собой нагрев точки пайки, вызывающий расплавление припоя и растекание его по паяным деталям. После прекращения нагрева припой застывает и соединяет тем самым вывод компонента и контактную площадку интерфейсной платы.

Подготовка к пайке: для обеспечения надёжного паяного соединения необходимо, чтобы место пайки и жало паяльника были чистыми. Обычно печатная плата и выводы компонентов достаточно чистые. К сожалению этого нельзя сказать о жале паяльника. Жало паяльника можно очистить влажной «кухонной» губкой, обтирая кончик жала паяльника об губку до металлического блеска. Если не удастся добиться металлического блеска кончика жала, то можно попробовать его *облудить*, на короткое время приложив к припою, а затем снова обтереть об губку. Если эти действия не помогут, то, скорее всего, придётся заменить жало паяльника.

Пайка: выполняются следующие действия:

1. Наберите на кончик жала немного припоя (на жале может находиться достаточное количество припоя от предыдущей пайки). Припой улучшает теплопередачу от жала к месту пайки.
2. Прогрейте жалом паяльника место пайки в течении нескольких секунд.
3. Распределите расплавленный припой по прогретому месту пайки, он должен равномерно растечься по паяемым элементам, при этом не должно возникать излишков припоя.

При распределении припоя по паяным элементам не пытайтесь прикладывать припой к месту пайки, это может привести к некачественной пайке. Если место пайки было хорошо прогрето, то припой хорошо растекается по нему. На **Рис. А.4** показана правильная пайка до и после откусывания торчащих выводов.

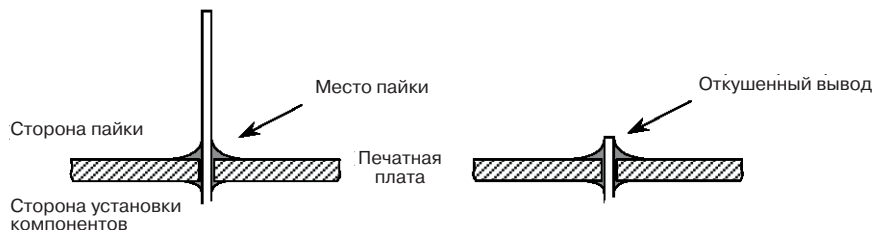


Рис. А.4. Правильно припаянный и откушенный вывод

Правила чтения принципиальной схемы

Следующие сведения помогут правильно понимать принципиальные схемы при изучении устройства узлов и их наладке:

- Обычно входы компонентов изображаются слева, а выходы справа.
- На цифровых входах и выходах могут изображаться небольшие окружности, вплотную к изображению элемента. Так обозначается *инверсия* сигнала или активный НИЗКИЙ уровень. Если это вход, то НИЗКИЙ уровень входного сигнала будет сигнальным. Если это выход, то появление выходного сигнала приведёт к НИЗКОМУ уровню на таком выходе.
- Небольшие точки на концах входных и выходных линий являются контактами печаной платы. Закрашенные точки означают электрические соединения цепей.
- VCC означает шину питания +5 В логических схем.
- Земля обозначается одним из двух символов: несколько штрихов в форме треугольника обозначают цифровую землю, незакрашенный треугольник аналоговую землю. На плате они образуют одну цепь и имеют потенциал 0 В.

Наладка

Наладка схем должна выполняться систематизировано, чтобы не затрачивалось много времени и усилий. Простейший метод наладки заключается в разделении всей схемы на отдельные простые узлы, которые налаживаются и проверяются по-отдельности.

При проверке работоспособности узлов схемы перед поверкой выходного сигнала необходимо подать соответствующий входной сигнал. Если узел не работает, то чаще всего это вызвано следующими причинами:

- Неправильные подключения сигналов.
- Использование компонентов несоответствующих типов.
- Неправильное подключение (ориентация) компонента.
- Плохая пайка или ненадёжный контакт.
- Короткое замыкание, вызванное растеканием припоя.
- Повреждённые проводники печатной платы (дефекты, возникшие в процессе производства или вследствие неправильного обращения с печатной платой).
- Неправильная подача напряжения питания на схему.
- Неисправные элементы.

Типичная последовательность действий при наладке схем:

1. Проверка правильности сборки схемы.
2. Проверка типов установленных компонентов и их ориентации на плате, убедиться в том, что компоненты установлены на свои места.
3. Проверить напряжение питания схемы. Если напряжение питания не соответствует норме, то сделать следующее:

- Проверить, не перегреты ли корпуса микросхем. Если обнаружен чрезмерный нагрев микросхемы, то неисправную микросхему нужно заменить. Проверить температуру остальных элементов схемы.
 - Визуально проверить проводники на предмет непредумышленных замыканий и разрывов, а также качество пайки. В случае необходимости целостность цепей можно проверять при помощи мультиметра, как описано ниже.
4. Проверить, что входные и выходные сигналы находятся в допустимых пределах.
 5. Проверить адекватность выходных сигналов входным сигналам. Если этого не наблюдается:
 - На выходе чрезмерная нагрузка: либо это вызвано подключением к цепям платы, либо коротким замыканием.
 - Компонент неисправен и его нужно заменить.

Проверка целостности цепи

Проверка целостности цепи выполняется с целью обнаружения коротких замыканий и разрывов. Для этого мультиметр переключается в режим измерения сопротивления (прозвонки). Проверка целостности цепей выполняется при ОТКЛЮЧЕННОМ напряжении питания.

Ранее говорилось, что для упрощения наладки нужно собирать и налаживать схему, разбив её на отдельные узлы: сначала собрать и наладить один блок, а затем приступить к следующему блоку. Наладка этих блоков описывается далее в том порядке, в каком они следуют по тексту книги.

Демонтаж компонентов

При удалении компонентов с печатной платы, используя обычные инструменты, следует придерживаться следующих требований, чтобы не повредить удаляемые компоненты. Для этого также можно пользоваться специализированными средствами, впрочем, большинство читателей таковыми не располагают. Компоненты с большим количеством выводов при удалении можно разрушать, чтобы минимизировать риск повреждения печатной платы. Следует учитывать, что печатные проводники печатной платы могут быть повреждены при чрезмерном или продолжительном нагревании.

Демонтаж дискретных компонентов: резисторов, конденсаторов и подобных им компонентов. Если у компонента гибкие выводы, то проще всего захватить один из выводов компонента плоскогубцами, нагреть паяльником место его пайки до расплавления припоя. Как только припой расплавится, вытащить вывод из платы. Те же действия повторяются с другим выводом.

Если выводы компонента жёсткие и невозможно выпаять выводы по отдельности, то выводы предварительно откусываются кусачками от компонента, а затем выпаиваются.

Демонтаж микросхем. Все микросхемы интерфейсной платы должны устанавливаться в панельки. В этом случае микросхемы легко устанавливать и извлекать,

не повреждая при этом печатную плату или саму микросхему. Если же микросхема напрямую запаяна в плату, то перед её демонтажом рекомендуется откусить кусачками все её выводы от корпуса. Потом останется только выпаять все ножки по одной.

Соединительные кабели и провода

В Табл. А.1 приведено всё необходимое для выполнения кабельных соединений. Кабели показаны на Рис. А.5 и Рис. А.6. Рекомендуется изготовить только те соединительные проводники, которые необходимы для конкретного проекта, а кабель параллельного порта типа DB25M-DB25M лучше всего приобрести готовый.

Таблица А.1. Соединительные провода и кабели

Количество	Описание
1	Кабель DB25M–DB25M, выводы разъёмов соединены один-в-один.
50	Контактные гнёзда, допускают подключение контактов диаметром 0.9...1.0 мм (на соединительных проводах)
7 м	Монтажный провод
1 м	Термоусадочная трубка диаметром 2.5...3 мм

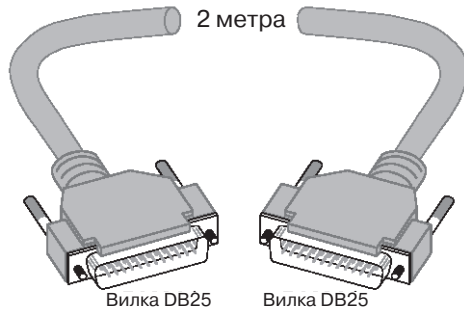


Рис. А.5. Кабель DB25M–DB25M с распайкой один-в-один

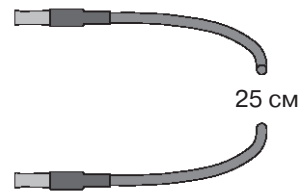


Рис. А.6. Соединительный проводник с двумя гнёздами

Изготовление соединительных проводников. Соединительные проводники используются для подключения выходов одних блоков ко входам других. Два выхода соединять вместе нельзя, в большинстве случаев это приведёт к выходу из строя соединённых компонентов. Изготовить соединительные проводники, Рис. А.6, можно следующим образом.

На оба конца отрезка провода длиной 25 см припаиваются контактные гнёзда. На гнездо надевается 15 мм отрезок термоусадочной трубки так, чтобы он захватывал и провод, и примерно половину гнезда (чтобы предотвратить замыкание рядом расположенных контактов). Для усадки трубки по месту необходимо нагреть её. Таким же образом поступают с другим концом проводника.

Проверка соединительных проводников. Механическая прочность проводников проверяется путём растягивания их в разные стороны за контактные гнёзда с умеренной силой. Если проводник разорвался, то его нужно собрать заново. Электрическая проводимость проверяется мультиметром в режиме измерения сопротивления. Сопротивление между концами проводника не должно превышать тысячных долей Ома.

Блок питания

Сборка. Нужно установить и припаять компоненты, перечисленные в Табл. А.2, согласно обозначениям на печатной плате, Рис. А.8 и Рис. А.9.

Таблица А.2. Блок питания – перечень элементов

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
2	Керамический конденсатор 0.1 мкФ	0.2 дюйма	C2, C3
2	Танталовый электролитический конденсатор 1 мкФ ≥ 16 В	0.2 дюйма	C13, C14
1	Электролитический конденсатор 4700 мкФ ≥ 16 В	0.5 или 0.35 дюйма	C1
1	Резистор 1 кОм	0.25 Вт	R94
1	Резистор 1.8 кОм	0.25 Вт	R52
3	Диод 1N4004		D1, D2, D10
1	Стабилизатор напряжения LM7805CT	TO-220	U1
1	Стабилизатор напряжения LM7809CT	TO-220	U2
1	Модуль трансформатора с выпрямителем, +12 В, 1 А		
3	Двухконтактная клеммная колодка	Шаг 5 мм	J2, J9, J10
16	Штыревой контакт, диаметр 0.9...1.0 мм		
2	Радиатор, тепловое сопротивление 12 °С/Вт		HS1, HS2
2	Винт М3, длина 6...10 мм		
2	Гайка М3		
2	Гровер М3		

Наладка. На Рис. А.7 приведена принципиальная схема блока питания, за исключением цепи питания –8 В, которая предназначена только для ЦАП и к блоку питания не относится. Интерфейсная плата может питаться от любого источника постоянного тока напряжением 13...18 В при токе не менее 1 А. Наиболее дешё-

вым решением является использование трансформаторного модуля на 12 В, 1 А. На выходе трансформаторного модуля напряжение превышает 12 В, пока ток меньше 1 А.

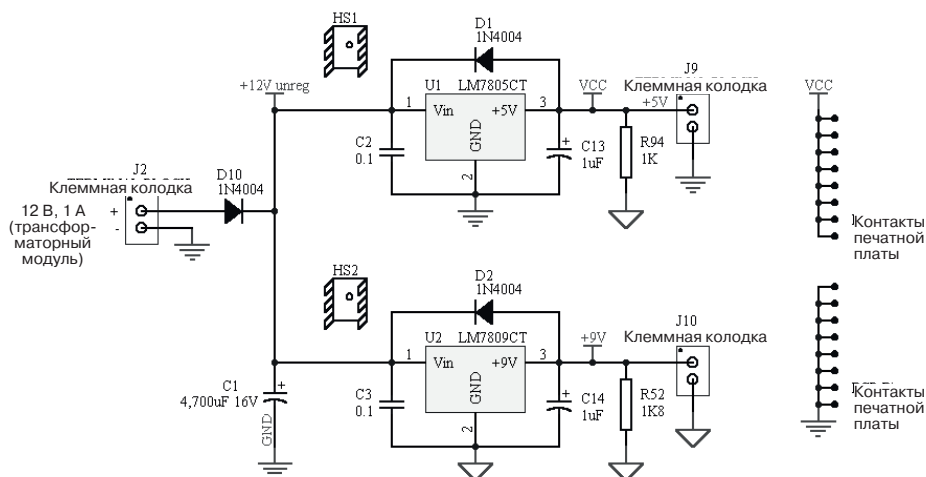


Рис. А.7. Принципиальная схема блока питания

На вход стабилизаторов напряжения +5 В и +9 В должно подаваться напряжение 12...18 В (величина напряжения зависит от трансформаторного модуля или внешнего источника напряжения). Если напряжение не соответствует норме:

- Убедиться в подключении трансформаторного модуля или внешнего источника напряжения, соблюдена ли полярность подключения к клеммной колодке J2.
- Проверить полярность подключения диода D10.

Выходное напряжение стабилизаторов +5 В и +9 В должно быть 4.75...5.25 В и 8.55...9.45 В, соответственно, что соответствует допускам их выходного напряжения. Если это условие не выполняется:

- Убедиться в соблюдении полярности электролитических конденсаторов C13, C14 и диодов D1 и D2.
- Убедиться, что ориентация стабилизаторов на плате совпадает с обозначенной на плате.
- Проверить схему на предмет коротких замыканий и обрывов в местах пайки и подключения проводников.
- Убедиться в наличии резисторов R52 и R94.

Примечание

Рядом с клеммной колодкой источника питания расположены 8 контактов, подключенных к земле, и 8 контактов, подключенных к шине питания +5 В. Они предназначены для подачи логических сигналов на входы схем и для тестирования схем.

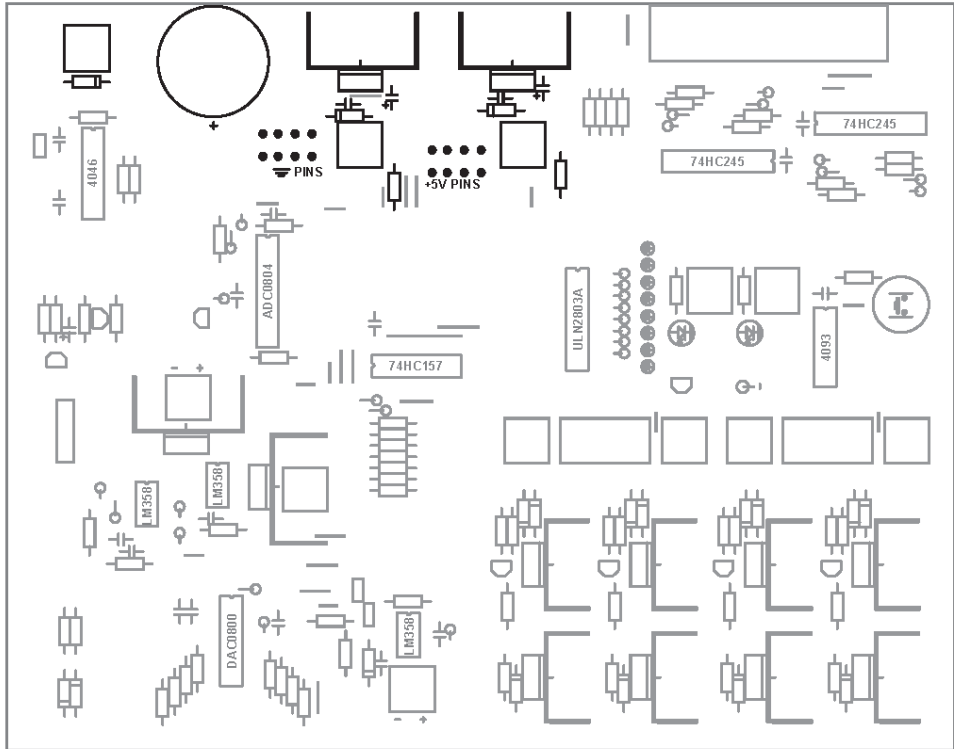


Рис. А.8. Расположение элементов блока питания на печатной плате

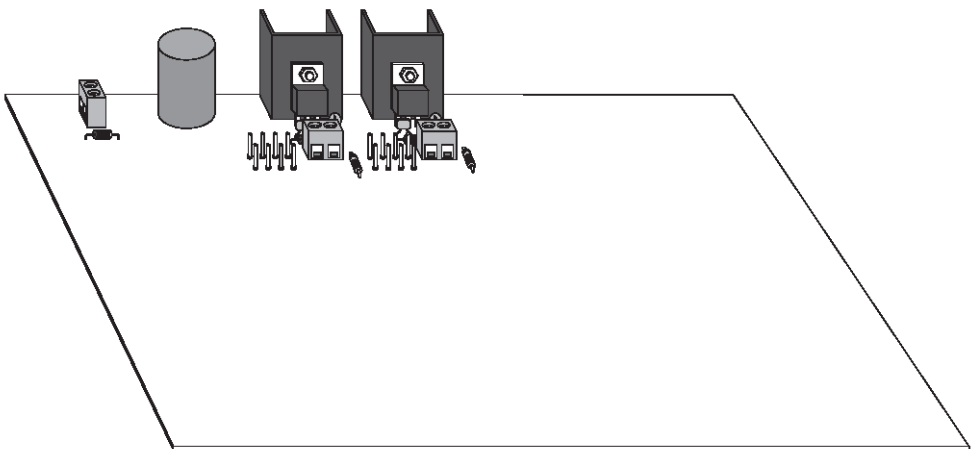


Рис. А.9. Печатная плата с установленными элементами блока питания

Интерфейс параллельного порта

Сборка. Нужно установить и припаять компоненты, перечисленные в Табл. А.3, согласно обозначениям на печатной плате, Рис. А.11 и Рис. А.12.

Таблица А.3. Интерфейс параллельного порта – перечень элементов

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
2	Керамический конденсатор 0.1 мкФ	0.2 дюйма	C20, C22
4	Резистор 470 Ом	0.25 Вт	R29, R30, R43, R44
16	Резистор 10 кОм	0.25 Вт	R78–R93
2	Микросхема 74HC245 (КМОП)	DIL20	U6, U13
2	Панелька для микросхемы	20 контактов	
1	Разъем типа D25 для установки на печатную плату		J1
23	Штыревой контакт, диаметр 0.9...1.0 мм		

Наладка. На Рис. А.10 приведена принципиальная схема интерфейса. Две буферных микросхемы предназначены для защиты цепей параллельного порта компьютера при возникновении неисправностей на плате.

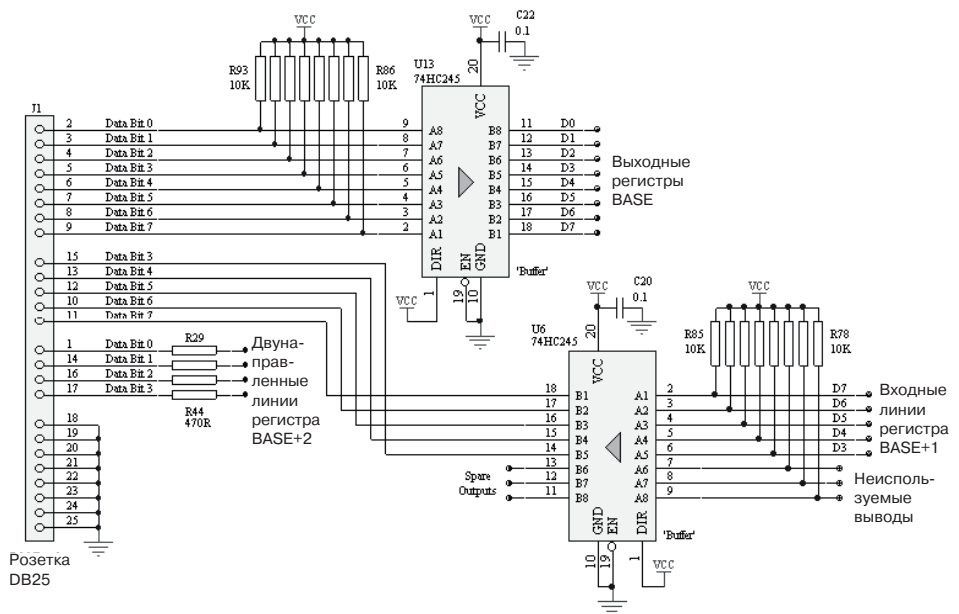


Рис. А.10. Принципиальная схема интерфейса параллельного порта

Перед наладкой следует убедиться, что оба буфера полностью отключены от других цепей, в том числе и от параллельного порта.

Напряжение питания каждого буфера (микросхема 74НС245, вывод 20) должно быть примерно равно +5 В (как и на выходе стабилизатора +5 В). Если это не так:

- Убедиться в правильности установки микросхемы, в правильности установки панельки, в отсутствии коротких замыканий и разрывов, в исправности микросхемы, в исправности источника питания +5 В.

Входы выбора направления передачи данных и разрешения выхода должны подключаться следующим образом:

- Вход DIR (вывод 1) подключается к шине +5 В.
- Вход EN (вывод 19) подключается к земле.

На входах данных буфера установлены *подтягивающие резисторы*. Они нужны для правильного функционирования ТТЛ логики параллельного порта. Неиспользованные входы микросхемы также подключены к шине +5 В посредством подтягивающих резисторов. Входы, на которых нет стабильного логического уровня, могут вызвать непредсказуемое поведение схемы. Подтягивающие резисторы обеспечивают на входах буфера ВЫСОКИЙ уровень даже при отключенном кабеле параллельного порта. На выходах данных буфера при этом тоже будет ВЫСОКИЙ уровень. Четыре резистора, включенные последовательно в цепь разъёма D25 предохраняют линии регистра BASE+2 от выхода из строя.

Во время наладки интерфейса он должен быть отключен от параллельного порта компьютера. Отдельный вход данных буфера (микросхема 74НС245, U13) можно проверить следующим образом:

- На входе данных (A1...A8) должно быть напряжение +5 В, то есть ВЫСОКИЙ уровень. На соответствующем ему выходе (B1...B8) тоже должен быть высокий уровень. Так проверяются все восемь входов.
- На вход данных (A1...A8) подаётся НИЗКИЙ уровень путём подключения его к земле при помощи соединительного проводника. На соответствующем ему выходе (B1...B8) тоже должен появиться НИЗКИЙ уровень. Таким же образом проверяются остальные входы.

Входы данных буфера U6 (74НС245) проверяются следующим образом:

- На входы данных (A1...A8) подаётся НИЗКИЙ уровень путём подключения его к земле при помощи соединительного проводника. На соответствующих выходах (B1...B8) тоже должен появиться НИЗКИЙ уровень.

При подключении входа к шине питания +5 В на выходе тоже должен появиться ВЫСОКИЙ уровень. Так проверяются все входы.

Если вышеприведённые тесты не проходят, то нужно проверить следующее:

- Правильность установки микросхемы, отсутствие коротких замыканий и обрывов, исправность микросхемы.

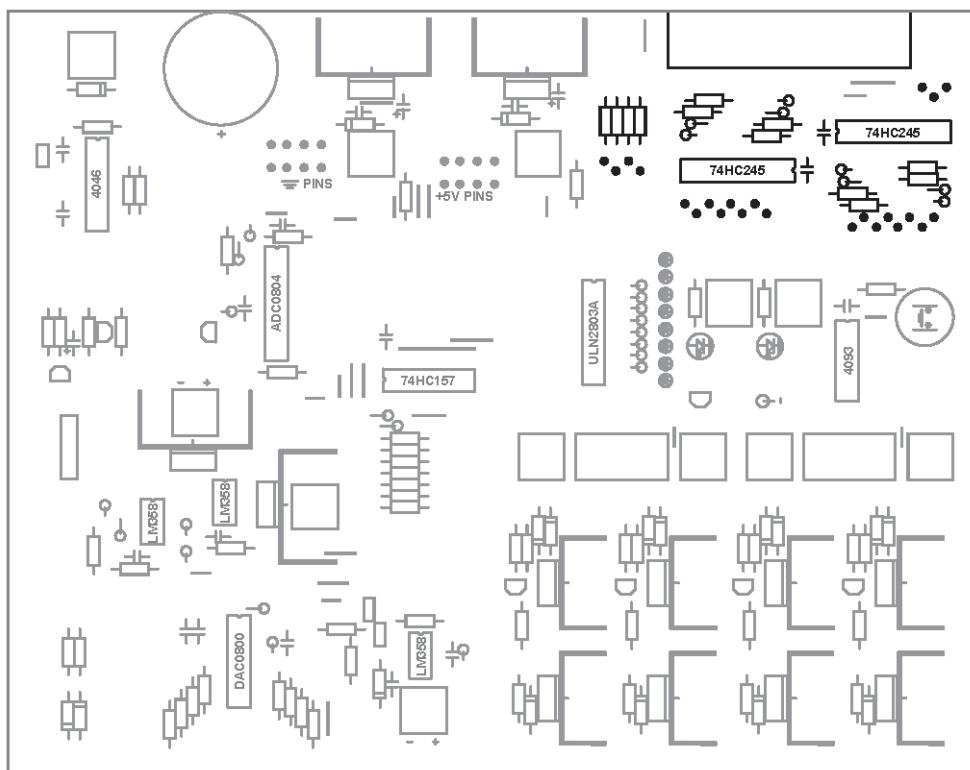


Рис. А.11. Расположение элементов интерфейса параллельного порта на печатной плате

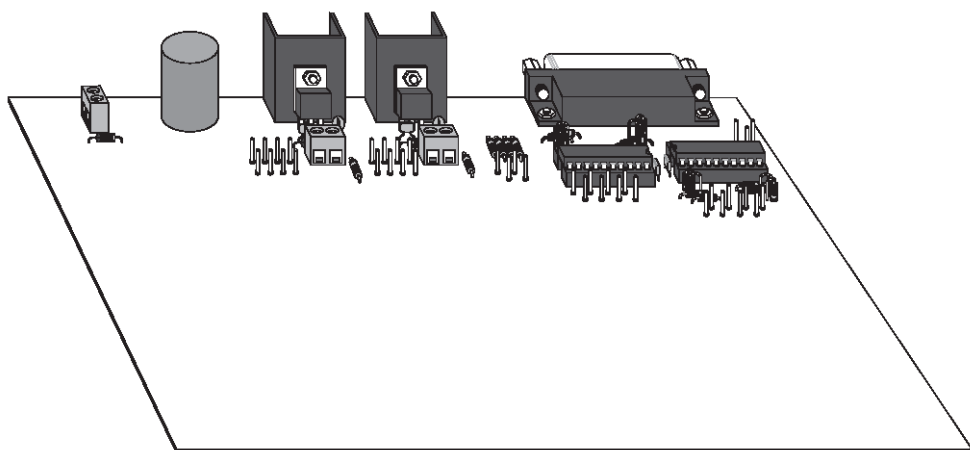


Рис. А.12. Печатная плата с установленными элементами интерфейса параллельного порта

Драйвер светодиодов

Сборка. Нужно установить и припаять компоненты, перечисленные в Табл. А.4, согласно обозначениям на печатной плате, Рис. А.14 и Рис. А.15.

Таблица А.4. Драйвер светодиодов – перечень элементов

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
8	Резистор 330 Ом	0.25 Вт	R1, ...R10 LED1–LED8
8	Красный светодиод, диаметр корпуса 3 мм		
16	Резистор 10 кОм	0.25 Вт	R78–R93
1	Набор ключей ULN2803A	DIL18	U3
1	Панелька для микросхемы	18 контактов	
8	Штыревой контакт, диаметр 0.9...1.0 мм		

Наладка. На Рис. А.13 приведена принципиальная схема драйвера светодиодов. При помощи этой схемы очень удобно проверять уровень цифрового сигнала, считываемого или формируемого программой, в частности, при отладке программы. Схема состоит из микросхемы ULN2803A и подключенных к ней светодиодов с резисторами. Внутри микросхемы находятся восемь независимых транзисторов Дарлингтона (составных транзисторов), используемых для управления током, протекающим через соответствующий выход микросхемы. Светодиоды можно зажигать и гасить, подавая на соответствующие входы микросхемы логические сигналы, таким образом, по светодиодам можно определить уровень цифрового сигнала на входе схемы.

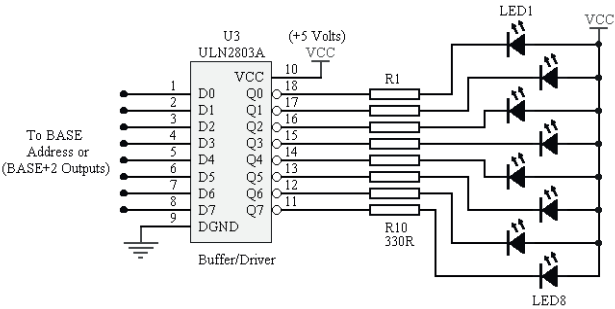


Рис. А.13. Принципиальная схема драйвера светодиодов

При сборке схемы нужно сначала проверить работу схемы с одним светодиодом и резистором, чтобы убедиться в правильности полярности светодиода. Определение полярности светодиодов обсуждалось ранее.

Напряжение питания микросхемы ULN2803A на выводе 10 должно составлять +5 В. На выводах светодиодов, которые не подключены к резисторам, тоже должно быть +5 В. Если это не так, то возможно следующее:

- Микросхема установлена неправильно, неправильно подключена панелька микросхемы, короткие замыкания и разрывы, неисправная микросхема или светодиоды, неисправен источник питания +5 В.

Микросхема ULN2803A работает следующим образом:

1. Если на вход микросхему (D0...D7) подан ВЫСОКИЙ уровень (то есть +5 В), соответствующий выход (Q1...Q7) микросхемы замыкается на землю. Тогда через светодиод начинает протекать ток от шины питания +5 В, через светодиод, резистор и выход микросхемы на землю, и светодиод загорается.
2. Если на входе микросхемы НИЗКИЙ уровень, то соответствующий выход микросхемы будет отключен от земли, ток через светодиод протекать не будет, значит и свечения светодиода не будет.

Если какой-либо светодиод не будет светиться, то нужно проверить:

- Правильность подключения светодиода, нет ли коротких замыканий и разрывов, исправность светодиода и резистора.

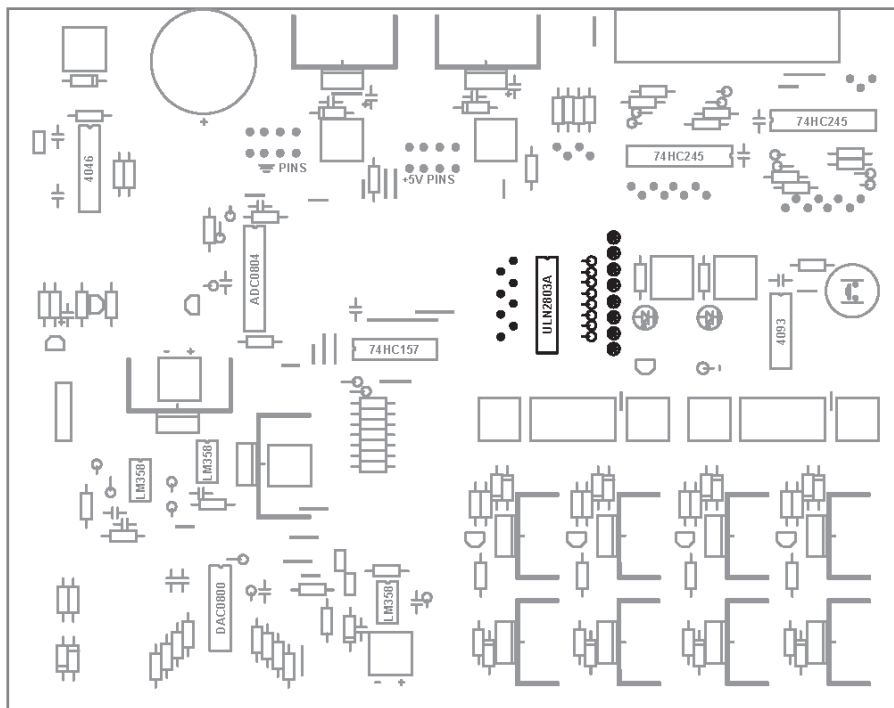


Рис. А.14. Расположение элементов драйвера светодиодов на печатной плате

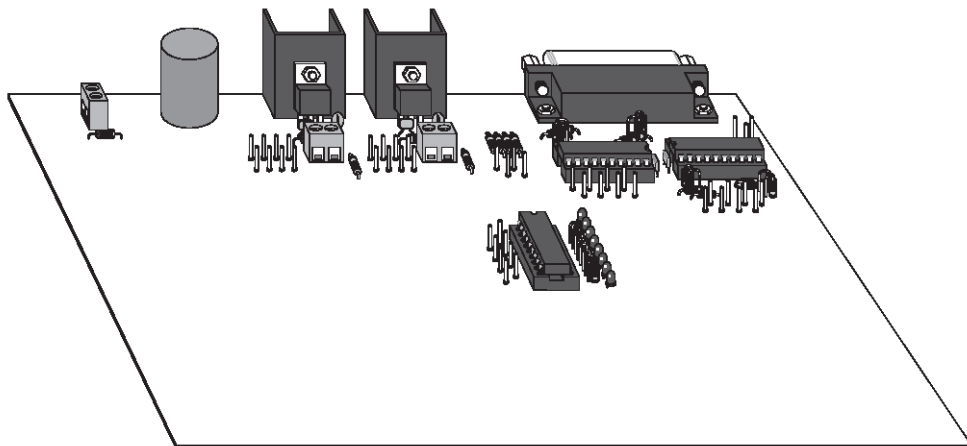


Рис. А.15. Печатная плата с установленными элементами драйвера светодиодов

Цифро-аналоговый преобразователь

Схема цифро-аналогового преобразователя показана на **Рис. А.16**. Она состоит и микросхемы цифро-аналогового преобразователя и буферного усилителя.

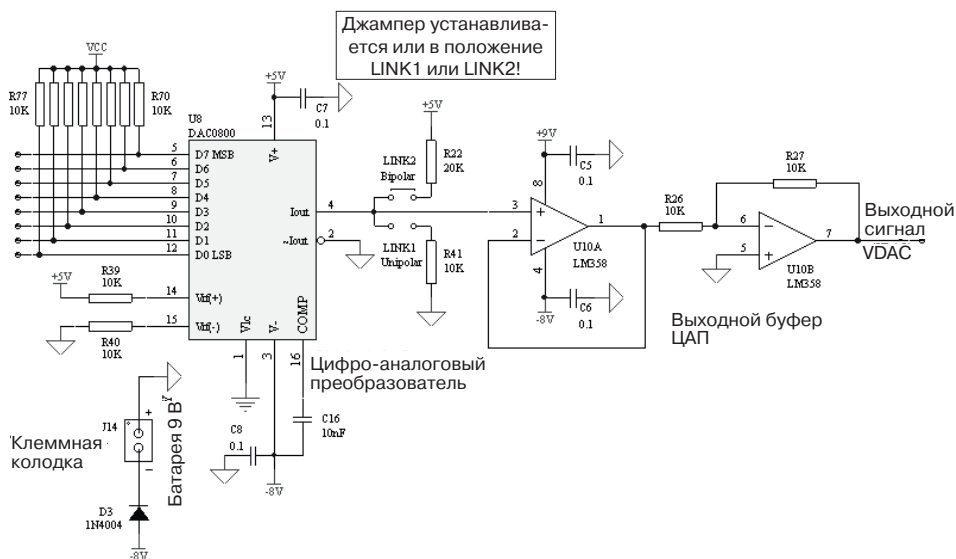


Рис. А.16. Принципиальная схема ЦАП с буфером

Сборка. Нужно установить и припаять компоненты, перечисленные в **Табл. А.5**, согласно обозначениям на печатной плате, **Рис. А.17** и **Рис. А.18**.

Наладка узла ЦАП. Сначала к клемме J14 нужно подключить исправную батарейку 9 В. Положительное напряжение питания ЦАП (U8) на выводе 13 (V+) должно быть +5 В, отрицательное напряжение питания на выводе 3 (V-) примерно -8 В. Если это условие не выполняется, то нужно проверить следующее:

- Правильно ли установлена микросхема, исправна ли панелька микросхем, что батарейка не разряжена, убедиться в отсутствии коротких замыканий или разрывов, проверить исправность микросхемы АЦП и источника питания +5 В.

Если цифровые входы ЦАП (**D0...D7**) никуда не подключены, то, благодаря подтягивающим резисторам, на них будет подаваться **ВЫСОКИЙ** уровень +5 В. Причиной отсутствия этого напряжения может служить:

- Короткие замыкания и разрывы, неисправные резисторы и некачественная пайка.

В позицию LINK1 устанавливают джампер. При **ВЫСОКОМ** уровне на всех цифровых входах ЦАП (когда они никуда не подключены) на выходе (вывод 4) должно быть напряжение -5 В. Если на всех цифровых входах **НИЗКИЙ** уровень, то на выходе ЦАП должно быть 0 В. Если эти условия не выполняются, то это может быть вызвано следующими причинами:

- Плохие контакты, короткие замыкания и разрывы, плохая пайка, неисправные компоненты, несоответствующие номиналы элементов.

Теперь джампер из позиции LINK1 нужно переставить в позицию LINK2 (биполярный режим). При неподключенных входах ЦАП (когда на них **ВЫСОКИЙ** уровень, +5 В) на выходе ЦАП должно быть -5 В. При **НИЗКОМ** уровне на входах ЦАП (0 В) на выходе должно быть +5 В.

Наладка буфера. Буфер выполняет единичное усиление и инверсию выходного сигнала ЦАП. Таким образом, большее число на входе ЦАП вызывает большее выходное напряжение VDAC.

Напряжение питания операционного усилителя U10 должно составлять +9 В на выводе 8 и примерно -8 В на выводе 4. Если это не так, то возможны следующие причины:

- Неправильная установка микросхемы, отсутствие контакта в колодке микросхемы, неисправность источника +9 В, разряженная батарейка на 9 В, короткие замыкания и разрывы, неисправная микросхема LM358.

Когда на выходе ЦАП -5 В (то есть на всех цифровых входах ЦАП +5 В), на неинвертирующем входе буферного усилителя U10 (вывод 3) тоже должно быть -5 В. Аналогично, на инвертирующем входе (вывод 2) и выходе (вывод 1) тоже должно быть -5 В. Если это не так, то возможно следующее:

- Короткие замыкания и разрывы, отсутствие контакта в колодке микросхем, неисправная микросхема LM358.

На обоих входах инвертирующего усилителя U10 (выводы 5 и 6) должно быть 0 В. На выходе этого усилителя (вывод 7) должно быть +5 В. Если эти условия не выполняются, то возможны следующие причины:

- Короткие замыкания и разрывы, отсутствие контакта в панельке микросхемы, несоответствие номиналов резисторов R26, R27 или их неисправность, а также неисправная микросхема LM358.

Таблица А.5. Цифро-аналоговый преобразователь – перечень элементов

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
Источник питания –8 В			
1	Диод 1N4004		D3
1	Двухконтактная клеммная колодка	Шаг 5 мм	J14
1	Клемма подключения батареи 9 В		
ЦАП			
1	Керамический конденсатор 10 нФ	0.2 дюйма	C16
2	Керамический конденсатор 0.1 мкФ	0.2 дюйма	C7, C8
11	Резистор 10 кОм	0.25 Вт	R39–R41, R70–R77
1	Резистор 20 кОм	0.25 Вт	R22
1	Микросхема DAC0800 (КМОП)	DIL16	U8
1	Панелька для микросхемы	16 контактов	
8	Штыревой контакт, диаметр 0.9...1.0 мм		
2	Контакты джампера	0.1 дюйма	LINK1, LINK2
1	Джампер (для позиций LINK1 и LINK2)	0.1 дюйма	
Буферный усилитель			
2	Керамический конденсатор 0.1 мкФ	0.2 дюйма	C5, C6
2	Резистор 10 кОм	0.25 Вт	R26–R27
1	Микросхема LM358	DIL8	U10
1	Панелька для микросхемы	8 контактов	
1	Штыревой контакт, диаметр 0.9...1.0 мм		

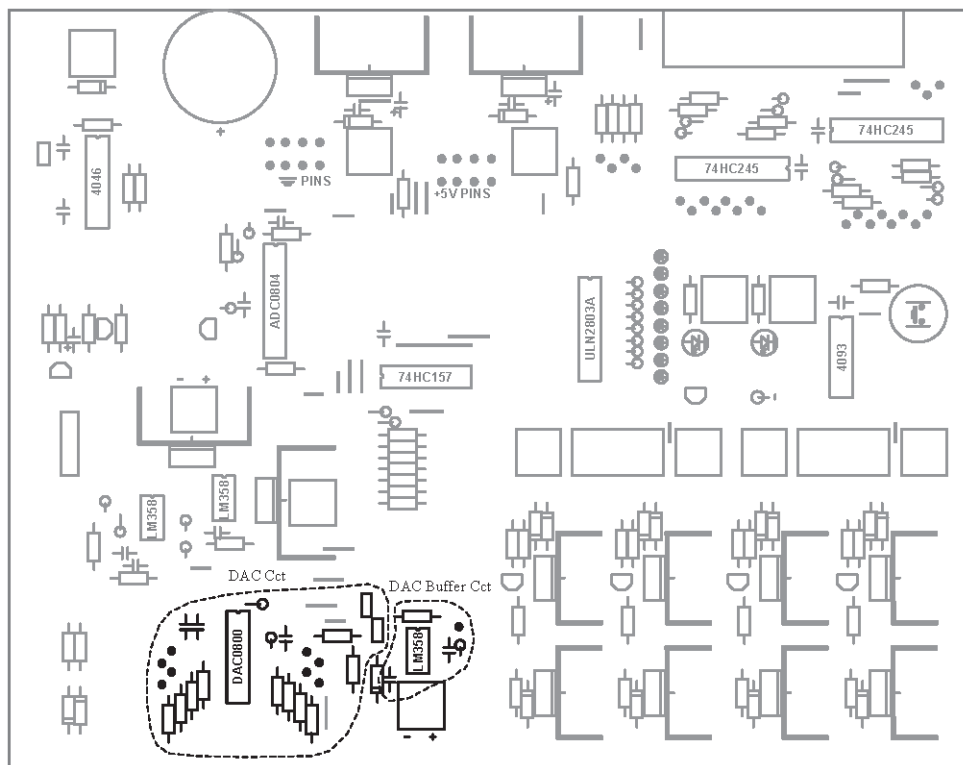


Рис. А. 17. Расположение элементов схемы ЦАП с буферизованным выходом на печатной плате

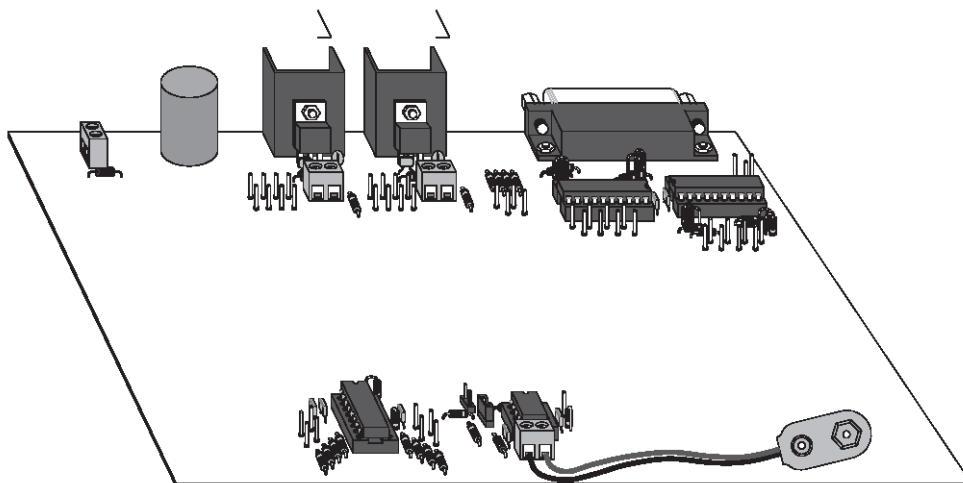


Рис. А. 18. Печатная плата с установленными элементами схемы ЦАП с выходным буфером

Схема управления двигателями

Сборка. Нужно установить и припаять компоненты, перечисленные в Табл. А.6, согласно обозначениям на печатной плате, Рис. А.20 и Рис. А.21.

Наладка. На Рис. А.19 показана принципиальная схема блока управления двигателями. Управление и коллекторными, и шаговыми двигателями осуществляется посредством двух мостов схемы. Входы всех двухконтактных клеммных колодок при наладке схемы нужно замкнуть проводочными перемычками. К четырёхконтактной клеммной колодке нужно подключить источник питания не более 30 В, подходящий для питания двигателей (или трансформаторный модуль 12 В, 1 А, если это возможно): положительный вывод источника подключается ко входам Vm1 и Vm2, а отрицательный к Vm1 GND и Vm2 GND.

На каждой двухконтактной клеммной колодке должно присутствовать напряжение питания двигателей. Если это условие не выполняется, то нужно проверить:

- Исправность трансформаторного модуля или внешнего источника питания, убедиться в наличии перемычек на клеммных колодках.

Каждый мост схемы состоит из четырёх ключевых транзисторов с независимым управлением-ключ «замыкается» при подаче ВЫСОКОГО логического уровня. Мост состоит из транзисторов двух типов:

1. Двух одинаковых транзисторов Дарлингтона типа $n-p-n$ в нижних плечах моста (Q5, например).
2. Двух одинаковых транзисторов Дарлингтона типа $p-n-p$ (например, Q8), каждый из которых управляется инвертором на транзисторе $n-p-n$ типа (например, Q2). Без $n-p-n$ транзистора в схеме инвертора будет невозможно управлять транзистором Дарлингтона $p-n-p$ типа, поскольку для закрытия этого транзистора в данном случае необходим сигнал, напряжение которого равно напряжению питания двигателя (Vm1). Инвертор меняет уровень входного логического сигнала и способен формировать напряжения вплоть до Vm1. Выходной сигнал такого инвертора управляет транзистором Дарлингтона $p-n-p$ типа.

Нужно проверить ключи обоих мостов. Вместо двигателя нужно подключить небольшую нагрузку, например резистор сопротивлением 1 кОм к клеммам M1 и M2.

Каждый раз проверяется совместная работа ключевого транзистора нижнего плеча одной половины моста и ключевого транзистора верхнего плеча другой половины моста. При этом все логические входы должны быть **отключены** от других цепей. Например, на входы В и С подаются следующие сигналы:

- Когда на входы В и С подаётся ВЫСОКИЙ логический уровень +5 В (Q5 и Q10 открыты), на нагрузке, подключенной к четырёхконтактной клеммной колодке, должно быть примерно 1 В на выводе M1 и (Vm – 1) В на выводе M2.

То же касается остальных транзисторов моста, управляемых входами А и D. Если эта цепь исправна, то полярность выходного напряжения должна поменять-

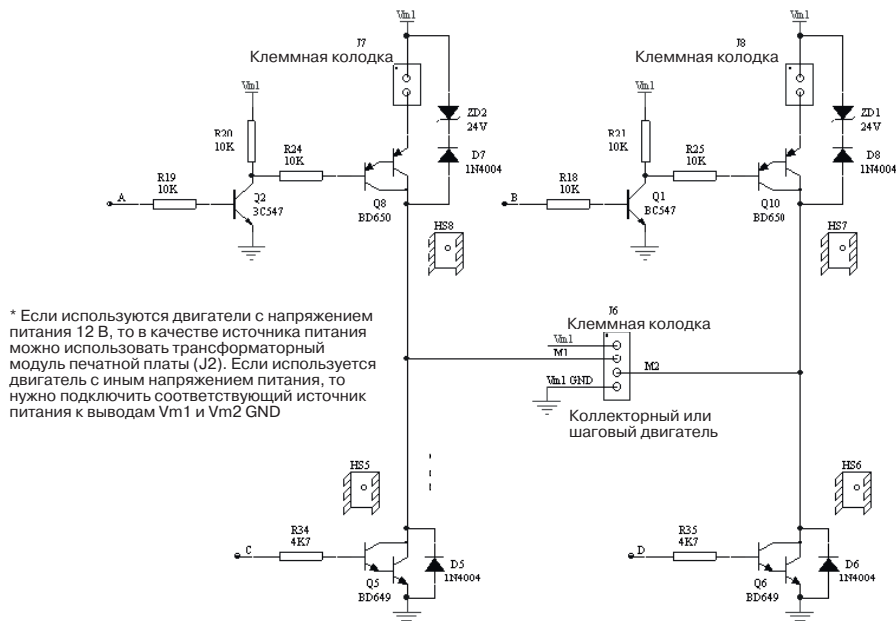
ся на противоположную, на М1 примерно ($V_m - 1$) В, а на М2 примерно 1 В. Если вышеприведённые условия не выполняются, то нужно проверить:

- Исправность источника питания, надёжность соединений, правильность подключений, нет ли коротких замыканий и разрывов, номиналы и типы компонентов, их исправность и правильность установки.

Следует отметить, что транзисторы Дарлингтона BD649 и BD650, используемые в качестве ключей моста, допускают коммутацию токов до 8 А. Вместо них можно использовать другие транзисторы с аналогичным максимальным током и расположением выводов. Последовательно с обычным диодом, подключенным к транзистору BD650, включен ещё и стабилитрон на 24 В, он нужен для гашения импульсов ЭДС самоиндукции, которые возникают в момент выключения тока. Диод у транзистора BD649 ограничивает отрицательное напряжение на нём величиной прямого падения напряжения на диоде, примерно 0.7 В.

Таблица А.6. Схема управления двигателями – перечень элементов

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
4	Резистор 4.7 кОм	0.25 Вт	R34, R35, R51, R53
12	Резистор 10 кОм	0.25 Вт	R18–R21, R24, R25, R45–R50
8	Диод 1N4004		D5–D8, D11–D14
4	Стабилитрон 24 В	1 Вт	ZD1, ZD2, ZD4, ZD5
4	Транзистор типа <i>n-p-n</i> BC547	ТО-92	Q1, Q2, Q13, Q14
4	Транзистор Дарлингтона типа <i>n-p-n</i> BD649 (или аналогичный)	ТО-220	Q5, Q6, Q15, Q16
4	Транзистор Дарлингтона типа <i>p-n-p</i> BD650 (или аналогичный)	ТО-220	Q8, Q10, Q17, Q18
4	Двухконтактная клеммная колодка	Шаг 5 мм	J7, J8, J12, J13
2	Четырёхконтактная клеммная колодка	Шаг 5 мм	J6, J11
8	Радиатор, тепловое сопротивление 20 °С/Вт		HS5–HS12
8	Штыревой контакт, диаметр 0.9...1.0 мм		
8	Винт М3, длина 6...10 мм		
8	Гайка М3		
8	Гровер М3		



* При использовании коллекторных двигателей на клеммные колодки нужно установить проволочные перемычки

* При использовании шаговых биполярных двигателей к клеммным колодкам нужно подключить резистор Rm

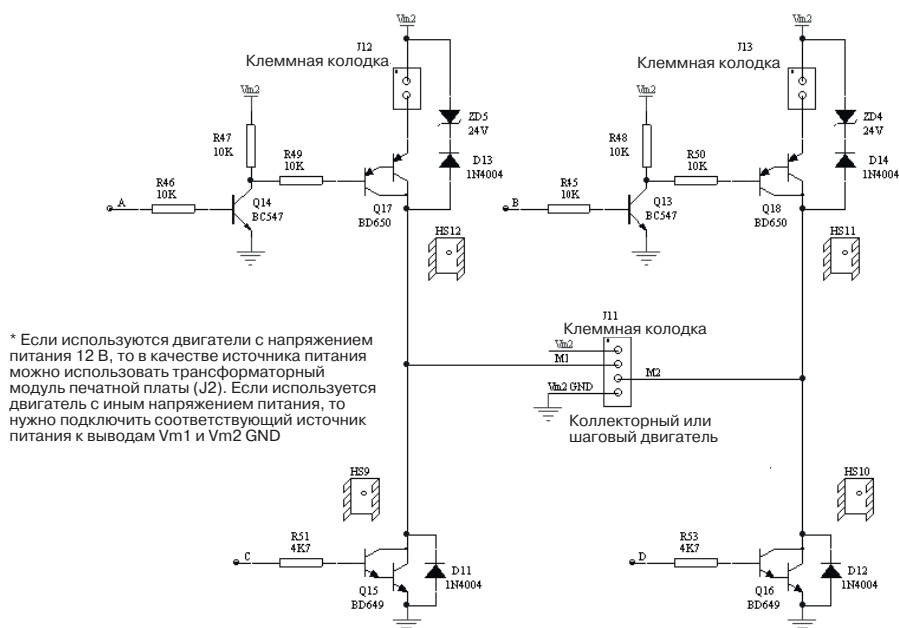


Рис. А.19. Принципиальная схема блока управления двигателями

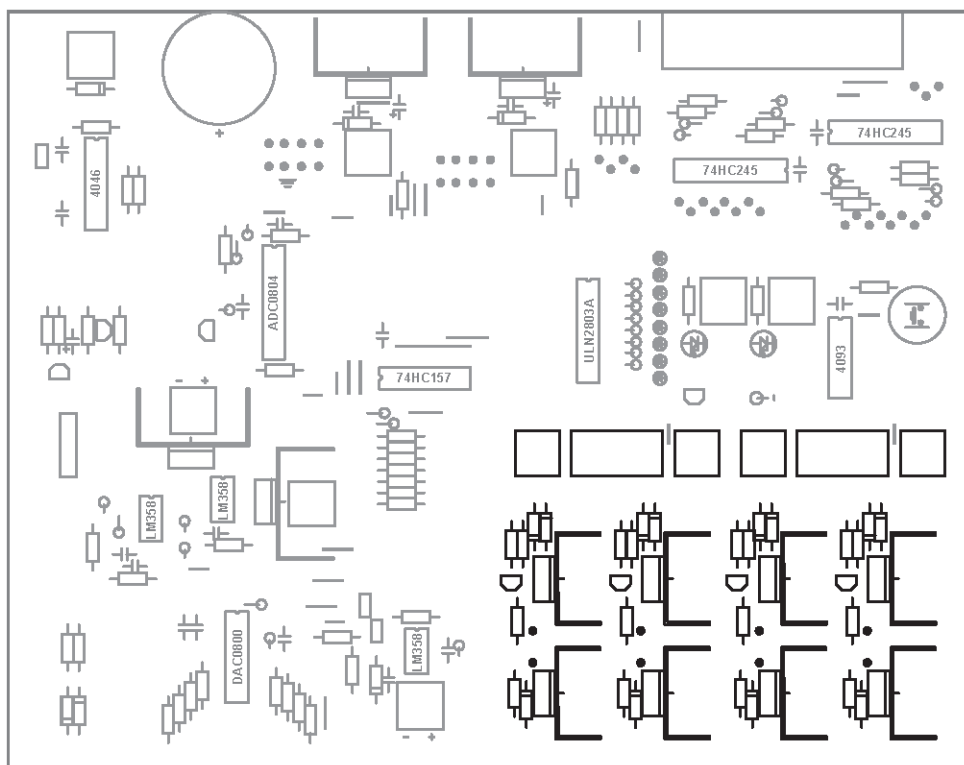


Рис. А.20. Расположение элементов схемы управления двигателями на печатной плате

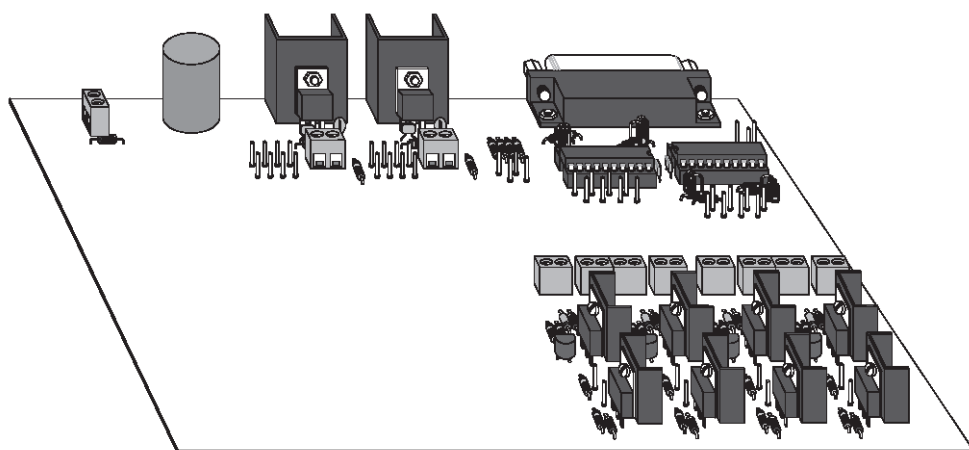


Рис. А.21. Печатная плата с установленными элементами схемы управления двигателями

Управляемый напряжением генератор импульсов

В некоторых проектах книги вместе с ГУН используется термистор, поэтому подключение термистора будет тоже рассмотрено здесь.

Сборка. Нужно установить и припаять компоненты, перечисленные в Табл. А.7, согласно обозначениям на печатной плате, Рис. А.23 и Рис. А.24.

Наладка ГУН. На Рис. А.22 приведена схема генератора, управляемого напряжением (собранный с использованием узлов микросхемы ФАПЧ). На выходе генератора формируются прямоугольные импульсы, частота которых пропорциональна входному напряжению.

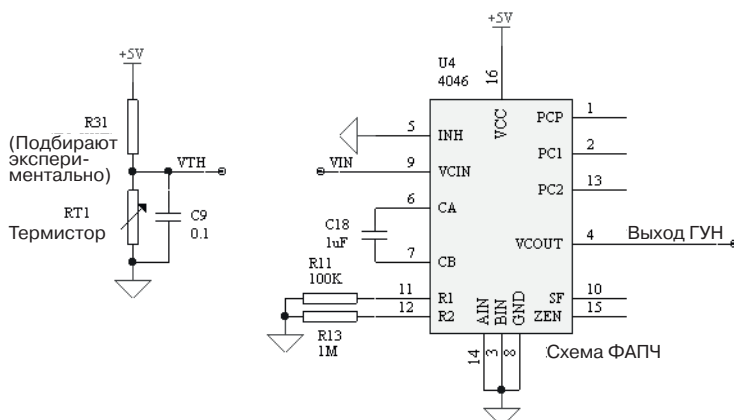


Рис. А.22. Принципиальная схема ГУН и цепи термистора

Напряжение питания микросхемы ФАПЧ U4 на выводе 16 должно составлять примерно +5 В, то есть столько же, сколько на выходе стабилизатора +5 В. Если это условие не выполняется, то причина может быть следующей:

- Неправильная установка микросхемы, короткие замыкания и разрывы, отсутствие контакта в колодке микросхемы, неисправная микросхема 4046, неисправность источника питания +5 В.

Вход ГУН VIN (вывод 9) нужно подключить к земле, а выход (вывод 4) на один из входов драйвера светодиодов. При этом на выходе ГУН будет цифровой сигнал низкой частоты, заставляющий мигать светодиод. Подключение входа к шине питания +5 В должно привести к значительному увеличению частоты. Препятствовать этому могут следующие причины:

- Короткое замыкание или разрыв цепи, несоответствующие номиналы резисторов и конденсаторов, неисправность компонентов, неправильная установка микросхемы, некачественная пайка.

Наладка цепи термистора. Термистор включен по очень простой схеме делителя напряжения, между выходом делителя и землёй включен конденсатор. Конденсатор нужен для подавления высокочастотных помех от источника питания +5 В.

На противоположном от вывода подключения конденсатора С9 выводе резистора R31 нужно проверить наличие напряжения +5 В. Если это условие не выполняется, то могут быть следующие причины:

- Неисправность источника +5 В, короткие замыкания и разрывы.

Сопротивление термисторов меняется с температурой, выходное напряжение VTH при этом тоже меняется. Корпус термистора нужно нагреть пальцами рук, сопротивление его поменяется, значит и выходной сигнал тоже изменится. Если этого не произошло, то возможные следующие причины:

- Неправильно выбранное сопротивление резистора R31, некачественная пайка компонентов, короткие замыкания и разрывы, неисправный резистор R31, неисправные термистор или конденсатор.

Таблица А.7. Схема управляемого напряжением генератора импульсов – перечень элементов

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
Схема ГУН			
1	Керамический конденсатор 1 мкФ	0.2 дюйма	C18
1	Резистор 100 кОм	0.25 Вт	R11
1	Резистор 1 МОм	0.25 Вт	R13
1	Микросхема 4046 (КМОП)	DIL16	U4
1	Панелька для микросхемы	16 контактов	
2	Штыревой контакт, диаметр 0.9...1.0 мм		
Цепь термистора			
1	Керамический конденсатор 0.1 мкФ	0.2 дюйма	C9
1	Резистор 100...470 кОм (величина зависит от термистора, глава 10)		R31
1	Термистор		RT1

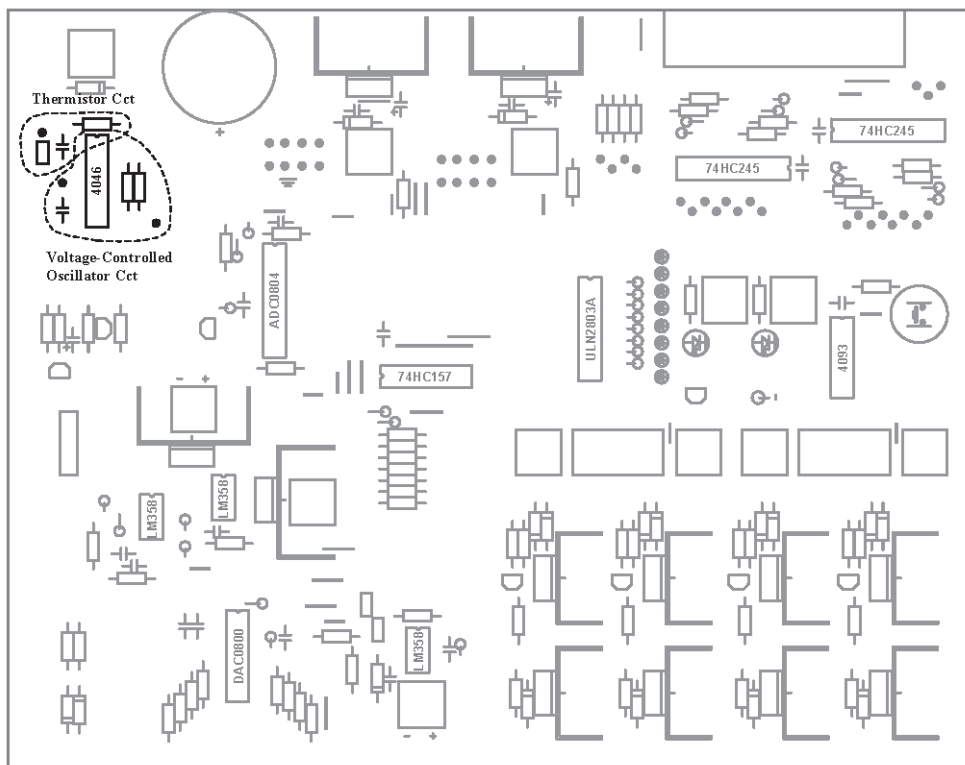


Рис. А.23. Расположение элементов ГУН с термистором на печатной плате

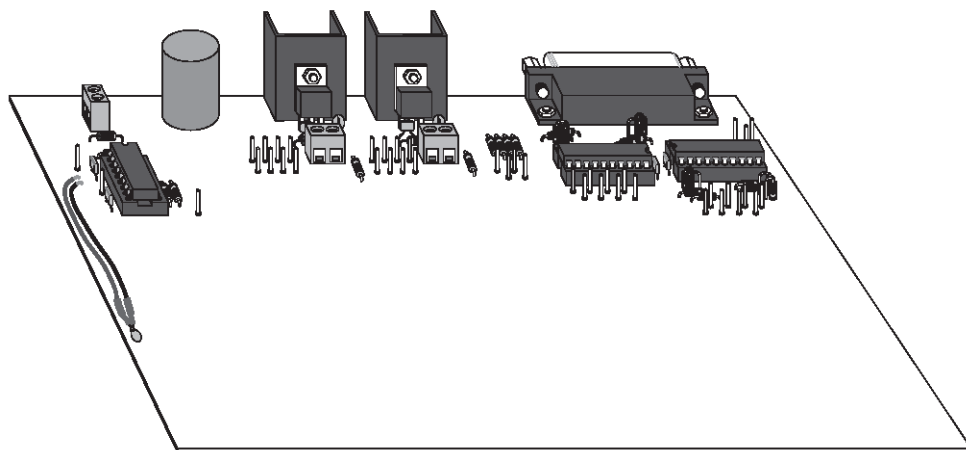


Рис. А.24. Печатная плата с установленными элементами ГУН и термистором

Аналого-цифровой преобразователь

Сборка. Нужно установить и припаять компоненты, перечисленные в Табл. А.8, согласно обозначениям на печатной плате, Рис. А.26 и Рис. А.27.

Наладка. На Рис. А.25 показана схема блока аналого-цифрового преобразования.

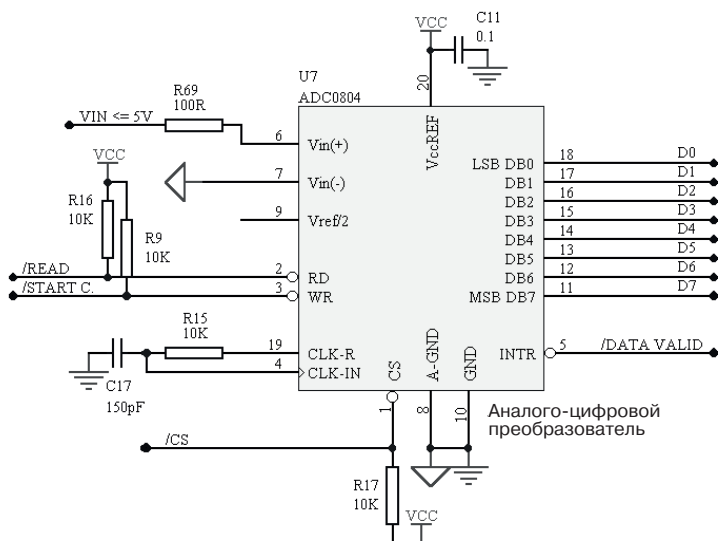


Рис. А.25. Принципиальная схема аналого-цифрового преобразователя

Аналого-цифровой преобразователь выполняет преобразование входного напряжения в 0...+5 В на входе VIN в целое число из диапазона 0...255 при помощи микросхемы АЦП ADC0804.

Напряжение питания микросхемы ADC0804 на выводе 20 должно быть +5 В. Если это не так, то нужно проверить следующее:

- Исправность источника питания +5 В, правильность установки микросхемы, нет ли коротких замыканий или разрывов, есть ли контакт в панельке микросхемы и исправность микросхемы ADC0804.

Если напряжение питания микросхемы составляет +5 В, но микросхема при этом горячая, то это может быть проявлением так называемого *тиристорного эффекта*, которым страдают некоторые микросхемы КМОП. Такой эффект выражен во внутреннем замыкании шины питания на землю. Проблему можно решить лишь отключив питание платы, затем нужно дать микросхеме остыть и только потом включить питание, контролируя при этом её температуру.

Три цифровых входа микросхемы ADC0804 (/READ, /START C. и /CS) должны быть подключены к шине +5 В через подтягивающие резисторы R16, R9 и R17. Если на этих входах нет ВЫСОКОГО логического уровня +5 В, то это может быть вызвано следующими причинами:

- Некачественная пайка компонентов, короткозамкнутые и разорванные цепи, неисправные резисторы, неправильная установка микросхемы или её неисправность.

Проверка АЦП завершается подключением сигналов /CS и /READ к земле, а к линии сигнала /START С. подключается соединительный проводник, другой конец которого оставляется неподключенным.

Выходные цифровые сигналы (**D0...D7**) АЦП подключаются к драйверу светодиодов для визуального отображения их состояния.

Аналоговый вход АЦП VIN нужно подключить к земле, затем на короткое время дотронуться проводом сигнала /START С. к земле, запустив тем самым процесс преобразования. Результат АЦП должен быть близок к 0, или в двоичной форме 0000 0000.

Те же действия нужно повторить для VIN, подключенного к шине питания +5 В, при этом на выходе должно быть значение, близкое к 255, или в двоичной форме 1111 1111.

Старшим значащим битом в ADC0804 является бит **D7** (вывод 11). Пока входное напряжение превышает 2.5 В, этот бит находится в ВЫСОКОМ состоянии.

Если вышеперечисленные тесты не проходят, то это может быть вследствие следующих причин:

- Неправильная установка микросхемы, некачественная пайка компонентов, короткие замыкания и разрывы, неисправные компоненты.

Таблица А.8. Схема управляемого напряжением генератора импульсов – перечень элементов

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
1	Керамический конденсатор 150 пФ	0.2 дюйма	C17
1	Керамический конденсатор 0.1 мкФ	0.2 дюйма	C11
1	Резистор 100 Ом	0.25 Вт	R69
4	Резистор 10 кОм	0.25 Вт	R9, R15–R17
1	Микросхема ADC0804 (КМОП)	DIL20	U7
1	Панелька для микросхемы	20 контактов	
13	Штыревой контакт, диаметр 0.9...1.0 мм		

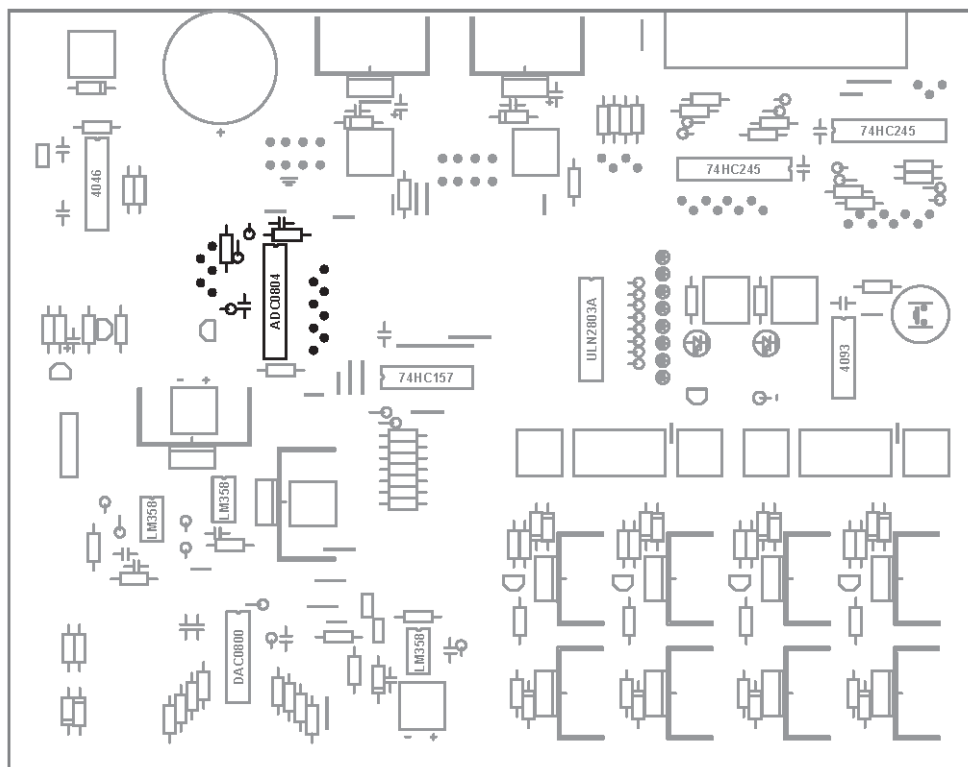


Рис. А.26. Расположение элементов АЦП на печатной плате

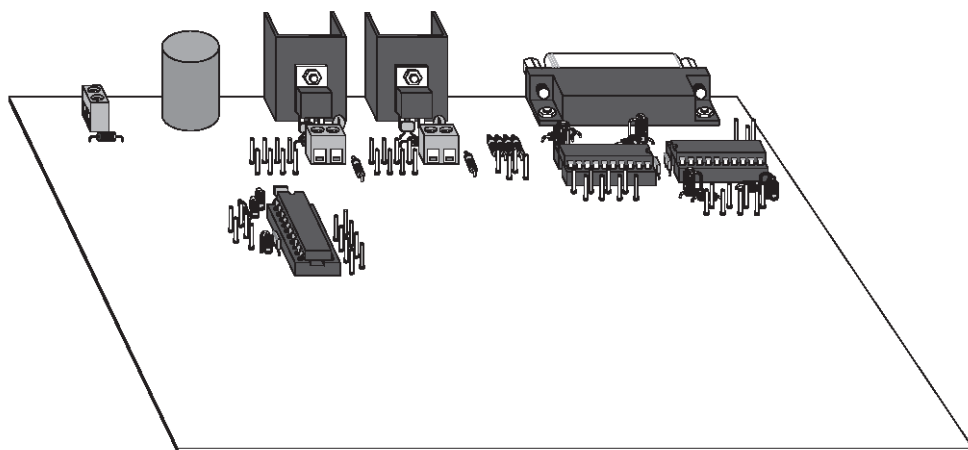


Рис. А.27. Печатная плата с установленными элементами АЦП

Мультиплексор

Сборка. Нужно установить и припаять компоненты, перечисленные в Табл. А.9, согласно обозначениям на печатной плате, Рис. А.29 и Рис. А.30.

Наладка. На Рис. А.28 показана принципиальная схема блока мультиплексора. Мультиплексор разбивает входной байт данных на две тетрады, передавая на выход нужную тетраду. Так можно передавать восьмибитные данные четырёхбитными группами, для чего нужны только четыре линии передачи данных. С другой стороны, при такой передаче затрачивается в два раза больше времени, чем при передаче целыми байтами.

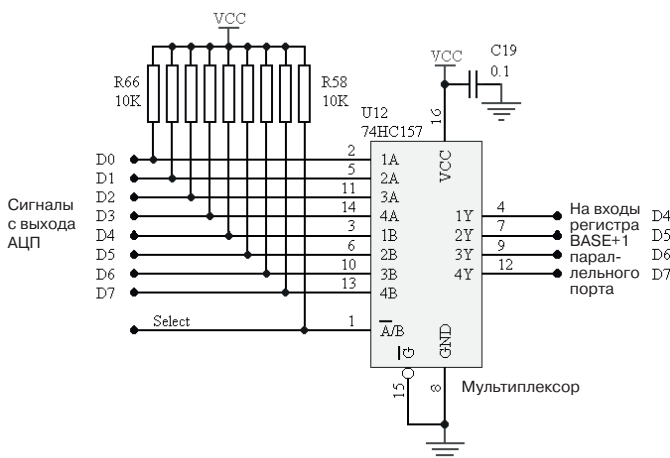


Рис. А.28. Принципиальная схема узла мультиплексора

Входной сигнал Select переключает тетрады 1A...4A и 1B...4B на выходах 1Y...4Y. Если на входе Select НИЗКИЙ уровень, то к выходу подключена входная тетрада A. Если на входе Select ВЫСОКИЙ уровень, то на выходе тетрада B.

Напряжение питания микросхемы мультиплексора 74HC157 на выводе 16 должно составлять примерно +5 В, то есть как на выходе стабилизатора +5 В. Если это условие не выполняется, то возможно следующее:

- Неправильно установлена микросхема, короткие замыкания и разрывы, отсутствие контакта в колодке микросхемы, неисправная микросхема 74HC157.

Первый тест мультиплексора:

1. Все входы и выходы должны быть неподключенными. При этом на всех входах будет ВЫСОКИЙ уровень (в том числе и на входе Select), подаваемый подтягивающими резисторами. Тогда на выходы 1Y...4Y должны подаваться сигналы со входов группы В (1B...4B, или биты данных **D4...D7**). В данном случае на всех выходах должен быть ВЫСОКИЙ уровень.
2. Всё остаётся как и в предыдущем случае, только вход Select подключается к земле. В этом случае на выходы 1Y...4Y будут подаваться сигналы группы А (1A...4A, или биты данных **D0...D3**), на всех выходах должен быть ВЫСОКИЙ уровень.

Если в вышеприведённом тесте выявлены отклонения выходных сигналов от указанных значений, то нужно проверить уровни сигнала на входах (входы должны оставаться неподключенными). В случае отсутствия уровней +5 В на входах нужно убедиться:

- В отсутствии разрывов в цепях подтягивающих резисторов (при отключенном напряжении питания). Если сопротивление цепи окажется больше номинального 10 кОм, то нужно проверить цепь на предмет разрывов и некачественной пайки.
- Правильно ли установлена микросхема, нет ли коротких замыканий и разрывов, есть ли контакт в панельке микросхемы, исправна ли микросхема 74НС157.

Второй тест мультиплексора:

1. Входы группы А (1А...4А) при помощи соединительных проводников нужно подключить к земле. При этом входы группы В (1В...4В) должны оставаться неподключенными. Вход Select подключается к земле. Тогда входы группы А будут подключены к выходным линиям (вывод битов **D0...D3**). Таким образом на всех выходах должен быть **НИЗКИЙ** уровень. Если этого не происходит, то возможны следующие причины:
 - Неправильная установка микросхемы, плохой контакт, короткие замыкания и разрывы, отсутствие контакта в панельке микросхемы и неисправная микросхема 74НС157
2. На вход Select подать **ВЫСОКИЙ** уровень (для чего достаточно только отключить его от земли). Тогда к выходам будут подключены входы группы В (1В...4В, вывод битов **D4...D7**). Теперь на выходах должен появиться **ВЫСОКИЙ** уровень. Если на входы 1В...4В подать **НИЗКИЙ** уровень, то на выходе тоже должен быть **НИЗКИЙ** уровень. Успешному выполнению теста могут препятствовать следующие причины:
 - Неправильная установка микросхемы, плохой контакт, короткие замыкания и разрывы, отсутствие контакта в панельке микросхемы, неисправная микросхема 74НС157.

Таблица А.9. Схема мультиплексора – перечень элементов

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
1	Керамический конденсатор 0.1 мкФ	0.2 дюйма	C19
9	Резистор 10 кОм	0.25 Вт	R58–R66
1	Микросхема 74НС157 (КМОП)	DIL16	U12
1	Панелька для микросхемы	16 контактов	
13	Штыревой контакт, диаметр 0.9...1.0 мм		

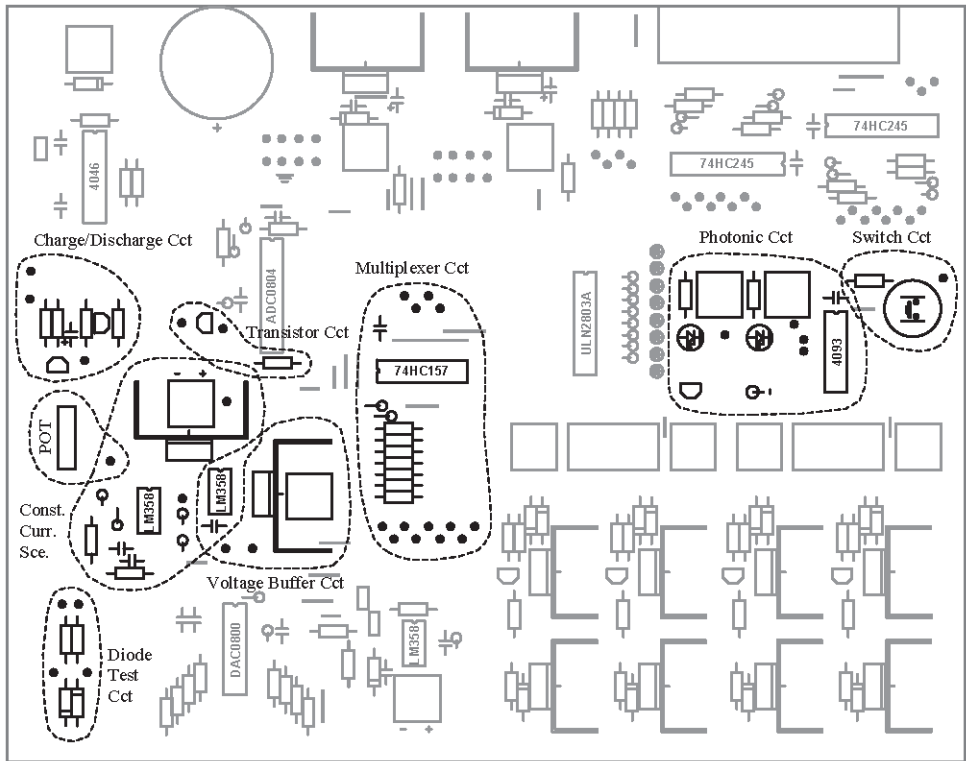


Рис. А.29. Расположение элементов различных устройств на печатной плате

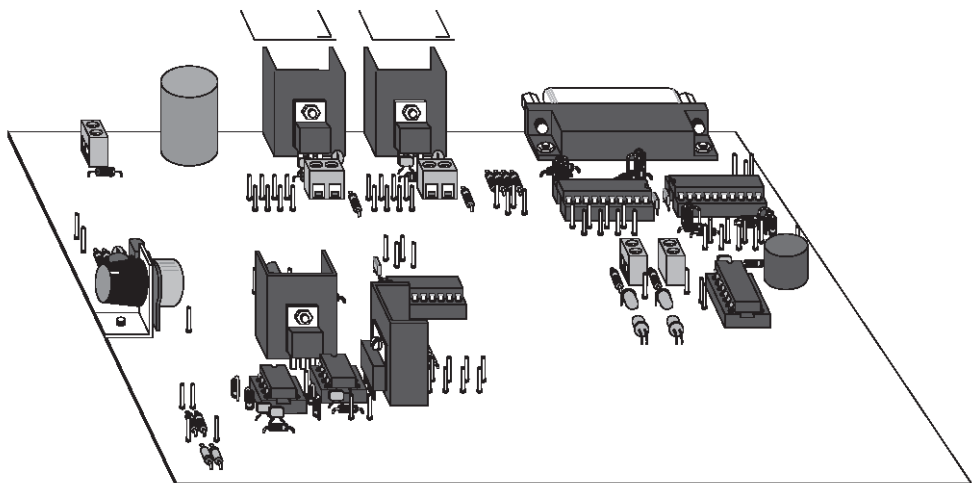


Рис. А.30. Печатная плата с установленными элементами устройств

Управляемый источник тока

Сборка. Нужно установить и припаять компоненты, перечисленные в Табл. А.10, согласно обозначениям на печатной плате, Рис. А.29 и Рис. А.30.

Таблица А.10. Схема управляемого источника тока – перечень элементов

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
3	Керамический конденсатор 0.1 мкФ	0.2 дюйма	C4, C10, C23
1	Резистор 3 кОм	0.25 Вт	R67
4	Резистор 10 кОм	0.25 Вт	R28, R55–R57
1	Резистор 12 кОм	0.25 Вт	R68
1	Транзистор Дарлингтона типа <i>p-n-p</i> BD650 (или аналогичный)	TO-220	Q9
2	Микросхема LM358	DIL8	U9, U11
2	Панелька для микросхемы	8 контактов	
2	Двухконтактная клеммная колодка	Шаг 5 мм	J4, J16
1	Радиатор, тепловое сопротивление 12 °C/Вт		HS4
2	Штыревой контакт, диаметр 0.9...1.0 мм		
1	Винт М3, длина 6...10 мм		
1	Гайка М3		
1	Гровер М3		

Наладка. На Рис. А.31 показана принципиальная схема управляемого источника тока. Управляемые источники тока могут использоваться, например, для заряда никель-кадмиевых аккумуляторов, в контрольно-измерительной аппаратуре. Источник формирует в нагрузке ток, величина которого пропорциональна входному напряжению 0...+5 В, поступающего с выхода ЦАП VDAC. Диапазон выходного тока задаётся внешним резистором R_{CUR} , который подключается посредством клеммной колодки J16.

Регулируемый источник тока состоит из трёх секций, выполненных на операционных усилителях, каждая из которых выполняет определённую функцию. В первой секции на усилителе U11B происходит умножение величины сигнала на $\frac{4}{5}$ при помощи делителя напряжения на резисторах R67 и R68. Тогда при максимальном входном сигнале +5 В на выходе делителя будет +4 В. Операционный U11B усилитель служит буфером, выходной сигнал которого подключен ко второй секции схемы (U11A).

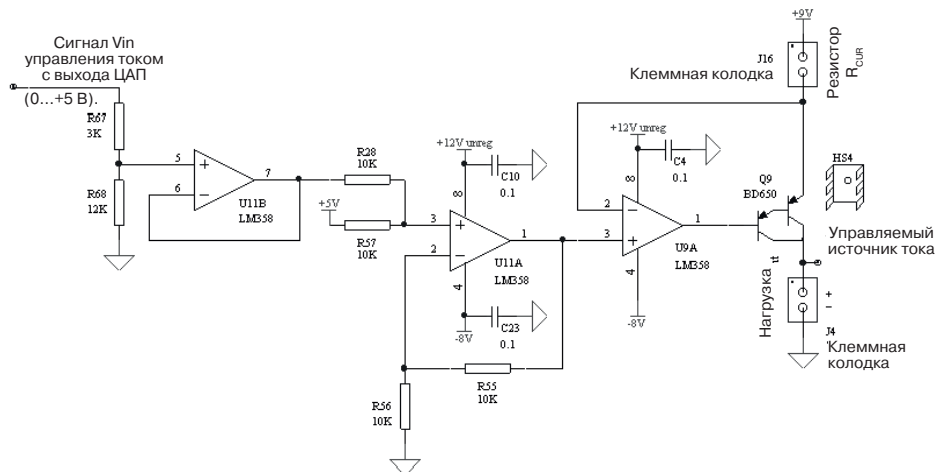


Рис. А.31. Принципиальная схема управляемого источника тока

Вторая секция представляет собой *неинвертирующий усилитель*, усиливающий входной сигнал на неинвертирующем входе в два раза. В неинвертирующем усилителе коэффициент усиления находится по формуле $1 + R_F/R$, где R_F сопротивление резистора обратной связи, а R сопротивление подключенного к земле резистора. Сигнал на неинвертирующий вход тоже подаётся через делитель напряжения ($R28$ и $R57$). Когда выходное напряжение первой секции $U11B$ (вывод 7) максимально и равно 4 В, на неинвертирующем входе $U11A$ будет 4.5 В. После увеличения этого сигнала в 2 раза усилителем $U11A$ на его выходе (вывод 1) будет +9 В.

Последняя секция представляет собой источник тока, управляемый напряжением, собранный на операционном усилителе $U9A$. На его вход подаётся напряжение с выхода $U11A$ и управляет током через транзистор Дарлингтона $Q9$, при этом напряжение с эмиттера транзистора (эмиттер обозначен стрелкой), подаваемое на инвертирующий вход, равно напряжению на неинвертирующем входе. Таким образом, когда на выходе предыдущей секции напряжение +9 В, на выводе клеммной колодки $J16$, подключенном к эмиттеру, тоже должно быть +9 В. Тогда на резисторе R_{CUR} , подключенном к клеммной колодке, падение напряжения составит 0 В, что означает отсутствие тока через этот резистор, через транзистор и через нагрузку, подключенную к клеммной колодке $J4$.

Когда входное напряжение V_{DAC} первой секции будет равно 0 В, на выходе $U11B$ тоже будет 0 В, на выходе второй секции будет +5 В, тогда и на подключенном к эмиттеру выводе клеммной колодки тоже будет +5 В, а падение напряжения на R_{CUR} составит 4 В. Величина тока через этот резистор определяется его сопротивлением. Выбором соответствующего сопротивления R_{CUR} ограничивают максимальный ток источника.

Такой источник тока можно использовать для различных целей, например:

- Виртуальный омметр. В этом случае необходимый ток задаётся резистором R_{CUR} , а измеряемый резистор подключается к клеммной колодке $J4$. Через

измеряемый резистор задаётся ток сигналом VDAC, при этом на резисторе будет падать напряжение, которое можно измерить при помощи АЦП (или ГУН, если известна его передаточная характеристика).

- Устройство зарядки никель-кадмиевых аккумуляторов. К клеммной колодке J4 подключается аккумулятор с соблюдением полярности, а затем выполняется её заряд требуемым током в течение заданного периода времени (или пока аккумулятор не зарядится до требуемого уровня напряжения).

Для проверки транзисторов и диодов (вывод их характеристик на экран). Исследуемый компонент подключается к клеммной колодке J4 с соблюдением полярности. Затем через компонент пропускается изменяемый ток, в то время как происходит измерение напряжения на нём.

Для проверки схемы нужно подключить резисторы к обоим клеммным колодкам, R_{CUR} должно быть 390 Ом, а к J4 подключить резистор 470 Ом.

Сначала проверяют напряжение питания микросхем U11 и U9 на выводе 8. Напряжение питания должно быть примерно +12 В. Напряжение отрицательного источника питания должно быть примерно –8 В на выводе 4 микросхем U11 и U9.

На вход подаются напряжения 0 и +5 В с выхода ЦАП (как это сделать описывалось ранее), при этом контролируются напряжения на входе и на выходе схемы. Следует помнить, что в правильно работающем операционном усилителе напряжения на инвертирующем и неинвертирующем входах равны (с точностью до нескольких микровольт).

Так инвертирующим и неинвертирующим входами ток практически не потребляется, то можно считать, что весь ток резистора R_{CUR} втекает в эмиттер транзистора Q9. При напряжении на входе 0 В на R_{CUR} (390 Ом) будет падать 4 В, что соответствует току примерно 10 мА. Практически весь этот ток будет протекать и через нагрузку, подключенную к клеммной колодке J4 (резистор 470 Ом), вызывая падение напряжения на нём порядка 4.7 В. Незначительный ток ответвляется в транзисторе Дарлингтона Q9 на выход операционного усилителя U9A, этот ток служит для управления напряжением на эмиттерной нагрузке, уравнивая напряжения на инвертирующем и неинвертирующем входах.

Вышеприведённые условия могут не выполняться в силу разных причин:

- Неисправность источников питания +12 В, –8 В (батарейки 9 В), неправильная установка микросхем и транзисторов, несоответствующие типы компонентов, плохая пайка, короткие замыкания и разрывы, отсутствие контакта в панельке микросхемы или неисправные компоненты.

Повторитель напряжения

Сборка. Нужно установить и припаять компоненты, перечисленные в Табл. А.11, согласно обозначениям на печатной плате, Рис. А.29 и Рис. А.30.

Наладка. На Рис. А.32 показана принципиальная схема повторителя напряжения.

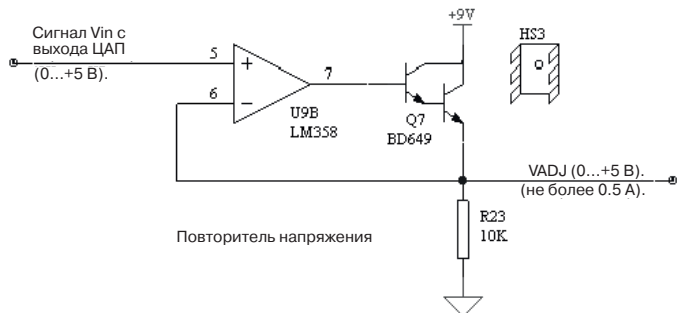


Рис. А.32. Принципиальная схема повторителя напряжения

На вход этой схемы поступает напряжение от слаботочного источника, выходной ток которого не превышает нескольких миллиампер, а на выходе появляется то же по величине напряжения, то выход способен отдавать в нагрузку значительный ток (до 0.5 А).

Таблица А. 11. Повторитель напряжения – перечень элементов

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
1	Резистор 10 кОм	0.25 Вт	R23
1	Транзистор Дарлингтона типа <i>n-p-n</i> BD649 (или аналогичный)	TO-220	Q7
1	Микросхема LM358	DIL8	U9
1	Панелька для микросхемы	8 контактов	
1	Радиатор, тепловое сопротивление 12 °С/Вт		HS3
2	Штыревой контакт, диаметр 0.9...1.0 мм		
1	Винт М3, длина 6...10 мм		
1	Гайка М3		
1	Гровер М3		

В схеме используется операционный усилитель U9B, выходной сигнал которого управляет силовым транзистором Дарлингтона Q7 так, что напряжение на эмиттерной нагрузке транзистора практически равно напряжению на неинвертирующем входе. Обратная связь обеспечивается резистором R23. Операционный усилитель отслеживает изменения входного сигнала, соответствующим образом управляя током транзистора.

Сначала проверяют напряжения питания операционного усилителя. Положительное напряжение питания должно быть примерно равно +12 В на выводе 8,

а отрицательное примерно -8 В на выводе 4. Между выходом VADJ и землёй нужно подключить нагрузку, например резистор сопротивлением 100 Ом. При входном напряжении V_{in} 0 В на выходе VADJ тоже должно быть 0 В. При подаче на вход V_{in} различных напряжений $0...+5$ В на выходе VADJ должны появляться те напряжения. Для такой проверки лучше всего подходит потенциометр, позволяющий регулировать напряжение от 0 до $+5$ В.

Так как для работы операционному усилителю U9 необходим источник -8 В, то в плате должны быть установлены компоненты этого источника, перечисленные в Табл. А.5 и подключена исправная батарейка 9 В.

Причиной неисправностей могут быть:

- Неисправность источников питания $+12$ В и -8 В (батарейки 9 В), неправильная установка микросхемы и транзистора, некачественная пайка, короткие замыкания и разрывы, отсутствие контакта в колодке микросхемы или неисправные компоненты.

Схема заряда/разряда

Сборка. Нужно установить и припаять компоненты, перечисленные в Табл. А.12, согласно обозначениям на печатной плате, Рис. А.28 и Рис. А.29.

Принципиальная схема устройства приведена на Рис. А.33 и служит для демонстрации переходных процессов в RC цепи. Выходной сигнал можно оцифровать при помощи АЦП и вывести осциллограмму сигнала на экран.

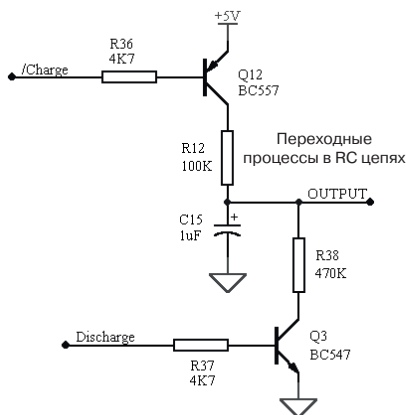


Рис. А.33. Принципиальная схема заряда/разряда RC цепи

Процесс заряда начинается после перевода сигнала /Charge в активный НИЗКИЙ уровень, в то время как сигнал Discharge (разряд) находится в неактивном НИЗКОМ состоянии. При этом происходит постепенное увеличение напряжения на конденсаторе C15. Открывание транзистора Q15 типа $p-n-p$ происходит, когда напряжение на базе транзистора становится меньше напряжения на эмиттере (обозначен стрелкой) примерно на 0.7 В. Ток начинает протекать через резистор R12, заряжая конденсатор C15 до напряжения $+5$ В.

Наладка. Транзистор Q3 типа *n-p-n* управляется сигналом Discharge и начинает проводить ток, когда напряжение на его базе превысит напряжение на эмиттере (подключенном к земле) примерно на 0.7 В.

Разряд происходит при неактивном уровне сигнала /Charge (то есть более 4.3 В, когда транзистор Q12 не проводит ток), а сигнал Discharge в активном ВЫСОКОМ состоянии. Тогда транзистор Q3 начинает проводить ток, позволяя стекать на землю накопленному заряду через резистор R38. Процесс разряда прекратится, когда напряжение на конденсаторе упадёт до нуля. Следует обратить внимание, что сопротивление разрядного резистора R38 примерно в пять раз больше сопротивления зарядного резистора R12, что приводит к разным длительностям заряда/разряда, **Рис. 13.6**.

Сначала нужно проверить напряжение на эмиттере транзистора Q12 (он подключен к широкой печатной дорожке на плате), оно должно составлять +5 В. Если это не так, то причина может быть следующей:

- Неисправен источник +5 В, неправильно установлен транзистор или конденсатор, некачественная пайка, короткие замыкания и разрывы, неисправные компоненты.

Теперь нужно проверить работу схемы, подавая сигналы /Charge и Discharge, как было описано выше. Если не происходит адекватной реакции схемы, то возможно следующее:

- Неправильная установка транзистора или конденсатора, некачественная пайка, короткие замыкания и разрывы, неисправные компоненты, неправильные уровни сигналов /Charge или Discharge, недостаточные для открывания/закрывания транзисторов.

Таблица А.12. Схема заряда/разряда RC цепи – перечень элементов

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
1	Танталовый электролитический конденсатор 1 мкФ \geq 16 В	0.2 дюйма	C15
2	Резистор 4.7 кОм	0.25 Вт	R36, R37
1	Резистор 100 кОм	0.25 Вт	R12
1	Резистор 470 кОм	0.25 Вт	R38
1	Транзистор типа <i>n-p-n</i> BC547	ТО-92	Q3
1	Транзистор типа <i>p-n-p</i> BC557	ТО-92	Q12
3	Штыревой контакт, диаметр 0.9...1.0 мм		

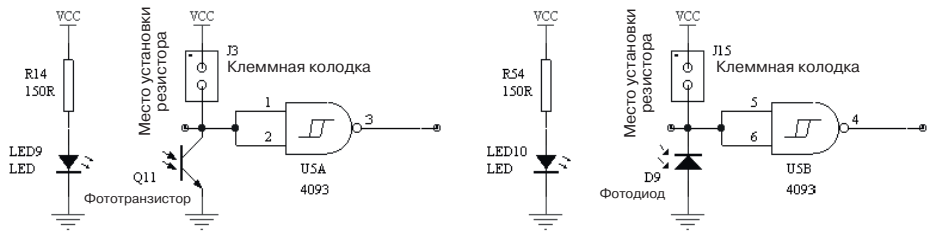
Пара светодиод-фотоприёмник

Сборка. Нужно установить и припаять компоненты, перечисленные в Табл. А.13, согласно обозначениям на печатной плате, Рис. А.29 и Рис. А.30.

Таблица А.13. Пара светодиод-фотоприёмник – перечень элементов

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
1	Керамический конденсатор 0.1 мкФ	0.2 дюйма	C21
2	Резистор 150 Ом	0.25 Вт	R14, R54
2	Светодиод (красный или инфракрасный)		LED9, LED10
1	Фотодиод (должен соответствовать типу светодиода)		D9
1	Фототранзистор (должен соответствовать типу светодиода)		Q11
1	Микросхема 4093 (КМОП)	DIL14	U5
1	Панелька для микросхемы	14 контактов	
2	Двухконтактная клеммная колодка	Шаг 5 мм	J3, J15
4	Штыревой контакт, диаметр 0.9...1.0 мм		

Наладка. На Рис. А.34 приведена принципиальная схема пары светодиод-фотоприёмник (фотоприёмником служат фотодиод или фототранзистор). Такие устройства могут применяться: для измерения освещённости, в качестве бесконтактных датчиков, в качестве датчиков положения/перемещения, а также в качестве датчиков скорости вращения или перемещения.



Оптические датчики скорости вращения вала двигателя и его положения

Рис. А.34. Принципиальная схема оптических датчиков

В клеммные колодки J3 и J15 нужно установить резисторы, задающие рабочие точки фотодатчиков (режимы их работы). Сопротивления этих резисторов нужно определить опытным путём. Фототранзистор можно заменить фотодиодом. Подбирать резисторы нужно в геометрической прогрессии, начиная со 100 Ом и умножая каждый раз сопротивление на 10. Затем уже более мелкими шагами сопротивления выбрать подходящее сопротивление.

Сначала проверяют наличие напряжения питания +5 В на всех четырёх резисторах. Если напряжение не соответствует норме, то это может быть вызвано следующими причинами:

- Неисправен источник питания +5 В, некачественная пайка, короткие замыкания и разрывы, неисправные компоненты.

Затем проверяется чувствительность схемы к свету: на фотоприёмник освещается светодиодом, при затемнении фотоприёмника выходное напряжение на соответствующем выводе должно увеличиться. При этом нужно учитывать, что бумага является прозрачной для инфракрасного излучения. Если реакции на свет не обнаружено, то возможны следующие причины:

- Неправильная установка светодиода или фотодатчика, некачественная пайка, короткие замыкания и разрывы, неисправные компоненты.

Логические элементы И-НЕ служат для преобразования выходного аналогового сигнала фотоприёмников в цифровой. Логические элементы обладают петлёй гистерезиса, что обозначено соответствующим символом. Петля гистерезиса элемента И-НЕ предотвращает случайные изменения его выходного сигнала при незначительных колебаниях уровня освещённости.

Кнопка, потенциометр, диод и стабилитрон

Сборка. Нужно установить и припаять компоненты, перечисленные в Табл. А.14, согласно обозначениям на печатной плате, Рис. А.29 и Рис. А.30.

Наладка. На Рис. А.35 показана принципиальная схема подключения кнопки, потенциометра, диода, стабилитрона и транзистора.

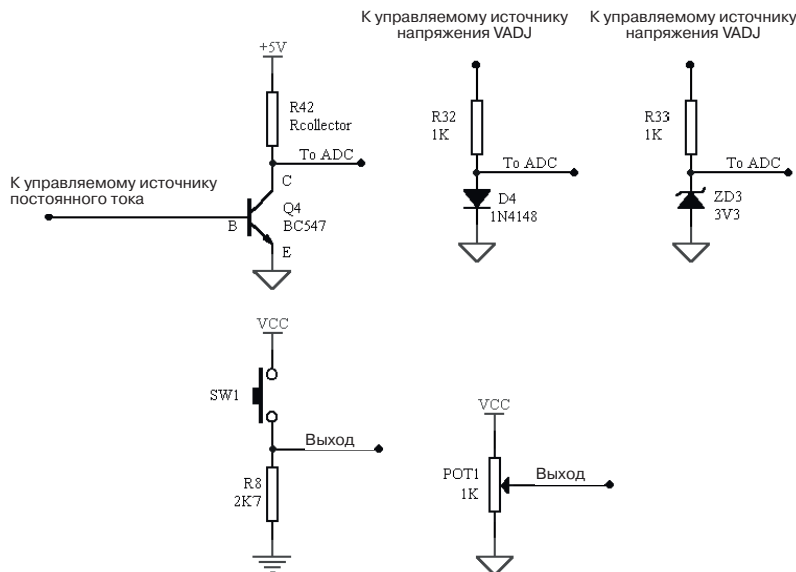


Рис. А.35. Принципиальная схема подключения кнопки, потенциометра, диода, стабилитрона и транзистора

Таблица А.14. Кнопка, потенциометр, диод и стабилитрон – перечень элементов

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
Кнопка			
1	Резистор 2.7 кОм	0.25 Вт	R8
1	Кнопка	Шаг 5 мм	SW1
1	Штыревой контакт, диаметр 0.9...1.0 мм		
Потенциометр			
1	Ручка на вал потенциометра	0.2 Вт	POT1
1	Потенциометр 1 кОм, 16 мм		
1	Штыревой контакт, диаметр 0.9...1.0 мм		
1	Прямоугольный кронштейн потенциометра		
Полупроводниковые приборы			
2	Резистор 1 кОм	0.25 Вт	R32, R33
1	Резистор, подбирается опытным путём из диапазона от 100 Ом до 10 кОм	0.25 Вт	R42
1	Диод 1N4148		D4
1	Стабилитрон 3.3 В	1 Вт	ZD3
1	Транзистор типа <i>n-p-n</i> BC547	TO-92	Q4
6	Штыревой контакт, диаметр 0.9...1.0 мм		

Кнопка. Когда кнопка не нажата на выходе должен быть НИЗКИЙ уровень, и наоборот, если кнопка нажата, то на выходе должен быть ВЫСОКИЙ уровень. Если кнопка не работает, как описано выше, то возможны следующие причины:

- Некачественная пайка, короткие замыкания и разрывы, неправильная установка кнопки, неисправные компоненты.

Если кнопка будет подключаться ко входам микросхем ТТЛ, то нужны небольшие изменения. Входной ток микросхем ТТЛ много больше, чем у схем КМОП. Если кнопку нужно подключить ко входу ТТЛ, то резистор R8 нужно уменьшить до 330 Ом.

Потенциометр. Сначала нужно убедиться, что подвижный контакт потенциометра не подключен к другим цепям. При повороте ручки потенциометра от одного крайнего положения до другого выходное напряжение должно измениться от 0 до +5 В.

Полупроводниковые приборы. Они предназначены для изучения характеристик диодов, стабилитронов и биполярных транзисторов.

На диод D4 подаётся или управляемое напряжение, или управляемый ток, тогда как напряжение на аноде измеряется: так снимается вольт-амперная характеристика диода.

На стабилитрон ZD3 тоже подаются регулируемые напряжение или ток при измерении напряжения на аноде для получения его вольт-амперной характеристики.

На биполярный транзистор Q4 подаётся регулируемый ток 0...1 мА от управляемого источника тока при подключенном резисторе R_{CUR} 3.9 кОм. При этом снимается характеристика прибора путём измерения напряжений на базе (В) и коллекторе (С). Ток можно задавать и от источника напряжения, тогда в цепь между источником и базой транзистора нужно включить резистор.

Из панельки для микросхемы можно изготовить две клеммы для подключения резистора, чтобы облегчить процесс подбора резистора R42.

Следует отметить, что на вход управляемого источника тока можно подключить выход Р0Т1 потенциометра.

Перечень элементов и материалов интерфейсной платы

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
1	Керамический конденсатор 150 пФ	0.2 дюйма	C17
1	Керамический конденсатор 10 нФ	0.2 дюйма	C16
14	Керамический конденсатор 0.1 мкФ	0.2 дюйма	C2–C11, C19–C23
1	Керамический конденсатор 1 мкФ	0.2 дюйма	C18
3	Танталовый электролитический конденсатор 1 мкФ \geq 16 В	0.2 дюйма	C13–C15
1	Электролитический конденсатор 4700 мкФ \geq 16 В	0.5 или 0.35 дюйма	C1
1	Резистор 100 Ом	0.25 Вт	R69
2	Резистор 150 Ом	0.25 Вт	R14, R54
8	Резистор 330 Ом	0.25 Вт	R1–R10
4	Резистор 470 Ом	0.25 Вт	R29, R30, R43, R44
3	Резистор 1 кОм	0.25 Вт	R32, R33, R94
1	Резистор 1.8 кОм	0.25 Вт	R52
1	Резистор 2.7 кОм	0.25 Вт	R8
1	Резистор 3 кОм	0.25 Вт	R67
6	Резистор 4.7 кОм	0.25 Вт	R34–R37, R51, R53

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
59	Резистор 10 кОм	0.25 Вт	R9, R15–R17, R18–R21, R23–R28, R39–R41, R45–R50, R55–R66, R70–R93
1	Резистор 12 кОм	0.25 Вт	R68
1	Резистор 20 кОм	0.25 Вт	R22
2	Резистор 100 кОм	0.25 Вт	R11, R12
1	Резистор 470 кОм	0.25 Вт	R38
1	Резистор 1 МОм	0.25 Вт	R13
1	Резистор 100...470 кОм (для термистора)		R31
1	Резистор от 100 Ом до 10 кОм	0.25 Вт	R42
1	Потенциометр 1 кОм, 16 мм	0.2 Вт	POT1
1	Ручка на вал потенциометра		
1	Термистор		RT1
1	Диод 1N4148		D4
12	Диод 1N4004		D1–D3, D5–D8, D10–D14
1	Стабилитрон 3.3 В	1 Вт	ZD3
4	Стабилитрон 24 В	1 Вт	ZD1, ZD2, ZD4, ZD5
8	Красный светодиод, диаметр корпуса 3 мм		LED1–LED8
2	Светодиод (красный или инфракрасный)		LED9, LED10
1	Фотодиод (должен соответствовать типу светодиода)		D9
1	Фототранзистор (должен соответствовать типу светодиода)		Q11
6	Транзистор типа <i>n-p-n</i> BC547	TO-92	Q1–Q4, Q13, Q14
1	Транзистор типа <i>p-n-p</i> BC557	TO-92	Q12
5	Транзистор Дарлингтона типа <i>n-p-n</i> BC649 (или аналогичный)	TO-220	Q5–Q7, Q15, Q16
5	Транзистор Дарлингтона типа <i>p-n-p</i> BC650 (или аналогичный)	TO-220	Q8–Q10, Q17, Q18
1	Микросхема 4046 (КМОП)	DIL16	U4
1	Микросхема 4093 (КМОП)	DIL14	U5

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
1	Микросхема 74HC157 (КМОП)	DIL16	U12
2	Микросхема 74HC245 (КМОП)	DIL20	U6, U13
1	Микросхема DAC0800 (КМОП)	DIL16	U8
3	Микросхема LM358	DIL8	U9, U10, U11
1	Микросхема ADC0804 (КМОП)	DIL20	U7
1	Микросхема LM7805CT	TO-220	U1
1	Микросхема LM7809CT	TO-220	U2
1	Микросхема ULN2803A (набор ключей)	DIL18	U3
3	Панелька для микросхемы	20 контактов	LINK1, LINK2 J2–J4, J7–J10, J12–J16 J6, J11 SW1 J1 HS1–HS4
1	Панелька для микросхемы	18 контактов	
3	Панелька для микросхемы	16 контактов	
2	Панелька для микросхемы	14 контактов	
3	Панелька для микросхемы	8 контактов	
2	Контакты джампера	0.1 дюйма	
1	Джампер (для позиций LINK1 и LINK2)	0.1 дюйма	
12	Двухконтактная клеммная колодка	Шаг 5 мм	
2	Четырёхконтактная клеммная колодка	Шаг 5 мм	
1	Клемма подключения батареи 9 В		
1	Кнопка	Шаг 5 мм	
1	Разъем типа D25 для установки на печатную плату		
1	Кабель DB25M- DB25M, выводы разъёмов соединены один-в-один		
50	Гнездо для штыревого контакта диаметром 0.9...1.0 мм		
111	Штыревой контакт, диаметр 0.9...1.0 мм		
7 м	Монтажный провод		
1 м	Термоусадочная трубка диаметром 2.5...3 мм.		
4	Радиатор, тепловое сопротивление 12 °C/Вт		

Количество	Описание элемента	Расстояние между выводами или тип корпуса	Позиционное обозначение
8	Радиатор, тепловое сопротивление 20 °C/Вт		HS5–HS12
1	Термопаста		
12	Винт М3, длина 6...10 мм, гайка М3, гровер М3		
4	Самоклеющаяся резиновая ножка для платы		
1	Трансформаторный модуль +12 В, 1 А		
1	Прямоугольный кронштейн для потенциометра		

В ПРИЛОЖЕНИЕ

Служебные слова C++

asm	near
auto	new
break	operator
case	private
catch	protected
char	public
class	register
const	return
continue	short
default	signed
delete	sizeof
do	static
double	struct
else	switch
enum	template
far	this
float	throw
for	try
friend	typedef
goto	union
if	unsigned
inline	virtual
int	void
interrupt	volatile
long	while

Приоритет операторов

В нижеприведённой таблице приведены операторы C++ по убыванию приоритета. Наивысший приоритет имеют операторы в первой строке, наименьший в последней. Операторы, перечисленные в одной строке, обладают одинаковым приоритетом. Операторы выполняются слева направо, за исключением строк, отмеченных *. Такие операторы выполняются в обратном порядке. Например, `a = b` выполняется справа налево, то есть значение `b` присваивается `a`.

Таблица В.1. Приоритет операторов

Операторы	
<code>() [] -> :: .</code>	
<code>! ~ + - ++ -- & *</code>	*
<code>sizeof new delete</code>	*
<code>.* -></code>	
<code>* / %</code>	
<code>+ -</code>	
<code><< >></code>	
<code>< <= > >=</code>	
<code>== !=</code>	
<code>&</code>	
<code>^</code>	
<code> </code>	
<code>&&</code>	
<code> </code>	
<code>? :</code>	*
<code>= *= /= %= += -= &= ^= = <<= >>=</code>	*
<code>,</code>	

Символы ASCII

NUL	DLE	SP	0	@	P	`	p
0x00	0x10	0x20	0x30	0x40	0x50	0x60	0x70
SOH	DC1	!	1	A	Q	a	q
0x01	0x11	0x21	0x31	0x41	0x51	0x61	0x71
STX	DC2	"	2	B	R	b	r
0x02	0x12	0x22	0x32	0x42	0x52	0x62	0x72
ETX	DC3	#	3	C	S	c	s
0x03	0x13	0x23	0x33	0x43	0x53	0x63	0x73
EOT	DC4	\$	4	D	T	d	t
0x04	0x14	0x24	0x34	0x44	0x54	0x64	0x74
ENQ	NAK	%	5	E	U	e	u
0x05	0x15	0x25	0x35	0x45	0x55	0x65	0x75
ACK	SYN	&	6	F	V	f	v
0x06	0x16	0x26	0x36	0x46	0x56	0x66	0x76
BEL	ETB	'	7	G	W	g	w
0x07	0x17	0x27	0x37	0x47	0x57	0x67	0x77
BS	CAN	(8	H	X	h	x
0x08	0x18	0x28	0x38	0x48	0x58	0x68	0x78
HT	EM)	9	I	Y	i	y
0x09	0x19	0x29	0x39	0x49	0x59	0x69	0x79
LF	SUB	*	:	J	Z	j	z
0x0A	0x1A	0x2A	0x3A	0x4A	0x5A	0x6A	0x7A
VT	ESC	+	;	K	[k	{
0x0B	0x1B	0x2B	0x3B	0x4B	0x5B	0x6B	0x7B
FF	FS	,	<	L	/	l	
0x0C	0x1C	0x2C	0x3C	0x4C	0x5C	0x6C	0x7C
CR	GS	-	=	M]	m	}
0x0D	0x1D	0x2D	0x3D	0x4D	0x5D	0x6D	0x7D
SO	RS	.	>	N	^	n	~
0x0E	0x1E	0x2E	0x3E	0x4E	0x5E	0x6E	0x7E
SI	US	/	?	O	_	o	DEL
0x0F	0x1F	0x2F	0x3F	0x4F	0x5F	0x6F	0x7F

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Символы

- 137
-- 137
-> 150
! 137
!= 137
? 141
. 149
* 147
& 147
+ 137
++ 137
< 137
<= 137
== 137
> 137
>= 137
~ 137

A

ASCII набор 233

B

BIOS 368
bioskey() 233
break 141

C

catch 173
char 23
conio.h 49
continue 141
cout 18
cprintf() 283

D

define 47
delete 170
dos.h 49
double 24

do-while 138
Dynamic torque 191

E

edit 12
endl 18

F

float 24

G

getch() 49, 232
getche() 232
getmaxx() 289
getmaxy() 289
goto 142
gotoxy() 283
grapherrormsg() 292
graphresult() 292

H

Holding torque 191

I

IDE 11
if 139
Include DirectPersonNameory 16
in-line функции 80
inportb() 49
int 23
iostream.h 17

K

kbhit() 206

L

LIFO 169
linker. См. Редактор связей
LSB 36

M

main 14
make 271
make-файл 271
MSB 36

N

new 170
NPersonNameotepad 12
NTC 275

O

outportb() 50

P

peekb() 233
printf() 49, 51
private 76
protected 76
public 76
Pull-in torque 191

R

Ramped step rate 191
RTL. См. Библиотека времени выполнения
Run-time library 12

S

Source file 12
stdio.h 49
switch-case 142

T

throw 173
try 173

U

unsigned char 23
unsigned int 24

V

VCO 274

W

while 138

X

XOR. См. Исключающее ИЛИ

A

Абстрактность подпрограмм 20
Автоматический деструктор 77
Автоматический конструктор 77, 78
Адрес операнда 147
Аналого-цифровое преобразование 298
Аналого-цифровой преобразователь 298
Аргументы функции 26
АЦП 298
АЦП двойного интегрирования 301
АЦП последовательного приближения 301

Б

Базовый класс 64, 67, 112
Байт 32
Библиотека времени выполнения 12
Библиотечные функции 16
Биполярный выход 105
Бит 33
Бифилярная намотка 186
Блок try 173
Буферирование 107

В

Виртуальная функция 195
Виртуальные функции 63, 64
Включаемый файл. См. Заголовочный файл
Вложенный цикл 137
Возврат каретки 12
Возврат результата по ссылке 327
Вольтамперная характеристика 44

Время выборки 310
Время захвата 307
Время преобразования 299
Время установления выхода 108
Выборка и хранение 307
Вызов функции 19, 22
Выражение возврата 26
Выражение выбора 142
Выражение инициализации 136
Выражение инкремента 137
Выражение условия 139

Г

ГУН 274

Д

Данные-члены 61
Двоичные данные 35
Двоичный формат 31
Двумерный массив 144
Декодирование адреса 315
Деструктор 62
Динамическое связывание. См. Позднее
связывание
Директивы препроцессора 12
Драйвер 288
Дружественные функции 327

З

Зависимости файлов 272
Заголовочный файл 16
Запас помехоустойчивости 30, 31

И

Идентификатор 18, 24
Иерархия 68
Иерархия классов 64, 66
Импульсное питание 190
Инвертор 107
Индекс массива 145
Инициализация 25
Инициализация массива 145
Инкапсуляция 62

Инстанцирование 78
Инстанцирование объекта 63
Интегральные схемы 31
Интегрированная среда разработки 11
Интегрирующие АЦП 301
Интерфейс 60
Исключающее ИЛИ 53
Исполняемый файл 14
Истина 136
Истинный блок 139
Исходный код 12
Исходный файл 12

К

Каталог заголовочных файлов 16
Квантование 299
Кварцевые кристаллы 275
Класс объекта 59, 61
Командная строка. См.
Комментарий 15
Компилирование 12
Компилятор 11, 12
Константная функция 332
Константный указатель 150
Конструктор 62, 77
Контекст вызова 28
Крутящий момент работающего
двигателя 190
Куча 28, 169

Л

Ловушка 173
Логические уровни 30, 31
Ложный блок 139
Ложь 136
Лужение 392

М

Макрос 168
Массив 19, 143
Машинный код 12
Микрошаговый метод 190
Младший значащий бит 36

Множественное использование кода 64, 69
Многофайловая программа 255
Момент выключенного двигателя 190
Момент выпадения из синхронизма 190
Момент потери фазы 190
Монотонная ошибка 300

Н

Наследование 64, 67, 71
Неинвертирующий усилитель 422
Нелинейность 108
Неопределённые ссылки 13

О

Область данных 28, 169
Область кода 28, 169
Обработка исключений 173
Обратная связь 204
Объектный файл 12
Объявление функции 21
Однополярный выход 105
Оператор доступа к членам 149
Оператор точка. См. Оператор доступа к членам
Операторы отношения 137
Операторы равенства 137
Определение класса 61
Определение функции 21
Остаточная поляризация диэлектрика 302
Отрицательная обратная связь 101
Ошибка дискретизации 307
Ошибка крутизны 108
Ошибка смещения 108

П

Параметры функции 19
Перевод строки 12
Перегрузка операторов 327
Перегрузка функций 78
Передача параметров по значению 330
Передача параметров по ссылке 327, 330
ПНЧ 274
Погрешность диапазона 108

Погрешность квантования 300
Подтягивающие резисторы 400
Позднее связывание 64, 193, 195
Полиморфизм 66, 124
Положительная обратная связь 101
Полупроводниковые диоды 275
Полуфазный режим 191
Порт просмотра 288
Последовательное соединение 45
Постинкремент 137
Поток 18
Поток ввода/вывода 348
Преинкремент 137
Препроцессор 12
Прерывание от таймера 360
Приведение типов 172
Принципиальная схема 35
Производный класс 64, 67, 112
Пространство ввода/вывода 33
Прототип функции 22
Процедурное программирование 58
Псевдо-код 247
Публичная функция 79

Р

Разделители 13
Разыменованное 147
Раннее вызывание 195
Расширенный набор 233
Регистр-защёлка 361
Регистр состояния 361
Регистр управления 362
Регистры 33
Редактор 11
Редактор связей 11, 14
Режим полных фаз 191
Резольвер 182

С

Свободная память 169. См. куча
Сегмент и смещение 234
Семейства логических схем 31
Семейство логических схем 30

Сигнал «выбор микросхемы» 315
Сигнал «запуск преобразования» 315
Сигнал «чтение» 315
Синтаксис 13
Скалярная величина 19
Скважность 181, 203
Скорость нарастания 307
Смещение 45
Составной оператор 135
Спецификатор доступа 113, 122, 123
Среда разработки программ 11
Ссылка 330
Ссылка на адрес. См. Разыменование
Старший значащий бит 36
Статическое связывание 195
Стек 169
Строб-импульс 366
Схема выборки и хранения 309
Схема с последовательным
сопротивлением 189
Счётчик 360

Т

Таймер 8254 359
Тактовый сигнал 359
Тело функции 17
Термисторы 275
Термопары 275
Термочувствительные конденсаторы 275
Тернарный оператор 141
Тетрада 37
Тип возвращаемого значения 18
Тип объекта 58, 61
Типы данных 23
Типы данных базовые 23
Тиристорный эффект 415
ТКС 275
Транзитные объекты 348
Третье состояние 305
Трёхмерный массив 144

У

Указатель 146
Унарные операторы 137
Условие цикла 136
Утечка памяти 170
Утилита make. См. make

Ф

Фаза синхронной работы 190
Файл проекта 271
Фактические аргументы 19
Флаг 294
Флэш-АЦП 301
Формальные аргументы 19
Функции-члены 61

Ц

Цифровые логические схемы 31

Ч

Чип 31
Чистые виртуальные функции 63, 196

Ш

Шаговые двигатели биполярные 191
Шаговые двигатели униполярные 191
Шестнадцатеричный формат 36
Шифратор 182

Э

ЭДС самоиндукции 182
Эквивалентная выборка 307, 312
Экземпляр класса 84
Элемент массива 143
Энкодер 182

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.**

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.aliants-kniga.ru.**

Оптовые закупки: **тел. +7 (499) 782-38-89.**

Электронный адрес: **books@aliants-kniga.ru.**

Янта Катупития (Jayantha Katupitiya)
Ким Бентли (Kim Bentley)

Управление электронными устройствами на C++

Разработка практических приложений

Главный редактор	<i>Мовчан Д. А.</i>
	dmkpress@gmail.com
Перевод с английского	<i>Бакомчев И. В.</i>
Корректор	<i>Синяева Г. И.</i>
Верстка	<i>Паранская Н. В.</i>
Дизайн обложки	<i>Мовчан А. Г.</i>

Формат 70×100 1/16. Гарнитура «Петербург».

Печать офсетная. Усл. печ. л. 35,91.

Тираж 100 экз.

Веб-сайт издательства: www.dmk.ru



Управление электронными устройствами на C++

Разработка практических приложений

Книга предназначена всем, кому интересно изучение C++ и управление электронными устройствами на реальных и интересных примерах. Читателю предоставлена возможность научиться писать программы для выполнения конкретных задач, а не просто скучное изложение материала с картинками. Также рассказывается как создавать программы, взаимодействующие с внешними устройствами посредством специально разработанной интерфейсной платы. Книга и прилагающееся программное обеспечение представляют собой набор простых и несложных для понимания устройств, таких как цифро-аналоговый преобразователь, аналого-цифровой преобразователь, устройство управления коллекторными и шаговыми электродвигателями, измерители температуры и напряжения, таймеры на базе компьютера и простое устройство сбора данных. Также материал книги содержит сведения из области автоматического управления, электроники и механотроники.

Издание будет полезно студентам, инженерам и научным работникам, техникам и радиолюбителям.

На сайте издательства www.dmkpress.com выложены программы для ряда компиляторов, работающих в DOS, Microsoft Windows и Linux.

Интернет-магазин:

www.dmkpress.com

Книга - почтой:

orders@alians-kniga.ru

Оптовая продажа:

"Альянс-книга"

Тел.: (499) 782-3889

books@alians-kniga.ru



www.dmk.pф

ISBN 978-5-97060-175-4



9 785970 601754 >