Белов А. В.

+ ВИРТУАЛЬНЫЙ ДИСК

ПРОГРАММИРОВАНИЕ МИКРОКОНТРОЛЛЕРОВ

Rescuer

ДЛЯ НАЧИНАЮЩИХ И НЕ ТОЛЬКО...



Сделай уверенный шаг от «чайника» до профи!

НиТ
издательство

От азов до создания собственных устройств.
Основы схемотехники и конструирования.
Два языка программирования: Ассемблер и СИ.
Реальная практическая книга + виртуальный CD!

А. В. Белов

ПРОГРАММИРОВАНИЕ МИКРОКОНТРОЛЛЕРОВ

ДЛЯ НАЧИНАЮЩИХ И НЕ ТОЛЬКО...

Книга + виртуальный диск

Наука и Техника, Санкт-Петербург
2016

Белов А. В.

Программирование микроконтроллеров для начинающих и не только... Книга + виртуальный диск. — СПб.: Наука и Техника, 2016. — 352 с.: ил.

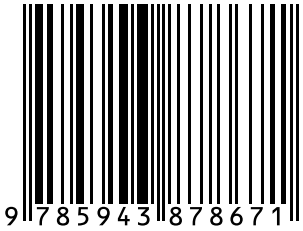
ISBN 978-5-94387-867-1

Вы хотите научиться разрабатывать самостоятельно интересные устройства на микроконтроллерах? Хотите легко научиться придумывать схемы и писать программы? Тогда эта книга для вас! Новейший самоучитель позволит уверенно пройти путь от уровня, получившего меткое название «чайник», до вполне готового специалиста, умеющего самостоятельно разрабатывать готовые работоспособные микроэлектронные устройства. Процесс освоения начинается с азов цифровой техники (базовые элементы цифровой логики, теория цифрового сигнала, основы Булевой Алгебры, системы исчисления и т.п.). Следуя постепенно от простого к сложному, вы изучите основы микропроцессорной техники, возможности семейства микроконтроллеров на примере AVR. Ну и, наконец, вы освоите основы схемотехники и конструирования микроэлектронных устройств.

Книга позволит изучить сразу два языка программирования для микроконтроллеров (язык Ассемблера и язык СИ). Научит транслировать, отлаживать программы, прошивать их в память микроконтроллера, правильно обращаться с инструментальными программами и разными видами программаторов.

Книгу сопровождает обновляемый виртуальный диск, доступный свободно в сети Интернет. На диске представлены тексты всех программных примеров, рассмотренных в книге, а также видеоуроки, помогающие глубже понять подаваемый в книге материал, ссылки на инсталляционные пакеты используемых в книге инструментальных программ: AVR Studio, PonyProg, Code Vision, а также программу, драйвер и прошивку для программатора USBasp. Там же вы найдете ссылку на виртуальный FUSE-калькулятор, позволяющий легко рассчитать значение FUSE переключателей, определяющих режимы работы конкретного микроконтроллера. На виртуальном диске представлено также множество других вспомогательных программ и материалов.

Книга предназначена для широкого круга читателей, стремящихся освоить основы программирования современных микроконтроллеров, создавать полезные электронные устройства.



ISBN 978-5-94387-867-1

Автор и издательство не несут ответственности за возможный ущерб, причиненный в ходе использования материалов данной книги.

Контактные телефоны издательства
(812) 412-70-25, 412-70-26

Официальный сайт: www.nit.com.ru

© Белов А. В.
© Наука и Техника (оригинал-макет), 2016

ООО «Наука и Техника».
Лицензия № 000350 от 23 декабря 1999 года.
198097, г. Санкт-Петербург, ул. Маршала Говорова, д. 29.
Подписано в печать . Формат 70×100 1/16.
Бумага газетная. Печать офсетная. Объем 22 п. л.
Тираж 1200 экз. Заказ № .

Отпечатано с готовых файлов заказчика
в АО «Первая Образцовая типография»
филиал «УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ»
432980, г. Ульяновск, ул. Гончарова, 14

СОДЕРЖАНИЕ

Глава 1. Цифровая техника — это очень просто 5
1.1. Начнем с микропроцессора 5
1.2. Считаем по-другому 11
1.3. Электронные цифры 17
1.4. Логические элементы 20
1.5. Простейший триггер 24
1.6. Хранение информации 29
1.7. Счетчики 32
1.8. Дешифраторы 39
1.9. Мультиплексоры 43

Глава 2. Как работают микропроцессоры и микроконтроллеры 45
2.1. Типовая схема микропроцессорной системы 45
2.2. Алгоритм работы микропроцессорной системы 52
2.3. Механизм прерываний 61
2.4. Прямой доступ к памяти 63
2.5. Микроконтроллеры 65

Глава 3. Вперед, к практике: знакомтесь — микроконтроллеры AVR 66
3.1. Общие сведения 66
3.2. Регистры общего назначения (РОН) 71
3.3. Регистры ввода-вывода 72
3.4. Память 73
3.5. Счетчик команд и стековая память 78
3.6. Подсистема ввода-вывода 81
3.7. Система прерываний 83
3.8. Таймеры-счетчики 85
3.9. Другие встроенные периферийные устройства 93
3.10. Другие ячейки 97

Глава 4. Осваиваем программы отладки и транслирования 99
4.1. Программная среда AVR Studio 99
4.2. Система программирования Code Vision AVR 125
4.3. Программаторы 133

Глава 5. Полезные устройства на микроконтроллерах своими руками 155
5.1. Общие положения 155
5.2. Простейшая программа 158
5.3. Переключающийся светодиод 184
5.4. Боремся с дребезгом контактов 193
5.5. Мигающий светодиод 200
5.6. Бегущие огни 205
5.7. Использование таймера 215
5.8. Использование прерываний по таймеру 223
5.9. Формирование звука 239
5.10. Музыкальная шкатулка 257
5.11. Кодовый замок 281
5.12. Кодовый замок с музыкальным звонком 317

Глава 6. Организация связи микроконтроллера и ПК через USB 334
6.1. Почему сегодня выбран именно USB интерфейс? 334
6.2. Принцип действия адаптера USB-FIFO 335
6.3. Внутренняя структура микросхемы FT245R 336
6.4. Типовая схема подключения микроконтроллера к ПК 339
6.5. Драйверы для адаптера и принципиальная схема 339

Приложение 1. Сводная таблица команд Ассемблера микроконтроллеров AVR 341
Группа команд логических операций 341
Группа команд арифметических операций 341
Группа команд операций с разрядами 341
Группа команд сравнения 342
Группа команд операций сдвига 342
Группа команд пересылки данных 343
Группа команд управления системой 343
Группа команд передачи управления (безусловная передача управления) 344
Группа команд передачи управления (пропуск команды по условию) 344
Группа команд передачи управления (передача управления по условию) 344

Приложение 2. Как работать с виртуальным диском 345
Раздел «Примеры программ из книги» 346
Раздел «Видеоуроки» 347
Раздел «Инструментальные программы» 348
Раздел «Справочные материалы» 349

Список литературы 351

Список полезных ссылок на ресурсы Интернет 351

ГЛАВА 1

ЦИФРОВАЯ ТЕХНИКА — ЭТО ОЧЕНЬ ПРОСТО

1.1. Начнем с микропроцессора

Что же такое микропроцессор

В современной электронике **микропроцессором** называют специальную микросхему, которая предназначена для выполнения некоего набора сложных функций по управлению тем либо иным электронным устройством. Микропроцессор — это сердце любого компьютера. Но не только. Те же технологии, которые применяются в компьютерах, с успехом применяются и в более простых электронных устройствах.

Микропроцессор незаметно завоевал весь мир. В последнее время на помощь человеку пришла целая армия электронных помощников. Мы привыкли к ним и часто даже не подозреваем, что во многих таких устройствах работает микропроцессор. Микропроцессорные технологии очень эффективны. Одно и то же устройство, которое раньше собиралось на традиционных элементах, будучи собрано с применением микропроцессора становится проще, не требует регулировки и меньше по размерам. Кроме того, с применением микропроцессоров появляются практически безграничные возможности по добавлению новых потребительских функций и возможностей.

Где же **применяются** микропроцессоры? Да просто везде! Посмотрите вокруг себя. У вас в квартире стоит современный телевизор? Не сомневайтесь: в нем есть, как минимум, один процессор. У вас есть на руке электронные часы? Современные часы строятся на основе специализированного микропроцессора. Ну, а мобильные телефоны, смартфоны — это вообще миниатюрные компьютеры!

Возможно, у вас есть игровая приставка, карманная электронная игра, современная микроволновая печь, стиральная машина, проигрыватель лазерных дисков, MP3-плеер, калькулятор. Во всех этих устройствах работает микропроцессор. Современный автомобиль нашпигован микропроцессорами, как фаршированная рыба. Не говоря уже о самолетах, кораблях, поездах и т. п. В общем, всего не перечесть.

Микропроцессор насчитывает достаточно долгую историю. До того, как изобрели микропроцессор (то есть процессор на одной микросхеме), существовали целые процессорные блоки в больших компью-

терах. Теперь же интеграция пошла до фантастических пределов. Одна микросхема содержит не только сам процессор, но и сопутствующие ему элементы. Целый компьютер в одной микросхеме. Такая микросхема называется **микроконтроллером**.

Что же это за сопутствующие элементы? Это очень важные составные части микропроцессорной системы. Без них не может обходиться ни один микропроцессор. Итак, мы подходим к первому важному вопросу — **составу типовой микропроцессорной системы**. Любая микропроцессорная система (**рис. 1.1**) состоит из следующих основных элементов: процессор, модуль памяти, порты ввода-вывода. Рассмотрим каждую из этих составляющих подробнее.

Память. Это специализированное электронное устройство, которое представляет собой набор ячеек, в каждой из которых может храниться одно число. Причем это не совсем то число, с которым мы с вами привыкли иметь дело. Это упрощенное компьютерное число. Обычно каждая ячейка памяти может хранить число, принимающее значения от нуля до 255. Подробнее об этом будет рассказано ниже (см. **раздел 2.1**).



Рис. 1.1. *Основные составляющие компьютерной системы*

Порты ввода-вывода. Это специальные микросхемы, при помощи которых микропроцессорная система может общаться с внешним миром. Причем можно говорить отдельно о портах ввода и портах вывода. Через **порты ввода** компьютерная система получает информацию извне, а посредством **портов вывода** она выдает результаты своей работы и управляет внешними устройствами. Только благодаря этим самым портам ввода-вывода к компьютеру подключаются такие устройства, как клавиатура, мышь, дисководы, CD-ROM и т. д.

Те читатели, которые знакомы с компьютерами, возможно, слышали термины «параллельный порт» (LPT) и «последовательный порт» (COM). Так вот, в данном случае речь идет совсем о другом понятии. Это просто схожие термины. Параллельный, и тем более, последовательный порты компьютера — это достаточно сложные схемы, которые, в свою очередь, управляются при помощи портов ввода-вывода. Не нужно также думать, что клавиатура и мышь используют только порты ввода, а дисплей — порт вывода.

Для управления большинством устройств компьютера используются как порты ввода, так и порты вывода микропроцессорной системы. Возможно, вас удивляет, что я называю внешними устройствами и

жесткий диск, и флоппи дисковод. Но когда мы начнем изучать типовую схему микропроцессорного устройства, вы убедитесь, что это именно так! Внутри компьютера скрыто еще много устройств, которые по отношению к микропроцессору являются внешними, хотя находятся зачастую не только внутри компьютера, но и непосредственно на материнской плате — главной плате компьютера.

Процессор — это самая главная часть, сердце всей системы. Он предназначен для того, что бы выполнять различные операции с числами. Последовательность этих операций называется **программой**. Каждая операция кодируется в виде числа и записывается в память. Те числа, с которыми процессор выполняет свои операции, называются **данными**. Данные также записаны в память. По сути дела, процессор — это цифровой автомат, способный выполнять определенный набор операций с числами. Но главной его особенностью является возможность запрограммировать любую последовательность его действий. Как происходит программирование, мы увидим далее (см. **раздел 2.2**).

Все три части вычислительной системы связаны между собой так называемыми **шинами данных**. По этим шинам передаются цифровые сигналы от процессора к модулю памяти, от процессора — к портам ввода-вывода. А также и в обратном направлении: от портов ввода вывода и памяти к процессору.

Какие же операции может выполнять процессор? Во-первых, все простейшие операции, которые можно произвести над числом. Он может читать число из любой ячейки памяти, складывать, вычитать, сравнивать, иногда умножать и делить прочитанные числа. Результат вычислений процессор записывает обратно в память. Кроме арифметических действий, процессор может выполнять логические операции с числами (Булевы функции). Что такое логические операции, будет подробно описано ниже (см. **раздел 1.4**).

Набор операций, которые процессор способен выполнять с участием портов ввода-вывода, гораздо меньше, чем операций с ячейками памяти. В них также можно записывать и считывать информацию. Однако хранение чисел — это не главное назначение портов.

Процессор считывает их в виде чисел и обрабатывает полученные числа в соответствии с алгоритмом управления.

Порт ввода *— это специальное электронное устройство, на которое извне поступают какие-либо электрические сигналы, предназначенные для управления микропроцессорным устройством. Например, сигналы, возникающие при нажатии клавиш на клавиатуре, сигналы, возникающие при срабатывании различных датчиков и т. п.*

Порт вывода *выполняет обратную функцию. В них процессор записывает различные числа, которые затем поступают на внешние устройства в виде электрических сигналов.*

Эти сигналы используются для управления. Управлять можно любым устройством, которое допускает электрическое управление, это индикаторы, дисплеи, электромагнитные реле, электромоторы, электропневмоклапаны, электрические нагреватели и т. д.

Нужно только усилить управляющие сигналы до требуемой мощности. Кроме перечисленных выше команд в любой микропроцессор заложен **набор специальных команд**, специфических для задач управления процессом вычислений. В дальнейшем мы остановимся подробнее на всех типах команд микропроцессора (см. **раздел 2.2**).

Итак, мы разобрались, что такое процессор. И вот такой простой цифровой автомат способен вытворять все те чудеса, которые мы привыкли наблюдать в исполнении современных компьютеров. Как же это возможно? Оказывается, все на свете можно описать цифрами. И текст, и изображение, и звуки, и музыку, и даже целые видеофильмы.

Хорошо поработали ученые-математики. Они сумели разработать математические модели всех этих процессов. Остальное оказалось делом техники. Главное, чтобы процессор мог выполнять операции как можно быстрее! А современные процессоры это могут!

Но обработка всех перечисленных выше процессов — это удел мощных микропроцессоров, применяемых в персональных компьютерах. Однако настоящая книга не ставит перед собой задачи изучения этих микропроцессоров. Предмет нашего изучения — небольшие **специализированные микропроцессорные схемы**, предназначенные для управления конкретными устройствами автоматики, электронной и бытовой техники. Подобные устройства управления имеют одно общее название — **микропроцессорные контроллеры**.

Теперь рассмотрим подробнее, **что же такое микропроцессор**. Как уже упоминалось, микропроцессор — это числовой автомат, который выполняет заложенные в нем операции в соответствии с заранее составленной программой. Программа — это некоторая последовательность команд, разработанная программистом и записанная в памяти (см. **рис. 1.1**) в виде чисел. Для выполнения программы в схеме микропроцессора заложен простой алгоритм. Этот алгоритм закладывается в микросхему микропроцессора при его производстве и состоит в том, что сразу после включения питания процессор начинает читать числа из той области памяти, которая отведена для хранения программ.

Чтение происходит последовательно, ячейка за ячейкой, начиная с самой первой. Каждое число (иногда несколько чисел) — это код

команды (иногда говорят «код операции»). Прочитав код команды, микропроцессор выполняет соответствующее ему действие. То есть одну из команд, о которых мы говорили выше. Таким образом, вся работа микропроцессора сводится к последовательному чтению и выполнению команд. Этот процесс начинается при включении питания и продолжается непрерывно, вплоть до выключения.

Производители микропроцессоров заботятся о том, чтобы заложить в микропроцессор достаточный набор команд для решения любых возможных задач. Используя эти команды, разработчик конкретной микропроцессорной системы может создать свою собственную программу, заставляющую микропроцессор выполнять именно те действия, какие ему нужны. Разработанная программа записывается в соответствующую область памяти и хранится там постоянно.

Виды памяти

Различают несколько разных **видов памяти**. С точки зрения микропроцессора все виды памяти идентичны. Это набор ячеек для хранения информации. Однако в реальном микропроцессорном устройстве применяют микросхемы памяти, изготовленные по разной технологии и имеющие различные свойства и назначение. В частности, для хранения программ чаще всего используется специальный вид микросхем памяти — так называемые **постоянные запоминающие устройства (ПЗУ)**. По-английски это звучит как **ROM (read only memory)**. Они называются постоянными потому, что допускают лишь однократную запись информации. После записи информации в ПЗУ она хранится там постоянно и не может быть изменена, что исключает случайную порчу или утерю программы.

Запись ПЗУ — это специальный процесс, выполняемый при помощи так называемых **программаторов**. Существует несколько видов микросхем ПЗУ. В разных видах ПЗУ для записи информации используются разные физические принципы. В любом случае, программатор изменяет структуру различных областей кристалла ПЗУ таким образом, чтобы в каждой ячейке прописалось нужное число.

После прошивки ПЗУ информация хранится в микросхеме даже после выключения питания. Микропроцессор может только читать информацию из такой памяти. Записать туда он ничего не сможет. Если же программист ошибется при составлении программы, составит ее таким образом, что процессор попытается записать в ПЗУ какую-либо информацию, ничего страшного не произойдет. В ПЗУ просто останется то, что там было до попытки записи.

Кроме постоянного запоминающего устройства, в любой микропроцессорной системе обязательно должна быть так называемая **оперативная память**, или по другому: **оперативное запоминающее**

устройство (ОЗУ). По-английски такая память называется **RAM**. В эту память процессор может как записывать информацию, так и читать из нее. Ни одна программа не обходится без некоторого количества ячеек памяти для хранения множества промежуточных результатов и вспомогательных величин. Для этих целей и служит **ОЗУ**.

Отличительной особенностью ОЗУ является то, что при выключении питания записанная в него информация теряется. Пока современная технология не умеет создавать микросхемы, позволяющие микропроцессору с достаточно большой скоростью записывать информацию и читать ее, но, при этом, не теряющие эту информацию при выключении питания.

Отдельным видом памяти следует считать **ЭСПЗУ (ПЗУ с электрическим стиранием информации)**. По зарубежной терминологии такие микросхемы называют **флэш-памятью**. Эти микросхемы позволяют процессору как записывать, так и считывать информацию. Кроме того, записанная в микросхемы информация сохраняется при выключении питания. Однако микросхемы флэш-памяти обладают очень низким быстродействием и не пригодны для хранения оперативной информации.

Кроме того, алгоритм записи информации в такие микросхемы более сложный, чем для предыдущих двух типов. Поэтому флэш-память применяется ограниченно. В основном для хранения некоторых редко изменяемых констант, значение которых должно сохраняться даже при выключении питания. Например, в системе управления автомобильной магнитолой во флэш-памяти сохраняются значение, фиксированные настройки, текущий выбранный диапазон, текущие уровни громкости, тембра, баланса и т. п.

Различия между микропроцессорами и микроконтроллерами

Обобщая вышесказанное, мы можем сформулировать следующее: **микроконтроллер** — это целая микропроцессорная система на одном кристалле! Одна микросхема содержит в себе все описанные выше составляющие: память; порты ввода-вывода; собственно процессор.

Кроме того, там часто располагаются некоторые дополнительные устройства: таймеры; устройства прерывания; компараторы и др.

Значения этих, пока что, возможно, непонятных терминов вы узнаете из последующего повествования (см. **главу 2**). Вообще, термины «микропроцессор» и «микроконтроллер» достаточно условны и иногда подменяют друг друга. Иногда для краткости микроконтроллер называют процессором. И в этом нет большой ошибки.

1.2. Считаем по-другому

Десятичная система исчисления

Если вы всерьез собрались научиться разрабатывать микропроцессорные устройства, вам просто необходимо знать, как же компьютер хранит и обрабатывает числа. К сожалению (а может, и к счастью), люди не придумали способа записывать в виде электронных сигналов числа в том виде, как мы их привыкли видеть. Поэтому математикам сначала пришлось потрудиться и придумать более подходящий способ их представления. И они потрудились на славу! Придуман не один, а огромное множество **способов представления чисел**.

Итак, что же это за способы? Любой грамотный человек хорошо знает, по крайней мере, два таких способа. Это, во-первых, хорошо нам известные **арабские цифры** и гораздо менее распространенные, но все же тоже всем известные, **римские цифры**. Одно и то же число можно записать как 2 или как II, как 5 или как V, как 22 или как XXII.

Очевидно, что перед нами два альтернативных способа написания чисел. Числа одинаковые, а способы написания разные. Ну а если есть два способа, почему бы не выдумать и третий, четвертый и т. д.? И математики давно нашли такие способы. И не три-четыре. Они нашли универсальный прием, позволяющий теоретически создавать бесконечное количество способов представления чисел. Эти способы назвали **системами исчисления**.

За основу взяли арабские числа. Очевидно, что этот способ представления числа гораздо более красивый, чем римский вариант, так как подчиняется строгому закону. Судите сами. В привычной для нас системе исчисления мы имеем 10 цифр: **1, 2, 3, 4, 5, 6, 7, 8, 9** и **0**. При помощи этих цифр мы легко можем представить любое число.

Если нам нужно записать число, величина которого меньшие десяти, мы используем всего одну из вышеуказанных цифр. Для представления чисел от десяти и выше мы вводим новый разряд, который для начала мы приравниваем единице. Подставляя в младший разряд по порядку те же десять базовых цифр, мы получаем следующий десяток. Как только все цифры в младшем разряде перебраны, мы увеличиваем наш старший разряд на единицу. И так, пока не переберем все цифры вплоть до 99.

Когда во втором разряде будут перебраны все возможные варианты, мы вводим третий разряд. Затем четвертый, пятый и так до бесконечности. Как видите, здесь просматривается набор четких правил, описывающих то, как при помощи десяти знаков (цифр) представить любое мыслимое число! Так как количество цифр равно десяти, описанная выше система исчисления называется **десятичной**. Сформулируем

Правила

1. *Для представления чисел используются десять цифр, каждая из которых обозначается своим знаком. Любое число записывается при помощи одной или нескольких рядом стоящих цифр, причем имеет значение как набор используемых цифр, так и их взаимное расположение.*
2. *Для записи чисел в пределах первого десятка используется только одна цифра. Эта цифра является младшим разрядом числа.*
3. *Для чисел следующего десятка вводится дополнительный разряд. Это еще одна цифра (следующий разряд числа), которая ставится впереди предыдущего разряда. Цифра, стоящая в этом разряде, означает количество десятков. Двухразрядные числа используются в пределах первой сотни.*
4. *При переполнении очередного разряда вводится следующий разряд, и так до бесконечности.*

набор правил, по которым строится любое число в десятичной системе исчисления.

Теперь зададим себе вопрос. **А почему именно десять цифр?** Видимо потому, что первобытные люди, которые изобретали эту систему, пользовались для счета пальцами рук. А их всего десять.

Восьмиричная система исчисления

А если бы природа распорядилась так, что у человека было бы не по пять, а по четыре пальца на каждой руке? Тогда бы цифр было восемь. И мы сейчас, наверное, использовали бы восьмиричную систему исчисления! Такая система действительно существует. И как же она выглядит? Ну, во-первых, она имеет только восемь цифр.

Для удобства используются первые восемь знаков десятичной системы: **0, 1, 2, 3, 4, 5, 6** и **7**. Попробуем представить, как нужно записывать числа в восьмиричной системе, при условии, что правила составления числа из отдельных цифр будут такими же, как в известной нам десятичной системе.

Числа от нуля до семи мы, как и в десятичной системе, будем записывать при помощи одной цифры. В данном случае числа, записанные в десятичной и восьмиричной системах, ничем не будут отличаться. Далее в десятичной системе идет число восемь. **Цифры 8 в восьмиричной системе нет!**

Поэтому мы поступим в строгом соответствии с описанными выше правилами. Мы добавим новый разряд, который будет равен у нас единице. Младший же разряд будет равен нулю. При этом число 8 в восьмиричной системе исчисления будет выглядеть как 10. Для того, чтобы не путаться, при записи чисел в разных системах исчисления в математике принято помечать каждое число **специальной меткой**, показывающей, в какой системе исчисления оно записано.

Рассмотрим несколько примеров записи числа в двух разных системах исчисления:

$9_{10} = 11_8$, $10_{10} = 12_8$, $11_{10} = 13_8$.

И так далее.

Пример

Число восемь в десятичной системе записывается так — 8_{10}. А то же число в восьмиричной системе записывается следующим образом — 10_8.

После числа 17_8 в восьмиричной системе следующее число 20_8. Эти два числа в десятичной системе равны: $17_8 = 15_{10}$, $20_8 = 16_{10}$. Аналогичным образом, в полном соответствии с описанными выше правилами, записываются и остальные числа в восьмиричной системе. После переполнения второго разряда появляется третий (после 77_8 идет 100_8). После третьего — четвертый, и так далее.

Количество цифр, используемых в системе исчисления, называют ее **основанием**. Десятичная система исчисления имеет основание 10, а восьмиричная — основание 8. Ну а теперь, когда мы знаем, как выглядит восьмиричная система, легко можно представить системы и с другим основанием. По большому счету можно выбрать любое число в качестве основания, лишь бы знаков (цифр) хватило для записи чисел.

Пример

Так записываются числа в двух разных системах исчисления:

$32_{10} = 40_8$, $131_{10} = 203_8$, $42_{10} = 52_8$, $1254_{10} = 2346_8$, $348_{10} = 534_8$ и т. д.

Реально же на практике нашли применение, кроме десятичной и восьмиричной, еще две системы исчисления: **шестнадцатиричная** и **двоичная**. Причем последняя (двоичная) система исчисления и является той самой системой представления чисел, которую инженеры без труда смогли смоделировать при помощи электронных схем. Восьмиричная и шестнадцатиричная системы исчисления очень удобны для записи компьютерных данных на бумаге и на экране компьютера. Раньше, на заре развития компьютерной техники, широко использовали восьмиричную систему. Сейчас она почти забыта. Теперь вместо нее более употребима шестнадцатиричная. Рассмотрим подробнее две еще не описанные нами системы.

Шестнадцатиричная система исчисления

В **шестнадцатиричной** системе исчисления, как вы догадались, используются шестнадцать цифр. Обычно для обозначения первых десяти цифр применяют те же символы, что и для десятичной системы. А недостающие шесть цифр заменяют буквами латинского алфавита. Вот полный набор цифр шестнадцатиричной системы:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E и **F.**

Пример

Вот еще несколько примеров шестнадцатиричных чисел:

$17_{10} = 11_{16}$ $206_{10} = 0CE_{16}$
$25_{10} = 19_{16}$ $698_{10} = 2BA_{16}$
$26_{10} = 1A_{16}$ $1235_{10} = 4D3_{16}$

Одним разрядом в шестнадцатиричной системе исчисления можно записать числа от нуля до пятнадцати. Число шестнадцать (16_{10}) записывается, как 10_{16}.

Применение букв может внести путаницу. Если число состоит только из букв, его можно принять за какое-то слово. Например, за название переменной в тексте программы. Поэтому перед шестнадцатиричным числом, начинающимся с буквы, принято ставить незначащий ноль.

Двоичная система исчисления

Среди всех систем исчисления особое место занимает **двоичная** система. Она использует для записи любого числа всего две цифры. Это цифры 0 и 1. Естественно, что при помощи одного разряда в двоичной системе можно записать только два числа: 0 и 1. Число два в двоичной системе будет выглядеть как 10_2. Используя уже знакомые правила, запишем другие числа: $3_{10} = 11_{2,}$ $5_{10} = 101_{2,}$ $_{10} = 111_{2,}$ $4_{10} = 100_{2,}$ $6_{10} = 110_{2,}$ $8_{10} = 1000_2$ и т. д.

Если у вас есть компьютер, на котором установлен Windows, вы сами легко можете поэкспериментировать с переводом чисел из одной системы исчисления в другую. Для этого запустите стандартный калькулятор, входящий в Windows. Обычно он запускается через меню Пуск / Стандартные / Калькулятор. Поэкспериментируйте с калькулятором для более полного усвоения материала.

Войдите в меню «Вид» калькулятора и выберите«Инженерный». После этого вид калькулятора изменится. Вы увидите множество новых клавиш и органов управления. И среди них дополнительные буквенные клавиши для набора латинских букв. Кроме того, в левой верхней части калькулятора появится переключатель систем исчисления. Он имеет четыре положения: **Hex** (шестнадцатиричная); **Dec** (десятичная); **Oct** (восьмиричная); **Bin** (двоичная).

Вы можете выбрать любую из систем исчисления, набрать на калькуляторе число, а затем переключить калькулятор на другую систему. Калькулятор автоматически переведет набранное вами число в новую систему исчисления.

Способы обозначения чисел в разных системах исчисления

Теперь немного о способах обозначения чисел в разных системах исчисления. До сих пор, во всех предыдущих примерах, для обозна-

чения типа используемой системы исчисления мы применяли способ, принятый в математике. Но такой способ далеко не всегда можно применить. Приписывать маленькие циферки внизу под записью числа удобно только от руки на бумаге. Ну, еще в Microsoft Word и аналогичных ему текстовых процессорах. При написании же текстов программ или, например, электронных писем используются редакторы, не отягощенные такими возможностями. Они не позволяют писать подстрочные надписи. Поэтому существуют другие способы записи таких чисел.

Способ 1. Самый распространенный способ — добавление в конце числа **латинской буквы**, обозначающей систему исчисления. Так, для обозначения шестнадцатиричных чисел используется латинская буква **H**. Для восьмиричных — латинская буква **O**, для двоичной системы — латинская буква **B**. Десятичные числа при таком способе обозначений записываются либо вообще без буквы, либо обозначаются латинской буквой **D**.

Пример

Запись чисел с использованием букв:

23D	*то же, что и*	23_{10}
03FH	*то же, что и*	$03F_{16}$
01101B	*то же, что и*	01101_{2}
375O	*то же, что и*	375_{8}

Способ 2. Другой широко распространенный способ использует, в частности, фирма Atmel при программировании для микроконтроллеров AVR. В этом способе систему исчисления определяет **дополнительный ноль** и **буква впереди числа**. Так для записи шестнадцатиричных чисел используется сочетание «0x», для двоичных «0b», а десятичные числа не имеют специальных обозначений. Восьмиричная система исчисления в данном случае не поддерживается.

Пример

Вот варианты записи десятичного, шестнадцатиричного и двоичного чисел.

2342 0x12C4D0 0b001101011

Способ 3. Еще один способ, который также используется при написании программ для микроконтроллеров AVR, — использование символа «$» для обозначения шестнадцатиричных чисел. В системах программирования фирмы Atmel эти два способа можно использовать одновременно.

Пример

Число 0x12C4D0 можно записать как $12C4D0.

Арифметическая операция сложения

В любой из вышеперечисленных систем исчисления можно выполнять любые арифметические операции, к которым мы привыкли в

десятичной системе. То есть сложение, вычитание, умножение, деление. Правда, на практике никто не занимается восьмиричной и шестнадцатиричной арифметикой. Это не имеет никакого смысла. А вот арифметика в двоичной системе была подробно проработана.

Надо же было обучить этому электронные устройства. На самом деле правила, по которым производятся все операции в любой из систем исчисления, взяты из десятичной системы. Но при вычислениях в двоичной системе это выглядит немного по-другому. Возьмем, например, сложение. Как и в десятичной, так и в двоичной системе два любых числа можно сложить столбиком. Только нужно помнить, что в этой системе каждый разряд может принимать лишь два значения: либо 0, либо 1. Возьмем для примера два двоичных числа. Например, 10011001110 + 11000101110. Записываем пример сложения.

```
  10011001110
+ 11000101110
-------------
101011111100
```

Правила

1. *Ноль плюс ноль — получится, естественно, ноль.*
2. *Один плюс ноль и ноль плюс один дадут в результате единицу.*
3. *При сложении двух единиц мы получим ноль в текущем разряде и единицу переноса в следующий разряд.*

Теперь **выполним сложение**. Как и в десятичной системе, будем складывать числа поразрядно, начиная с младшего разряда. При сложении значений каждого разряда будем учитывать следующие правила.

Сложив все разряды, **результат** запишем под чертой. Складывая значение очередного разряда, не забывайте учитывать перенос из предыдущего. При сложении двух единиц плюс перенос из предыдущего разряда получим единицу и перенос в следующий.

Арифметическая операция умножения

Умножение в двоичной системе также делается столбиком. Но в двоичной системе есть одна особенность, которая сильно облегчает задачу. Очевидно, что любое число, умноженное на ноль, дает в результате ноль. А число, умноженное на единицу, дает в результате само себя. Вот **пример** умножения столбиком двух двоичных чисел:

```
   10011001110
 x        1011
-------------
   10011001110
+  10011001110
  10011001110
-------------
110100110110 10
```

Как легко убедиться из примера, умножение в двоичной системе исчисления сводится к сложению одного и того же числа (множимого), сдвинутого относительно самого себя. Как видите, при работе с двоичными числами даже умножать не приходится. Достаточно уметь сдвигать разряды числа и складывать числа между собой. Это важно при построении вычислительных устройств.

Простейшие микропроцессоры в составе своих команд не имеют команды умножения. Однако любой микропроцессор имеет команды сдвига и сложения. Одну команду умножения всегда можно заменить небольшим набором команд сложения и сдвига. Точно так же команду деления легко заменить сдвигом и вычитанием.

1.3. Электронные цифры

Представление чисел на ПК

Изобретение двоичной системы исчисления дало возможность научить компьютер работать с числами. Теперь настало время узнать, каким образом это делается. Посмотрим, как числа представляются в компьютере. Во всех современных вычислительных системах это делается следующим образом. Представим себе некий узел вычислительной системы. Допустим, он должен передавать на последующие узлы числа в электронном виде. Для этой цели такой узел имеет **группу выходов** (обычно их количество равно или кратно восьми).

Обозначим эти выходы, как это принято в вычислительных системах, D0, D1, D2, D3, D4, D5, D6 и D7 (см. **рис. 1.2**). Эти выходы подключаются к соответствующим входам последующего узла, как показано на **рис. 1.3**. Для передачи числа используется вся группа выходов одновременно. Передаваемое число представляется в двоичной системе исчисления. Каждый из выходов передает один разряд двоичного числа и может находиться в одном из двух состояний:

- состояние логического нуля — когда напряжение на выходе отсутствует;
- состояние логической единицы — когда на выходе присутствует напряжение (в этом случае оно обычно равно или близко к напряжению питания).

Причем схема каждого из выходов устроена таким образом, что исключает появление на любом из выходов промежуточных значений напряжения. Такая группа выходов называется **цифровой шиной данных**.

Каждый разряд шины имеет свой «вес». Именем D0 обозначают разряд, который имеет самый маленький «вес» — вес, равный единице. Это

значит, что когда в этом разряде установлена логическая единица, а во всех остальных разрядах — логический ноль, то все число равно единице.

Разряд D1 имеет «вес», равный двум (10_2). Это означает, что, если значение разряда D1 равно единице, а во всех остальных разрядах ноль, то все число, передаваемое шиной, будет равно двум.

Вес D2 равен четырем (100_2). D3 — восьми (1000_2). И так далее. Вес последнего разряда шины (D7) равен 128 (10000000_2). Значение числа, которое передается по шине, всегда можно найти путем сложения весов тех разрядов шины, значение которых в данный момент равно единице.

Пример

Для того, что бы передать по шине число 25 (11001_2), нужно выставить на шине следующие значения: на трех выходах D0, D3 и D4 должен быть единичный сигнал, на всех остальных выходах должен быть ноль.

Правило

По восьмиразрядной шине можно передавать числа от 0 до 255 (то есть 256 разных значений). Это важно знать, так как восьмиразрядная шина является своего рода стандартом в вычислительной технике.

Проверим теперь, что получится, если мы сложим веса всех выходов, находящихся в единичном состоянии. Вес разряда D0 равен 1. Вес разряда D3 равен 8. Вес D4 равен 16. **Итого**: 1+8+16=25. Что и требовалось доказать.

Очевидно, что для передачи числа, максимального для данной шины, нужно установить все разряды шины в единичное состояние. В этом случае число, передаваемое по нашей шине данных, будет равно 1+2+4+8+16+32+64+128 = 255.

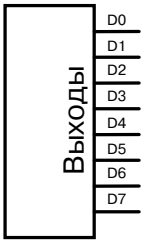


Рис. 1.2. *Узел с цифровыми выходами*

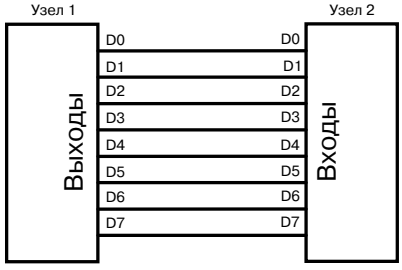


Рис. 1.3. *Соединение двух цифровых узлов*

Более подробно к описанию шин мы вернемся, когда перейдем к описанию микропроцессорной системы.

Двухуровневый сигнал

А сейчас остановимся подробнее на двухуровневом сигнале, имитирующем двоичный разряд числа. Каждый из этих уровней имеет свое название.

Различают **сигнал логического нуля** и **сигнал логической единицы**. Другое название — **высокий логический уровень** и **низкий логический уровень**. Эти термины в дальнейшем будут часто встречаться в книге.

Для получения сигналов логической единицы и логического нуля используются так называемые **триггерные схемы**, или **схемы с двумя устойчивыми состояниями**. Напряжения на выходе любого реального устройства неидеальны и имеют разброс в значениях.

Пример

Сигналы для микросхем, выполненных по так называемой ТТЛ-технологии (транзисторно-транзисторная логика), допускают следующие отклонения (при напряжении питания 5 В):

- *для логического нуля на выходе допускается присутствие напряжения до 0,4 В;*
- *для логической единицы напряжение на выходе должно быть не менее 2,4 В.*

По ТТЛ технологии выполнены микросхемы таких серий, как 155, 555, 533, 1533 и др. В микросхемах других серий уровни сигнала имеют другие значения.

Это происходит благодаря наличию у цифровых входов порога срабатывания. Если входной сигнал выше этого порога, внутренний триггер устанавливается в состояние логической единицы. Если входное напряжение ниже порога, то триггер переходит в состояние логического нуля.

Правило

Существует правило, верное для любых типов цифровых микросхем: если напряжение на выходе цифровой микросхемы соответствует логическому нулю (входит в соответствующий диапазон), то другая микросхема, на которую приходит данный цифровой сигнал, надежно распознает его как ноль. Если сигнал на выходе соответствует логической единице (входит в эти допустимые пределы), то принимающее устройство всегда распознает его как единицу.

Триггер — это как раз такое устройство, которое может находиться лишь в одном из двух состояний. Промежуточное состояние исключено. Такой подход повышает надежность работы цифровых схем. Благодаря триггерному эффекту такие явления, как тепловые шумы, дрейф нуля и электромагнитные помехи гораздо меньше влияют на качество передачи цифрового сигнала. Вернее, они вообще не влияют, пока уровень сигнала помехи не превысит порога.

Правда, если помехи все же превысят порог, то наступает **полный сбой**. В этом случае цифровой сигнал, передаваемый по шине данных, будет полностью искажен, а полезная информация будет утеряна.

Однако в том случае, когда уровень помех невысок, при передаче цифрового сигнала полностью отсутствуют типичные недостатки, присущие аналоговым способам передачи информации. Отсутствует посте-

пенное ухудшение качества с увеличением дальности передачи или при большом количестве перезаписей с одного носителя на другой.

Электронное представление числа, рассмотренное выше, дает возможность не только передавать эти числа от одного устройства к другому. В следующем разделе мы увидим, каким образом можно производить операции сложения с электронными числами. Используя электронные числа, можно также хранить цифровую информацию, производить описанные выше арифметические операции (сложение, вычитание, умножение, деление) и многое другое. Но прежде нам нужно познакомиться с особым классом электронных элементов, которые и являются основой для построения схем, выполняющих операции с числами. Они называются «Логические элементы».

1.4. Логические элементы

Знакомство с логическими элементами

Многим из вас наверняка знакомо понятие ЛОГИЧЕСКИЕ ЭЛЕМЕНТЫ. Если вы хоть немного работали с цифровыми схемами, то наверняка знакомы с подобным понятием. В настоящее время логические элементы и другие цифровые компоненты можно встретить в схемах, очень далеких от микропроцессоров и вычислительной техники. Если принципы работы цифровой логики вам хорошо знакомы, можете пропустить данную главу. Для тех, кто этого не знает или желает систематизировать свои знания, начнем с самого начала.

Из всего разнообразия цифровых элементов большинство можно отнести к разряду составных. **Составными** я называю те элементы, которые можно составить из других, более простых. А в основе всего разнообразия цифровых устройств лежат всего три простейших логических элемента. На **рис. 1.4** изображены эти три кита цифровой техники.

Следует заметить, что логические элементы на **рис. 1.4** изображены в соответствии со стандартом, принятым в свое время в СССР и теперь еще широко используемым во всех странах СНГ. По этим стандартам цифровые элементы изображаются в виде прямоугольника. Все входы рисуются слева, а выходы — справа. Именно таким образом в этом стандарте можно отличить входы элемента от его выходов. Правда, в случае более сложных элементов это правило соблюсти не всегда возможно, так как часто бывает, что один и тот же выход служит одновременно и входом. Но для простых элементов это условие всегда соблюдается.

В западных стандартах для цифровых схем приняты другие условные обозначения. В задачу настоящей книги не входит изучение западных стандартов. При желании вы сможете это сделать самостоятельно.

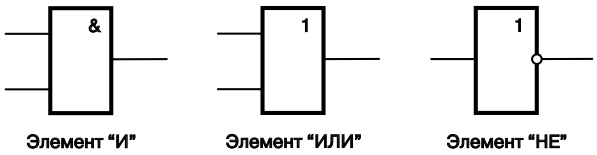


Рис. 1.4. *Простейшие логические элементы*

Все логические элементы работают **с цифровыми сигналами**. Это значит, что сигнал на любом из входов элемента должен принимать значения либо логического нуля, либо логической единицы. На выходе каждый элемент также обеспечивает цифровой сигнал, который, в зависимости от логики работы схемы, принимает значение либо логической единицы, либо логического нуля. На **рис. 1.4** изображены двухвходовые варианты элемента «И» и элемента «ИЛИ». На самом деле эти элементы могут иметь любое количество входов. Теоретически количество входов может быть увеличено до бесконечности. Тип элемента определяется не количеством входов, а логикой его работы. Какова же эта логика? Рассмотрим каждый элемент по отдельности.

Простые логические элементы

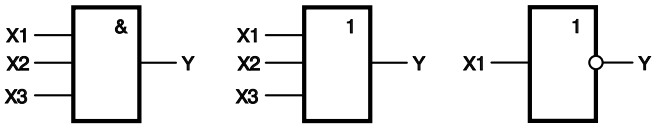
Элемент «И». На выходе этого элемента сигнал логической единицы появляется тогда и только тогда, когда на всех его входах будет присутствовать логическая единица. То есть единица должна быть **И** на первом, **И** на втором, **И** на третьем (если он есть), **И** на всех имеющихся входах. Если хотя бы на одном входе будет ноль, то и на выходе тоже будет ноль.

Элемент «ИЛИ». На выходе этого элемента сигнал логической единицы появится тогда и только тогда, когда хотя бы на одном из его входов появится единица. То есть единица должна быть **ИЛИ** на первом, **ИЛИ** на втором, **ИЛИ** на третьем — на любом из имеющихся входов или на нескольких сразу. Логический ноль на выходе будет только тогда, когда на всех входах будет сигнал логического нуля.

Элемент «НЕ», или инвертор. У этого элемента не может быть больше одного входа. Инвертор имеет один вход и один выход. И логика его работы очень проста. Когда на входе у инвертора сигнал логического нуля, на выходе логическая единица. И наоборот, когда на входе логическая единица, на выходе логический ноль.

Таблица истинности

Для отображения логики работы того или иного элемента принято составлять так называемые таблицы истинности. **Таблица истинности** — это такая таблица, которая имеет столбцы для всех входов и выхо-



Входы			Вых.
X1	X2	X3	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Входы			Вых.
X1	X2	X3	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Вход	Выход.
X1	Y
0	1
1	0

Рис. 1.5. *Примеры построения таблицы истинности*

дов конкретного элемента. В строках таблицы отображаются все возможные состояния элемента. Каждая строка соответствует одному из возможных состояний. На **рис. 1.5** приведены таблицы истинности для трех основных логических элементов. Для наглядности использованы трехвходовые варианты элемента «И» и элемента «ИЛИ».

Составные логические элементы

Итак, исходный материал — три основных элемента — у нас есть. Теперь начнем составлять из них остальные. Полученные при этом новые элементы часто имеют свое самостоятельное значение, и поэтому многие элементы, представляемые нами как составные, имеют свое собственное схемное обозначение. Рассмотрим это подробнее.

Для начала соединим элемент «И» с инвертором так, как это показано на **рис. 1.6.** В результате у нас получится новый элемент. Такой элемент имеет название **«И-НЕ»**, и на схеме его часто изображают так, как показано на **рис. 1.7**. Логика работы нового для нас логического элемента «И-НЕ» очевидна. Сигнал на выходе будет равен нулю в том и только в том случае, когда на всех его входах присутствует логическая единица. Точно таким же образом легко составить элемент **«ИЛИ-НЕ»**.

Попробуем теперь синтезировать более сложный логический элемент под названием **«ИСКЛЮЧАЮЩЕЕ ИЛИ»**. Его тоже часто можно встретить в различных электронных схемах.

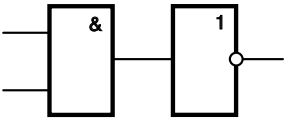


Рис. 1.6. *Составной элемент «И-НЕ»*

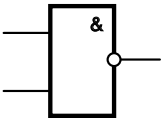
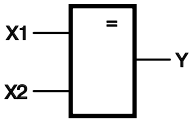


Рис. 1.7. *Условное обозначение элемента «И-НЕ»*

Схемное обозначение элемента «ИСКЛЮЧАЮЩЕЕ ИЛИ» и его таблица истинности приведены на **рис. 1.8**. Как видно из таблицы, логика работы элемента соответствует его названию. Это тот же элемент «ИЛИ» с одним небольшим отличием. Если значение на обоих входах равно логической единице, то на выходе элемента «ИСКЛЮЧАЮЩЕЕ ИЛИ», в отличие от элемента «ИЛИ», не единица, а ноль.



Входы		Вых.
X2	X3	Y
0	0	0
0	1	1
1	0	1
1	1	0

Рис. 1.8. *Элемент «Исключающее ИЛИ»*

Эквивалентная схема элемента «ИСКЛЮЧАЮЩЕЕ ИЛИ» изображена на **рис. 1.9**. Советую самостоятельно разобрать работу этой схемы. Для этого мысленно нужно подставлять на входы X1 и X2 различные варианты логических сигналов и для каждого варианта прослеживать, какими будут сигналы на выходе каждого элемента. И так поэлементно проследить, какой будет сигнал на выходе всей схемы.

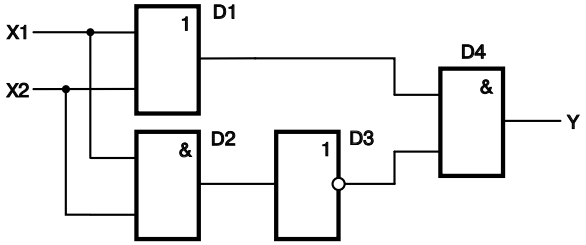
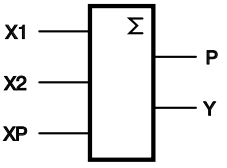


Рис. 1.9. *Одно из возможных схемных решений элемента «Исключающее ИЛИ»*

Если внимательно посмотреть на таблицу истинности элемента «ИСКЛЮЧАЮЩЕЕ ИЛИ», то легко заметить, что логика работы элемента очень похожа на таблицу сложения двух одноразрядных двоичных чисел. И действительно:

- ноль плюс ноль равняется ноль;
- один плюс ноль и ноль плюс один равняется один;
- сумма двух единиц дает ноль в этом разряде и единицу переноса в следующий разряд, не хватает только переноса.

Элемент, формирующий также и перенос, и выполняющий суммирование входных сигналов, называется **сумматором**. Его еще можно составить из простых логических элементов. Схемное изображение и логика работы сумматора приводятся на **рис. 1.10**. Здесь **X1** и **X2** — это входы складыва-



Входы			Выходы	
XP	X1	X2	Y	P
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Рис. 1.10. *Сумматор*

емых разрядов. **Y** — выход суммы. **P** — выход переноса в старший разряд. **XP** — вход переноса с младшего разряда. Один сумматор производит суммирование двух одноразрядных двоичных чисел. Для суммирования многоразрядных цифровых сигналов сумматоры **соединяют каскадом**. При этом сигнал с выхода P сумматора одного разряда подается на вход XP сумматора следующего разряда.

На практике отдельные микросхемы-сумматоры почти никогда уже не применяются. Где-то внутри микропроцессора обязательно есть сумматор, который является его частью. Я привел здесь его описание лишь как пример сложного элемента цифровой техники.

При разработке микропроцессорных систем нам чаще придется иметь дело с другими элементами, такими как триггеры, регистры, дешифраторы, мультиплексоры и т. д. Прямо сейчас мы этим и займемся.

1.5. Простейший триггер

Что такое триггер

Триггер — это качественно новый составной элемент цифровой техники. Его уже нельзя отнести к логическим элементам. **Триггер** можно назвать элементом нового класса. Выше мы уже говорили о том, что логические входы обладают триггерным эффектом. В случае с логическими входами мы имели дело с триггерным эффектом, работающим по уровню сигнала.

Мы говорили также, что триггер — это устройство, которое может находиться в двух (и только в одном из двух) устойчивых состояниях. На самом деле входы логических элементов обладают слабым триггерным эффектом. Между областью входных напряжений, соответствующих логическому нулю, и областью напряжений, соответствующих логической единице, всегда существует **промежуточный диапазон логической неопределенности**. Если напряжение на логическом входе попадает в этот диапазон, то поведение логического элемента непредсказуемо.

Некоторые элементы даже могут переходить в линейный режим работы (усиливают аналоговый сигнал). Более четкое срабатывание обеспечивает так называемый **триггер Шмитта**. Этот триггер имеет совершенно конкретный порог срабатывания. При переходе входного напряжения через этот порог триггер Шмитта переключается из одного устойчивого состояния в другое.

Второй особенностью триггера Шмитта является наличие не одного, а двух порогов срабатывания. Первый порог действует, когда напряжение на входе повышается. При достижении порога триггер срабатывает,

и на выходе появляется логическая единица. При понижении напряжения на входе действует второй порог. Когда напряжение на входе снизится ниже этого порога, триггер переключается снова, и на выходе устанавливается логический ноль.

Второй порог всегда немного ниже первого. Наличие двух порогов называется **гистерезисом**. Гистерезис увеличивает стабильность работы триггера при напряжениях, близких к пороговому. В отсутствии гистерезиса при входных напряжениях, близких к порогу срабатывания, любая помеха на входе вызовет многократное переключение триггера, что обычно крайне нежелательно. Триггеры Шмитта часто используются для преобразования аналоговых колебаний в прямоугольные импульсы, которые затем уже используются в цифровой технике.

Кроме триггеров, срабатывающих по уровню, существует целый класс триггеров, переключение которых происходит при воздействии различных сочетаний логических сигналов. Такие триггеры имеют не менее двух цифровых входов и один или два цифровых выхода. Сигналы на таких выходах всегда имеют противоположные значения. Один из выходов называется прямым выходом триггера, а второй — инверсным выходом.

Такой триггер имеет два устойчивых состояния: **единичное** и **нулевое**. В единичном состоянии на прямом выходе триггера устанавливается сигнал логической единицы, а на инверсном, соответственно, сигнал логического нуля. Если же триггер находится в нулевом состоянии, сигналы на выходах меняют свои значения: на прямом выходе появляется ноль, а на инверсном — единица. Переход из одного устойчивого состояния в другое происходит под воздействием сигналов на входах триггера.

Устройство и работа RS-триггера

Для разных триггеров используется разные сочетания сигналов. Далее в этой главе мы подробно рассмотрим все существующие виды триггеров и логику их работы. А начнем мы с самого элементарного, простейшего триггера, так называемого **RS-триггера.** На **рис. 1.11** показана схема такого триггера, а на **рис. 1.12** — его условное обозначение.

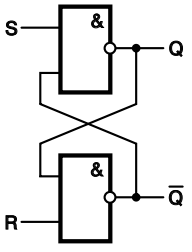


Рис. 1.11. *RS-триггер*

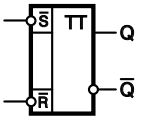


Рис. 1.12. *Условное обозначение RS-триггера*

Как видно из **рис. 1.11**, схема RS-триггера состоит из двух элементов «И-НЕ». Триггер имеет два входа, которые называются S и R. **Вход S** называют **входом установки (**от слова Set — установить). **Вход R** — это **вход сброса** (Reset). Два выхода триггера обозначаются как Q и $\overline{Q}$.

Буква $\overline{Q}$ с **чертой сверху** читается как **«не кю»**. Черта над именем любого выхода или входа означает, что данный вывод (вход) инверсный. Поэтому Q и $\overline{Q}$ — это, соответственно, прямой и инверсный выходы триггера.

Посмотрим, **как работает RS-триггер**. Для правильной работы такого триггера на оба его входа необходимо подать сигналы логической единицы. Перевод триггера из одного устойчивого состояния в другое производится путем кратковременной подачи на один из входов нулевого сигнала. При подаче нуля на вход S (Set) триггер переходит в единичное состояние. При подаче сигнала на вход R (Reset) триггер сбрасывается в ноль.

Одновременная подача двух нулей на оба входа триггера недопустима, так как в этом случае работа триггера непредсказуема. В промежутке между сигналами, когда на обоих входах единица, триггер сохраняет ранее установленное состояние.

Если на обоих входах присутствует единица, установленное состояние триггера сохраняется все время, пока на схему подано напряжение питания. Таким образом, триггер можно использовать для хранения информации. При выключении питания информация теряется. Если питание было выключено, то в момент включения питания (до прихода первых входных импульсов) триггер устанавливается в случайное положение. Если точнее, то это положение зависит от того, какой из элементов триггера оказался более быстродействующим.

Рассмотрим подробнее, **как происходит переключение триггера**. Обратимся для этого к схеме на **рис. 1.11**. Допустим, что после включения триггер сбросился в нулевое состояние. То есть на выходе Q триггера — логический ноль. Этот ноль поступает на соответствующий вход нижнего элемента триггера (см. **рис. 1.11**). На втором входе того же элемента, как мы помним, — логическая единица. В соответствии с логикой работы элемента «И-НЕ», на его выходе так же устанавливается единица. Эта единица поступает на выход $\overline{Q}$ и на соответствующий вход верхнего по схеме элемента. На втором входе верхнего элемента, как мы помним, — тоже единица. Две единицы на входе «И-НЕ» и дают ноль на его выходе. Итак, круг замкнулся. Все сигналы подтверждают сами себя. На выходе Q — логический ноль, на выходе $\overline{Q}$ — логическая единица. Триггер находится в устойчивом состоянии.

Теперь посмотрим, **как происходит переход триггера из одного устойчивого состояния в другое**. Для переключения триггера в единичное состояние подадим на вход S сигнал логического нуля. Равновесие сразу нарушится. На входах верхнего элемента уже не две единицы, а единица и ноль. Поэтому на его выходе сразу устанавливается единица. Она поступает на соответствующий вход нижнего элемента.

И теперь уже на нижнем элементе две единицы на обоих входах. Он тут же выдает на выходе ноль. Теперь можно снимать нулевой сигнал со входа S. Триггер находится уже в новом состоянии. И это состояние тоже устойчивое. Единичный сигнал на входе S не переведет триггер назад в нулевое состояние, так как на нижнем входе верхнего по схеме элемента логический ноль.

Переключение триггера в нулевое состояние происходит точно так же, как и переключение в единичное. Только нулевой сигнал для переключения в данном случае подается на вход R. Процессы, происходящие в триггере, при этом полностью соответствуют процессам, происходящим в предыдущем случае, только с точностью до наоборот. Если триггер уже стоит в единичном состоянии, то подача нулевого импульса на вход S ничего не изменит. Точно так же подача импульса на вход R не изменит состояния триггера, если перед этим он находился в нулевом состоянии.

RS-триггер можно считать простейшим устройством для хранения одного бита цифровой информации.

Один бит — это один двоичный разряд или величина, которая может принимать только два значения (0 и 1). Логично, что для хранения единицы триггер переводится в единичное состояние. Для хранения нуля — в нулевое.

Обратите внимание на обозначение входов сброса и установки на **рис. 1.12**. Оба входа снабжены кружочками, а их имена снабжены чертой. Это означает, что входы инверсные. RS-триггер имеет не только инверсный выход, но и оба его входа также инверсные. Входы считаются инверсными потому, что активный сигнал для каждого из них — это низкий логический уровень. Такое обозначение достаточно условно. В RS-триггере допускается обозначение входов как с инверсией, так и без. Оба варианта будут верны.

Борьба с дребезгом контактов

На самом деле RS-триггеры редко используются для хранения двоичных чисел. Для этого существуют другие, более сложные триггеры, о которых мы поговорим немного позже. Но все же RS-триггеры достаточно широко применяются в цифровой и микропроцессорной технике. В качестве примера я хотел бы остановиться на одном из таких применений. RS-триггер — идеальное устройство для борьбы с **дребезгом контактов**. Возможно, вы не знаете, что такое дребезг контактов. Поэтому предлагаю остановиться на этом поподробнее.

В цифровой и микропроцессорной технике редко удается обойтись без различных кнопок или контактов. С их помощью на микро-

процессорное устройство подаются различные команды, реализуются, например, разнообразные датчики. Применение механических контактов приносит дополнительную проблему. Как бы качественно ни был выполнен контакт, он никогда не замыкается и не размыкается мгновенно. В момент замыкания, когда два контакта еще только-только коснулись друг друга и еще не плотно прижаты, происходит досадное явление, называемое **дребезгом**.

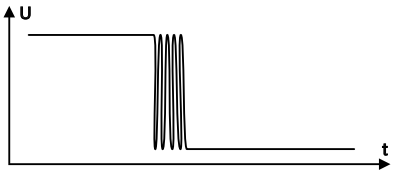


Рис. 1.13. *Дребезг контактов*

Дребезг представляет собой многократное замыкание и размыкание цепи. В результате на вход микропроцессорного устройства поступает не единичный перепад напряжения, а целая пачка импульсов. Примерная форма сигнала на таких контактах в момент замыкания показана на **рис. 1.13**.

Цифровые микросхемы обладают настолько большим быстродействием, что для них такая пачка импульсов выглядит как несколько нажатий клавиши. Если бы не применялись антидребезговые устройства, то мы никогда бы не смогли набрать текст на клавиатуре компьютера. При нажатии на каждую клавишу выскакивала бы не одна, а несколько одинаковых букв. Существует множество схемных и программных решений, позволяющих избавиться от дребезга контактов. Одно из таких решений основано на применении RS-триггера.

На **рис. 1.14** показана схема антидребезгового устройства на основе RS-триггера. Такая схема применяется в том случае, когда кнопка или датчик выполнены в виде группы переключающихся контактов. Как видно из схемы, на оба входа RS-триггера через токоограничивающие резисторы подано напряжение питания. Благодаря этому, на том входе RS-триггера, который не подключен в данный момент к подвижному контакту, присутствует сигнал логической единицы (входное сопротивление логической микросхемы обычно столь велико, что оно не влияет на величину входного напряжения).

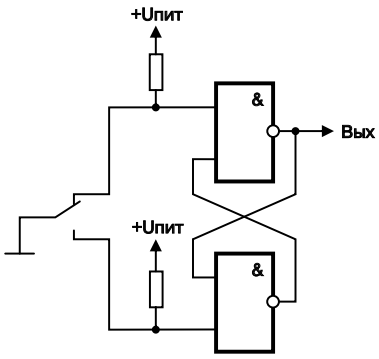


Рис. 1.14. *Схема антидребезга на основе RS-триггера*

Если подвижный контакт замыкает вход на общий провод, то напряжение на нем падает до нуля. А это соответствует низкому логическому уровню. При нажатии и отпускании кнопки (срабатывании датчика) подвижный контакт соединяет с общим проводом то

один, то другой вход RS-триггера. При этом триггер переключается из одного устойчивого положения в другое. Допустим, подвижный контакт переходит в нижнее по схеме положение.

В момент замыкания контактов происходит их дребезг. Как только на вход триггера приходит первый отрицательный импульс из пачки импульсов, обусловленных дребезгом, триггер переключается, и на выходе устройства устанавливается логический ноль. Остальные импульсы уже не изменят состояния триггера.

Это состояние изменится на обратное только тогда, когда подвижный контакт сначала разомкнется с нижним по схеме контактом, преодолеет расстояние от нижнего контакта до верхнего, а затем замкнется с верхним. Как только на верхний по схеме вход RS-триггера поступит первый отрицательный импульс, наш триггер переключится, и на выходе устройства появится логическая единица. В единичном состоянии триггер будет находиться до тех пор, пока контакт опять не переключится в нижнее положение.

1.6. Хранение информации

Устройство и работа D-триггера

В **разделе 1.5** мы узнали, что для хранения информации можно использовать даже простейший RS-триггер. В цифровой и вычислительной технике для этой цели чаще используются другие, более совершенные триггеры, из которых даже составляют целые регистры. Начнем с отдельных триггеров. Пожалуй, самый распространенный вид триггера — это так называемый **D-триггер**. Схемное обозначение D-триггера приведено на **рис. 1.15**.

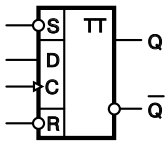


Рис. 1.15. *Схемное обозначение D-триггера*

Главным атрибутом D-триггера являются два новых входа: **D-вход** и **C-вход**. Входы R и S имеют то же самое назначение, что и у RS-триггера (для сброса и установки). Как у любого другого триггера, у D-триггера имеются два выхода: прямой и инверсный. Следует заметить, что наличие RS-входов, так же как и инверсного выхода, необязательно. Рассмотрим логику работы D-триггера. Для начала разберемся с новыми для нас входами.

Вход D — это вход данных (от английского DATA). В процессе работы на этот вход подается логический уровень, который необходимо записать в D-триггер.

Вход C называется тактовым. На него поступает тактовый импульс, синхронизирующий запись данных.

Обратите внимание, что на условном обозначении триггера тактовый вход отмечен стрелкой в виде маленького треугольника. Такой треугольник означает, что данный вход импульсный. До сих пор мы имели дело с **потенциальными входами**. Потенциальный вход реагирует на потенциал поступающего на него сигнала. Про такой вход говорят: **срабатывает при поступлении логической единицы**. Или **срабатывает от логического нуля.**

Импульсный вход не чувствителен к уровню сигнала. Такой вход срабатывает в момент перехода от одного уровня к другому. Про такие входы говорят: **срабатывает по переднему фронту** (то есть при переходе с нуля на единицу) или **срабатывает по заднему фронту** (то есть при переходе от единицы к нулю). Иногда применяют другие технические термины для описания работы импульсного входа. В литературе можно прочитать: «вход срабатывает по фронту сигнала» или «вход срабатывает по спаду сигнала».

Теперь рассмотрим подробнее **логику работы D-триггера**. Для переключения триггера в нужное нам состояние сначала на вход D необходимо подать соответствующий логический сигнал. Для записи единицы на вход D подаем единицу, для записи нуля — ноль. Затем на вход C необходимо подать тактовый импульс. По спаду этого импульса триггер установится в нужное нам состояние (сигнал на D-входе запишется в триггер). Такая логика работы D-триггера делает его очень удобным устройством для хранения одного бита цифровой информации (одного разряда двоичного числа).

Параллельный регистр

Для хранения двоичного числа, состоящего из большего количества разрядов, используют несколько параллельно соединенных D-триггеров. На **рис. 1.16** показана схема, предназначенная для хранения четырехразрядного двоичного числа.

Такая схема называется **параллельным регистром**. Для того, чтобы сохранить какое-либо число в таком регистре, нужно подать это число поразрядно на входы D0—D3. Затем на вход C схемы подается импульс записи. По заднему фронту этого импульса число записывается в регистр. Причем каждый разряд числа записывается в свой отдельный D-триггер. Записанное в регистр число можно считывать с выходов Q0—Q3.

В схеме регистра присутствует также вход сброса $\overline{R}$. Он объединяет входы $\overline{R}$ всех триггеров и используется для начальной установки всех разрядов регистра в нулевое состояние. В цифровой технике это называется «начальная установка».

В реальных микропроцессорных устройствах чаще используются восьмиразрядные параллельные регистры. На **рис. 1.17** изображено

схемное обозначение одного из таких регистров. Его внутренняя структура и назначение выводов аналогичны структуре и назначению выводов регистра, изображенного на **рис. 1.16**.

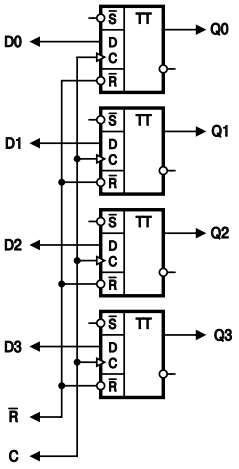


Рис. 1.16. *Простейший параллельный регистр*

Параллельный регистр с расширенными возможностями

Более сложный регистр изображен на **рис. 1.18**. Это регистр приспособлен для работы с параллельной шиной данных. Для этого в регистр введены два новых входа:

- вход выбора микросхемы ($\overline{CS}$);
- вход перевода выходов в высокоимпедансное состояние (OE).

Разберемся подробнее с этими новыми входами и режимами работы. Вход выбора микросхемы $\overline{CS}$ (Chip Select) предназначен для ее включения и выключения в разные моменты времени. Такие входы можно часто встретить у микросхем, предназначенных для микропроцессорной техники. Особенно в больших многофункциональных микросхемах. Наличие таких входов позволяет соединять несколько подобных микросхем параллельно по входам, но работать с каждой микросхемой по отдельности. В случае параллельного соединения одноименных входов данные будут записаны только в тот из регистров, на входе $\overline{CS}$ которого в момент записи будет присутствовать низкий логический уровень. Состояние остальных регистров останется неизменным.

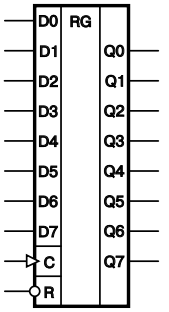


Рис. 1.17. *Восьмиразрядный параллельный регистр*

Вход OE, напротив, используется при параллельном объединении нескольких регистров по их выходам. Такое объединение возможно только в том случае, если в каждый момент времени будут работать выходы только одной из микросхем. Выходы остальных параллельно соединенных микросхем должны уметь автоматически отключаться от схемы. Для этой цели микросхема, изображенная на **рис. 1.18,** имеет специальный режим.

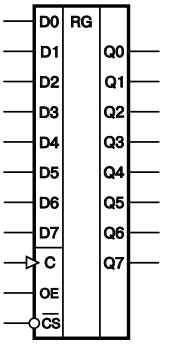


Рис. 1.18. *Параллельный регистр с расширенными возможностями*

В этом режиме все выходы микросхемы отключаются и не влияют на работу остальной схемы. Такое состояние выходов называется высокоим-

педансным. **Импеданс** — это полное сопротивление цепи. Если импеданс высокий, то можно считать, что соответствующий выход просто отключен. Микросхема переводит свои выходы в высокоимпедансное состояние при подаче логической единицы на вход OE. Если же на вход OE подать логический ноль, то выходы микросхемы перейдут обратно в рабочее состояние.

Устройство и работа JK-триггера

Ну, и в заключении этого раздела хочу опять вернуться к триггерам и описать еще один вид. Это, пожалуй, самый сложный из триггеров. Называется он **JK-триггер**. Условное обозначение такого триггера приведено на **рис. 1.19**. Как видно из **рис. 1.19**, JK-триггер сильно напоминает D-триггер. Но вместо одного D-входа такой триггер имеет два новых, пока не известных нам входа, которые имеют обозначение J и K.

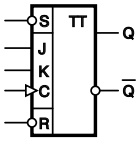


Рис. 1.19. *JK-триггер*

В общем и целом, входы J и K частично выполняют те же функции, что и D-вход. Но логика работы такого триггера более сложна. Если на J-вход подать сигнал логической единицы, а на K-вход — сигнал логического нуля, то по спаду тактового сигнала на входе C триггер установится в единичное состояние.

Если на J подать логический ноль, а на K — логическую единицу, то по спаду тактового сигнала триггер установится в нулевое состояние. Если на входы J и K одновременно подать логическую единицу, то по каждому спаду тактового импульса триггер будет переключаться в противоположное состояние. То есть, с единицы в ноль и с нуля в единицу. И, наконец, если и на J, и на K подать логический ноль, то триггер перестанет реагировать на тактовые импульсы, и его состояние будет оставаться неизменным.

На первый взгляд логика работы триггера чересчур сложна, и область применения таких триггеров неочевидна. Однако виртуозы схемотехники умудряются при помощи JK-триггеров создавать более компактные и рациональные схемы делителей и счетчиков. А вот что такое счетчики и делители, мы узнаем из следующего раздела.

1.7. Счетчики

Работа делителя частоты

Счетчиками в цифровой технике называются специальные элементы, позволяющие подсчитывать число поступивших на вход импульсов. Понятие «счетчик импульсов» тесно связано с понятием «делитель частоты». По сути дела, это одно и то же устройство. Но рассмотрим все по порядку.

В качестве простейшего **делителя частоты** может выступать рассмотренный в предыдущей главе JK-триггер (см. **рис. 1.19**). Для того, чтобы этот триггер работал как делитель, нужно на оба входа J и K подать высокий логический уровень. Теперь, если на вход C подать импульсный сигнал некоторой постоянной частоты, то по спаду каждого входного импульса триггер будет переключаться в противоположное состояние.

В результате на выходе JK-триггера мы получим другой сигнал с частотой следования импульсов в два раза меньшей, чем частота импульсов на его входе. Этот процесс наглядно показан на **рис. 1.20**. Как видно из рисунка, период сигнала на выходе делителя ровно в два раза больше периода входного сигнала. А частота выходного сигнала, соответственно, в два раза ниже входного.

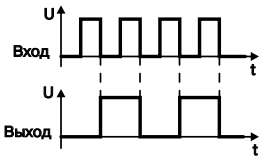


Рис. 1.20. *Деление частоты*

Второй вариант делителя частоты приведен на **рис. 1.21**. Он построен на основе D-триггера. Для того, чтобы перевести D-триггер в счетный режим, нужно соединить инверсный выход триггера $\overline{Q}$ с его D-входом так, как это показано на **рис. 1.21**. Теперь, если подать сигнал на вход C, такая схема тоже будет работать как делитель. Выходной сигнал такого делителя снимается с выхода Q триггера.

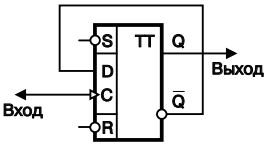


Рис. 1.21. *Простейший делитель частоты*

Рассмотрим подробнее работу этой схемы. Предположим, что после включения триггер установился в единичное состояние. Это означает, что на инверсном выходе триггера ($\overline{Q}$) присутствует логический ноль. Этот ноль поступает на D-вход. Подадим на вход делителя некоторый цифровой сигнал, такой же, как мы подавали и в предыдущем случае (см. **рис. 1.20**).

По спаду первого входного импульса D-триггер перейдет в нулевое состояние, так как на его D-входе сигнал логического нуля. После этого на инверсном выходе триггера устанавливается логическая единица. Поэтому по спаду следующего входного импульса триггер переключится в единичное состояние. И так далее.

Результат работы делителя на D-триггере точно такой же, как и делителя на JK-триггере, и выходной сигнал нового варианта так же полностью соответствует **рис. 1.20**. Следует заметить, что в настоящее время JK-триггеры применяются довольно редко. Гораздо большее распространение благодаря своей простоте и универсальности получили D-триггеры.

Делители широко используются в цифровой технике. Цепочка последовательно соединенных D-триггеров позволяет получить сигналы требуемой частоты путем деления импульсов задающего генератора.

Пример

Соединенные последовательно два делителя позволят получить сигнал с частотой в четыре раза меньшей, чем входная. Трехкаскадный делитель (три последовательно соединенных D-триггера) дадут деление на восемь. Четыре каскада будут делить на шестнадцать. И так далее.

На **рис. 1.22** изображена схема четырехкаскадного делителя частоты на D-триггерах. Импульсы тактового генератора поступают на вход первого каскада деления. Если частота сигнала на входе равна f, то на выходах делителя мы получим сигналы со следующими частотами: Q0 — f/2; Q1 — f/4; Q2 — f/8; Q3 — f/16.

Счетчики прямого счета

Приведенную на **рис. 1.22** схему можно использовать не только в качестве делителя частоты, но и в качестве **счетчика входных импульсов**. Представьте, что выходы Q0—Q3 — это разряды некоторого двоичного числа. Выход Q0 — это младший разряд, а выход Q3 — самый старший.

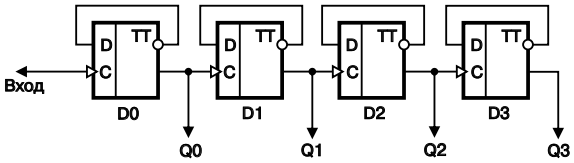


Рис. 1.22. *Четырехкаскадный делитель частоты*

Предположим, что перед началом счета все четыре триггера установлены в нулевое состояние. На вход схемы поступает некоторое количество импульсов. Входящие в схему триггеры будут переключаться согласно описанному выше алгоритму. Состояние триггеров в процессе счета показано в **табл. 1.1**. Как видно из таблицы, после прихода первого входного импульса триггер D0 переходит в единичное состояние. После прихода второго импульса D0 возвращается в ноль, зато в единичное состояние переходит D1.

Дальнейшее поведение всех четырех триггеров хорошо видно из таблицы. А теперь внимательно посмотрите, что у нас получилось. Если воспринимать совокупность цифровых сигналов на выходах счетчика как четырехразрядное двоичное число, то мы видим перед собой последовательность чисел от 0000 до 1111. Десятичный эквивалент этих чисел показан в правой крайней колонке **табл. 1.1**.

Итак, перед началом счета на выходе делителя — ноль. После прохождения первого импульса на выходе — единица, после второго — два,

Логика работы делителя *Таблица 1.1*

Входные импульсы	Состояние выходов				Десятичный эквивалент
	Q3	Q2	Q1	Q0	
Не было ни одного	0	0	0	0	0
Один импульс	0	0	0	1	1
Два импульса	0	0	1	0	2
Три импульса	0	0	1	1	3
-	0	1	0	0	4
-	0	1	0	1	5
-	0	1	1	0	6
-	0	1	1	1	7
-	1	0	0	0	8
-	1	0	0	1	9
-	1	0	1	0	10
-	1	0	1	1	11
-	1	1	0	0	12
-	1	1	0	1	13
Четырнадцать импульсов	1	1	1	0	14
Пятнадцать импульсов	1	1	1	1	15

и так далее. Каждый входной импульс увеличивает значение двоичного числа на выходе счетчика на одну единицу. Поэтому в любой момент времени счетчик содержит число, равное количеству импульсов, пришедших к этому моменту на его вход.

Максимальное число импульсов, которое может посчитать счетчик, схема которого изображена на **рис. 1.22,** — это 16. После прихода шестнадцатого импульса счетчик вернется в нулевое состояние. Счетчики широко применяются в цифровой технике в том случае, когда необходимо подсчитать количество каких-либо импульсов. Причем совсем необязательно, чтобы входные импульсы поступали равномерно с постоянным периодом. Это могут быть одиночные импульсы. Например, импульсы с какого-нибудь датчика, кнопки и т. п.

Счетчики с обратным отсчетом

Кроме счетчиков прямого отсчета, к которым относится схема, изображенная на **рис. 1.22**, существуют **счетчики с обратным отсчетом** (иногда такой счетчик называют **инверсным**). В таком счетчике при поступлении каждого входного импульса содержимое уменьшается на единицу. Кроме того, бывают задачи, для которых требуются универсальные счетчики, которые могут считать как в прямом, так и в инверсном направлении. Если задаваться задачей построения таких счетчиков на отдельных триггерах и логических элементах, то мы получим довольно сложную схему. На практике используют специальные микросхемы — счетчики импульсов. Современная промышленность

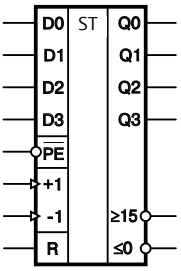


Рис. 1.23. *Реверсивный счетчик*

предлагает большой ассортимент таких микросхем. На **рис. 1.23** изображена микросхема К555ИЕ7. Это одна из микросхем 555 серии, которая широко выпускалась в свое время в СССР, и сейчас ее можно свободно найти в продаже на радиорынках стран СНГ.

Микросхема 555ИЕ7 — это реверсивный четырехразрядный счетчик/делитель с возможностью предустановки. Он имеет два счетных входа, обозначенных как «+1» и «–1». По спаду каждого импульса на входе «+1» содержимое счетчика увеличивается на единицу. По спаду каждого импульса на входе «–1» содержимое счетчика уменьшается на единицу. Счетчик имеет прямые выходы всех своих разрядов: Q0—Q3. Вход сброса $\overline{R}$ служит для установки всех разрядов счетчика в нулевое состояние.

Еще одно полезное свойство описываемого счетчика — это наличие режима предустановки. Используя этот режим, можно в любой момент записать во все разряды счетчика любое четырехразрядное двоичное число. Для этого счетчик имеет несколько дополнительных входов. **Во-первых,** это входы данных D0—D3. А, во вторых, это вход предустановки $\overline{PE}$. Предустановка счетчика осуществляется следующим образом. Сначала на входы D0—D3 подается код, который требуется записать в разряды счетчика. Затем на вход $\overline{PE}$ подается сигнал низкого логического уровня. По этому сигналу код, установленный на входах D0—D3, запишется в счетчик и тут же появится на его выходах Q0—Q3. Дальнейший счет импульсов будет производиться уже от этого нового значения.

Выходы «≥15» и «≤0» — это **выходы переполнения**. Они используются при последовательном соединении нескольких таких счетчиков. В процессе счета уровень сигнала на обоих этих выходах равен единице. На выходе «≥15» логический ноль появляется в том случае, если в процессе прямого счета содержимое счетчика достигнет своего максимального значения 1111_2, и на вход «+1» поступит очередной счетный импульс. Выход «≤0» работает аналогично, но при обратном счете. Сигнал логического нуля появляется на этом выходе в тот момент, когда счетчик досчитает до своего нижнего предела — 0000_2, и на вход «–1» поступит очередной счетный импульс.

При последовательном соединении двух счетчиков выходы «≥15» и «≤0» первого счетчика соединяется соответственно с входами «+1» и «–1» второго. В результате, соединив последовательно два таких счетчика, мы получим восьмиразрядный реверсивный счетчик, который также будет иметь возможность предустановки. Таким способом можно соединять последовательно любое количество счетчиков 555ИЕ7.

Одно из применений микросхемы 555ИЕ7 — построение делителей с переменным коэффициентом деления. Простой делитель частоты, рассмотренный в начале этой главы, дает фиксированный набор коэффициентов деления, который к тому же можно выбирать лишь из ограниченного ряда значений, являющихся степенью числа 2.

Делители с переменным коэффициентом деления

В цифровой и микропроцессорной технике часто требуются делители с произвольным коэффициентом деления. При этом желательно, чтобы коэффициент деления можно было оперативно менять. На **рис. 1.24** изображена схема делителя с программируемым коэффициентом деления на основе реверсивного счетчика К555ИЕ7. Для хранения коэффициента деления используется специальный четырехразрядный параллельный регистр, обозначенный на схеме как DD1. Коэффициент деления такого делителя может изменяться от 1 до 15.

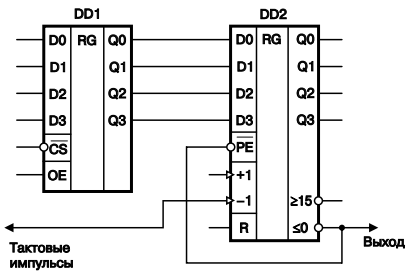


Рис. 1.24. *Делитель с переменным коэффициентом деления*

Работа счетчика начинается с установки всех его разрядов в ноль при помощи входа R. Обратите внимание на то, что в микросхеме К555ИЕ7 используется прямой, а не инверсный вход сброса. Поэтому сброс происходит при подаче на этот вход сигнала логической единицы. После того, как счетчик сброшен, для нормальной работы счетчика на вход R должен быть подан нулевой уровень.

Входной сигнал поступает на вход «–1». Поэтому счетчик работает в режиме обратного счета. Поэтому первый же входной импульс после сброса счетчика вызовет сигнал переполнения на выходе «≤0». Этот импульс поступит на вход $\overline{\text{PE}}$. В результате в счетчик будет записано двоичное число с выхода регистра DD1. Это число соответствует выбранному коэффициенту деления. Допустим, что в регистр DD1 мы записали число 10 (1010_2). Тогда именно это число будет записано в разряды счетчика DD2.

Каждый последующий входной импульс будет уменьшать содержимое счетчика на единицу. Так будет продолжаться до тех пор, пока содержимое счетчика снова не уменьшится до нуля. Для этого потребуется как раз 10 тактовых импульсов. По приходу одиннадцатого импульса на выходе «≤0» снова появится сигнал переполнения, и в счетчик будет опять записано число десять из регистра DD1.

Описанный процесс будет повторяться все время, пока приходят входные импульсы. Период следования импульсов на выходе «≤0», а,

значит, и на выходе всей схемы в нашем случае будет в 11 раз больше периода входных сигналов. А частота выходных импульсов будет, соответственно, в 11 раз меньше. То есть наш счетчик будет делить на 11. Записывая в регистр DD1 различные значения, можно легко менять коэффициент деления описанной схемы. Забегая вперед скажу, что запись числа в регистр коэффициента деления может производить микропроцессор. В этом случае мы можем создать делитель, управляемый от микропроцессора.

Таймеры

Подобную схему можно использовать также для формирования различных интервалов времени. Если на вход «–1» подавать тактовые импульсы фиксированной частоты, а в качестве управляющего входа использовать вход R, то на выходе мы можем получать импульс заданной длительности. И эту длительность можно программировать, записывая в регистр D1 различные коэффициенты.

*Схемы, предназначенные для формирования различных интервалов времени, называются **таймерами**.*

Обычно одни и те же цифровые элементы при определенном способе включения могут с успехом выступать в любой из трех описанных выше ролей: либо как делители, либо как счетчики, либо как таймеры.

Пример

Микроконтроллеры серии AVR имеют от одного (в микросхеме AT90S1200) до четырех (в микросхеме ATmega128) встроенных таймеров/счетчиков. Это позволяет при формировании временных интервалов обойтись без внешних таймеров.

Существуют и специализированные микросхемы-таймеры. Например, микросхема К580ВИ53 — это универсальный программируемый трехканальный счетчик-таймер. Такая микросхема имеет множество режимов работы, которые должны выбираться программным путем при помощи микропроцессора.

Современные микроконтроллеры, или, как их еще называют, **однокристальные микроЭВМ**, обычно всегда содержат в своем составе один или несколько встроенных таймеров-счетчиков.

1.8. Дешифраторы

Устройство и прицип действия дешифратора

Еще один элемент, без которого не обойтись при изучении микропроцессорной техники, — это **дешифратор цифровых сигналов**. Существует много разных типов дешифраторов. В общем случае дешифратор — это устройство, преобразующее цифровой сигнал, представленный в какой-либо одной из кодировок, в другую, незакодированную форму. Нас в данном случае будет интересовать классический линейный дешифратор. Схемное обозначение одного из вариантов такого дешифратора приведено на **рис. 1.25**. Описываемый дешифратор имеет три входа данных D0, D1 и D2, вход выбора микросхемы $\overline{CS}$, а также восемь выходов, обозначенных цифрами от 0 до 7.

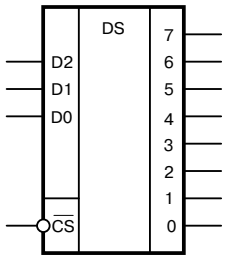


Рис. 1.25. *Простейший дешифратор*

Логика работы микросхемы такова: на входы данных микросхемы подается цифровой код. В данном случае — это любое трехразрядное двоичное число. Смысл работы такого дешифратора — выдать активный сигнал только на одном из своих выходов. На том выходе, номер которого соответствует двоичному коду, присутствующему на его входах D0—D2.

В большинстве современных дешифраторов активным сигналом на выходе считается низкий логический уровень. Это значит, что при поступлении на входы D0—D1 сигнала 000_2, на выходе «0» будет логический ноль, а на всех остальных выходах — единица. Состояние выходов дешифратора при других значениях входного сигнала приведено в **табл. 1.2**.

Табл. 1.2 — это таблица истинности данного дешифратора. Важное значение имеет вход выбора микросхемы $\overline{CS}$. Этот вход позволяет включить или выключить всю микросхему. Микросхема работает только в том случае, если на вход $\overline{CS}$ дешифратора подан разрешающий нулевой сигнал. В противном случае микросхема выключается и на всех ее выходах устанавливается сигнал логической единицы.

Главное **назначение** линейного дешифратора — функция выбора одного из нескольких электронных устройств. **Например**, выбор одной из нескольких микросхем памяти. Каждая такая микросхема должна иметь свой собственный вход $\overline{CS}$. К каждому выходу дешифратора подключается одна такая микросхема. Вернее, подключается ее вход $\overline{CS}$.

Теперь, если на входы D0—D2 дешифратора подать номер выбираемой микросхемы, она включится, а остальные семь микросхем отклю-

Таблица истинности дешифратора *Таблица 1.2*

Входы				Выходы							
D2	D1	D0	CS	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	1	1	1	0
0	0	1	0	1	1	1	1	1	1	0	1
0	1	0	0	1	1	1	1	1	0	1	1
0	1	1	0	1	1	1	1	0	1	1	1
1	0	0	0	1	1	1	0	1	1	1	1
1	0	1	0	1	1	0	1	1	1	1	1
1	1	0	0	1	0	1	1	1	1	1	1
1	1	1	0	0	1	1	1	1	1	1	1
x	x	x	1	1	1	1	1	1	1	1	1

чатся. При выключении самого дешифратора отключаются и все подключенные к нему микросхемы.

Селектор памяти ячеек ОЗУ

Хороший пример использования дешифратора — селектор ячеек памяти ОЗУ. Схема простейшего модуля ОЗУ, состоящего всего из четырех ячеек, приведена на **рис. 1.26**. В качестве ячеек памяти в схеме используются параллельные регистры с возможностью перевода в высокоимпедансное состояние. Такие регистры нам уже знакомы по **разд. 1.6** (см. **рис. 1.18**).

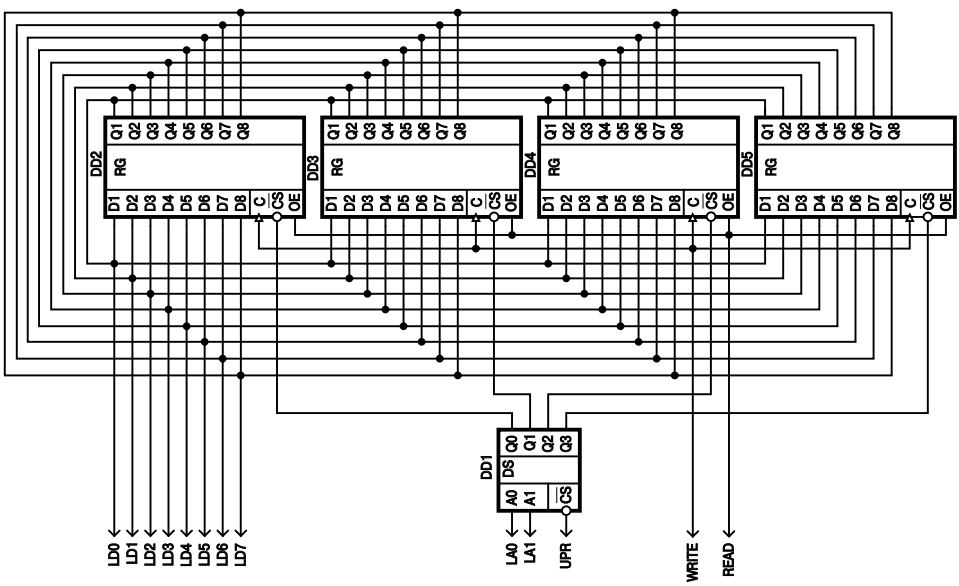


Рис. 1.26. *Схема простейшего модуля ОЗУ*

Рассмотрим внимательно схему на **рис. 1.26**.

Линии LD0–LD7 — это восьмиразрядная шина данных. Она используется как для записи чисел в память, так и для чтения из нее.

Входы LA0, LA1 — это так называемые входы адреса.

Вход UPR — вход выбора для всего устройства.

Входы WRITE и READ, соответственно, — вход команды записи и вход команды чтения.

Входы UPR, WRITE и READ — инверсные. То есть в отсутствии сигнала на каждом из них должен присутствовать высокий логический уровень. Активным сигналом для этих входов является логический ноль.

Для того, чтобы записать число в одну из ячеек такого ОЗУ, нужно сначала на вход UPR подать нулевой сигнал (выбрать устройство). Затем на лини LD0—LD7 от внешнего источника цифрового сигнала подать восьмиразрядное двоичное число, предназначенное для записи. Затем на линии LA0, LA1 подается число, соответствующее номеру нужной ячейки памяти (адрес ячейки). Номер выбранной ячейки поступает на дешифратор DD1.

Предположим, что мы хотим выбрать нулевую ячейку памяти. Для этого мы пододим на входы LA0, LA1 сигнал 00_2. В результате на выходе Q0 дешифратора появляется нулевой сигнал, а на всех остальных его выходах — единичный. С выхода Q0 дешифратора нулевой сигнал поступает на вход $\overline{CS}$ параллельного регистра DD2 и включает его. Все остальные регистры остаются отключенными. Теперь для того, чтобы записать число в выбранную ячейку памяти, нужно подать короткий нулевой импульс на вход WRITE. Он поступит на входы C всех регистров. Но число запишется только в регистр DD2.

Как видно из схемы на **рис. 1.26**, лини D0—D7 объединяют в себе не только входы регистров, но и их выходы. Однако такая схема включения не мешает работе устройства. Это достигается благодаря тому, что все регистры имеют выходы с тремя состояниями. К двум обычным состояниям (ноль и единица) добавлено третье — высокоимпендансное. Это состояние включается при подаче на вход OE любого регистра логической единицы.

В режиме чтения информации лини LD0—LD7 используются как выходы. Для того, чтобы прочитать число из любой ячейки памяти, нужно сначала подать адрес ячейки на входы LA0, LA1. Это приводит к тому, что нужный нам регистр включается. Включается точно так же, как это происходило при записи.

Теперь для того, чтобы прочитать число из выбранной ячейки, достаточно подать на вход READ сигнал логического нуля. В результате выходы выбранного регистра перейдут из высокоимендансного в рабочее состояние. На них появится записанное в регистр число, кото-

рое поступит на выход всей схемы. Все остальные регистры останутся отключенными и не будут мешать процессу чтения.

Каскадирование дешифраторов

Прежде, чем покончить с дешифраторами хочу рассмотреть вопрос их каскадирования. Благодаря наличию входа $\overline{CS}$, несколько дешифраторов можно объединять вместе, образуя составной дешифратор, имеющий большее число входов и выходов. Пока мы имели дело с двумя видами дешифраторов. Первый дешифратор (**рис. 1.25**) имел три входа и восемь выходов. Дешифратор, который мы использовали в примере схемы ОЗУ (**рис. 1.26**), имел всего два входа и четыре выхода. Логика работы любого такого дешифратора одна и та же. Различие состоит лишь в количестве входных и выходных разрядов.

Для того, чтобы в краткой форме обозначить характеристики конкретного дешифратора, иногда применяют следующее обозначение: «**Дешифратор 2X4**» или «**Дешифратор 3X8**». На **рис. 1.27** показан способ, как при помощи каскадного соединения нескольких дешифраторов создать новый дешифратор с формулой «5X32». Первый каскад состоит из одного дешифратора (DD1). Этот дешифратор управляет четырьмя другими дешифраторами (DD2—DD5), которые составляют второй каскад.

Все дешифраторы, рассмотренные нами до сих пор, относятся к разряду **полных дешифраторов**. Поясню, что это такое. Если дешифратор имеет пять входов, то максимальное число значений, которые могут при-

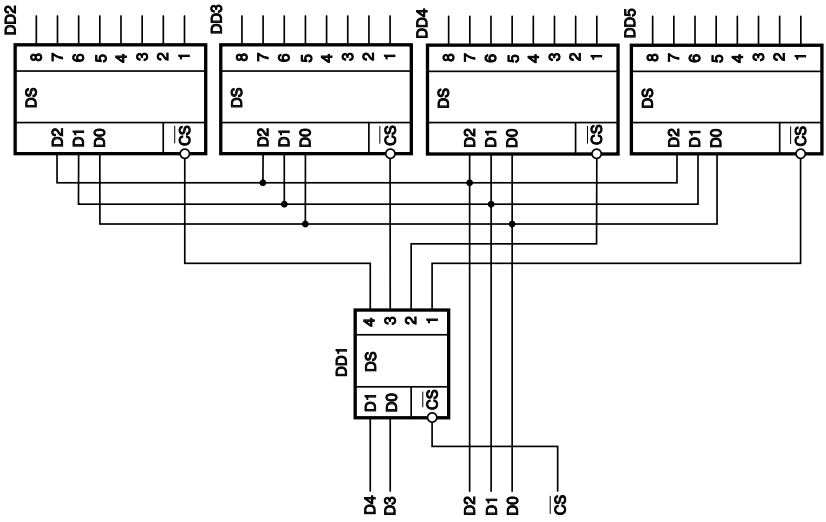


Рис. 1.27. *Каскадная схема включения дешифраторов*

нимать эти входы, равно 32. Если при этом дешифратор имеет 32 выхода, то при подаче на его входы любого возможного числа, на одном из выходов обязательно появится активный уровень сигнала. В некоторых случаях наличие всех возможных выходов не обязательно.

1.9. Мультиплексоры

Ну и последний элемент, на котором я хотел бы остановиться, — это мультиплексоры. **Мультиплексор** — это устройство, обратное простому линейному дешифратору. Вообще, термин «мультиплексирование» означает собирание сигнала с нескольких разных входов. А точнее — переключение нескольких входов на один выход. Схема одного из вариантов мультиплексора изображена на **рис. 1.28**.

Любой мультиплексор имеет информационные входы, один информационный выход, а также входы выбора адреса для управления мультиплексированием. Так, мультиплексор, изображенный на **рис. 1.28**, имеет восемь входов данных (D0—D7) и три входа адреса (A0—A2). На адресные входы должен подаваться цифровой код, соответствующий номеру информационного входа, сигнал с которого должен поступить на выход. То есть, если на входы A0—A2 подать код 000H, то с выходом будет соединен вход D0. Если на адресные входы подать число 001H, то с выходом соединится вход D1. И так далее.

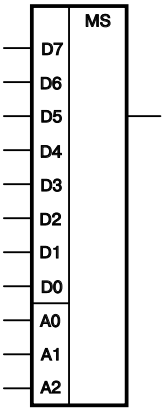


Рис. 1.28. *Схема мультиплексора*

Мультиплексоры могут иметь разное количество информационных входов и входов адреса. Но на их количество распространяется тот же самый закон, который

Пример

Дешифратор для работы с так называемыми двоично-десятичными числами. Двоично-десятичное число — это число, записанное в двоичной форме, но принимающее значение от 0 (0000_2) до 10 (1010_2). При работе с такими числами используют неполный дешифратор 4X10. Такой дешифратор имеет четыре входа и всего десять выходов. При подаче на входы такого дешифратора двоичных чисел в диапазоне от 0000_2 до 1010_2 активизируются выходы «0» — «9».

При этом подразумевается, что числа выше чем 1010_2, на входы дешифратора подаваться не будут. Однако ничего не мешает подать на входы дешифратора эти числа. Как будет вести себя неполный дешифратор в таком случае, определяется его разработчиком. Чаще всего в этом случае все выходы дешифратора остаются неактивными. Промышленность выпускает несколько видов неполных дешифраторов. Например, К555ИД1, К555ИД6, К555ИД10.

действует при определении соотношения входов и выходов дешифратора. Число информационных входов должно быть равно максимально возможному количеству адресов, которые можно подать на входы адреса. При двух адресных входах должно быть четыре информационных, при трех адресных — восемь информационных, при четырех — шестнадцать. И так далее. Если информационных входов окажется меньше, мультиплексор будет неполным.

Описанные выше мультиплексоры называются **цифровыми**. Кроме цифровых мультиплексоров, существуют **аналоговые мультиплексоры**. Отличие между этими двумя видами мультиплексоров в способе соединения входа и выхода. Цифровой мультиплексор может работать лишь с цифровым сигналом. Этот сигнал должен подаваться на один из входов, а сниматься с выхода. В обратную сторону сигнал не распространяется.

Аналоговый мультиплексор работает по принципу цифрового ключа. Один из входов соединяется с выходом напрямую. Две разные электрические цепи объединяются в одну общую цепь. В случае использования аналогового мультиплексора в качестве информационных сигналов можно использовать как цифровые, так и аналоговые сигналы, подавать их как на вход, так и на выход мультиплексора. В последнем случае выход становится входом. **Примером** аналогового мультиплексора может служить микросхема К561КП2.

На этом мы заканчиваем изучение основ цифровой логики и переходим к следующему этапу — микропроцессорам.

ГЛАВА 2

КАК РАБОТАЮТ МИКРОПРОЦЕССОРЫ И МИКРОКОНТРОЛЛЕРЫ

2.1. Типовая схема микропроцессорной системы

Структурная схема типичной микропроцессорной системы

Основным действующим элементом современной микропроцессорной системы является **микроконтроллер**. Однако для того, чтобы понять основополагающие принципы работы, сначала все же необходимо остановиться на **микропроцессоре**. Сразу нужно сказать, что микропроцессор не работает сам по себе. Микропроцессор — это всего лишь часть той или иной микропроцессорной системы.

Кроме собственно микропроцессора, в состав микропроцессорной системы входят и другие, не менее важные элементы. На **рис. 2.1** приведена обобщенная структурная схема типичной микропроцессорной системы. Рассмотрим детально, как она устроена. Как вы видите, названия всех элементов системы даны как в русском, так и в английском варианте.

Русские названия когда-то пытались внедрить в практику в бывшем СССР. Поэтому и сейчас они иногда встречаются в литературе. Однако в настоящее время более широкое распространение получили английские, а точнее — международные наименования. Каждое наименование как в русском, так и в английском варианте представляет собой определенное сокращение полного названия элемента. Ниже приведена их расшифровка.

- **CPU** (Central Processing Unit) — центральное процессорное устройство (ЦПУ).
- **RAM** (Random Access Memory) — устройство с произвольным доступом, или оперативное запоминающее устройство (ОЗУ).
- **ROM** (Read Only Memory) — память только для чтения, или постоянное запоминающее устройство (ПЗУ).
- **Port I/O** (Port Input/Output) — порт ввода-вывода.

Теперь рассмотрим все эти элементы подробнее. Что такое процессор и для чего он нужен, вы уже немного знаете из **главы 1** нашей книги. Замечу только, что процессор не всегда был **МИКРОпроцессором**. Были

времена, когда процессор представлял собой одну или даже несколько электронных плат, набитых радиоэлементами.

Виды памяти

Два вида памяти (ОЗУ и ПЗУ) предназначены для хранения информации (данных и программ). Оба вида памяти представляют собой **набор ячеек**, в каждой из которых может храниться одно двоичное число. Деление на постоянную и оперативную память достаточно условно. С точки зрения процессора, оба эти вида памяти практически идентичны. Однако все же между ними есть одно довольно существенное различие.

После того, как информация записана в ОЗУ, она хранится там лишь до тех пор, пока подано напряжение питания. Как только питание будет отключено, информация, записанная в ОЗУ, тут же теряется. Об этом мы уже говорили выше. Классический пример ячейки ОЗУ — это простейший регистр, построенный на D-триггерах (см. **рис. 1.26**).

В такой регистр можно записывать информацию и читать ее оттуда. Однако если отключить, а затем включить питание, то все триггеры, из которых состоят регистры ОЗУ, установятся в случайное состояние. Информация будет утеряна. Современные микросхемы памяти строятся на основе совсем других технологий. Но и по сей день не придумано достаточно быстродействующее устройство памяти, не теряющее информации при выключении питания.

Самая распространенная на сегодняшний день технология построения ОЗУ — это так называемая **динамическая память**. Хранение информации в микросхемах динамической памяти осуществляется при помощи динамически подзаряжаемых миниатюрных емкостей (конденсаторов), выполненных интегральным способом на кристалле кремния.

Каждый конденсатор хранит один бит информации. Если значение бита должно быть равно единице, то схема управления заряжает конденсатор. Если в ячейке должен быть логический ноль, то конденсатор разряжается. Заряженный конденсатор может хранить свой заряд, а, значит, и записанную в него информацию в течение всего нескольких миллисекунд. Для того, что бы информация не потерялась, используют регенерацию памяти.

Специальная схема периодически считывает содержимое каждой ячейки памяти и подзаряжает конденсаторы для тех битов, где записана единица. Для ускорения процесса регенерации все ячейки памяти каждой микросхемы разбиваются на строки. Считывание и обновление производится сразу для целой строки. Для нормальной работы динамического ОЗУ регенерации должна непрерывно работать в течение всего времени, пока включено питание. В современных ОЗУ схема регенерации встраивается внутрь самих микросхем.

Постоянное запоминающее устройство (ПЗУ) предназначено для долговременного хранения информации и не теряет записанную информацию даже после выключения питания. При изготовлении микросхем ПЗУ применяются совершенно другие технологии. На заре микропроцессорной техники микросхемы ПЗУ осуществляли хранение информации благодаря прожиганию внутренних **микроперемычек** на кристалле. Занесенная таким образом информация не могла быть изменена. Если информация устаревала, микросхему просто выбрасывали и заменяли на другую.

На смену однократно программируемым ПЗУ пришли **ПЗУ с ультрафиолетовым стиранием**. Такие микросхемы ПЗУ допускали многократное использование. Пережигаемые перемычки получили возможность восстанавливаться. Перед повторным использованием микросхему нужно было «стереть». То есть восстановить перемычки. Для этого кристалл микросхем подвергался облучению световым потоком ультрафиолетового диапазона, для чего микросхемы снабжались специальным окошечком в верхней части корпуса.

Количество циклов записи-стирания для таких микросхем было ограничено. Микросхемы с ультрафиолетовым стиранием просуществовали достаточно долго. Они и сейчас работают во множестве микропроцессорных устройств, изготовленных на рубеже прошлого и нынешнего веков.

Современные же микросхемы ПЗУ строятся по так называемой **флэш-технологии (Flash)**. Такие микросхемы также основаны на применении специальных пережигаемых перемычек с возможностью восстановления. Но стирание информации в данном случае происходит электрическим путем. Поэтому такие микросхемы еще называют **ЭСПЗУ (электрически стираемые ПЗУ)**. Весь процесс стирания осуществляется внутри микросхемы. Для запуска процесса стирания достаточно подать определенную комбинацию сигналов на ее входы.

Будучи включенными в состав микропроцессорной системы, микросхемы ОЗУ и микросхемы ПЗУ работают как единая память программ и данных. Хотя процессор и работает с обоими видами памяти одинаково, но из ПЗУ он может только читать информацию. Запись информации в ПЗУ невозможна. Если микропроцессор все же попытается произвести запись, то ничего страшного не произойдет. Просто в ячейке останется то, что там было до попытки записи.

Порты ввода-вывода

Без портов теряется весь смысл микропроцессорной системы. Она не может работать сама по себе. Микропроцессор должен чем-то управлять, а иначе зачем он? Через порты ввода процессор получает внешние воз-

Порты ввода-вывода *— это специальные устройства, при помощи которых микропроцессорная система может общаться с внешним миром.*

действия (управляющие сигналы). Например, сигналы от кнопок, датчиков. При помощи портов вывода процессор управляет внешними устройствами (реле, моторами, световыми индикаторами, дисплеями).

Процессор работает с портами ввода-вывода практически так же, как и с ячейками памяти. Работа с портами сводится к тому, что процессор просто читает число из порта ввода или записывает число в порт вывода. В качестве порта вывода чаще всего выступает обыкновенный параллельный регистр. Порт ввода еще проще. Это простая ключевая схема, которая по команде с центрального процессора подает внешние данные на его входы.

Процессор и цифровые шины

Главным управляющим элементом всей микропроцессорной системы является **процессор**. Именно он, за исключением нескольких особых случаев, управляет и памятью, и портами ввода-вывода. Память и порты ввода-вывода являются пассивными устройствами и могут только отвечать на управляющие воздействия.

Для того, чтобы процессор мог управлять микропроцессорной системой, он соединен со всеми ее элементами при помощи цифровых шин. Как мы уже говорили, **шина** — это набор параллельных проводников, по которым передается цифровой сигнал. Эти проводники называются **линиями шины**.

В каждый момент времени по шине передается одно двоичное число. По каждой линии передается один разряд этого числа. В любой микропроцессорной системе имеется, по крайней мере, три основных шины. Все они изображены на **рис. 2.1**, но даны только русскоязычные названия шин.

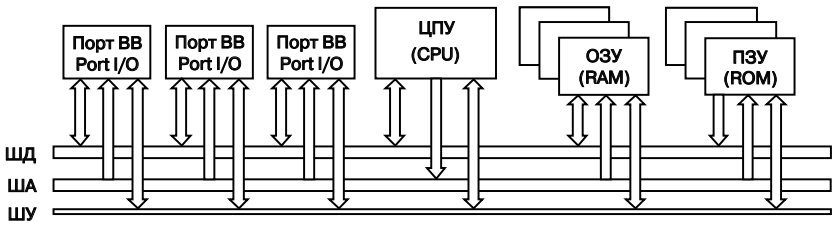


Рис. 2.1. *Структурная схема типовой микропроцессорной системы*

Ниже приведена расшифровка этих названий и их англоязычный эквивалент:

- **ШД** — шина данных (DATA bus);

- **ША** — шина адреса (ADDR bus);
- **ШУ** — шина управления (CONTROL bus).

Все вместе эти три шины образуют системную шину. Рассмотрим подробнее назначение каждой шины.

Шина данных

Шина данных предназначена для передачи данных от микропроцессора к периферийным устройствам, а также в обратном направлении. Разрядность **шины данных** определяется типом применяемого процессора. В простых микропроцессорах шина данных обычно имеет 8 разрядов. Современные процессоры могут иметь шину данных в 16, 32, 64 разрядов. Количество разрядов всегда кратно восьми.

*Двоичное число, имеющее восемь разрядов, называется **байтом**.*

В вычислительной технике байт, по сути, стал минимальной (после бита) единицей информации. Шестнадцатиразрядная шина данных может за раз передавать до двух байтов. 32-разрядная шина передает до четырех байт. 64-разрядная — до восьми. Какой бы ни была разрядность шины, она всегда имеет возможность при необходимости передать всего один байт. И это не случайно. Любой процессор должен иметь возможность записать информацию в одну отдельную ячейку памяти или в один отдельный порт ввода-вывода. А также прочитать информацию из одной ячейки или одного порта.

Шина адреса

Как и шина данных, **шина адреса** представляет собой набор проводников, по которым происходит передача двоичных чисел в электронной форме. Однако, в отличие от шины данных, двоичные числа, передаваемые по шине адреса, имеют другой смысл и назначение. Они представляют собой адрес ячейки памяти или номер порта ввода/вывода, к которому в данный момент обращается процессор. Количество разрядов адресной шины отличается большим разнообразием.

Пример

Микропроцессор серии К580ИК80 имеет 16 разрядов адреса. Это можно считать минимальным количеством для микропроцессора. Процессор Intel 8086, на котором собран компьютер IBM PC-XT, родоначальник всех PC-совместимых персональных компьютеров, имеет 20 разрядов шины адреса. Современные процессоры имеют до 32 разрядов и больше.

От количества разрядов шины адреса зависит то, какое количество ячеек памяти может адресовать процессор. Процессор, имеющий шестнадцатиразрядную шину

Один килобайт в вычислительной технике не равен 1000 байтов, как этого можно было бы ожидать. Число 1000 не является круглым числом в двоичной системе. В двоичной системе круглыми числами удобнее считать степени числа 2. Например, 4, 8, 16, 32, 64 и т. д. Ближайшей степенью двойки для числа 1000 будет 2^{10}, то есть число 1024. Поэтому 1 килобайт равен 1024 байтам. Точно так же 1 мегабайт равен 1024 килобайтам. А один гигабайт равен 1024 мегабайтам.

данных, может обращаться к 2^{16} (то есть к 65536) ячейкам памяти. Это число называется **объемом адресуемой памяти**.

Реальный объем подключенной памяти может быть меньше, но никак не больше этой величины. Если все же есть необходимость в подключении большего объема памяти, применяют специальные схемные ухищрения (переключаемые банки памяти). В каждый момент времени к микропроцессору подключается свой банк памяти. Переключением банков управляет сам микропроцессор.

Объем памяти определяется в байтах. Сколько ячеек памяти, столько и байт. Существует понятие килобайт, мегабайт, гигабайт, терабайт и т. д. Однако в вычислительной технике используется необычный способ подсчета количества байт в килобайте.

Пример

Для адресации 1024 ячеек памяти нужна шина адреса, имеющая ровно 10 разрядов. То есть к адресной шине в 10 разрядов максимально можно подключить 1024 ячейки памяти. Если бы мы подключили 1000 ячеек, то нам все равно пришлось бы использовать 10 разрядов адреса, которые, в этом случае, использовались бы не полностью. Поэтому вы никогда не встретите микросхему памяти, имеющую 1000 ячеек. Именно по этой причине реальный объем памяти любой микропроцессорной системы, даже если она меньше максимально возможной для данной разрядности шины адреса, всегда будет кратным степени двойки.

Такой способ подсчета может показаться странным. Но это только на первый взгляд. На самом деле тут действуют те же закономерности, что и в случае с дешифратором. Вспомните полный и неполный дешифраторы (см. **разд. 1.8**). Для того, чтобы это было более понятно, приведу один небольшой пример.

Для адресации портов ввода-вывода используется та же самая шина адреса. Но микропроцессору обычно не требуется так много портов, как ячеек памяти. Поэтому чаще всего для адресации портов используется не вся шина данных, а только несколько его младших разрядов. **Например**, в микропроцессоре К580ИК80 для адресации портов используется только 8 младших разрядов шины адреса.

Шина управления

Шина управления в строгом понимании не является цельной цифровой шиной. Просто для управления процессами обмена информации микропроцессорная система должна иметь некий **набор линий**, передающих специальные управляющие сигналы. Эти линии и принято объединять в так называемую шину управления. Что же это за линии и что за сигналы? Ниже приведен примерный набор линий шины управления.

- **RD** (Read) — сигнал чтения.
- **WR** (Write) — сигнал записи.
- **MREQ** — сигнал инициализации устройств памяти (ОЗУ или ПЗУ).
- **IORQ** — сигнал инициализации портов ввода-вывода.

Кроме того, к сигналам шины управления относятся:

- **READY** — сигнал готовности;
- **RESET** — сигнал сброса.

И еще несколько специальных сигналов, о которых мы поговорим позже (**разделы 2.3** и **2.4**).

Принцип действия микропроцессорной системы

Теперь, когда мы изучили все элементы микропроцессорной системы, настало время разобраться, как же она работает. Обратимся вновь к **рис. 2.1**. Как уже говорилось выше, основным элементом системы является **центральный процессор (CPU)**.

По отношению к любым периферийным устройствам процессор может выполнять в каждый момент времени одну из **четырех основных операций**: чтение из ячейки памяти; запись в ячейку памяти; чтение из порта; запись в порт. При работе с памятью процессор активизирует сигнал на выходе MREQ. Сигнал на выходе IORQ остается неактивным. При работе с портами ввода-вывода наоборот: сигнал IORQ активный, а сигнал MREQ — неактивный. Активным уровнем обычно является логический ноль.

Теперь рассмотрим подробнее, как происходят процессы записи и чтения памяти. Для того, чтобы прочитать байт из ячейки памяти, процессор сначала выставляет на шине адреса адрес нужной ячейки. Затем процессор переводит в активное состояние (логический ноль) сигнал RD. Этот сигнал поступает как на устройства памяти, так и на порты ввода-вывода.

Однако порты не реагируют на него, так как они отключены высоким уровнем сигнала IORQ. Устройство памяти, напротив, получив сигналы RD и MREQ, выдает на шину данных байт информации из ячейки памяти, адрес которой присутствует на шине адреса.

Процесс записи данных в память происходит в следующей последовательности. Сначала, как и при чтении, центральный процессор выставляет на адресную шину адрес нужной ячейки памяти. Затем на шину данных он выставляет байт, предназначенный для записи в эту ячейку. После этого процессор переводит в ноль сигнал WR. Получив все эти сигналы, ОЗУ производит запись байта в выбранную ячейку.

При работе с системами памяти часто используется еще один сигнал. Это сигнал готовности. Он необходим в том случае, если модули памяти имеют низкое быстродействие. Такая память может не успеть выдать информацию или произвести ее запись так же быстро, как это способен сделать центральный процессор.

Для согласования работы медленных устройств памяти с быстрыми процессорами существует сигнал READY (готовность). Сразу после того, как процессор установит сигнал чтения или записи в активное состояние, устройство памяти выдает сигнал «не готов». То есть переводит линию READY в пассивное нулевое состояние.

Сигнал READY поступает на процессор. Процессор приостанавливает свою работу и переходит в режим ожидания. Когда устройство памяти закончит выполнение процесса чтения (записи), оно устанавливает сигнал READY в единицу (состояние «Готов»). Получив этот сигнал, процессор возобновляет свою работу.

Операции чтения из порта и **записи** в порт происходят аналогично операциям чтения/записи ОЗУ. Различие лишь в том, что вместо сигнала MREQ в активное состояние переходит сигнал IORQ, разрешающий работу портов. Для работы с медленными внешними устройствами также используется сигнал READY.

2.2. Алгоритм работы микропроцессорной системы

Возможности процессора

Мы узнали, как микропроцессор осуществляет работу с периферийными устройствами — процессор читает числа из этих устройств и записывает информацию. Однако, кроме чтения и записи, процессор производит множество других операций по обработке полученной информации.

С электронными числами процессор способен производить любые виды преобразований, которые вообще возможны с числами. Может складывать числа, вычитать, сравнивать между собой. Кроме того, он способен производить сдвиг разрядов двоичного числа вправо или влево, поразрядные логические операции. К логическим операциям относятся:

- логическое умножение (операция «И»);
- логическое сложение (операция «ИЛИ»);
- инверсия (операция «НЕ»).

Некоторые процессоры умеют производить также умножение и деление. Однако нужно понимать, что все эти операции микропроцессор проделывает с простыми восьмиразрядными (иногда с шестнадцатиразрядными) двоичными числами. Все перечисленные выше операции производятся **аппаратным образом**.

Для этого микропроцессор содержит ряд логических модулей, подобных тем, которые мы рассматривали в предыдущей главе. Все они состоят из набора логических элементов, регистров и т. п. **Примером** может служить рассмотренный выше **сумматор**. Главная же **особенность** микропроцессора в том, что все эти операции выполняются между простыми двоичными числами размером в один (иногда два) байта. Но этого вполне достаточно.

Любую более сложную задачу всегда можно разложить на более простые составляющие и свести к операциям с байтами. Например, для того, чтобы перемножить два многоразрядных десятичных числа, их можно сначала перевести в двоичный вид и записать каждое такое число в несколько ячеек памяти. Затем составить небольшую программку, которая будет перемножать эти числа байт за байтом, а результаты складывать с учетом разрядности. Главное — составить правильный алгоритм.

Программа

Как же заставить процессор выполнять все эти операции в нужной нам последовательности? Для этого нужно создать **программу**. Каждая операция, которую способен выполнять конкретный микропроцессор, кодируется некоторым числом. Это число называется **кодом операции**. Коды операций записываются последовательно, один за другим в память микропроцессорной системы (в ОЗУ либо в ПЗУ).

> ***Программа*** *— это последовательность операций, которую должен выполнять процессор. Она записывается в память в виде последовательности кодов.*

Процессор последовательно, байт за байтом, читает коды операций и выполняет их по мере поступления.

В простых процессорах для кодирования всех команд достаточно одного байта. Одна команда — один байт. В более сложных процессорах число команд возрастает настолько, что одного байта для их кодирования не хватает. В этом случае для кодирования команд используют два, а иногда и более байтов. Причем чаще всего применяют гибкую систему кодирования. В такой системе одни команды кодируются одним бай-

том, другие двумя и так далее. Причем кодировка учитывает частоту применения той или иной команды. Чем чаще команда встречается в программах, тем короче ее код.

Для кодирования команд недостаточно указать код, определяющий вид операции. Например, для команды сложения двух чисел процессору нужно указать:

- адрес ячейки, где хранится первое число;
- адрес второго числа;
- адрес, куда нужно поместить результат.

Поэтому каждая команда, кроме кода операции, может содержать еще один или несколько параметров. Это не обязательно адреса операндов (как в приведенном примере). Параметрами могут служить константы, номера вспомогательных регистров и другие специальные коды.

Существуют самые разные **способы кодировки команд**. Иногда команда состоит из двух байт. Первый байт — это код операции, а второй — параметр. Бывает так, что вся команда вместе с параметрами укладывается в один байт. При этом часть битов этого байта — код операции, а другая часть — параметр.

Бывают и другие варианты. В микропроцессорах серии AVR для кодировки каждой команды используется одно двухбайтовое слово. Для некоторых особо сложных команд кодировка может состоять из двух и более слов. Система команд микропроцессоров AVR приведена в **Приложении 1**.

Процесс выполнения команды

Теперь рассмотрим сам **процесс выполнения программы**. Для того, чтобы микропроцессор мог последовательно читать команды из памяти, внутри него имеется специальный регистр, называемый **регистром адреса** или **счетчиком команд**. В этом регистре хранится адрес текущей выполняемой команды.

Работа микропроцессора всегда начинается с **процедуры начального сброса**. Сброс микропроцессора сводится к установке всех его регистров в исходное состояние. В регистр адреса после сброса записывается адрес начала программы. Адрес начала программы зависит от модели микропроцессора и определяется его разработчиком. Чаще всего этот адрес равен нулю.

Сразу по окончании процесса начального сброса начинается выполнение программы. Для начала процессор читает число из программной памяти, т. е. из ячейки, адрес которой записан в регистр адреса. В нашем случае — из ячейки с нулевым адресом. Прочитанное число он воспринимает как **код первой команды**.

Процессор анализирует код и выполняет соответствующую команду. Если команда предполагает наличие еще одного или нескольких байтов с параметрами, то перед тем, как выполнять команду, процессор читает нужное количество байт из последующих ячеек памяти. При этом он каждый раз увеличивает содержимое регистра адреса.

После выполнения первой команды процессор снова увеличивает значение счетчика команд на единицу и приступает к чтению и выполнению следующей команды. Этот процесс повторяется бесконечно, пока на процессор подано напряжение питания. Таким образом, нормально работающий процессор всегда находится в процессе выполнения программы.

Правда, существуют **несколько исключений**. В частности, в системе команд микропроцессора обычно имеется специальная **команда останова**. Если в процессе выполнения программы встретится такая команда, процессор останавливается. То есть прекращает выполнение программы. Запуск процессора в этом случае возможен лишь после системного сброса либо в результате внешнего прерывания. О том, что такое прерывание, мы еще поговорим.

Второе исключение — **режим сна**. Некоторые модели микроконтроллеров способны переходить в специальный режим низкого потребления энергии, который называется режимом сна или спящим режимом. В спящем режиме выполнение программы также приостанавливается. Режим сна удобен в том случае, когда микропроцессорная система вынуждена долгое время находиться в состоянии ожидания. Например, ожидание нажатия кнопки «Пуск». Такой процессор способен в нужный момент пробудиться из режима сна и продолжить работу.

Рабочие регистры

Кроме регистра адреса, любой микропроцессор обязательно имеет несколько рабочих регистров (так называемых **регистров общего назначения**). Эти регистры наряду с ячейками памяти предназначены для хранения промежуточных результатов вычислений. Преимущество внутренних регистров для хранения данных перед памятью данных — в скорости доступа. При доступе к этим регистрам не нужно указывать адрес, как в случае с доступом к ячейке памяти. Так, команда записи в память состоит минимум из двух байт: кода операции и адреса ячейки памяти.

Иногда для записи адреса одной ячейки не хватает. Если шина адреса процессора имеет 16 разрядов, то для адресации ячейки памяти потребуется два байта. Поэтому такая команда будет содержать не менее трех байт. Команды, работающие с внутренними регистрами микропроцессора, обычно состоят из одного байта.

Сам **код операции** содержит информацию о номере используемого регистра. Чтение и выполнение таких команд происходит гораздо быстрее. Кроме того, внешняя память часто сама имеет меньшее быстродействие, чем процессор. Наличие нескольких внутренних регистров с быстрым доступом позволяет оптимально использовать память. Часто используемые данные стараются помещать в рабочие регистры.

Микроконтроллеры серии AVR для хранения программ используют отдельную память, каждая ячейка которой состоит из одного шестнадцатиразрядного слова. То есть каждая команда состоит минимум из двух байт. Эти два байта содержат и код операции, и параметры. Если двух байт не хватает, то добавляется еще два байта. То есть байты всегда читаются парами.

Команды микропроцессора

Теперь остановимся подробнее на выполняемых командах. Весь набор команд любого микропроцессора можно разделить на несколько групп.

Первая группа — это **команды перемещения данных**. Повинуясь этим командам, процессор копирует содержимое одной ячейки памяти в другую, копирует информацию из ячейки памяти в один из внутренних регистров либо, наоборот, копирует содержимое регистра в одну из ячеек памяти. Кроме того, данные могут копироваться из одного внутреннего регистра в другой.

Следует заметить, что так называемые **команды перемещения**, по сути, не перемещают данные из ячейки в ячейку, а копируют эти данные. Операция перемещения в цифровой технике бессмысленна. В общепринятом понимании переместить означает убрать из одного места и поместить в другое. Но убрать данные из ячейки памяти невозможно!

Любой бит любой ячейки памяти всегда равен либо нулю, либо единице. То есть всегда содержит какое-либо число. Поэтому команды перемещения считывают байт данных из ячейки-источника и записывают их в ячейку-приемник. При этом в ячейке-источнике данные не изменяются.

Ко **второй группе** относятся **команды преобразования данных**. Именно в эту группу входят команды сложения, вычитания, логических преобразований, сдвига разрядов и т. д.

К **третьей группе** относятся **команды передачи управления**. Вот об этом классе команд я хотел бы поговорить подробнее. Сложно представить себе программу, состоящую лишь из одной последовательной цепочки команд. Подавляющее число алгоритмов требуют разветвления программы. Это значит, что программа должна уметь выполнять разные последовательности действий в зависимости от некоторого условия.

Очевидно, что для реализации данного алгоритма придется прервать последовательное выполнение команд. Для того, чтобы программа имела возможность менять алгоритм своей работы в зависимости от какого-либо условия, в системе команд любого процессора обязательно имеются команды передачи управления. К командам передачи управления относятся следующие виды команд: команды условного перехода; команды безусловного перехода; команды перехода к подпрограмме; команды организации цикла. Рассмотрим все эти виды команд по порядку.

Пример

Допустим, что наше микропроцессорное устройство имеет кнопки управления. При нажатии каждой кнопки должно выполняться свое определенное действие. Например, при нажатии одной кнопки исполнительный механизм должен повернуться влево. При нажатии другой — повернуться вправо и т. п. Для того, чтобы это было возможно, программа периодически считывает состояние кнопок. Затем программа должна оценить их состояние. Если нажата первая кнопка, выполняется некая последовательность команд, выдающая на соответствующий порт код, приводящий к включению двигателя в прямом направлении. Если нажата вторая кнопка, то выполняется другая последовательность команд, выдающая в тот же порт совсем другой код. Этот код должен вызвать включение двигателя в реверсном направлении.

Команды условного и безусловного перехода

Оба этих вида команд предназначены для того, чтобы прерывать последовательное выполнение программы и вызывать так называемый **переход**. Причем условный переход происходит только при соблюдении какого-либо условия. Безусловный переход выполняется всегда, как только программа встретит соответствующую команду. В качестве условий перехода может выступать одно из следующих логических выражений:

- величина А *равна* величине В;
- величина А *не равна* величине В;
- величина А *меньше* величины В;
- величина А *больше* величины В;
- величина А *меньше или равна* величине В;
- величина А *больше или равна* величине В.

В качестве величин для сравнения может выступать содержимое любых внутренних регистров процессора, содержимое любых ячеек памяти или просто константы.

Допустим, что условный переход производит сравнение кода нажатой клавиши с некоторой константой. Тогда действие, выполняемое

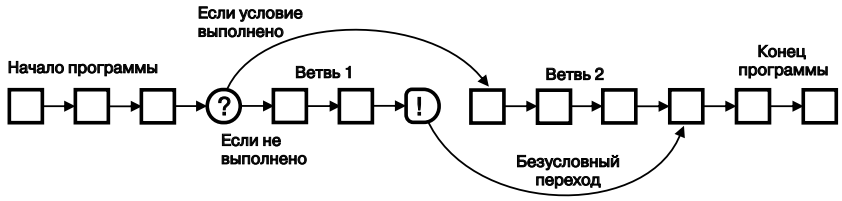


Рис. 2.2. *Работа операторов условного и безусловного переходов*

Пример

Рассмотрим пример применения условного и безусловного переходов. Для наглядности изобразим цепочку команд в программной памяти в виде последовательности графических элементов (см. ***рис. 2.2****). Ход выполнения программы показан при помощи стрелок. Квадратиками обозначены обычные команды (команды перемещения и команды преобразования данных). Кружочек с вопросом — это команда условного перехода. Скругленный элемент с восклицательным знаком — это безусловный переход. Такая программа имеет две ветви. В случае, если условие есть ложь, выполняется ветвь номер 1. В случае, если условие — истина, выполняется ветвь номер 2.*

условным оператором, можно записать так: *«Если код нажатой клавиши равен 0, перейти к выполнению ветви номер 2»*. Соответственно, в случае невыполнения условия (например, считанное число равно 1), программа продолжит свою работу в обычном режиме и перейдет, таким образом, к выполнению ветви номер 1.

В конце ветви номер 1 стоит оператор безусловного перехода. Он служит для того, чтобы программа не начала выполнять ветвь номер 2 сразу после выполнения ветви номер 1. В данном случае выполнение перехода обязательно и никакого условия не требуется.

Технически переход выполняется путем записи в регистр адреса нового значения. Изменение значения регистра адреса возможно только при помощи команд передачи управления.

Команда организации цикла

Циклическое выполнение группы команд — очень эффективное средство для сокращения программного кода. Иногда требуется выполнить одну и ту же группу команд несколько раз. Вместо того, чтобы много раз записывать одни и те же команды, можно заставить любой участок программы выполняться многократно. Для этого и служат команды организации цикла.

Допустим, мы хотим создать простейшую программу, предотвращающую ложное срабатывание кнопки. Допустим, нажатая кнопка при

считывании дает единицу, отпущенная — ноль. Для повышения надежности мы будем считывать состояние кнопки не один, а несколько раз. Все полученные таким образом числа мы сложим между собой.

Затем мы легко можем определить, каких результатов было больше: нулевых либо единичных. Допустим, что мы будем производить подряд 20 операций чтения-сложения. Теперь, если полученная сумма окажется больше десяти, то кнопку можно считать нажатой. В противном случае она считается отпущенной. Такой алгоритм называется **цифровой интегрирующий фильтр**.

Операции считывания состояния кнопки и сложения полученных результатов удобно оформить в виде цикла. На **рис. 2.3** показан ход выполнения подобной программы. Как и на предыдущем рисунке, квадратиками обозначены обычные операторы. Кружком с буквой Ц обозначен **оператор цикла**. Часть программы, называемая **телом цикла**, выполняется нужное количество раз. Каждое такое выполнение называется **проходом цикла**.

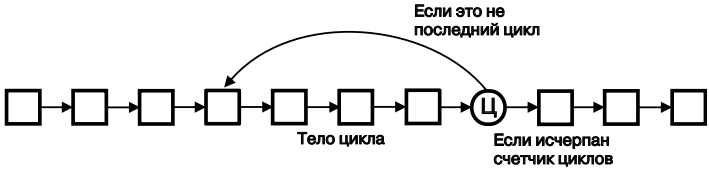


Рис. 2.3. *Демонстрация работы оператора цикла*

Важным элементом оператора цикла служит так называемый **параметр цикла**. Параметр цикла — это число, которое сначала равно количеству проходов. При каждом новом проходе параметр цикла уменьшается. Обычно параметр цикла записывается в один из рабочих регистров процессора. В нашем случае параметр цикла равен 20. На рисунке показаны несколько команд, которые выполняются до начала цикла. Среди этих команд обязательно должна быть команда, записывающая в соответствующий регистр значение параметра цикла. Затем выполняется тело цикла. В нашем случае тело цикла — это команды считывания состояния клавиши и сложения полученных результатов. После выполнения тела цикла наступает очередь оператора цикла. Этот оператор выполняет следующие действия:

- уменьшает параметр цикла на единицу;
- проверяет, не равен ли параметр после уменьшения нулю;
- если не равен, то оператор осуществляет переход к началу цикла;
- если же параметр равен нулю, переход не производится и выполнение программы продолжается в обычном режиме.

В результате такой оператор вызывает многократное выполнение тела цикла до тех пор, пока содержимое параметра цикла не достигнет

нуля. При достижении нуля цикл заканчивается, и программа продолжает выполняться в обычном режиме.

Команды перехода к подпрограмме

Для организации подпрограмм любой процессор содержит как минимум две специальные команды:

- команду перехода к подпрограмме;
- команду выхода из подпрограммы.

> ***Подпрограмма*** *— это некоторый участок программы, к выполнению которого программа может возвращаться несколько раз. Такой прием применяется в том случае, если одни и те же действия нужно выполнять в разных местах программы. Для этого любую последовательность команд можно оформить в виде подпрограммы. В нужном месте основная программа вызывает подпрограмму. После выполнения подпрограммы управление передается в то место, откуда произошел ее вызов. Одна и та же подпрограмма может быть вызвана любое количество раз из самых разных мест основной программы.*

Существуют также команды перехода к подпрограмме по условию. Процесс обращения к подпрограмме показан на **рис. 2.4**. Слева от многоточия показана цепочка команд, составляющих основную программу. Точками обозначена та часть основной программы, которая нас сейчас не интересует. Где-то после окончания основной программы в памяти расположен текст подпрограммы.

Как и в предыдущих случаях, квадратиками обозначены обычные команды. Элемент с буквой «П» — это команда перехода к подпрограмме. Буквой «В» обозначена команда возврата из подпрограммы. По команде перехода к подпрограмме микропроцессор запоминает текущий адрес (значение счетчика программ). Затем управление передается на начало подпрограммы.

В конце подпрограммы обязательно должна стоять команда выхода из подпрограммы. Встретив эту команду, процессор извлекает из памяти адрес, откуда произошел вызов подпрограммы, и переходит

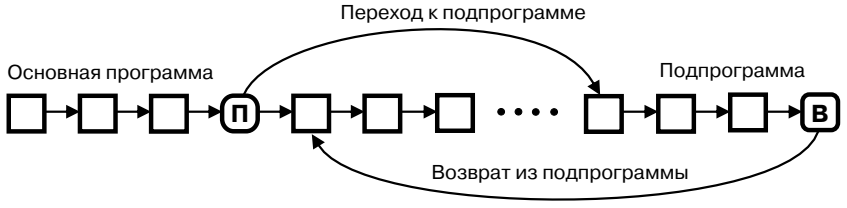


Рис. 2.4. *Демонстрация работы операторов организации подпрограммы*

к команде, непосредственно следующей за этим адресом. После этого программа выполняется в обычном режиме.

Использование подпрограмм позволяет увеличить стуктурированность вашей программы. При чтении текста незнакомой программы каждая подпрограмма воспринимается как отдельная законченная процедура. Каждая такая процедура представляет собой законченный программный блок со своими свойствами и назначением. Из этих блоков, как из кирпичиков, удобно строить основную программу.

Написанная таким образом программа становится удобнее для понимания. Поэтому иногда подпрограммы используют даже в том случае, когда в основной программе они используются только один раз.

2.3. Механизм прерываний

Итак, мы рассмотрели основные принципы работы микропроцессорной системы. Все описанное выше относится к основному режиму работы. Однако для повышения эффективности работы большинство процессоров обычно имеют еще два не менее важных режима работы. Это **режим прерывания** и **режим прямого доступа к памяти**. На структурной схеме типового микропроцессорного устройства (**рис. 2.1**) для простоты не показаны элементы, которые и предназначены для работы в двух дополнительных режимах. Теперь настал момент познакомиться с ними.

Начнем с **механизма прерываний**. Представим себе задачу.

Задача

На вход некоего микропроцессорного устройства поступают некие импульсы, которые процессор должен постоянно считать. Например, импульсы от датчика вращения колеса автомобиля в системе электронного спидометра. Спидометр должен постоянно подсчитывать эти импульсы, чтобы получить величину пробега автомобиля. Одновременно спидометр должен заниматься измерением скорости и другими вспомогательными операциями. Например, опрашивать клавиатуру, выполнять введенные с нее команды, выводить информацию на индикаторы.

Если включить команду опроса порта, на который поступают данные импульсы в общую управляющую программу, то в момент поступления импульса с датчика процессор может быть занят другой операцией. В результате некоторые импульсы могут быть пропущены. Именно для таких случаев и был придуман механизм прерывания.

Для этого любой процессор имеет как минимум один специальный вход запроса на прерывание. Именно на этот вход

необходимо подать наши импульсы, предназначенные для подсчета. Когда очередной импульс приходит на вход запроса прерывания, включается внутренний алгоритм обработки прерывания, заложенный в микропроцессор.

Выполнение основной программы прерывается, а управление передается на специальную процедуру обработки прерывания. Эта процедура является составной частью основной управляющей программы, которую разрабатывает программист при создании микропроцессорной системы.

Адрес начала процедуры обработки прерывания определяется типом микропроцессора. Процедура обработки прерывания должна выполнить те действия, которые необходимы при поступлении сигнала. В нашем случае это может быть алгоритм цифровой фильтрации и увеличение на единицу содержимого ячейки памяти, где хранится количество поступивших импульсов.

Подпрограмма обработки прерывания завершается специальной командой выхода из прерывания. Эта команда подобна команде выхода из подпрограммы. Поэтому после окончания процедуры обработки прерывания микропроцессор возвращается к тому месту основной программы, где произошло прерывание.

Процедура обработки прерывания и подпрограмма очень похожи. Отличие только в том, что переход к подпрограмме происходит по специальной команде, а вызов процедуры обработки прерывания — по приходу внешнего сигнала. Поэтому прерывание может произойти на любом этапе выполнения основной программы.

В результате применения механизма прерывания мы достигаем эффекта независимости двух процессов. Получается, что основная программа выполняется как бы сама по себе, а подсчет импульсов от датчика при этом выполняется как бы отдельно. Машинное время микропроцессора распределяется между двумя этими процессами таким образом, что они оба выполняются независимо друг от друга. Механизм прерываний широко применяется как в микропроцессорной технике, так и в больших компьютерах.

Хороший пример задачи, решаемой при помощи прерывания, — это работа манипулятора «мышь» персонального компьютера. Какую бы сложную программу ни выполнял компьютер, но указатель всегда свободно бегает по экрану, повинуясь движениям мыши.

Бывает, что программа или несколько программ «зависли». Но указатель мыши живет. Мышь обычно зависает только в самом крайнем случае при серьезном сбое системы. Все это происходит благодаря тому, что манипулятор «мышь» работает по прерыванию. Когда вы переме-

щаете манипулятор по столу, специальный механизм внутри мыши преобразует эти перемещения в электронные сигналы, которые передаются на один из входов компьютера.

Специальная схема внутри компьютера принимает эти сигналы и вырабатывает запрос на прерывание для микропроцессора. Получив этот запрос, процессор прерывает выполнение основной программы и выполняет процедуру перемещения изображения мышиного курсора по экрану.

При каждом прерывании курсор перемещается всего лишь на один шажок. Затем процессор возвращается к выполнению своей основной программы. В результате вы наблюдаете свободное перемещение курсора мыши по экрану на фоне выполняющихся программ.

Механизм прерывания в персональном компьютере используется не только для мыши. Это очень распространенный прием. Любой современный процессор имеет сложную многоуровневую систему прерываний, позволяющую обрабатывать прерывания одновременно от нескольких источников. По прерыванию работают такие устройства, как клавиатура, жесткий диск, внутренние системные часы, порт принтера и многое другое.

2.4. Прямой доступ к памяти

Второй специальный режим работы микропроцессорной системы называется **режимом прямого доступа к памяти**. В этом режиме нарушается основной принцип всей микропроцессорной системы. Теперь системой управляет не микропроцессор, а специальный контроллер прямого доступа к памяти (контроллер ПДП).

Прямой доступ к памяти применяется для ускорения работы в том случае, когда необходимо записать в память либо прочитать из памяти большой блок информации.

Для реализации этого режима микропроцессорная система должна иметь в своем составе специальное устройство — **контроллер прямого доступа** (см. **рис. 2.5**).

Пример

Хороший пример — печать текста на скоростном принтере. Если производить печать в обычном режиме работы, то необходимо написать некую подпрограмму, которая должна побайтно читать данные из памяти и выдавать их на порт принтера. Для чтения каждого байта процессор должен выполнить 3-4 команды. Если принтер скоростной, то процессор может не обеспечить требуемую скорость печати. В этом случае применяют прямой доступ к памяти.

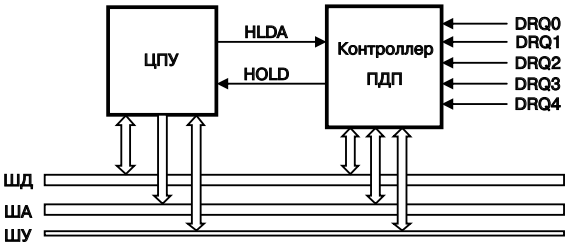


Рис. 2.5. *Подключение контроллера ПДП*

Сначала всем управляет микропроцессор. Перед тем, как начать печать, он производит подготовку сеанса прямого доступа. Для этого он программирует микросхему контроллера. Программирование сводится к записи в управляющие регистры контроллера специальных кодов, которые и определяют в дальнейшем его работу. При помощи этих кодов процессор должен определить начальный и конечный адрес блока данных в памяти, подлежащих передаче и направление передачи информации.

Затем центральный процессор подает команду на контроллер, которая запускает сеанса прямого доступа. С этого момента управление системой берет на себя микроконтроллер ПДП, а центральный процессор отключается. Контроллер отвечает за формирование адреса на соответствующей шине, а также именно он формирует управляющие сигналы RD, WR, MREQ и IORQ на шине управления. При помощи этих сигналов контроллер прямого доступа выполняет довольно простую операцию.

В нашем примере он читает байт за байтом данные из заранее определенной области памяти и выдает их в порт принтера. По завершении процесса передачи контроллер подает специальный сигнал микропроцессору, тот включается и берет на себя управление системой.

Кроме выдачи данных из памяти в порт, контроллер прямого доступа может также осуществлять считывание блока данных из порта и запись его в заранее заданную область памяти. А также может осуществлять передачу данных из одной области памяти в другую.

Рассмотрим подробнее **схему подключения контроллера ПДП** (**рис. 2.5**). Как уже говорилось, микросхема контроллера ПДП — это программируемое устройство. В данном случае понятие «**программируемое**» означает то, что микросхема имеет внутри **специальные регистры**, которые подключаются к системной шине как порты ввода-вывода. Процессор может записывать в эти порты различные коды и, тем самым, менять режимы работы микросхемы.

Для того, чтобы запустить процесс прямого доступа к памяти, центральный процессор должен сначала запрограммировать контроллер. Затем, при помощи тех же самых программируемых регистров, процес-

сор передает на контроллер команду запуска процесса ПДП, а сам продолжает свою работу. Получив команду запуска, контроллер ПДП ждет сигнала на одном из специальных входов DRQ0—DRQ4 (см. **рис. 2.5**).

В нашем случае сигнал запроса ПДП поступает от принтера в тот момент, когда он окажется готовым к работе. Этот сигнал называется **«сигнал запроса на ПДП»**. Получив запрос на ПДП, контроллер формирует запрос на захват центрального процессора (сигнал HOLD). Получив сигнал HOLD, процессор приостанавливает выполнение текущей программы и переходит в пассивный режим. Все выводы процессора отключаются от шины данных, шины адреса и шины управления и не мешают дальнейшей работе системы.

О своей готовности к работе **в режиме захвата** процессор сообщает контроллеру ПДП при помощи сигнала подтверждения захвата (HLDA). Лишь после этого начинается передача данных под управлением контроллера ПДП. По окончании процесса передачи данных контроллер ПДП отключается от системной шины и снимает с процессора сигнал HOLD. После снятия сигнала процессор возобновляет свою работу в обычном режиме.

В микроконтроллерах режим прямого доступа к памяти применяется очень редко. Зато в любом персональном компьютере прямой доступ применяется достаточно широко. Кроме порта принтера, прямой доступ используют сетевые и звуковые карты, а также накопители на гибких и жестких дисках.

2.5. Микроконтроллеры

В классической микропроцессорной системе, изображенной на **рис. 2.1**, используются отдельная микросхема процессора, отдельные микросхемы памяти и отдельные порты ввода вывода. Стремительное развитие микропроцессорной техники требует все большей и большей степени интеграции микросхем.

Именно поэтому были разработаны микросхемы, которые объединяют в себе сразу все элементы микропроцессорной системы. Такие микросхемы называются **микроконтроллерами**. В советское время такие микросхемы называли «**Однокристальные микроЭВМ**».

Для однокристальных микроконтроллеров понятие «центральный процессор» обычно не употребляется. Так как процессор — это все-таки отдельное устройство. Функции процессора в микроконтроллере заменяет **арифметико-логическое устройство (АЛУ)**.

Кроме АЛУ, микроконтроллер содержит в своем составе: тактовый генератор; память данных; память программ; порты ввода-вывода.

ГЛАВА 3

ВПЕРЕД, К ПРАКТИКЕ: ЗНАКОМТЕСЬ — МИКРОКОНТРОЛЛЕРЫ AVR

3.1. Общие сведения

Особенности новой серии микроконтроллеров

Итак, в предыдущей главе мы рассмотрели общие принципы работы микропроцессорной системы. Теперь рассмотрим, как это выглядит на примере конкретных микроконтроллеров. В первом моем Самоучителе (2005 г.) в качестве примера был использован микроконтроллер AT89C2051. Это одна из микросхем популярной в свое время серии AT89 фирмы Atmel. Главным преимуществом этой серии была совместимость с микроконтроллерами iMCS-51 фирмы *Intel*, которые, в свою очередь, были очень распространены в 80-е годы прошлого (двадцатого) столетия.

Однако в настоящее время все эти микросхемы устарели. На смену им пришли новые типы микроконтроллеров. Фирма Atmel тоже не стоит на месте. Самая современная разработка этой фирмы на сегодняшний день — это **микроконтроллеры серии AVR**. Микросхемы этой серии достаточно распространены и популярны во всем мире. Поэтому в настоящей книге мы остановимся именно на этих микроконтроллерах.

В пределах серии микроконтроллеры подразделяются на несколько семейств. Устаревшее семейство «Classic» в настоящее время уже не используется. Основу серии составляет **семейство «Tiny»** и **семейство «Mega».** Семейство микроконтроллеров «Tiny» — это микроконтроллеры минимальной конфигурации и, преимущественно, небольших габаритов, предназначенные для простых недорогих и миниатюрных электронных устройств управления. Они имеют минимальный набор возможностей и невысокую цену.

Микроконтроллеры семейства «Mega», напротив, имеют развитую архитектуру и предназначены для более мощных микропроцессорных систем. Кроме того, фирма Atmel выпускает еще несколько видов микроконтроллеров, которые она также относит к серии AVR. Полный

состав этой серии вы найдете в интернет на виртуальном диске (ссылка в конце книги).

В данной главе мы рассмотрим основные возможности и особенности построения всей серии микроконтроллеров AVR. Таким образом, мы перейдем от теоретических рассуждений к практическим вопросам конкретного устройства микросхем. Все микросхемы AVR строятся по единому принципу и имеют единую систему команд, которая для разных моделей микроконтроллеров может отличаться лишь в наличии либо отсутствии нескольких непринципиальных команд. Поэтому целесообразно изучать всю серию микросхем как единое целое.

Если попытаться сравнить микросхемы AVR с предыдущей серией микросхем (серия AT89), то мы обнаружим очень много отличий. Разрабатывая новую серию, компания Atmel решила не привязываться к старым своим разработкам и кардинально изменила как внутреннюю архитектуру новых микроконтроллеров, так и систему команд.

Состав серии AVR

Итак, что же представляют собой микроконтроллеры серии AVR? **Семейство AVR** включает в себя микроконтроллеры самой разной конфигурации, с разным объемом памяти и разным количеством встроенных портов ввода-вывода и других дополнительных устройств.

Конструктивное исполнение микроконтроллеров также очень разнообразно. Применяется несколько типов корпусов (см. **рис. 3.1**). Это традиционные корпуса типа PDIP с количеством ножек от 8 до 40. Корпуса типа SOIC с количеством выводов от 8 до 20. А вот большинство микроконтроллеров семейства Mega AVR выполнятся либо в сорокавыводных PDIP-корпусах, либо в современных многовыводных корпусах типа TQFP или MLF (до 64 выводов).

В задачу данной книги не входит полное описание всех микросхем серии AVR. Здесь мы познакомимся лишь с общими сведениями, позволяющими понять основные принципы построения микросхем. Более подробно мы остановимся лишь на одной микросхеме из этой серии, и произойдет это в **главе 4**. В качестве примера выбрана **микросхема ATtiny2313**.

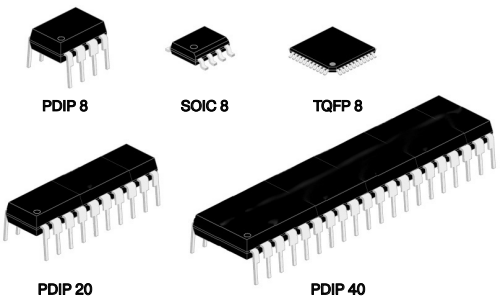


Рис. 3.1. *Микроконтроллеры серии AVR*

Если в справочной литературе не окажется нужной вам микросхемы, рекомендую скачать из Интернета оригинальное описание нужной вам микро-

схемы, так называемый даташит (Datasheet). Любой даташит всегда найдется на сайте производителя (**www.atmel.ru** или **www.atmel.com**).

К сожалению, выложенная там документация существует только на английском языке.

Особенности серии AVR

Микроконтроллеры серии AVR относятся к **классу восьмиразрядных микроконтроллеров**. Это значит, что подавляющее большинство операций процессоры производят с восьмиразрядными двоичными числами. По этой причине встроенная шина данных у этих контроллеров тоже восьмиразрядная. Все ячейки памяти и большинство регистров микроконтроллера также восьмиразрядные.

Для обработки шестнадцатиразрядных чисел некоторые внутренние регистры могут объединяться попарно. Каждая такая пара может работать как один шестнадцатиразрядный регистр. Исключение составляет память программ. Она целиком состоит из шестнадцатиразрядных ячеек.

Микроконтроллеры AVR изготавливаются по КМОП-технологии, благодаря которой они имеют достаточно высокое быстродействие и низкий ток потребления. Большинство команд микроконтроллера выполняется за один такт. Поэтому быстродействие контроллеров может достигать 1 миллиона операций в секунду при тактовой частоте 1 МГц.

Внутренняя память

Микроконтроллеры AVR имеют в своем составе **три вида памяти**. **Во-первых**, это ОЗУ (оперативная память для данных). В документации фирмы Atmel эта память называется **SRAM**. Объем ОЗУ для разных контроллеров варьируется от полного ее отсутствия (в микросхеме AT90S1200) до 2 Кбайт.

Второй вид памяти — это память программ. Она выполнена по Flash-технологии и предназначена для хранения управляющей программы. В фирменной документации она так и называется — **Flash-память**. Объем программной памяти в разных микросхемах этой серии составляет от 1 до 64 Кбайт. Программная память допускает стирание записанной туда информации и повторную запись. Однако количество циклов записи/стирания ограничено.

Программная память микроконтроллеров AVR допускает до 1000 циклов записи/стирания. Запись информации в память программ производится при помощи специальных устройств (программаторов). Последние модели микроконтроллеров AVR имеют режим автоперезаписи памяти программ. То есть управляющая программа самого микроконтроллера способна сама себя переписывать.

Третий вид памяти — это **энергонезависимая память для данных**. Она также выполнена по Flash-технологии, но в технической документации она называется EEPROM. Основное назначение этого вида памяти — долговременное хранение данных. Данные, записанные в эту память, не теряются даже при выключенном источнике питания.

Управляющая программа микроконтроллера может в любой момент записать данные в EEPROM или прочитать их оттуда. Память EEPROM допускает до 100000 циклов записи/стирания. Количество циклов чтения из EEPROM неограничено. Объем памяти EEPROM сравнительно небольшой. Для разных микросхем он составляет от 64 байт до 2 Кбайт. Для большинства задач этого вполне достаточно.

Записывать информацию в EEPROM можно также при помощи **программатора**. Причем для записи информации в память программ и в EEPROM используется один и тот же программатор. Такой порядок доступа к памяти позволяет при необходимости отказаться от программной перезаписи EEPROM и использовать эту память для хранения любых неизменяемых констант. Это увеличивает гибкость системы.

Способы программирования Flash- и EEPROM-памяти

Микроконтроллеры AVR допускают несколько способов программирования Flash- и EEPROM-памяти. Основные способы такие:

- параллельное программирование (Self-Prog);
- последовательное программирование с использованием SPI-интерфейса.

При **параллельном программировании** программатор передает в микросхему записываемые данные побайтно, параллельным способом. То есть при помощи восьмипроводной шины.

При **последовательном программировании** используется специальный последовательный интерфейс, получивший название **SPI**. Посредством этого интерфейса данные передаются в микросхему последовательно, бит за битом, с использованием всего трех проводников. Последовательный способ гораздо медленнее, чем параллельный. Зато он более универсален и допускает программирование микросхемы без извлечения ИМС из схемы.

Порты ввода-вывода

Порты ввода-вывода — это обязательный атрибут любого микроконтроллера. Их количество для каждой конкретной микросхемы разное. Все порты микроконтроллеров AVR восьмиразрядные, но в некоторых случаях отдельные разряды не используются. Это связано с ограниченным количеством выводов (ножек) у микросхемы.

У одних портов используются все восемь его линий. У других семь, шесть или даже три. Но для процессора порты остаются восьмиразрядными. Процессор всегда пишет в такие порты и читает из них полноценный байт информации. Неиспользуемые биты при записи просто теряются. При чтении байта из порта неиспользуемые разряды равны нулю.

Периферийные устройства

Кроме указанных выше элементов, любой микроконтроллер AVR обязательно содержит набор так называемых **периферийных устройств**. Периферийные они по отношению к центральному процессорному устройству (ЦПУ) микроконтроллера. Но находятся они также внутри микросхемы. Ниже перечислены все возможные периферийные устройства, которые могут входить в состав микроконтроллера AVR.

Встроенные таймеры/счетчики. Микроконтроллеры AVR могут содержать от одного до четырех таймеров/счетчиков. Причем используются как восьми-, так и шестнадцатиразрядные таймеры. Их количество на один микроконтроллер может составлять от одного до шести.

Генератор сигнала с широтно-импульсной модуляцией (ШИМ). Генерация сигнала ШИМ — это просто один из режимов работы таймера/счетчика. Одна микросхема может иметь от 2 до 12 каналов ШИМ, а может не иметь ни одного.

Аналоговый компаратор. Входит в состав практически во всех микроконтроллеров AVR.

Аналогово-цифровой преобразователь (АЦП). АЦП микроконтроллеров AVR могут иметь от четырех до шестнадцати каналов. То есть могут преобразовывать в цифровой эквивалент до 16 входных аналоговых сигналов. На самом деле канал АЦП всегда один. Но на его входе стоит система переключения (аналоговый мультиплексор). Поэтому АЦП способен подключаться к нескольким разным источникам аналогового сигнала.

Последовательный интерфейс. Микросхемы AVR способны поддерживать несколько разных видов последовательных интерфейсов. Каждый такой интерфейс реализует один или несколько известных стандартов передачи информации. Один из видов такого интерфейса поддерживает тот же стандарт, что и COM-порт персонального компьютера. Есть также интерфейс, поддерживающий стандарт широко известной в микроэлектронике так называемой I^2C шины.

Сюда же относится и **SPI-интерфейс**, который может использоваться как для последовательного программирования памяти программ, так и для связи нескольких микроконтроллеров в мультипроцессорной системе. Любой последовательный интерфейс предназначен

для передачи информации последовательным способом. Каждый байт передается последовательно, бит за битом.

Другие устройства

Кроме перечисленных выше устройств, микроконтроллеры серии AVR обязательно содержат систему прерываний, охранный таймер, систему начального сброса, систему контроля питающего напряжения и т. д. Все описанные выше устройства управляются центральным процессором при помощи регистров. И вот здесь мы вплотную подходим к такому понятию, как **архитектура микроконтроллеров AVR**.

*Под **архитектурой** понимается внутреннее строение микросхемы и взаимодействие всех ее элементов.*

К элементам архитектуры можно отнести объем и структуру всех видов памяти микроконтроллера, количество и свойства так называемых регистров общего назначения, устройство портов ввода-вывода и методы доступа к ним, устройство системы прерываний, способы управления встроенными периферийными устройствами. А начнем мы с изучения всех видов памяти.

3.2. Регистры общего назначения (РОН)

Для хранения промежуточных результатов вычислений каждый микроконтроллер AVR имеет тридцать два **регистра общего назначения (**сокращенно — **РОН)**. Для того, чтобы регистры можно было использовать в программе, каждый имеет свое **собственное имя**. Вот эти имена: R0, R1, R2 — R31.

Все РОН составляют так называемый **файл регистров общего назначения**. Все команды преобразования данных (сложения, вычитания и т. д.) микроконтроллера AVR построены таким образом, что обязательно используют РОН. Каждая команда в качестве операндов использует либо содержимое двух разных РОН, либо содержимое РОН и константу. Результат вычислений также помещается в один из РОН.

Пример

Команда ADD R0, R1 производит сложение содержимого регистров R0 и R1. Сумма помещается в R0. Команда ADD R5, #7 прибавляет к содержимому регистра R5 число семь. Результат помещается в R5.

Регистры общего назначения используются также и в командах перемещения данных. Перемещать данные можно из одного РОН в другой, из РОН в ячейку памяти и

в обратном направлении. Перемещение данных возможно также между РОН и регистрами ввода-вывода, о которых мы поговорим **в следующем разделе**.

Некоторые команды имеют ограничения по использованию РОН. Например, все команды обмена информацией с регистрами ввода-вывода не могут использовать регистры R0—R15. Существуют и другие ограничения. Подробнее это можно узнать из описания системы команд (см. **Приложение 1**).

Все регистры общего назначения микроконтроллеров AVR восьмиразрядные. Однако шесть последних регистров (R26—R31) способны объединяться в регистровые пары. Такая пара в некоторых операциях выступает как самостоятельный шестнадцатиразрядный регистр. При этом не теряется возможность чтения каждого регистра пары отдельно. Регистровые пары имеют свои названия. Пара, объединяющая регистры R26—R27, называется **регистром X**. Пара регистров R28—R29 называется **регистром Y**. А пара регистров R30—31 называется **регистром Z**.

Весь набор регистров общего назначения присутствует в любом микроконтроллере серии AVR.

3.3. Регистры ввода-вывода

То, что в **главе 2** мы называли **портами ввода-вывода**, в микроконтроллерах AVR называется **регистрами ввода-вывода**.

Смещение понятий произошло потому, что микроконтроллеры AVR для обмена информацией с внешними устройствами используют достаточно сложные электронные схемы, имеющие несколько разных режимов работы, а также возможность выбора программным путем направления передачи данных.

Именно они и получили название портов ввода-вывода. Чуть позже мы подробно рассмотрим их устройство.

Простые же регистры, служащие для связи центрального процессора с периферийными устройствами, получили более подходящее в данном случае название: **регистры ввода-вывода**. Эти регистры позволяют обмениваться информацией лишь со встроенными периферийными устройствами самой микросхемы. Такими как таймеры, компараторы, каналы последовательной передачи информации, система прерывания, АЦП и т. д.

Каждый регистр ввода-вывода имеет свой номер, то есть адрес в адресном пространстве ввода-вывода. Номера регистров могут иметь значение от $00 до $3F. Это означает, что максимально возможное количество РВВ равно 64. Однако реальное количество регистров любого

Пример

В микроконтроллере семейства «Tiny» регистр номер $1E предназначен для управления EEPROM. Этот регистр имеет имя EEAR. Второй регистр управления EEPROM имеет номер $1D и имя EEDR. Для разных микроконтроллеров регистры, имеющие одинаковое назначение, обычно имеют и одинаковое имя. А вот номер регистра может и отличаться.

***Имя регистра** — это условное понятие, придуманное лишь для удобства программистов. Сам же микроконтроллер работает исключительно с номером регистра.*

микроконтроллера всегда меньше. Разные микроконтроллеры имеют разный набор регистров ввода-вывода.

Каждый регистр ввода-вывода, помимо номера, имеет свое уникальное имя.

3.4. Память

Общие сведения

Как уже говорилось, микроконтроллеры AVR имеют три вида памяти: память программ (Flash); оперативную память данных (SRAM); энергонезависимую память данных (EEPROM).

Объем каждого вида памяти для разных микросхем вы можете видеть посмотреть в разделе «Справочная информация» на виртуальном диске. Память EEPROM имеется не во всех микроконтроллерах. Кроме того, в некоторых моделях отсутствует оперативная память (SRAM). В таких микросхемах для оперативного хранения данных используются только регистры общего назначения.

Каждый из этих трех видов памяти имеет свое собственное адресное пространство, и доступ к разным видам памяти осуществляется независимо друг от друга. Такое построение микроконтроллеров называется **архитектурой Гарвардского типа**.

Память программ

Память программ предназначена для хранения управляющей программы микроконтроллера. Каждая ячейка этой памяти имеет 16 разрядов, то есть хранит одно шестнадцатиразрядное двоичное число. Каждое такое число содержит как код операции, так и один или несколько параметров команды. Кроме того, в памяти программ можно хранить данные. Такие данные будут доступны только для чтения.

В памяти программ обычно хранят некоторые постоянные константы, таблицы символов и другие неизменяемые величины. Данные записываются в виде восьмиразрядных двоичных чисел (байтов). При

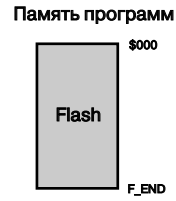


Рис. 3.2. *Адресное пространство программной памяти*

этом каждая шестнадцатиразрядная ячейка программной памяти используется как две восьмиразрядные. При чтении данных из программной памяти микроконтроллер может обратиться к каждой такой половинке отдельно. Мы еще поговорим об этом подробно.

Для разных микроконтроллеров память программ имеет разный объем. Однако при любом объеме памяти она представляет собой непрерывную область и начинается с ячейки, имеющей нулевой адрес.

На **рис. 3.2** изображено адресное пространство памяти программ в графическом виде. Такое условное изображение адресного пространства часто применяется в технической литературе. Изображенное на рисунке адресное пространство включает в себя ячейки Flash-памяти с адресами от $000 до F_END.

Условное обозначение F_END означает **адрес последней ячейки памяти**. Значение этого адреса будет разным для разных микроконтроллеров. Например, для микроконтроллера ATtiny2313 адрес последней ячейки памяти будет равен $7FF. Адрес последней ячейки памяти всегда на единицу меньше объема этой памяти.

Так, для микроконтроллера ATtiny2313 объем программной памяти равен 2 килобайтам. То есть 2048 байт. Если записать это число в шестнадцатиричном виде, получим $800. Учитывая то, что адресация начинается с нулевого адреса, то для адресации такого количества ячеек мы должны использовать адреса с 0 по 2047. Или в шестнадцатиричном виде от $000 до $7FF.

Некоторые адреса программной памяти зарезервированы. То есть используются для неких специальных целей. И первым зарезервированным адресом можно считать **нулевой адрес**. Он называется **вектором системного сброса**. Именно с этого адреса начинается выполнение программы после системного сброса микроконтроллера. Остальные зарезервированные адреса — это **векторы прерываний**.

> ***Вектор прерывания*** *— это адрес в программной памяти, с которого начинается выполнение процедуры обработки прерывания.*

Так как любой микроконтроллер AVR имеет несколько источников прерывания, то и векторов прерывания тоже несколько: по одному на каждый вид прерывания. Адреса векторов прерываний находятся сразу за вектором сброса. То есть занимают ячейки с адресами $001, $002 и т. д. Количество векторов прерываний для разных микросхем разное.

В этой графе отдельно показано количество внутренних и количество внешних прерываний.

В микроконтроллерах семейства «Tiny» эта область начинается с адреса $001. Для большинства микроконтроллеров семейства «Mega» таблица векторов прерываний начинается с адреса $002. При разработке программы для микроконтроллера программист по своему усмотрению может использовать, но может и не использовать механизм прерываний.

> ***Внутренним прерыванием*** *называется прерывание, вызванное одним из встроенных периферийных устройств самого микроконтроллера. Например прерывание по таймеру, аналоговому компаратору, АЦП и т. д.*
>
> ***Внешнее прерывание*** *— это прерывание по сигналу, поступающему от внешнего источника на специальный вход микроконтроллера.*
>
> *Область адресов, зарезервированных под векторы прерываний, называют* ***таблицей векторов прерываний****.*

Если прерывания не используются, то ячейки, зарезервированные под вектора прерываний, можно использовать как обычные ячейки для хранения программы. Если же вы решили в своей программе использовать прерывания, то по адресу $000 необходимо записать команду безусловного перехода, которая должна передавать управление на любой адрес за пределами таблицы векторов прерываний.

Именно там и должна начинаться основная программа. В каждую ячейку, соответствующую тому либо иному вектору прерывания, тоже записывается команда безусловного перехода. Каждый такой переход передает управление на начало соответствующей процедуры обработки прерывания.

Оперативная память микроконтроллеров AVR

Память данных микроконтроллеров AVR представляет собой отдельное адресное пространство с адресами от $0000 до $FFFF. То есть максимальный объем адресуемой памяти составляет 64 Кбайта. Однако большинство микроконтроллеров имеет гораздо меньшую память. В таких микроконтроллерах часть адресов не используется. Структура же памяти всегда одинакова. В графическом виде эта структура изображена на **рис. 3.3**. Посмотрите внимательно на этот рисунок.

Оперативная память микроконтроллеров AVR делится на три области.

- **$0000—$001F** — область памяти, совмещенная с регистрами общего назначения (РОН).
- **$0020—$005F** — область памяти, совмещенная с регистрами ввода-вывода (РВВ).
- **$0060—$FFFF** — не совмещенная ни с чем область памяти.

Эта последняя область предназначена просто для хранения данных. Эту область в свою очередь можно разделить на *область*

внутреннего ОЗУ (**$0060—RAMEND**) и *область внешнего ОЗУ* (***RAMEND+1—$FFFF***).

Под RAMEND понимается адрес последней ячейки внутреннего ОЗУ конкретного микроконтроллера. Рассмотрим каждую область памяти подробнее.

Область памяти, совмещенная с набором регистров общего назначения (РОН)

Эта область существует во всех микроконтроллерах AVR. Она занимает ячейки с адресами с $0000 по $001F. Все ячейки этой области памяти одновременно являются регистрами общего назначения (смотри выше). То есть записывая байт данных в ячейку памяти с адресом $0000, вы на самом деле записываете ее в регистр R0. И наоборот. Соответствие ячеек памяти и регистров общего назначение показано на **рис. 3.3**. Двойной доступ к РОН существенно увеличивает гибкость программ.

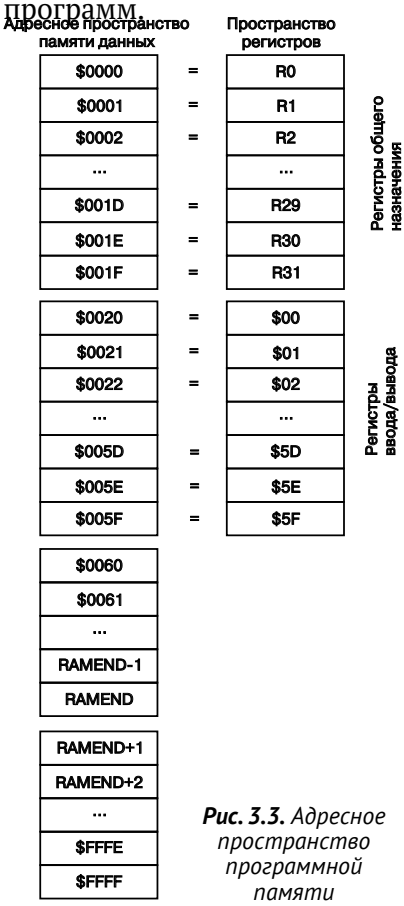


Рис. 3.3. *Адресное пространство программной памяти*

Область памяти, совмещенная с регистрами ввода-вывода (РВВ)

Область памяти с адреса $0020 по адрес $005F совмещена с регистрами ввода-вывода. В адресном пространстве ОЗУ это соответствует адресам $0020—$005F. Каждому регистру ввода-вывода соответствует своя ячейка в ОЗУ. Как уже говорилось, реальное количество регистров ввода-вывода почти всегда гораздо меньше их максимально возможного количества. Однако данная область памяти всегда используется только для этой цели.

Если регистр существует, то существует и соответствующая ячейка памяти. Остальные же ячейки из этой области ОЗУ просто отсутствуют. На **рис. 3.3** показано соответствие регистров общего назначения и ячеек памяти. Из рисунка видно, что адрес ячейки памяти всегда больше номера соответствующего РВВ на постоянную величину, равную 32 ($20).

Область внутреннего ОЗУ

Пространство ОЗУ с адреса $0060 и выше не выполняет никаких дополнительных функций и предназначено исключительно для оперативного хранения данных. Объем (а, значит, и конечный адрес) этой области ОЗУ для разных микросхем разный.

Область внешнего ОЗУ

Большинство микроконтроллеров AVR имеют лишь встроенное ОЗУ. Однако в состав серии входят микросхемы, допускающие подключение внешних микросхем ОЗУ. В результате объем ОЗУ микроконтроллера может быть расширен до 64 Кбайт. При этом общий объем оперативной памяти может достигать значения $FFFF.

Энергонезависимая память данных (EEPROM)

В современных микропроцессорных устройствах часто возникает необходимость в хранении таких данных. **Примером** может служить микропроцессорная система управления автомагнитолой. Такая система управления где-то обязательно должна хранить множество констант. У любой магнитолы есть несколько фиксированных настроек.

***EEPROM** — это специальная внутренняя память, выполненная по Flash-технологии и предназначенная для долговременного хранения данных.*

Кроме того, принято при выключении запоминать все режимы работы магнитолы и восстанавливать их после включения. Все эти настройки в виде чисел обычно записываются в энергонезависимую память. Можно, конечно, использовать внешнюю микросхему памяти. Но встроенная память гораздо удобнее.

Для подобных задач обычно не требуется больших объемов EEPROM-памяти. Поэтому микроконтроллеры AVR имеют объем EEPROM от 64 байт до 8 Кбайт. EEPROM — необычная память. Поэтому к этой памяти ЦПУ микроконтроллера обращается не так, как к остальным видам памяти. Для центрального процессора не существует адресного пространства EEPROM. К этому виду памяти микроконтроллер обращается при помощи регистров ввода-вывода. Для микроконтроллеров с объемом EEPROM менее 256 байт таких регистров всего три:

- **EEAR** — регистр адреса EEPROM;
- **EEDR** — регистр данных EEPROM;
- **EECR** — регистр управления EEPROM.

Если объем EEPROM превышает 256 байт, то вместо одного регистра адреса (EEAR) такой микроконтроллер имеет два регистра: **EEARH** и **EEARL**. Регистры доступа к EEPROM имеют следующие номера:

- **EEAR** — $1E;
- **EEARH** — $1F:
- **EEDR** — $1D:
- **EEARL** — $1E:
- **EECR** — $1C.

Регистры адреса EEAR (или EEARH, EEARL) работают только на запись. При помощи этих регистров микроконтроллер выбирает ячейку, куда нужно записать или откуда нужно прочитать данные.

Регистр данных (EEDR) работает как на запись, так и на чтение. Через этот регистр в EEPROM поступает записываемый байт. Через него же процессор получает байт при чтении из EEPROM.

Регистр управления (EECR) определяет режимы работы. Именно через него подаются команды чтения и записи EEPROM. В **главе 5** на конкретных примерах мы подробно рассмотрим алгоритм чтения и записи в EEPROM.

3.5. Счетчик команд и стековая память

Два важных регистра, которые существуют в любом микропроцессоре или микроконтроллере, — это **счетчик команд** и **указатель стека**.

Счетчик команд *— это специализированный внутренний регистр микроконтроллера, в котором хранится адрес текущей выполняемой команды.*

Этот регистр не доступен для программиста в том смысле, что не существует команд прямой записи или чтения его содержимого. Размер счетчика команд составляет для разных микроконтроллеров AVR от 9 до 12 разрядов. Количество разрядов счетчика команд зависит от размера адресуемой программной памяти конкретного микроконтроллера. После сброса микроконтроллера в счетчик команд записывается ноль.

Затем процессор переходит в **режим выполнения программы**. В процессе выполнения программы счетчик всегда указывает на текущую выполняемую команду. При считывании кода команды значение счетчика увеличивается на один или два (в зависимости от длины команды). При выполнении команд безусловного и условного переходов содержимое счетчика резко меняется. В него записывается новое значение адреса.

Новое значение адреса называется ***адресом перехода****.*

Кроме традиционных команд условного перехода, которые мы уже рассматривали в предыду-

щей главе, микроконтроллеры серии AVR имеют еще один вид команд, который можно рассматривать как их модификацию. Это **команды типа «проверка/пропуск»**. В командах этого типа производится проверка некоего условия, и результат проверки влияет на выполнение следующей команды. Если условие истинно, то следующая команда игнорируется.

То есть сама команда не выполняется, изменяется лишь содержимое счетчика команд. Это содержимое увеличивается либо на единицу, либо на две единицы, в зависимости от длины пропускаемой команды. Если условие ложно, то команда не пропускается, а выполняется как обычно. Теперь перейдем к указателю стека.

Запись в стековую память и чтение из нее производится по принципу магазина автомата Калашникова. Патроны в такой магазин вставляются через входное отверстие один за другим. Извлекаются патроны из магазина в обратном порядке по принципу «последний зашел — первый вышел». Стековую память очень часто используют при программировании. Особенно удобно использовать стек для сохранения данных при входе в подпрограмму и восстановления их перед выходом. В дальнейшем мы убедимся в этом на примерах. В настоящий же момент я хочу остановиться на методах организации стековой памяти.

> ***Указатель стека*** *— это специальный регистр, который предназначен для организации так называемой стековой памяти. Стековая память широко применяется в вычислительной технике. Вообще,* **стек** *— это некий буфер, состоящий из нескольких ячеек памяти, имеющий один вход, который одновременно является и выходом.*

В микроконтроллерах серии AVR применяется широко распространенный **способ организации стековой памяти**, когда в качестве стека используется часть ОЗУ. Для реализации принципа «последний зашел — первый вышел» и служит регистр–указатель стека. В зависимости от размеров ОЗУ, разрядность указателя стека бывает разная. В микроконтроллерах с небольшим объемом ОЗУ используется восьмиразрядный указатель стека. Он представляет собой один регист р ввода-вывода и доступен для свободного считывания и записи. Называется такой **регистр `SPL`**. Для ОЗУ больших размеров к регистру `SPL` добавляется еще один регистр `SPH`. Вместе они составляют один шестнадцатиразрядный указатель стека.

Перед началом работы в указатель стека необходимо **записать адрес вершины стека**. Это некий адрес ячейки ОЗУ, которая является старшей ячейкой области памяти, выделенной под стек. Определять размер стековой памяти и адрес ее вершины должен сам программист.

Для работы со стеком в системе команд микроконтроллера есть две специальные команды:

- команда записи в стек (`push`);
- команда извлечения из стека (`pop`).

Выполняя команду `push`, микроконтроллер записывает содержимое одного из РОН в ОЗУ по адресу, на который указывает указатель стека, а затем уменьшает значение указателя на единицу. Новая команда `push` запишет значение другого РОН в следующую ячейку ОЗУ. А указатель передвинется еще дальше. Таким образом происходит заполнение стека.

Выполняя команду `pop`, микроконтроллер сначала увеличивает содержимое указателя стека на единицу, а затем извлекает содержимое ячейки ОЗУ, на которое указывает указатель. Считанное значение помещается в один из РОН. В результате из стека считывается последнее записанное туда число. Следующая команда `pop` опять сначала увеличит указатель стека и прочитает предпоследнее записанное туда число. Благодаря регистру-указателю стека и описанному выше алгоритму реализуется полноценная стековая память.

Сразу после сброса микроконтроллера содержимое указателя стека равно нулю. Если оставить это содержимое без изменений, то все команды, связанные со стеком, работать не будут. Если вы собираетесь использовать стек в вашей программе, то в самом ее начале вам необходимо записать в регистр-указатель стека значение его вершины.

Обычно вершину стека устанавливают равной адресу самой старшей ячейки ОЗУ. Кроме того, при составлении программы вы должны следить, чтобы она не использовала в процессе своей работы область ОЗУ, выделенную вами для стека.

Одной из команд, активно использующих стековую память, является **команда перехода к подпрограмме.** При вызове подпрограммы текущий адрес из счетчика программ автоматически записывается в стек. При выходе из подпрограммы микроконтроллер извлекает адрес из стека и продолжает выполнение программы с этого адреса. Команда перехода к подпрограмме использует тот же самый стек, что и команды `push` и `pop`.

Это нужно обязательно учитывать при составлении программы. Если записать данные в стек, а затем перейти к подпрограмме, то в теле подпрограммы прочитать эти данные из стека будет уже невозможно. Если вы все же попытаетесь в данной ситуации извлечь данные из стека, вместо записанных данных вы получите адрес возврата из подпрограммы. Мало того, что вы не получите нужные вами данные, так вы еще и сделаете невозможным правильный выход из подпрограммы, так как в стеке уже не будет адреса выхода.

3.6. Подсистема ввода-вывода

Микроконтроллеры серии AVR всегда имеют в своем составе от одного до семи портов ввода-вывода. Каждый разряд такого порта подсоединен к одному из выводов (контактов) микросхемы. Порты ввода-вывода служат для обмена информацией с внешними устройствами. Как уже говорилось, порты могут быть полные и неполные. Полный порт содержит 8 разрядов. В неполных портах задействованы они не все. Каждый порт имеет свое имя. Они именуются латинскими буквами от A до G.

Для управления каждым портом ввода-вывода используется три специальных РВВ. Это регистры PORTx, DDRx и PINx. Под «x» здесь подразумевается конкретная буква — имя порта. **Например**, для порта A имена регистров управления будут такими: PORTA, DDRA и PINA.

Рассмотрим теперь назначение каждого из этих регистров:

- **PORTx** — регистр данных (используется для вывода информации);
- **DDRx** — регистр направления передачи информации;
- **PINx** — регистр ввода информации.

Отдельные разряды приведенных выше регистров также имеют свои имена. Разряды регистра PORTx обычно именуются как **Pxn**. Где «n» — это номер разряда. **К примеру**, разряды регистра PORTA будут именоваться следующим образом: PA0, PA1, PA2—PA7.

Разряды порта DDRx именуются как DDxn (для порта A — DDA0, DDA1—DDA7).

Разряды порта PINx именуются как PINxn (для порта A — PINA0, PINA1—PINA7).

Для других портов буква A заменяется соответственно на B, C, D, E, F, G.

Любой порт ввода-вывода микроконтроллера серии AVR устроен таким образом, что каждый его разряд может работать как на ввод, так и на вывод. То есть он может быть входом, а может быть выходом. **Для переключения режимов работы** служит регистр DDRx. Каждый разряд регистра DDRx управляет своим разрядом порта. Если в каком-либо разряде регистра DDRx записан ноль, то соответствующий разряд порта работает как вход.

Если же в этом разряде единица, то разряд порта работает как выход. Для того, чтобы выдать информацию на внешний вывод микросхемы, нужно в соответствующий разряд DDRx записать логическую единицу, а затем записать байт данных в регистр PORTx. Содержимое соответствующего бита этого байта тут же появится на внешнем выводе микросхемы и будет присутствовать там постоянно, пока не будет заменено другим, либо пока данная линия порта не переключится на ввод.

Для того, чтобы прочитать информацию с внешнего вывода микроконтроллера, нужно сначала перевести нужный разряд порта в режим

ввода. То есть записать в соответствующий разряд регистра DDRx ноль. Только после этого на данный вывод микроконтроллера можно подавать цифровой сигнал от внешнего устройства. Далее микроконтроллер просто читает байт из регистра PINx. Содержимое соответствующего бита прочитанного байта соответствует сигналу на внешнем выводе порта.

Порты ввода-вывода микроконтроллеров AVR имеют еще одну **полезную функцию**. В режиме ввода информации они могут при необходимости подключать к каждому выводу порта внутренний нагрузочный резистор. Внутренний резистор позволяет значительно расширить возможности порта. Такой резистор создает вытекающий ток для внешних устройств, подключенных между выводом порта и общим проводом.

Благодаря этому резистору упрощается подключение внешних контактов и кнопок. Обычно контакты требуют внешнего резистора. Теперь без внешнего резистора можно обойтись. Включением и отключением внутренних резисторов управляет регистр PORTx, если порт находится в режиме ввода. Это хорошо видно из **табл. 3.1**, в которой показаны все режимы работы порта.

Конфигурирование порта ввода-вывода *Таблица 3.1*

DDxn	Pxn	Режим	Резистор	Примечание
0	0	Вход	Отключен	Вывод отключен от схемы
0	1	Вход	Подключен	Вывод является источником тока
1	0	Выход	Отключен	На выходе «0»
1	1	Выход	Отключен	На выходе «1»

На **рис. 3.4** показана упрощенная схема одного разряда порта ввода-вывода. Эта схема дает представление о работе порта. Схема, изображенная на рисунке, — это лишь универсальная часть схемы вывода порта. На самом деле любой вывод кроме основных функций имеет ряд дополнительных. Поэтому реальная схема сложнее. В каждую такую схему добавлены элементы, реализующие все дополнительные функции.

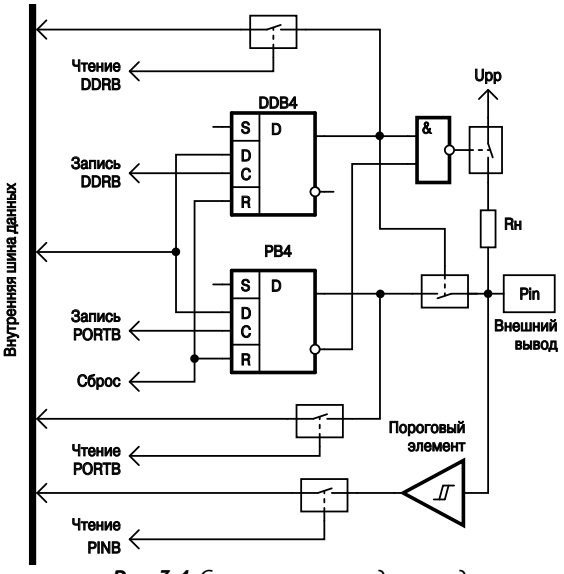


Рис. 3.4. *Схема порта ввода-вывода*

3.7. Система прерываний

Назначение системы прерываний

Важным элементом микроконтроллера является **система прерываний**. Система прерываний присутствует в любом современном микроконтроллере. Она также есть во всех микроконтроллерах AVR. Как уже говорилось, система прерываний микроконтроллера обслуживает несколько источников прерываний. Количество источников прерываний для разных микроконтроллеров различно.

Самое минимальное количество источников прерывания имеет микроконтроллер ATtiny11. Два внутренних источника прерываний (от таймера/счетчика и от встроенного компаратора), одно внешнее прерывание по сигналу на входе INT0 и одно прерывание по изменению сигналов на любом из входов, которое тоже считается внутренним.

К источникам прерываний фирма Atmel относит также начальный сброс микроконтроллера. Вектор начального сброса обычно также включают в таблицу векторов прерываний. Так что получается, что у микроконтроллера ATtiny11 имеется четыре внутренних источника прерываний и один внешний. Другие микросхемы серии AVR имеют более сложные системы прерываний.

Самая развитая на сегодняшний день система прерываний — у микроконтроллера ATmega1281. Этот микроконтроллер способен в общей сложности обслуживать 48 внутренних и 17 внешних источников прерываний. Вообще, источниками прерываний служат все встроенные таймеры, компараторы, АЦП, любой последовательный канал, система управления EEPROM.

Управление системой прерываний

Управление системой прерываний осуществляется при помощи специальных регистров ввода-вывода. Определяющим регистром здесь является **регистр SREG** (регистр состояния системы). Этот регистр предназначен для хранения флагов состояния. Каждый бит регистра — это один из флагов. Седьмой бит регистра состояния называется «флаг I». Это флаг глобального разрешения прерываний. Когда значение этого флага равно нулю, все прерывания в микроконтроллере запрещены.

Для разрешения прерываний нужно установить этот флаг в единицу. Однако чаще всего нам не нужны все виды прерываний одновременно. Для того, чтобы запретить одни прерывания и разрешить другие, применяются так называемые маскирующие регистры (регистры маски).

В микроконтроллерах AVR применяются два регистра маски. **Регистр GIMSK** управляет всеми видами прерываний, кроме прерываний от таймеров. В некоторых микроконтроллерах семейства «Mega»

Регистр маски *— это обычный регистр ввода-вывода, служащий для управления отдельными источниками прерываний. Каждому биту в регистре маски соответствует один источник. Если бит сброшен в ноль, прерывание этого вида запрещено. Если бит установлен в единичное состояние, прерывание разрешено.*

этот регистр называется **GICR**. Для управления прерываниями от таймеров имеется специальный **регистр TIMSK**.

Кроме регистров маски для управления процессом выполнения прерываний существуют еще два регистра. Это **регистры флагов прерываний**. Каждый бит такого регистра — это флаг одного из видов прерываний. При поступлении запроса на прерывание флаг устанавливается в единицу. По состоянию флага программа может судить о наличии запроса.

В определенных режимах после установки флага процедура обработки прерывания вызывается автоматически. Сразу после вызова процедуры соответствующий флаг сбрасывается. Микроконтроллеры AVR имеют два регистра флагов: регистр GIFR (обслуживает те же прерывания, что и регистр GIMSK) и регистр TIFR (флаги прерываний от таймеров).

Алгоритм работы системы прерываний

Общий алгоритм работы системы прерываний следующий. После сброса микроконтроллера все прерывания запрещены (флаги разрешения сброшены). Если программист планирует использовать один из видов прерываний, он должен предусмотреть в своей программе включение этого прерывания.

Для включения прерывания программа должна установить флаг I регистра SREG в единицу и записать в регистры маски такой код, который разрешит лишь нужные в данный момент прерывания. Разрешив, таким образом, прерывания, программа приступает к выполнению своей главной задачи.

При поступлении запроса на прерывание устанавливается флаг соответствующего прерывания. Флаг устанавливается даже в том случае, если прерывание запрещено. Если прерывание разрешено, то микроконтроллер приступает к его выполнению. Текущая программа временно приостанавливается, и управление передается на адрес соответствующего вектора прерывания.

В тот же момент флаг I автоматически сбрасывается, запрещая обработку других прерываний. Флаг, соответствующий вызванному прерыванию, также сбрасывается, сигнализируя о том, что микроконтроллер уже приступил к его обработке. Подпрограммы обработки прерывания обязательно должны оканчиваться **командой возврата из прерыва-**

ния (RETI). По этой команде управление передается в ту точку основной программы, в которой прервалась ее работа. Флаг I при этом автоматически устанавливается в единицу, разрешая новые прерывания.

Следует заметить, что без принятия специальных мер невозможны вложенные прерывания. Пока обрабатывается одно прерывание, все остальные прерывания запрещены. Однако ни одно прерывание не остается без обработки. При получении запроса на прерывание соответствующий флаг обязательно будет установлен. В этом состоянии он будет находиться до тех пор, пока данное прерывание не будет обработано.

После окончания обработки очередного прерывания происходит проверка остальных флагов, и если имеется хоть одно необработанное прерывание, микроконтроллер переходит к его обработке. Если необработанных прерываний окажется несколько, то применяется закон приоритетов. Из всех прерываний выбирается то прерывание, приоритет которого выше. Чем меньше адрес вектора прерывания, тем выше его приоритет.

И в заключении, в качестве примера, приведу таблицу векторов прерываний для микроконтроллеров семейства «Ttiny» (см. **табл. 3.2**).

Адреса векторов прерываний микроконтроллеров семейства «Tiny» *Таблица 3.2*

Источник	Описание	Tiny11x	Tiny12x	Tiny15L	Tiny28x
INT0	Внешнее прерывание 0	$001	$001	$001	$001
INT1	Внешнее прерывание 1	–	–	–	$002
PIN_CANGE	По изменению сигнала на любом из выводов	$002	$002	$002	–
LOW_LEVEL	По низкому уровню на входе порта B	–	–	–	$003
TIMER1 COMPA	По совпадению показаний таймера/счетчика T1 с содержимым контрольного регистра	–	–	$003	–
TIMER1 OVF	Переполнение таймера/счетчика T1	–	–	$004	–
TIMER0 OVF	Переполнение таймера/счетчика T0	$003	$003	$005	$004
EE_RDY	По готовности EEPROM	–	$004	$006	–
ANA_COMP	По сигналу от аналогового компаратора	$004	$005	$007	$005
ADC	По завершению преобразования в АЦП	–	–	$008	–

3.8. Таймеры-счетчики

Общие сведения

Любой микроконтроллер серии AVR содержит несколько встроенных таймеров. Причем по своему назначению их можно разделить на две категории. К **первой категории** относятся таймеры общего назначения. **Вторую категорию** составляет сторожевой таймер. Сторожевой таймер предназначен для автоматического перезапуска микроконтроллера в случае «зависания» его программы.

Зависанием *называют зацикливание программы в результате ошибки, допущенной программистом, либо в результате действия внешней помехи.*

Для каждой микросхемы нужен всего один сторожевой таймер. В любом микроконтроллере AVR такой таймер имеется.

Таймеры общего назначения используются для формирования различных интервалов времени и прямоугольных импульсов заданной частоты. Кроме того, они могут работать в режиме счетчика и подсчитывать тактовые импульсы заданной частоты, измеряя таким образом длительность внешних сигналов, а также при необходимости подсчитывать количество любых внешних импульсов.

По этой причине данные таймеры называют **«таймеры/счетчики»**. В микросхемах AVR применяются как восьмиразрядные, так и шестнадцатиразрядные таймеры/счетчики. Их количество для разных микроконтроллеров изменяется от одного до четырех. Точное количество таймеров/счетчиков для каждой микросхемы серии AVR можно узнать в разделе «Справочные материалы» виртуального диска. Все таймеры **обозначаются** числами от 0 до 3.

Пример

Timer/Counter0, Timer/Counter1 и т. д. В русскоязычной литературе их чаще именуют сокращенно T0, T1, T2, T3. Таймеры T0 и T2 в большинстве микроконтроллеров — восьмиразрядные. Таймеры T1 и T3 — шестнадцатиразрядные. Таймер T0 имеется в любой микросхеме AVR. Остальные добавляются по мере усложнения модели.

Каждый восьмиразрядный таймер представляет собой один восьмиразрядный регистр, который для микроконтроллера является регистром ввода-вывода. Этот регистр хранит текущее значение таймера и называется счетным регистром. Шестнадцатиразрядные таймеры имеют шестнадцатиразрядный **счетный регистр**. Каждый счетный регистр имеет свое имя.

Счетный регистр восьмиразрядного таймера именуется TCNTx, где «x» — это номер таймера. Для таймера T0 регистр называется TCNT0. Для таймера T2 — TCNT2. Шестнадцатиразрядные регистры именуются похожим образом. Отличие в том, что каждый шестнадцатиразрядный счетный регистр для микроконтроллера представляет собой два регистра ввода-вывода.

Один предназначен для хранения старших битов числа, а второй — для хранения младших битов. К имени регистра старших разрядов добавляется буква H, а для регистра младших разрядов добавляется буква L. Таким образом, счетный регистр таймера T1 — это два регистра ввода-вывода: TCNT1H и TCNT1L. Счетный регистр таймера T3 — это два регистра TCNT3H и TCNT3L.

Микроконтроллер может записать в любой счетный регистр любое число в любой момент времени, а также в любой момент прочитать содержимое любого счетного регистра. Когда таймер включается в режим счета, то на его вход начинают поступать счетные импульсы. После прихода каждого такого импульса содержимое счетного регистра увеличивается на единицу. **Счетными импульсами** могут служить как специальные тактовые импульсы, вырабатываемые внутри самого микроконтроллера, так и внешние импульсы, поступающие на специальные входы микросхемы. При переполнении счетного регистра его содержимое обнуляется, и счет начинается сначала.

Любой таймер жестко завязан с системой прерываний. Вызвать прерывание может целый ряд событий, связанных с таймером. Например, существует прерывание по переполнению таймера, по срабатыванию специальной схемы совпадения. Отдельные прерывания может вызывать сторожевой таймер.

Режимы работы таймеров

Таймеры микроконтроллеров семейства AVR могут работать в нескольких режимах. Разные микроконтроллеры имеют разные наборы режимов для своих таймеров. Для выбора режимов работы существуют специальные регистры — регистры управления таймерами. Для простых таймеров используется один регистр управления. Для более сложных — два регистра. Регистры управления таймером называются TCCRx (где «x» — номер таймера). Например, для таймера T0 используется один регистр с именем TCCR0. Для управления таймером T1 используется два регистра: TCCR1A и TCCR1B. При помощи регистров управления производится не только выбор соответствующего режима, но и более тонкая настройка таймера. Ниже перечислены все основные режимы работы таймера и их описание.

Режим Normal

Это самый простой режим. В этом режиме таймер производит подсчет приходящих на его вход импульсов (от тактового генератора или внешнего устройства) и вызывает прерывание по переполнению. Этот режим является единственным режимом работы для восьмиразрядных таймеров большинства микроконтроллеров семейства «Tiny» и для части микроконтроллеров семейства «Mega». Для всех остальных восьмиразрядных и всех шестнадцатиразрядных таймеров это всего лишь один из возможных режимов.

Режим «Захват» (Capture)

Суть этого режима заключается в сохранении содержимого счетного регистра таймера в определенный момент времени. Запоминание

происходит либо по сигналу, поступающему через специальный вход микроконтроллера, либо от сигнала с выхода встроенного компаратора.

Этот режим удобен в том случае, когда нужно измерить длительность какого-либо внешнего процесса. **Например** время, за которое напряжение на конденсаторе достигнет определенного значения. В этом случае напряжение с конденсатора подается на один из входов компаратора, а на второй его вход подается опорное напряжение.

Микроконтроллер должен одновременно запустить два этих процесса: подать напряжение на конденсатор; запустить таймер в режиме Capture.

Конденсатор начнет заряжаться, напряжение на нем при этом будет плавно расти. Одновременно счетчик таймера будет отсчитывать тактовые импульсы заданной частоты. В тот момент, когда напряжение на конденсаторе сравняется с опорным напряжением, логический уровень на выходе компаратора изменится на противоположный. По этому сигналу текущее значение счетного регистра запоминается в специальном **регистре захвата**. Имя этого регистра ICRx (для таймера T0 это будет ICR0, для T1 — ICR1 и т. д.). Одновременно вырабатывается запрос на прерывание.

Используя принцип измерения времени зарядки, удобно создавать простые схемы, работающие с различными аналоговыми датчиками (температуры, давления и т. д.). Если принцип работы датчика состоит в изменении его внутреннего сопротивления, то такой датчик можно включить в цепь зарядки конденсатора. Емкостные датчики можно подключать напрямую.

Режим «Сброс при совпадении» (CTC)

Для работы в режиме CTC используется специальный регистр — **регистр совпадения**. Если микроконтроллер содержит несколько таймеров, то для каждого из них существует свой отдельный регистр совпадения. Причем для восьмиразрядных таймеров регистр совпадения — это один восьмиразрядный регистр. Для шестнадцатиразрядных таймеров регистр совпадения — это два восьмиразрядных регистра.

Регистры сравнения также имеют свои имена. **Например**, регистр совпадения таймера T1 состоит из двух регистров: OCR1L и OCR1H. В ряде микроконтроллеров существуют два регистра совпадения. Так, во всех микроконтроллерах семейства «Tiny» существует два регистра совпадения для таймера T1. Это регистры OCR1A и OCR1B. Два регистра совпадения для таймера T1 имеет и микроконтроллер ATmega8x. Во втором случае как таймер, так и его регистры совпадения имеют шестнадцать разрядов.

Если регистр совпадения шестнадцатиразрядный, то физически он состоят из двух регистров ввода-вывода. **Например**, два регистра

совпадения таймера T1 микросхемы ATmega8x представляют собой четыре регистра ввода-вывода с именами OCR1AL, OCR1AH, OCR1BL, OCR1BH.

Как же используются регистры совпадения? Эти регистры включаются в работу только тогда, когда выбран режим CTC. В этом режиме, как и в предыдущем, таймер производит подсчет входных импульсов. Текущее значение таймера из его счетного регистра постоянно сравнивается с содержимым регистров совпадения.

Если таймер имеет два регистра совпадения, то для каждого из этих регистров производится отдельное сравнение. Когда содержимое счетного регистра совпадет с содержимым одного из регистров совпадения, произойдет вызов соответствующего прерывания. Кроме вызова прерывания, в момент совпадения может происходить одно из следующих событий:

- сброс таймера (верно только для регистров совпадения OCR1 и OCR1A);
- изменение состояния одного из выводов микроконтроллера (верно для всех регистров).

Произойдет или не произойдет одно или оба события из вышеперечисленных, определяется при настройке таймера.

Режим «Быстродействующий ШИМ» (Fast PWM)

> ***ШИМ*** *— расшифровывается как Широтно-Импульсная Модуляция. На английском это звучит как «Pulse Width Modulation» (PWM). Сигнал с ШИМ часто используется в устройствах управления.*

Сигнал с ШИМ можно, **например**, использовать для регулировки скорости вращения электродвигателя постоянного тока. Для этого вместо постоянного напряжения на двигатель подается прямоугольное импульсное напряжение. Благодаря инерции двигателя импульсы сглаживаются, и двигатель вращается равномерно. Меняя скважность импульсов (то есть отношение периода импульсов к их длительности), можно изменять среднее напряжение, приложенное к двигателю и, тем самым, менять скорость его вращения.

Точно таким же образом можно управлять и другими устройствами. **Например,** нагревательными элементами, осветительными приборами. **Преимущество** импульсного управления — в высоком КПД. Импульсные управляющие элементы рассеивают гораздо меньше паразитной мощности, чем управляющие элементы, работающие в аналоговом режиме.

Для формирования сигнала ШИМ используются те же самые регистры совпадения, которые работают и в режиме CTC. Формирование сигнала ШИМ может осуществляться несколькими разными способами. Работа таймера в режиме Fast PWM проиллюстрирована на **рис. 3.5**.

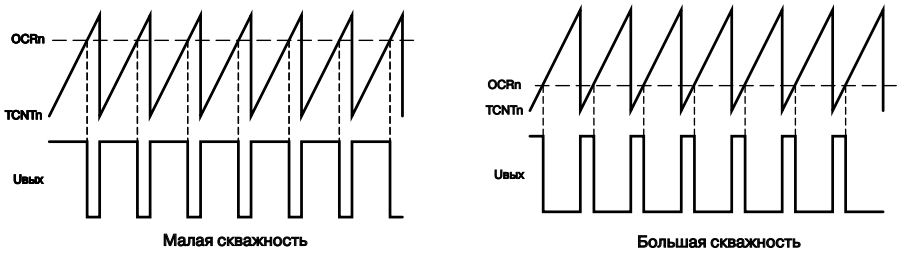


Рис. 3.5. *Работа таймера в режиме Fast PWM*

Сигнал с ШИМ формируется на специальном выходе микроконтроллера. На вход таймера подаются импульсы от **системного генератора**. Таймер находится в состоянии непрерывного счета. При переполнении таймера его содержимое сбрасывается в ноль, и счет начинается сначала. В режиме ШИМ переполнение таймера не вызывает прерываний. На **рис. 3.5** это показано в виде пилообразной кривой, обозначенной как **TCNTn**. Кривая представляет собой зависимость содержимого счетного регистра от времени.

Содержимое счетного регистра непрерывно сравнивается с содержимым регистра совпадения. Пока число в регистре OCRn больше, чем число в счетном регистре таймера (TCNTn), напряжение на выходе ШИМ равно логической единице. Когда же в процессе счета содержимое счетного регистра TCNTn станет больше содержимого OCRn, на выходе ШИМ установится нулевой потенциал.

В результате на выходе мы получим прямоугольные импульсы. Скважность этих импульсов будет зависеть от содержимого регистра OCRn. Чем меньше число в OCRn, тем выше скважность выходных импульсов. На **рис. 3.5** показана скважность импульсов для двух разных значений регистра OCRn.

Если содержимое OCRn достигнет своего максимального значения, то импульсы на выходе ШИМ исчезнут, и там постоянно будет присутствовать логическая единица. При уменьшении числа в OCRn появятся импульсы малой скважности (длительность почти равна периоду). Если плавно уменьшать число в OCRn, то скважность будет плавно уменьшаться. Когда содержимое OCRn достигнет нуля, импульсы на выходе ШИМ также исчезнут, и там установится логический ноль.

Режим «ШИМ с точной фазой» (Phase Correct PWM)

Описанный в предыдущем разделе режим ШИМ имеет один **недостаток**. При изменении длительности импульсов меняется и их фаза. Центр каждого импульса как бы сдвигается во времени. При управлении электродвигателем такое поведение фазы нежелательно. Поэтому в микроконтроллерах AVR предусмотрен еще один режим ШИМ. Это

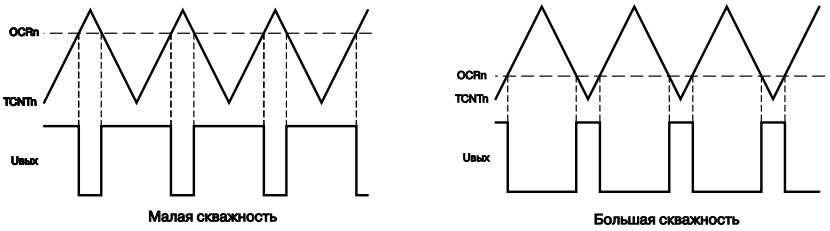


Рис.3.6. *Работа таймера в режиме Phase Correct PWM*

ШИМ с точной фазой. Принцип работы таймера в этом режиме изображен на **рис. 3.6**.

Отличие режима «Phase Correct PWM» от режима «Fast PWM» заключается в режиме работы счетчика. Сначала счетчик считает так же, как и в предыдущем режиме (от каждого входного импульса его значение увеличивается на единицу). Достигнув своего максимального значения, счетчик не сбрасывается в ноль, а переключается в режим реверсивного счета.

Теперь уже от каждого входного импульса его содержимое уменьшается на единицу. В результате пилообразная кривая, отображающая содержимое счетного регистра TCNTn, становится симметричной, как показано на **рис. 3.6**. Система совпадения работает так же, как и в предыдущем случае.

Благодаря симметричности сигнала на таймере, фаза выходных импульсов в процессе регулировки скважности не изменяется. Середина каждого импульса строго привязана к точке смены направления счета таймера.

Недостатком режима «Phase Correct PWM» можно считать в два раза меньшую частоту выходного сигнала. Это существенно уменьшает динамичность регулирования. Кроме того, при использовании внешних фильтров для преобразования импульсного сигнала ШИМ в аналоговый, схема с более низкой частотой потребует применения комплектующих с большими габаритами и массой.

Асинхронный режим

В некоторых моделях микроконтроллеров таймер может работать в **асинхронном режиме**. В этом режиме на вход таймера подается либо частота от внутреннего кварцевого генератора, либо от внешнего генератора. Счетчик не вырабатывает никаких прерываний и дополнительных сигналов. В этом режиме он работает в качестве часов реального времени. Микроконтроллер может предустанавливать содержимое счетного регистра. А затем в любой момент он может считать это содержимое, получив, таким образом, текущее значение реального времени.

Предделители таймеров/счетчиков

Как уже говорилось ранее, каждый таймер микроконтроллера может работать от двух разных источников тактовых импульсов. Либо это внешние импульсы, либо импульсы, вырабатываемые внутренней схемой микроконтроллера. Какой бы источник сигналов ни был выбран, перед тем, как попасть на вход таймера, этот сигнал проходит схему предварительного делителя. **Предварительный делитель** предназначен для того, чтобы расширить диапазон формируемых частот и длительностей таймера. Каждая микросхема AVR имеет свою структуру предварительного делителя для таймеров/счетчиков. Упрощенная схема одного из вариантов предварительного делителя приведена на **рис. 3.7**.

Как видно из схемы, частота внутреннего тактового генератора CLK поступает на **специальный десятиразрядный делитель**. С выходов делителя снимаются сигналы CLK/8, CLK/32, CLK/64, CLK/128, CLK/256 и CLK/1024. Все эти сигналы поступают на входы данных мультиплексора. На адресные входы мультиплексора поступают сигналы от трех разрядов регистра управления таймером (TCCRn).

Таким образом, записывая в разряды CSn0, CSn1, CSn2 различные значения, можно выбирать один из восьми **режимов работы предделителя**. В зависимости от выбранного режима, на выход схемы могут поступать сигнал с одного из выходов десятиразрядного делителя, прямой сигнал с тактового генератора либо нулевой логический уровень (входа D0). В последнем случае сигнал на входе таймера будет отсутствовать, и его работа приостанавливается.

Схема, приведенная на **рис. 3.7**, не является стандартом для всех микроконтроллеров серии AVR. Она отражает лишь общий принцип построения предделителей. В разных моделях это сделано немного по-разному.

На **рис. 3.8** приведена еще одна схема предделителя. Эта схема, в отличие от предыдущей, предусматривает подачу на входы таймеров

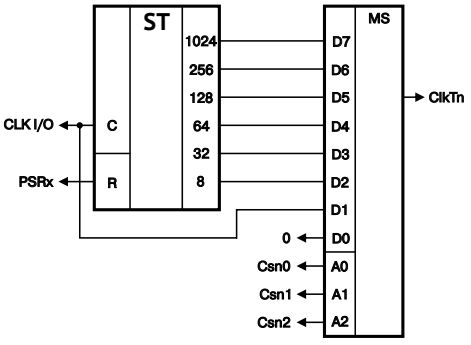


Рис. 3.7. *Предделитель для таймера*

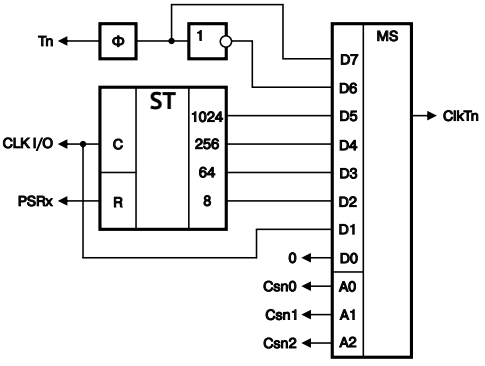


Рис. 3.8. *Предделитель с входом для внешнего сигнала*

тактового сигнала от внешнего источника. Для этого количество сигналов, снимаемых с десятиразрядного делителя, уменьшено до четырех. CLK/32 и CLK/128 исключены. Зато в схеме появились цепи, через которые на вход таймера может поступать внешние импульсы.

Эти импульсы должны подаваться на вход Tn. С этого входа импульсы поступают на формирователь, который осуществляет их предварительную обработку (приближает их форму к прямоугольной). Затем импульсы поступают на вход D7 дешифратора. На вход D6 поступают те же импульсы, но только в инвертированном виде. В результате для схемы, показанной на **рис. 3.8**, мы получаем следующие восемь режимов работы:

- **режим 0** — отсутствие импульсов;
- **режим 1** — прямой сигнал от внутреннего генератора;
- **режимы 2…5** — один из сигналов с делителя;
- **режим 6** — инверсный сигнал с внешнего входа;
- **режим 7** — прямой внешний сигнал.

3.9. Другие встроенные периферийные устройства

Аналоговый компаратор

Мы уже упоминали о **компараторе**. Он **предназначен** для сравнения напряжений на двух специальных внешних входах. Такие входы имеют **названия**: AIN0 (неинвертирующий); AIN1 (инвертирующий).

Не забываем, что каждый из этих входов совмещен с одной из линий какого-либо порта ввода-вывода. Если напряжение на входе AIN0 больше, чем напряжение на входе AIN1, то на выходе компаратора — логическая единица. В противном случае там логический ноль.

Этот результат сохраняется в одном из разрядов специального регистра ввода-вывода, предназначенного для работы с компаратором. Регистр называется **ACSR**. А разряд, куда выводится выходной сигнал компаратора, тоже имеет свое название. Он называется **ACO**. Другой разряд под названием **ACD** того же регистра отвечает за включение/выключение компаратора. Еще два разряда **ACIS0** и **ACIS1** определяют способ влияния сигнала с выхода компаратора на последующие схемы. Есть **три варианта**: любое изменение на выходе; изменение с единицы на ноль; изменение с ноля на единицу.

Как видите, отдельные разряды некоторых регистров тоже иногда различаются не по номерам, а по названиям. Это позволяет в разных микроконтроллерах использовать для одной и той же цели разные разряды регистров. В этом случае имя разряда остается прежним. Хотя чаще всего номера разрядов не меняются.

Схема компаратора имеет специальный внутренний источник опорного напряжения, который может быть подключен к неинвертирующему входу компаратора. Подключением внутреннего источника управляет разряд ACBG регистра ACSR. Кроме того, на инвертирующий вход компаратора можно подать сигнал с любого входа АЦП. Этим переключением управляют остальные разряды регистра ACSR. Что такое АЦП и какие АЦП применяются в микроконтроллерах серии AVR, мы рассмотрим в следующем разделе.

Аналого-цифровой преобразователь

Аналого-цифровой преобразователь (АЦП) **предназначен** для преобразования аналогового напряжения в цифровую форму. На вход АЦП поступает обычное аналоговое напряжение. Преобразователь измеряет величину этого напряжения и выдает на выходе цифровой код, соответствующий этой величине.

АЦП **применяются** в микропроцессорных системах управления, которые должны управлять различными аналоговыми процессами. **Например**, микропроцессорный стабилизатор напряжения, цифровой вольтметр и т. п.

Встроенный АЦП имеют не все микроконтроллеры AVR. Это и понятно. Ввод аналоговой информации нужен далеко не всегда. В микроконтроллерах AVR применяется десятиразрядное АЦП последовательного приближения. Микросхемы, имеющие в своем составе встроенный АЦП, обязательно имеют раздельное питание для цифровой и для аналоговой частей схемы. Поэтому они имеют два вывода питания и два вывода общего провода.

Кроме того, один из выводов зарезервирован для подачи на микросхему внешнего опорного напряжения. Опорное напряжение используется в схеме АЦП для оценки уровня входного сигнала. От стабильности опорного напряжения зависит точность измерения.

Каждый АЦП снабжен **многоканальным аналоговым коммутатором (мультиплексором)**, который позволяет измерять аналоговое напряжение с нескольких разных входов. Количество входов АЦП у разных микросхем различное. Существуют варианты в 4, 6, 8 и 11 входов.

Обычно измеряемый сигнал прикладывается между соответствующим входом АЦП и аналоговым общим проводом. Такие входы называются **несимметричными**. В некоторых микроконтроллерах имеется режим, в котором входы АЦП объединяются попарно и образуют **дифференциальные входы**. Дифференциальные входы отличаются от обычных тем, что измеряемый сигнал прикладывается между двумя **входами**: **прямым** и **инверсным**. При этом наводимые помехи компенсируются, а полезный сигнал проходит без изменений. Такие входы называются **симметричными**.

Процесс преобразования напряжения в код занимает 13 или 14 тактов. За это время происходит подбор кода методом последовательных приближений. По окончании процесса преобразования вырабатывается запрос на прерывание. Результат преобразования записывается в пару регистров ADCH, ADCL. Из шестнадцати разрядов этой регистровой пары используются только 10. Остальные всегда равны нулю. Причем могут использоваться либо десять старших разрядов (ADCH7—ADCH0, ADCL7, ADCL6), либо десять младших разрядов (ADCH1, ADCH0, ADCL7—ADCL0). Это зависит от выбранного вами режима работы.

АЦП могут работать как **в одиночном режиме**, так и в **непрерывном**. В непрерывном режиме преобразования идут один за другим. В одиночном режиме процесс преобразования запускается однократно от одного из следующих событий:

- прерывания от аналогового компаратора; внешнего прерывания INT0;
- прерывания по событию «Совпадение» одного из таймеров;
- прерывания по переполнению одного из таймеров;
- прерывания по событию «Захват» одного из таймеров.

Управление всеми режимами работы АЦП производится при помощи двух специальных регистров **ADMUX** и **ADCSR**. Регистр ADMUX предназначен для управления входным аналоговым мультиплексором. Регистр ADCSR предназначен для выбора режима работы АЦП.

Процесс преобразования в АЦП синхронизируется от внутреннего генератора микроконтроллера. Тактовый сигнал от генератора поступает на АЦП через предварительный делитель с программируемым коэффициентом деления. **Коэффициент деления** зависит от значения разрядов ADPS0, ADPS1 и ADPS2 регистра ADCSR и может принимать значения 2, 4, 8, 16, 32, 64 и 128.

Наибольшая точность преобразования достигается тогда, когда тактовая частота преобразования находится в диапазоне 50—200 кГц. Поэтому рекомендуется выбирать такой коэффициент деления, чтобы тактовая частота модуля АЦП находилась в этом диапазоне.

Последовательный канал (UART/USART)

Некоторые микроконтроллеры серии AVR имеют:

- встроенный универсальный последовательный асинхронный приемопередатчик (UART);
- универсальный последовательный синхронно/асинхронный приемопередатчик (USART).

Некоторые модели имеют даже сразу два таких канала (см. раздел «Справочные материалы» виртуального диска). Каналы UART (USART) **предназначены** для обмена информацией между микроконтролле-

ром и любым внешним устройством. Протокол UART (USART) — это довольно распространенный протокол последовательной передачи информации. Такой протокол, в частности, использует последовательный порт компьютера (COM-порт). При помощи UART (USART) можно организовывать линию связи не только между двумя микроконтроллерами, но и между микроконтроллером и компьютером.

Для обмена информацией UART (USART) использует две линии: **RxD** и **TxD**. Одна линия используется для приема информации, другая — для передачи. В модулях UART посылка может быть восьми- или девятиразрядной. В модуле USART ее длина может составлять от 5 до 9 разрядов. Кроме того, модули могут вырабатывать и контролировать разряд четности.

Скорость передачи определяется специальным внутренним программируемым делителем и частотой тактового генератора микроконтроллера. Коэффициент деления делителя может изменяться от 2 до 65536. Для того, чтобы последовательный канал мог нормально обмениваться информацией с внешними устройствами, необходимо так подобрать коэффициент деления и частоту тактового генератора, чтобы получить одну из стандартных скоростей передачи информации. **Например,** 2400, 4800, 9600, 14400, 19200, 28800 бит в секунду.

Последовательный периферийный интерфейс (SPI)

Это специальный последовательный интерфейс, разработанный для связи микроконтроллеров между собой. Канал SPI использует для передачи информации три линии: линию MISO (Master Input / Slave Output); линию MOSI (Master Output / Slave Input); линию SCK (Тактовый сигнал).

В микроконтроллерах AVR канал SPI может выполнять двоякую функцию. Так, при помощи этого интерфейса можно не только организовать последовательный канал обмена информацией между двумя микроконтроллерами, но и между микроконтроллером и любым периферийным устройством, имеющим SPI-интерфейс.

Существует целый набор подобных устройств: цифровые потенциометры, ЦАП/АЦП, внешние Flash-ПЗУ и др.

Второе предназначение канала SPI — программирование микроконтроллера. Именно через этот канал осуществляется последовательное программирование памяти программ и внутреннего EEPROM. Такой усеченный канал SPI имеется практически в каждом микроконтроллере AVR. Преимущество программирования через SPI состоит в том, что такой способ позволяет программировать микросхему, не вынимая ее из отлаживаемого устройства. Это так называемое внутрисхемное программирование. Необходимо лишь позаботиться, чтобы другие сигналы на выводах, служащих линиями SPI интерфейса, отключались в момент программирования. Обычно в плате отлаживаемого

устройства предусматривают специальный разъем, куда и подключается **программатор**. Подробнее о программаторах будет рассказано в **главе 4** (**раздел 4.3**).

Последовательный двухпроводный интерфейс (TWI)

Этот интерфейс является полным аналогом шины I²C фирмы Philips, получившей широкое распространение в различных системах управления бытовой и промышленной техникой. Интерфейс позволяет объединить вместе до 128 устройств, подключив их к одной двухпроводной шине. Линии шины I²C имеют следующие названия: **линия SCL** (линия тактового сигнала); **линия SDA** (линия передачи данных).

Интерфейс позволяет обмениваться данными между ведущим устройством, которым обычно является микроконтроллер, и любым из внешних устройств, подключенных к двухпроводной линии. При этом ведущее устройство может как передавать данные на ведомое, так и принимать данные из него.

Наличие интерфейса для работы с I²C шиной позволяет применять микроконтроллеры в системах управления телевизоров, радиоприемников и т. п. Специализированные микросхемы для телевизионных приемников, радиоприемников, магнитол с I²C-интерфейсом в настоящее время становятся фактически стандартом. Кроме того, в настоящее время широко применяются контроллеры дисплеев на жидких кристаллах, микросхемы Flash-памяти и другие устройства, управляемые по I²C-шине.

3.10. Другие ячейки

Конфигурационные ячейки

Все контроллеры AVR имеют множество режимов работы. Некоторые из режимов невозможно переключить программным путем, используя регистры управления. **Например**, в большинстве моделей микроконтроллеров в качестве тактового генератора можно применять встроенный параметрический генератор с подстраиваемой частотой. Два освободившихся контакта плюс контакт аппаратного сброса (Reset) можно использовать как дополнительный трехразрядный порт ввода-вывода. Естественно, что перевести в такой режим микросхему нужно еще до включения в схему.

Для подобных целей фирма Atmel ввела в свои микроконтроллеры новый настроечный элемент — **программируемые переключатели режимов**. Эти переключатели выполнены в виде специальных ячеек, которые, по сути, являются еще одним видом перепрограммируемой

энергонезависимой памяти. Все конфигурационные ячейки объединяются в байты. Различные микросхемы AVR имеют от одной до трех байтов конфигурационных ячеек.

Каждый конфигурационный переключатель предназначен для того, чтобы изменять какой-либо один параметр или режим работы микроконтроллера. В документации каждый такой переключатель имеет свое определенное имя. Некоторые биты конфигурационных ячеек объединены в группы. Например, группа из четырех битов **CKSEL3—0** позволяет выбирать режимы синхронизации. Разные модели микроконтроллеров имеют различные наборы конфигурационных ячеек. По терминологии фирмы Atmel, конфигурационные ячейки называются **Fuse Bits**. Поэтому для удобства и краткости эти ячейки часто называют «Фусами», или **Fuse-ячейками**.

Запись и чтение конфигурационных ячеек возможны только при помощи программатора. Все незапрограммированные Fuse-ячейки содержат единицу. При программировании в ячейку записывается ноль. Некоторые ячейки программируются еще на заводе. Состояние всех конфигурационных ячеек для каждой конкретной микросхемы смотрите в документации на эту микросхему.

Ячейки защиты и идентификации

Исторически сложилось так, что даже самые первые модели микроконтроллеров имели программируемые ячейки защиты информации. Микроконтроллеры AVR также имеют такую защиту. Это специальные ячейки, подобные конфигурационным.

Каждый микроконтроллер имеет, как минимум, две защитные ячейки LB1 и LB2. Запись и чтение этих ячеек возможны только в режиме программирования. При записи нуля в LB1 блокируется запись данных во Flash- и EEPROM-память. Одновременно блокируется возможность изменять конфигурационные ячейки.

Если записать ноль еще и в LB2, то блокируется и возможность чтения всех данных. После этого узнать содержимое вашей программы будет невозможно. Для повторного использования микроконтроллера нужно подать команду «Стирание микросхемы». При этом вся информация теряется, а способность чтения и модификации возвращается.

В микроконтроллерах семейства «Mega» имеются дополнительные ячейки защиты BLB02, BLB01, BLB12, BLB11. Они служат для ограничения доступа к различным областям памяти программ.

Еще одна группа ячеек — это **ячейки идентификации**. Любой микроконтроллер имеет три ячейки идентификации. Эти ячейки доступны только для чтения и содержат информацию о производителе и модели микроконтроллера.

ГЛАВА 4

ОСВАИВАЕМ ПРОГРАММЫ ОТЛАДКИ И ТРАНСЛИРОВАНИЯ

4.1. Программная среда AVR Studio

Отладка программы

На данном этапе нашего обучения, казалось бы, логично было бы перейти к главному вопросу всего курса — рассказать, наконец, на примерах, как пишутся программы для микроконтроллеров. В предыдущих изданиях книги именно так и было сделано. Но, изучив отзывы читателей, и вопросы, которые задают читатели после прочтения прежнего варианта книги, автор пришел к пониманию, что главу, посвященную программированию очень полезно предварить небольшой главой, раскрывающим еще один важный и полезный вопрос. Дело в том, что хитрости программирования очень трудно постигать чисто теоретически. Очень трудно бывает представить себе, как работает изучаемая программа, просто читая описание алгоритма. Гораздо эффективнее было бы попробовать работу программы на практике. И компьютерные технологии позволяют это сделать! Попробовать программу в работе программистам позволяет специальная компьютерная программа, которая называется **отладчик**.

Программы-отладчики существуют столько же, сколько существует само программирование. Основное назначение отладчика — поиск ошибок в разрабатываемой программе. Английское название такого вида программ — Debugger, а процесс отладки называется Debug. От слова Bug — ошибка. Второе значение английского слова bug — жук или клоп. Так что отладка программ сродни ловле блох! В нашем случае, когда о разработке новых программ говорить еще рано, отладчик будет нам полезен, как способ своими руками попробовать в работе изучаемую программу. Подборку уже готовых программных примеров вы увидите в следующей главе. Одна из главных функций любого отладчика — возможность пошагового выполнения разрабатываемой программы. После выполнения каждого очередного шага программист может видеть состояние всех регистров и ячеек памяти микроконтроллера. А точнее, видеть какое значение, записано в каждый регистр или ячейку памяти.

Выбор программы — отладчика тесно связан с языком программирования, на котором написана сама программа. И больше того, тут важна так же конкретная реализация языка программирования. Именно по этому в настоящее время почти не применяют отдельных программ-отладчиков. Вместо этого получили распространения программные среды. **Программная среда** — это большая многофункциональная программа, включающая в себя редактор текста программ, транслятор программы, который на основе текста программы генерирует машинный код для прошивки его в микроконтроллер. Эта же среда включает в себя программный отладчик, а так же, часто в программную среду включают программу для управления программатором. **Программатор** — это некое внешнее электронной устройство, подключаемое к компьютеру, при помощи которого подготовленный программный код прошивается в программную память микроконтроллера. В нашей книге для программирования на языке Ассемблера мы будем использовать программную среду AVR Studio фирмы Atmel. А для программирования на языке СИ используем систему CodeVision румынской фирмы «HP Infotech».

В процессе выполнения программы под управлением отладчика программист может на экране компьютера:

- видеть содержимое любого регистра микроконтроллера;
- видеть содержимое ОЗУ и EEPROM;
- наблюдать за последовательностью выполнения команд, контролируя правильность отработки условных и безусловных переходов;
- наблюдать за работой таймеров, отработкой прерываний.

В процессе отладки программист также может наблюдать логические уровни на любом внешнем выходе микроконтроллера. А также имитировать изменение сигналов на любом входе. Процесс отладки позволяет программисту убедиться в том, что разрабатываемая им программа работает так, как он задумал. Большинство ошибок в программе обнаруживаются именно в процессе отладки.

Существует **три основных вида отладчиков**: программные, аппаратные, комбинированные программно-аппаратные.

Программный отладчик *– это компьютерная программа, которая имитирует работу процессора на экране компьютера. Она не требует наличия реальной микросхемы или дополнительных внешних устройств и позволяет отладить программу чисто виртуально.*

Программный отладчик

Программный отладчик позволяет проверить только логику работы программы. При помощи такого отладчика невозможно проверить работу схемы в режиме реального времени или работу

всего микропроцессорного устройства в комплексе. То есть невозможно гарантировать правильную работу и всех подключенных к микроконтроллеру дополнительных микросхем и элементов.

Аппаратный отладчик

При помощи этих выводов отладочная плата может быть включена в реальную схему. Возникающие в процессе отладки электрические сигналы можно наблюдать при помощи осциллографа. Можно нажимать реальные кнопки и наблюдать работу светодиодов и других индикаторов.

Второй вид отладчиков *— аппаратный отладчик. Основа такого отладчика — специальная плата, подключаемая к компьютеру, работающая под его управлением и имитирующая работу реальной микросхемы микроконтроллера. Плата имеет выводы, соответствующие выводам реальной микросхемы, на которых в процессе отладки появляются реальные сигналы.*

В то же самое время на экране компьютера мы так же, как и в предыдущем случае, можем видеть всю информации об отлаживаемой программе:

- наблюдать содержимое регистров, ОЗУ, портов ввода-вывода;
- контролировать ход выполнения программы.

В аппаратном отладчике мы можем так же, как и в программном, выполнять программу в пошаговом режиме и применять точки останова. **Недостатком** аппаратного отладчика является его высокая стоимость.

Полнофункциональные программные имитаторы электронных устройств

Существует и **третий вид отладчиков**. Это полнофункциональные программные имитаторы электронных устройств. Такие программы позволяют на экране компьютера «собрать» любую электронную схему, включающую в себя самые разные электронные компоненты:

- транзисторы;
- резисторы;
- конденсаторы;
- операционные усилители;
- логические и цифровые микросхемы, в том числе и микроконтроллеры.

Такие программы обычно содержат обширные базы электронных компонентов и конструктор электронных схем. Собрав схему, вы можете виртуально записать в память микроконтроллера вашу программу, а затем «запустить» всю схему в работу.

Для контроля результатов работы схемы имитатор имеет **виртуальные вольтметры**, **амперметры** и **осциллографы**, которые вы можете

«подключать» к любой точке вашей схемы, «измерять» различные напряжения, а также «снимать» временные диаграммы.

Такие программы в настоящее время получают все большее распространение. Они позволяют разработать любую схему с микроконтроллером или без него, без использования паяльника и реальных деталей. На экране компьютера можно полностью отладить свою схему и лишь потом браться за паяльник.

Недостатком данного отладчика является то, что он требует значительных вычислительных ресурсов. Особенно в том случае, когда отлаживается схема, включающая как микроконтроллер, так и некоторую аналоговую часть. Кроме того, имитатор не всегда верно имитирует работу некоторых устройств. Однако подобные программы имеют очень большие перспективы. В рамках данной книги я не буду рассматривать подобную программу, так как такая задача достойна отдельной книги.

Внутренний отладчик микроконтроллеров AVR

Еще один аппаратный способ отладки заложен конструктивно в некоторые модели микроконтроллеров AVR. В частности, микроконтроллер ATtiny2313 поддерживает такой способ отладки.

Для обеспечения возможности аппаратной отладки такие микроконтроллеры имеют, во-первых, с**пециальную однопроводную линию debugWIRE**, которая обычно совмещена с входом RESET. Эта линия используется специальной платой-отладчиком для управления микроконтроллером в процессе отладки. Кроме того, в систему команд такого микроконтроллера включена команда `break`, которая может использоваться **для создания программных точек останова**.

Для того, чтобы использовать подобный режим отладки, необходимо иметь в своем распоряжении **специальную отладочную плату**, которая должна поддерживать этот режим. Кроме того, подобный режим должна поддерживать и **инструментальная программа-отладчик**.

В процессе отладки программист проставляет на экране компьютера в нужных местах отлаживаемой программы точки останова. Затем он запускает эту программу под управлением отладчика. Отладчик автоматически вставляет в отлаживаемую программу команды `break` в тех местах, где программист поставил точки останова. А команды, которые должны быть записаны в месте вставки команд `break`, запоминает в своей памяти.

Затем он автоматически «прошивает» полученный таким образом текст программы в программную память отлаживаемого микроконтроллера и запускает ее в работу. Микроконтроллер выполняет заложенную в него программу до тех пор, пока не встретится команда `break`.

Получив эту команду, микроконтроллер приостанавливает выполнение программы и передает управление отладчику.

Далее отладчик управляет микроконтроллером при помощи **интерфейса debugWIRE**. Этот интерфейс позволяет считать содержимое всех регистров микроконтроллера и других видов памяти. Прочитанная информация отображается на экране компьютера. Затем отладчик ждет команд от оператора. Под управлением отладчика микроконтроллер может принудительно выполнить любую команду из своей системы команд.

Это дает возможность легко реализовать пошаговое выполнение программы, а также выполнение тех команд, которые были заменены на `break`. Все управление осуществляется посредством интерфейса debugWIRE, который позволяет передавать информацию как от отладчика в микроконтроллер, так и в обратном направлении.

Преимуществом такого способа отладки является то, что в данном случае происходит не имитация микроконтроллера, а используется реальная микросхема. При этом работа в режиме отладки наиболее полно приближается к реальному режиму работы.

Недостаток — частое «перешивание» программной памяти микроконтроллера. Изменять содержимое этой памяти приходится каждый раз при установке новых или снятии старых точек останова. Если учесть, что допустимое количество перезаписи программной памяти составляет 10000 циклов, то при длительном процессе отладки это количество может исчерпаться, и микросхема выйдет из строя.

Программная среда «AVR Studio»

Именно этот отладчик удобнее всего использовать для детального изучения программных примеров в следующей (пятой) главе книги.

Фирма Atmel, разработчик микроконтроллеров AVR, очень хорошо позаботилась о сопровождении своей продукции. Для написания программ, их отладки, трансляции и прошивки в память микроконтроллера фирма разработала и бесплатно распространяет **специализированную среду разработчика под названием «AVR Studio»**. Инсталляционный пакет этой инструментальной программы можно свободно скачать с сайта фирмы. Подсказка для поиска программы на сайте: Home > Microcontrollers > Atmel AVR 8- and 32-bit Microcontrollers > megaAVR > AVR Studio 4. Точный адрес вы найдете в файле readme.txt в каталоге с инсталляционным пакетом AVR Studio на прилагаемом к книге диске.

Программная среда «AVR Studio» — это мощный современный программный продукт, позволяющий производить все этапы разработки программ для любых микроконтроллеров серии AVR. Пакет включает в себя специализированный текстовый редактор для написания программ, мощный программный отладчик.

Кроме того, «AVR Studio» позволяет управлять целым рядом подключаемых к компьютеру внешних устройств, позволяющих выполнять аппаратную отладку, а также программирование («прошивку») микросхем AVR.

Познакомимся подробнее с этим удобным программным инструментом для программистов. Программная среда «AVR Studio» работает не просто с программами, а с проектами. Для каждого проекта должен быть отведен свой отдельный каталог на жестком диске. В AVR Studio одновременно может быть загружен только один проект.

При загрузке нового проекта предыдущий проект автоматически выгружается. Проект содержит всю информацию о разрабатываемой программе и применяемом микроконтроллере. Он состоит из целого набора файлов.

Главный из них — **файл проекта**. Он имеет расширение `aps`. Файл проекта содержит сведения о типе процессора, частоте тактового генератора и т. д. Он также содержит описание всех остальных файлов, входящих в проект. Все эти сведения используются при отладке и трансляции программы.

Кроме файла `aps`, проект должен содержать хотя бы один **файл с текстом программы**. Такой файл имеет расширение `asm`. Недостаточно просто поместить файл `asm` в директорию проекта. Его нужно еще включить в проект. Как это делается, мы увидим чуть позже. Проект может содержать несколько файлов `asm`. При этом один из них является главным. Остальные могут вызываться из главного при помощи оператора `.include`. На этом заканчивается список файлов проекта, которые создаются при участии программиста.

Но типичный проект имеет гораздо больше файлов. Остальные файлы проекта появляются в процессе трансляции. Если ваша программа не содержит критических ошибок и процесс трансляции прошел успешно, то в директории проекта автоматически появляются следующие файлы: файл, содержащий результирующий код трансляции в `hex` формате, файл `map`, содержащий все символьные имена транслируемой программы со своими значениями, листинг-трансляции (`lst`) и другие вспомогательные файлы. Однако для нас будет важен лишь hex-файл (файл с расширением `hex`). Именно он будет служить источником данных при прошивке программы в программную память микроконтроллера.

Главная панель программы «AVR Studio»

Теперь перейдем к описаню интерфейса. На **рис. 4.1** показано, как выглядит главная панель программы «AVR Studio». На самом деле «AVR Studio» имеет очень гибкий интерфейс, и внешний вид может сильно

отличаться от варианта, показанного на рисунке. Но мы будем рассматривать случай, когда выбраны установки по умолчанию.

Главная панель программы AVR Studio разделена на пять основных областей. На **рис. 4.1** они обозначены цифрами 1, 2, 3, 4 и 5. Главная область помечена номером 2. Здесь открывается одно или несколько окон, содержащих текст программы на ассемблере. Области 1, 3, 4, 5 — вспомогательные. Каждая из них представляет из себя отдельное окно и имеет свое назначение.

Окно «Project» (область 1). Содержит полную информацию по текущему загруженному проекту. Информация представлена в виде дерева. Разные ветви этого дерева описывают все исходные и результирующие файлы проекта, все метки, процедуры и присоединяемые файлы. В процессе отладки тут же появляется вкладка отражающая шаги программы и время ее выполнения от начала до текущего шага.

Окно «I/O View» (область 3). Просмотр ввода-вывода. Это самое полезное окно. На нем в графическом виде показаны все ресурсы микроконтроллера:

- порты ввода-вывода;
- таймеры;
- компараторы;
- АЦП;
- регистры общего назначения и т. д.

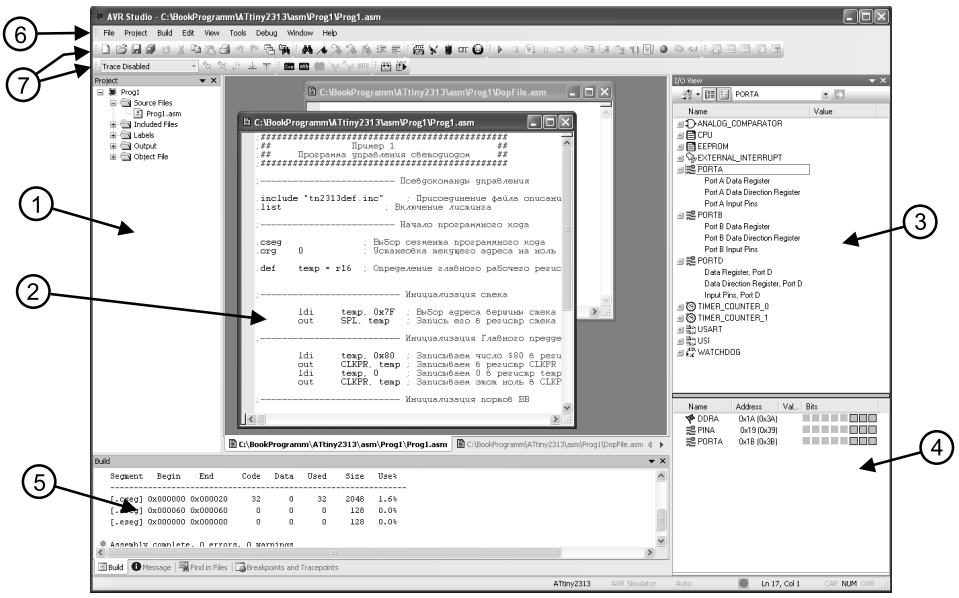


Рис. 4.1. *Программа AVR Studio*

Вся информация также представлена **в виде дерева**. Каждая «ветвь» этого дерева — это отдельный элемент. Если какой-либо элемент состоит из других элементов, то его можно раскрыть и увидеть эти элементы.

Каждый элемент (порт ввода-вывода, таймер, система прерывания и т. п.) имеет в своем составе несколько регистров. Эти регистры отображаются в специальном окне (область 4). Каждый регистр, в свою очередь, так же можно раскрыть.

Если какая-либо ветвь может быть раскрыта, то в своем основании она имеет квадратик с плюсиком внутри. Двойной щелчок на этом плюсике раскрывает ветвь. В раскрытой ветви плюсик превращается в минус. Повторный двойной щелчок по квадратику закрывает раскрытую ветвь.

На **рис. 4.2** для наглядности раскрыты ветви всех трех портов ввода-вывода и регистры, связанные с EEPROM. Вы можете видеть:

- полный состав управляющих регистров для каждого из устройств;
- их названия и адреса;
- состав и название каждого бита (если биты имеют свои названия).

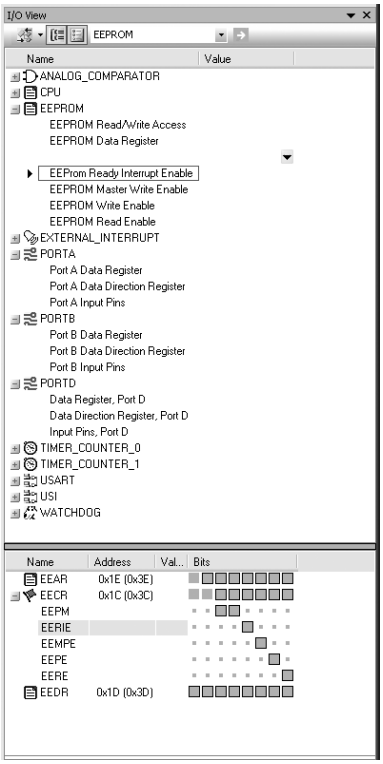


Рис. 4.2. *Окно ресурсов микроконтроллера*

Для наглядности на **рис. 4.2** раскрыта ветвь, соответствующая регистру EECR, и вы можете видеть все его биты.

В процессе отладки в этом окне вы увидите не только название и состав всех ресурсов, но и их содержимое. **Содержимое** будет отображаться как в шестнадцатиричном виде, так и путем затемнения квадратиков, отображающих отдельные биты конкретных регистров.

Затемненный квадратик означает, что бит равен единице. **Светлый квадратик** говорит о том, что бит равен нулю. Вы также можете оперативно менять это содержимое прямо в этом окне. Для изменения значения бита достаточно двойного щелчка мышки в соответствующем квадратике. Существуют и другие способы изменения содержимого различных регистров и ячеек памяти в процессе отладки.

В нижней части главной панели находится **окно «Build» (область 5).** Это окно служит, в основном, для вывода различных сообщений. Оно также содер-

жит ряд вкладок. По умолчанию их четыре. **Первая вкладка называется «Build»**. На вкладке «Build» отражается процесс трансляции. На эту вкладку выводятся сообщения об различных этапах трансляции, сообщения о синтаксических ошибках и различные предупреждения (Warnings).

Если трансляция заканчивается нормально (отсутствуют критические ошибки), то сюда же выводятся статистические данные о полученном результирующем коде. Эти данные показывают размеры и процент использования всех видов памяти микроконтроллера. **Например**, после трансляции нашего примера №11 (**листинг 5.21**) программа выдаст следующее сообщение:

```
ATtiny2313 memory use summary [bytes]:
Segment Begin End Code Data Used Size Use%
----------------------------------------------------
[.cseg] 0x000000 0x0004f2 508 758 1266 2048 61.8%
[.dseg] 0x000060 0x00009d   0  61   61  128 47.7%
[.eseg] 0x000008 0x000045   0  61   61  128 47.7%
Assembly complete, 0 errors. 0 warnings
```

Сообщение означает, что в программном сегменте использованы ячейки с адреса 0x000000 по адрес 0x0004f2. При этом собственно код программы занимает 508 байт. Данные в программной памяти занимают 758 байт. Всего использовано в программной памяти 1266 байт (сумма предыдущих двух чисел). Размер программной памяти для этого микроконтроллера составляет 2048 байт. Процент использования программной памяти 61,8%.

Точно такие же сведения приведены для памяти данных (ОЗУ) и для EEPROM. Естественно, что два последних вида памяти не содержат программного кода. Поэтому в соответствующем столбике стоят нули. Последняя строка содержит сообщения об ошибках. В данном случае сообщение переводится так: «Ассемблирование прошло успешно, 0 ошибок, 0 предупреждений».

Следующая вкладка второго окна называется «Message». Здесь выводятся разные системные сообщения о загрузке модулей программы и т. п.

Третья вкладка второго окна называется «Find in Files» (поиск в файлах). В этом окне отражаются результаты выполнения команды «Поиск в Файлах». Эта команда позволяет производить поиск заданной последовательности символов сразу во всех файлах проекта. По окончании поиска во вкладке «Find in Files» отражаются все найденные вхождения с указанием имени файла и строки, где найдена искомая последовательность.

Последняя вкладка называется «Breakpoints and Tracepoints» (точки останова и точки трассировки). Эти точки проставляются в тексте программы перед началом процесса отладки и дублируются в данном окне. Как проставлять точки останова, мы узнаем чуть позже.

Точки останова используются для того, чтобы приостановить выполнение программы в том или ином месте программы для того, чтобы убедиться, что программа выполняется правильно. При создании точки останова в тексте программы она автоматически появляется во вкладке «Breakpoints and Tracepoints».

Вкладка позволяет увидеть все точки останова программы в одном месте. Кроме того, на вкладке против каждой записи, описывающей точку останова, автоматически появляется «Check box» (поле выбора), при помощи которого можно в любой момент временно отключить любую точку останова.

Трассировка *— это особый вид отладочного процесса, когда программа запускается и выполняется в автоматическом режиме.*

Точки трассировки используются для управления процессом трассировки.

Но в процессе работы она оставляет сообщения в специальном окне. Сообщения отражают каждый шаг выполняемой программы. Точки трассировки могут отменить и заново разрешить трассировку на разных участках программы.

Программная среда «AVR Studio» поддерживает трассировку только при работе с отладочной платой ICE50. Это достаточно дорогое устройство. Поэтому в этой книге мы остановимся лишь на программном отладчике без применения каких-либо аппаратных средств отладки.

Поверьте, этого вполне достаточно для разработки микропроцессорных устройств практически любой сложности. Аппаратные отладчики необходимы в условиях промышленного производства для ускорения работ по разработке новых изделий.

Любую из вкладок любого вышеописанного окна можно скрыть или, наоборот, превратить в отдельное свободно перемещаемое окно. Для этого достаточно щелкнуть правой клавишей мыши по заголовку соответствующей вкладки и выбрать в открывшемся меню нужный режим. Пункт «Hide» этого меню означает «Скрытое» (невидимое), «Floating» означает «Свободное» (перемещающееся), «Docking» — «Закрепленное».

Для некоторых пользователей бывает затруднительно вернуть вкладку на место после того, как она превратится в свободно перемещаемое окно. В программе «AVR Studio» используется нестандартный, довольно оригинальный механизм управления окнами. Предположим, что мы случайно превратили в плавающее окно вкладку «Breakpoints and Tracepoints» окна номер два. Посмотрим, как можно поставить ее на место.

Если перемещать это окно при помощи мыши (удерживая его за заголовок), то на основной панели программы появляются специальные указатели размещения, так как это показано на **рис. 4.3**. Они представляют собой стилизованные стрелки синего цвета, расположенные по всему полю главного окна программы. Одновременно появляются восемь таких стрелок. Четыре из них объединены в центральный блок, в который включена еще и круглая кнопка посередине.

Этот блок автоматически располагается в центре того окна, в пределах которого в данный момент перемещается курсор. На **рис. 4.3** этот указатель расположен в центре окна номер два. Оставшиеся четыре стрелки располагаются сверху, снизу, справа и слева основного окна программы. Достаточно переместить курсор мыши вместе с перемещаемым плавающим окном на одну из этих стрелок и отпустить кнопку мыши, и окно тут же встроится в одно из вспомогательных окон программы в виде вкладки либо образует новое вспомогательное окно. Причем вы еще до отпускания кнопки мыши можете увидеть, куда попадет окно.

Как только ваш курсор совместится с одной из стрелок, программа закрасит синим цветом эту область. **Поэкспериментируйте** сами с

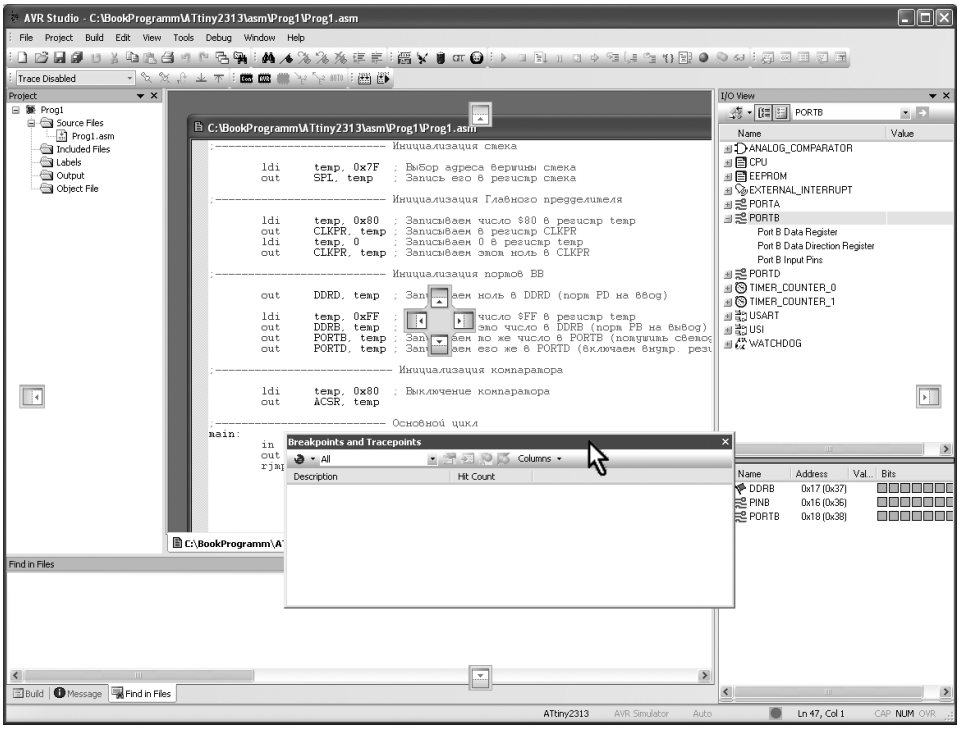


Рис. 4.3. *Перемещение окна*

размещением окон. Помните только, что стрелки превращают ваше плавающее окно в еще одно фиксированное дополнительное окно в разных частях интерфейса. А круглая кнопка посреди центрального блока превращает плавающее окно в дополнительную вкладку уже существующего окна. Именно при помощи этой кнопки оторванное от своего привычного места плавающее окно на **рис. 4.3** можно вернуть на свое место.

Основное и **дополнительные окна** позволяют легко изменять свои размеры. Для изменения размера достаточно перетащить границу окна при помощи мыши. Можно даже скрыть любое из этих окон, закрыв все его вкладки. Закрыть вкладку можно двумя способами. Либо щелкнуть по ее «корешку» правой кнопкой мыши, а в появившемся меню выбрать пункт «Hide». Либо щелкнуть мышью в крестик в верхнем правом углу вкладки. Открыть закрытые вкладки можно при помощи меню «View/Toolbars».

Особую роль играет **область 2 экрана**. Это даже не самостоятельное окно, а оставшаяся часть от главного окна программы. Если закрыть все остальные окна, область 2 займет все пространство программной панели. В этой области появляются различные рабочие окна.

Во-первых, это окна с текстами программ на Ассемблере. А **во-вторых**, здесь могут появляться окна любых открытых программой файлов. Это могут быть текстовые файлы или файлы других программ. Каждый такой файл по умолчанию открывается в виде отдельного плавающего окна. Для определенности будем называть такие окна текстовыми окнами. Текстовые окна будут «плавать» только внутри окна 3.

Для каждого нового текстового окна в нижней части окна 3 появляется «корешок», при помощи которого можно быстро перейти к нужному окну, если оно не находится на переднем плане. Если произвести двойной щелчок левой кнопкой мыши по заголовку любого текстового окна, оно раскроется на всю ширину окна 3. Иногда именно так удобно работать с тестами программ.

В области 2 можно открывать не только все тексты ассемблерных программ текущего проекта, но и тексты программ других проектов, а также тексты программ, написанных на других языках программирования. Такой прием очень удобен, если нужно переделать программу, написанную для старого микроконтроллера на старой версии Ассемблера, на новый лад. Все открытые текстовые окна запоминаются и затем открываются автоматически при открытии проекта.

Любое текстовое окно имеет **подсветку синтаксиса**. Разные части помещенного туда текста программы подсвечиваются разными цветами. Так, все операторы Ассемблера высвечиваются **голубым** цветом. Комментарии выделяются **зеленым**. Остальной текст (параметры команд,

псевдооператоры, метки, переменные и константы) остается **черным**. Это очень удобно. Если написанный вами оператор окрасился в голубой цвет, то это значит, что вы не ошиблись в синтаксисе. Если вы написали комментарий, но перед текстом комментария забыли поставить точку с запятой, то этот комментарий не окрасится в зеленый цвет. Таким образом, многие ошибки видны уже в процессе написания программы.

Кроме двух вспомогательных и одного основного окна, главная панель программы имеет **строку меню** (отмечена цифрой **4** на **рис. 4.1**), а также несколько инструментальных панелей (отмечены цифрой **5**). Как и в любой другой программе под Windows, при помощи меню вызываются все функции программы AVR Studio и переключаются все ее режимы. Панели инструментов дублируют часто используемые функции меню.

Создание проекта

Предположим, что программа AVR Studio установлена на ваш компьютер, запущена и находится в исходном состоянии. Приступим к созданию нового проекта.

Для этого выберем в меню «Project» пункт «New Project». На экране появится окно построителя. В поле «Project Type:» выбираем **тип будущего проекта**. Программа предлагает два варианта:

- проект на Ассемблере (Atmel AVR Assembler);
- проект на языке СИ++ (AVR GCC).

Выбираем **Ассемблер**. Затем в поле «Project name:» выбираем **имя проекта**. Например, Prog1. Сразу под полем с именем проекта расположены два элемента выбора режимов. Так называемые «Чек-боксы» (Check box). По умолчанию оба чек-бокса выбраны (то есть, в соответствующих квадратиках проставлены «галочки»).

Первый чек-бокс (Create initialize file) определяет, нужно ли автоматически создавать главный программный файл. Если у вас уже есть файл с тестом программы на Ассемблере и вы просто хотите создать проект, а затем подключить туда готовый программный файл, снимите соответствующую «галочку». Если вы создаете проект «с нуля», оставьте «галочку» нетронутой.

Второй чек-бокс (Create folder) определяет, нужно ли автоматически создавать отдельный каталог для данного проекта. Если вы заранее уже создали нужный каталог средствами Windows, снимите пометку. Если нет, оставьте.

Следующее поле называется «Initial file». Оно должно содержать имя файла, куда будет записываться текст программы. По умолчанию имя файла уже вписано в это поле. Оно соответствует имени проекта. Советую оставить его без изменений.

Еще одно поле, требующее нашего вмешательства, — это **поле «Location»**. Здесь вы должны указать путь к тому месту на вашем жестком диске, где будет храниться проект. Путь нельзя ввести непосредственно с клавиатуры. Для изменения пути нужно нажать кнопку справа, на которой в качестве названия поставлено многоточие («**...**»).

Откроется диалог «Select folder», при помощи которого вы и должны выбрать директорию. Просто войдите в нужную директорию и нажмите кнопку «Select». При выборе директории нужно учитывать значение чек-бокса «Create folder». Если там стоит «галочка», то при выборе в качестве Location каталога «c:\AVR\myprog», программа поместит ваш проект в каталог «c:\AVR\myprog\Prog1».

На этом можно закончить работу с первым окном построителя. Но прежде, чем нажимать кнопку «Next>>», обратите внимание, что в нижней части окна имеется **еще один чек-бокс**. Он называется **«Show dialog at startup»**. При выборе этого элемента, диалог создания проекта будет автоматически запускаться каждый раз при запуске программы AVR Studio.

Для перехода **к следующему этапу** построения проекта нажмите кнопку «Next>>». Содержимое окна построителя изменится. Появятся два больших поля под общим названием «Select debug platform and device» (Выбор отладочной платформы и микроконтроллера). В списке Отладочных платформ («Debug platform») перечислены все отладочные платы, которые поддерживает данная программа.

Мы не будем использовать внешних плат, поэтому выберем пункт «AVR Simulator» (Программный имитатор AVR). В поле «Device» выбираем нужный тип микросхемы. В нашем случае это ATtiny2313. Теперь все настройки закончены. Для завершения процесса нажмите кнопку «Finish». После нажатия этой кнопки программа создает проект и записывает его в выбранную вами директорию.

Сразу после создания новый проект состоит всего из двух файлов:

- собственно файл проекта `Prog1.aps`;
- файл, куда будет помещен текст программы на Ассемблере `Prog1.asm`.

Файл текста программы автоматически открывается в центре экрана (область 2). Причем он пока абсолютно пустой. Теперь вы можете **приступать к набору этого текста**. Если речь идет о программе Prog1, то просто наберите текст, приведенный в **листинге 5.1**. При наборе текста вы можете пользоваться всеми возможностями, какие обычно предоставляет любой современный текстовый редактор.

Встроенный текстовый редактор программы AVR Studio поддерживает все необходимые сервисные функции:

- выделение текстовых фрагментов;
- вырезание;
- копирование;
- вставку;
- перетаскивание мышью;
- поиск и замену.

Для управления всеми этими возможностями используется стандартный интерфейс, знакомый вам по многим текстовым редакторам, в частности, по популярному редактору Microsoft Word. Набранный тест программы не забудьте записать на диск при помощи команды «Save» меню «File» или при помощи соответствующей кнопки на панели инструментов (). Кнопка  позволяет записать сразу все открытые текстовые файлы.

Для программ, приведенных в этой книге, проекты создавать не обязательно. Достаточно скопировать файл с электронными версиями программ с диска, прилагаемого к книге, или скачать с сайта **http://book.mirmk.net**, распаковать архив и поместить его содержимое в директорию **c:\BookProgramm** на жестком диске вашего компьютера. Если вы копируете с диска, воспользуйтесь специальной функцией копирования программы-оболочки диска. После распаковки у вас появится целый набор директорий, в каждой из которых помещен свой проект. Причем архив содержит не только проекты на Ассемблере, но и на СИ. Любой проект на Ассемблере можно открыть при помощи пункта «Open Project» меню «Project».

Трансляция программы

Рассмотрим форматы файлов. После того, как текст программы набран и записан на жесткий диск, необходимо произвести **трансляцию программы**. В процессе трансляции создается результирующий файл, который представляет собой ту же программу, но в машинных кодах, предназначенную для записи в программную память микроконтроллера. Результирующий файл имеет расширение `hex`.

Кроме hex-файла транслятор создает еще несколько вспомогательных файлов. И главное, файл с расширением `eep`. Этот файл имеет точно такую же внутреннюю структуру, как файл `hex`. А содержит он информацию, предназначенную для записи в EEPROM. Такая информация появляется в том случае, когда в тексте программы переменным, размещенным в сегменте eeprom, присвоены начальные значения. В приведенных выше примерах мы этого не делали. Поэтому файл с расширением `eep` во всех проектах будет пустой (содержать лишь завершающую строку).

Теперь немного разберемся с форматом файлов `hex` и `eep`. В обоих случаях применяется так называемый HEX-формат, который практиче-

ски является стандартом для записи результатов транслирования различных программ. Он поддерживается практически всеми трансляторами с любого языка программирования.

В принципе, программисту не обязательно знать структуру этого формата. Достаточно понимать, что в hex-файле определенным способом закодирована программа в машинных кодах. Именно этот файл используется программатором для «прошивки» программной памяти микроконтроллера. Любой программатор поддерживает hex-формат и распознает записанные туда коды автоматически. Однако для тех, кому это интересно, приведу краткое описание hex-формата.

Формат HEX-файла. Если вы посмотрите содержимое такого файла при помощи редактора «Блокнот», то вы увидите, что это текстовый файл, в котором данные закодированы в виде текстовых строк. Ниже приведено содержимое hex-файла, полученного в результате трансляции программы Prog1 (**листинг 5.1**):

```
:020000020000FC
:100000000FE70DBF00E806BD00E006BD01BB0FEF26
:1000100007BB08BB02BB00E808B900B308BBFDCFB3
:00000001FF
```

Как видите, данный файл состоит из четырех строк. Первая и последняя строки несут служебную информацию. Наличие первой строки необязательно. Система AVR Studio при трансляции программы всегда добавляет в hex-файл первую строку именно такого содержания. Последняя строка — это стандартный конец для любого hex-файла.

Оставшиеся две строки как раз и содержат информацию о кодах программы. В каждой такой строке закодирована цепочка байтов и адрес в памяти, где эти байты должны размещаться.

Строка начинается с двоеточия. Двоеточие — обязательный элемент, который служит для идентификации hex-формата. Все остальные символы в строке — это шестнадцатиричные числа, записанные слитно без пробелов. Отдельные числа отличают по их позиции в строке. Так первые два знака занимает шестнадцатиричное число, означающее длину цепочки.

В нашем случае длина обеих цепочек равна 0x10 (то есть 16) байт. Следующие четыре символа — это начальный адрес, куда эти байты должны быть помещены. **Первая цепочка** будет размещена в памяти, начиная с нулевого адреса. **Вторая цепочка** — с адреса 0x0010. Очередные два знака занимает код вида строки. В интересующих нас строках он равен «00», что означает, что эти строки предназначены для записи данных (в первой строке такой код равен «02», а в последней «01»).

Сразу после кода вида строки начинаются собственно данные. Каждый байт данных занимает два знака. Самые последние два сим-

вола — это контрольная сумма. Она рассчитывается по специальной формуле с использованием значений всех байтов цепочки и служит для проверки на отсутствие ошибок.

Но **вернемся к процедуре трансляции**. Для того, чтобы запустить процесс трансляции текущего проекта, нужно выбрать в меню «Build» пункт, который тоже называется «Build», или нажать кнопку  . Длительность процесса трансляции зависит от размеров программы. Сразу же после начала процесса вкладка «Build» (область экрана 5) выходит на передний план.

В процессе трансляции сюда выводятся служебные сообщения. К таким сообщениям относятся: сообщения о завершении различных этапов трансляции, сообщения об ошибках (Error), а также предупреждения (Warning).

В готовой отлаженной программе ошибок и предупреждений быть не должно. Если программа обнаружит критическую ошибку (Error), то процесс трансляции будет приостановлен, и результирующие файлы созданы не будут. В этом случае необходимо устранить ошибки и повторить трансляцию.

Естественно, транслятор не в состоянии найти все виды ошибок. Он находит только явные ошибки, которые можно найти автоматически. К таким ошибкам относятся:

- ошибки синтаксиса (неправильное написание имени команды);
- неверное количество параметров у оператора;
- попытка использования неописанных переменных и т. п.

Например, сообщение «Unknown instruction or macro» означает, что найдена «Неизвестная инструкция или макрокоманда».

Предупреждения — это тоже ошибки, но некритические. При возникновении некритической ошибки процесс трансляции завершается как обычно. Все результирующие файлы создаются в полном объеме. Однако прежде чем зашивать такую программу в микроконтроллер, тщательно проанализируйте сообщение и постарайтесь определить, как оно повлияет на результаты работы. В любом случае, лучше изменить программу таким образом, чтобы устранить все предупреждения.

Все сообщения во вкладке «Build» появляются по мере их поступления. Для наглядности каждое сообщение помечено цветным кружочком в начале строки:

- сообщения об ошибках помечаются кружочком красного цвета;
- предупреждения помечаются желтым кружочком;
- сообщения об успешном выполнении каждого очередного этапа трансляции помечаются зеленым кружочком.

Если сообщения не вмещаются в окно, то они скрываются в верхней его части. Однако, используя полосу прокрутки, их всегда можно

просмотреть. В случае успешного завершения процесса трансляции в качестве последнего сообщения выводится статистическая информация.

Каждое сообщение об ошибке во вкладке «Build» содержит точное указание места в программе, где произошла эта ошибка. При этом указывается:

- имя файла;
- номер строки;
- фрагмент текста программы, содержащий ошибку;
- ее расшифровка.

Для того, чтобы быстро перейти к фрагменту программы, содержащему эту ошибку, достаточно двойного щелчка по сообщению об ошибке. Окно с текстом программы выйдет на передний план, и в этом окне автоматически отобразится нужный участок текста. На левой границе окна напротив строки, содержащей ошибку, вы увидите синюю стрелочку — указатель ошибки.

Иногда программа неверно определяет место, где возникла ошибка. Это происходит из-за несовершенства анализатора синтаксиса. Дело в том, что очень сложно разработать идеальный алгоритм анализа ошибок. Если в какой-либо строке транслятор показывает ошибку, а вы ошибок не наблюдаете, посмотрите на предыдущие строки. Возможно, ошибка где-то там.

Отладка программы

Если вы исправили все ошибки и добились отсутствия предупреждений, то это значит, что программа успешно оттранслирована. В принципе, вы можете записывать ее в программную память и пробовать ее работу «в железе». Но в большинстве случаев отсутствие синтаксических ошибок еще не означает отсутствие ошибок как таковых. Можно написать команду правильно, да не ту. Но самая **главная неприятность — ошибки алгоритма или его реализации**.

Программист может упустить какой-либо шаг или неправильно поставить условие. Всех возможных ошибок алгоритма не перечесть. Но в результате программа может работать неправильно либо совсем не работать. По этой причине перед тем, как записывать программу в программную память микроконтроллера, необходимо попытаться выявить все эти ошибки.

Вообще, процесс написания программы процентов на 60—70 состоит из поиска и устранения ошибок. И основное количество ошибок выявляется при отладке программы. Все программные примеры, приведенные в этой книге, прежде чем появились на ее страницах, прошли процесс отладки.

И несмотря на простоту этих программ и достаточный опыт в программировании, мне пришлось исправить немало ошибок. По этому поводу существует народная программистская шутка: ***«Если ты написал программу, транслятор не обнаружил в ней ни одной ошибки, посмотри, все ли в порядке с транслятором!»***.

С большим юмором подошли к этому вопросу англичане. По-английски процесс отладки называется Debug (Дебаг). Слово «Bug» — означает блоха. А «Debug» — это процесс избавления от ошибок или процесс ловли блох. Именно этим вам и придется заняться.

Процесс отладки начинается с перевода программы в соответствующий режим. Если проект открыт, а все его программы записаны и оттранслированы, то для перехода в режим отладки выберите пункт «Start Debugging» в меню «Debug» или нажмите кнопку ▶ на панели задач.

Программа начнет **процесс подготовки**. Процесс длительный. Пока идет подготовка, в нижней части основной панели будет двигаться полоса, показывающая процент выполнения операции. По окончании процесса подготовки программа переходит непосредственно в **отладочный режим**. В области 1 появляется новая вкладка «Processor». В этой вкладке вы будете видеть состояние программного счетчика, счетчика цикла, текущего времени выполнения программы, значения в памяти на которые указывают регистровые пары X, Y, Z. Содержимое области 4 немного изменяется. Для каждого элемента в дереве ресурсов появится поле, отображающее его значение. В области 5 на передний план выходит вкладка «Breakpoints and Tracepoints», где теперь будут отображаться все точки останова. В панели инструментов активизируются все инструменты, относящиеся к режиму отладки (до этого они были неактивны). В области 2 на первый план выходит текст главного программного файла. На левой границе окна этого файла появляется желтая стрелка — указатель текущей выполняемой команды. Причем этот указатель помещается в начало программы (напротив первой исполняемой команды). Теперь все готово для отладки.

Отладка может выполняться **разными методами**. Самый простой метод — **пошаговое выполнение**. Для того, чтобы сделать один шаг, выберите в меню «Debug» пункт «Step into» («Шаг в») либо нажмите кнопку на панели инструментов.

Можно также просто нажать кнопку «F11». В результате программа выполнит одну текущую команду. Указатель текущей команды (желтая стрелка) переместится в следующую позицию. Содержимое регистров изменится в соответствии с выполненной операцией. Вы можете это проверить, найдя нужный регистр в окне 3 и посмотрев его значение в окне 4. Убедившись, что команда выполнена правильно, делайте сле-

дующий шаг. И так далее. При этом вы можете проследить последовательность выполнения операций, правильность выполнения условных переходов и многое другое.

В любой момент вы можете **вручную изменить** содержимое любого из элементов в дереве ресурсов. Причем можно изменять как содержимое любого отдельного разряда, так и всего регистра в целом. Для изменения содержимого разряда достаточно щелкнуть при помощи мыши по одному из квадратиков, символизирующему нужный разряд (см. **рис. 4.4**).

При этом состояние квадратика изменится на противоположное (единица изменится на ноль либо наоборот). Для изменения значения всего регистра необходимо произвести двойной щелчок мышью по изображению содержимого регистра (в шестнадцатиричном виде). Откроется окно редактирования. Введите новое значение реги-

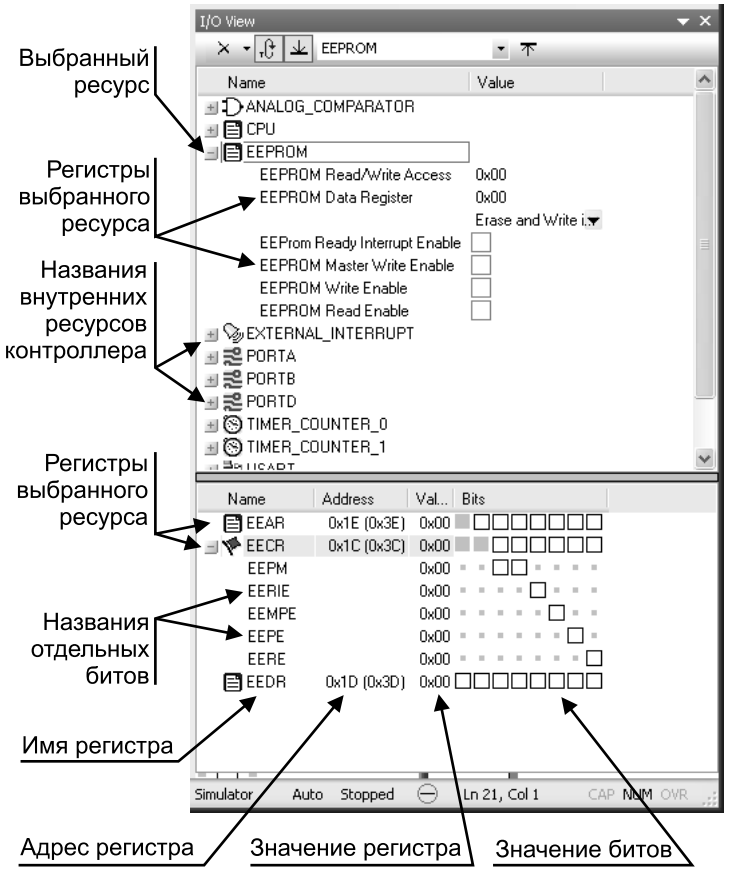


Рис. 4.4. *Изменение содержимого регистров*

стра в десятичном или шестнадцатеричном виде и нажмите Enter. Шестнадцатеричные числа должны начинаться с 0X…

Изменяя содержимое регистра, вы можете моделировать различные ситуации. **Например**, имитировать изменение сигналов на входе порта или принудительно изменять значение счетного регистра таймера, чтобы не ждать, пока он досчитает до нужного значения.

Кроме директивы «Шаг в», имеется еще несколько ее модификаций. Их назначение и способы вызова приведены в **табл. 4.1**.

Директивы пошагового выполнения программы *Таблица 4.1*

Название	Пункт меню «Debug»	Кнопка	Горячая клавиша	Описание
Шаг в	Step into		F11	Выполнить очередную команду
Шаг через	Step over		F10	Выполнить очередную подпрограмму
Шаг из	Step out		Shift+F11	Завершить текущую подпрограмму
Выполнить до	Run to cursor		Ctrl+F10	Выполнять с текущей строки и до строки, где стоит курсор

Директива «Шаг через» используется в том случае, если при пошаговом выполнении программы встретится команда вызова подпрограммы. Если вы не хотите пошагово выполнять всю подпрограмму, вы можете выполнить ее за один шаг. При этом желательно, чтобы подпрограмма не содержала ошибок.

Директива «Шаг из» применяется в том случае, если вы все же вошли в подпрограмму, но затем поняли, что ее пошаговое выполнение излишне. Выбрав данную директиву, можно за один шаг выполнить все оставшиеся команды подпрограммы.

Директива «Выполнить до» применяется в том случае, когда какая-либо часть программы не оформлена в виде подпрограммы, но ее желательно выполнить за один шаг. В этом случае в конце выбранного фрагмента вы можете установить текстовый курсор (мигающую вертикальную полоску) и выбрать директиву «Выполнить до». Отладчик за один шаг выполнит все команды, начиная с текущей (отмеченной желтой стрелкой) и вплоть до текстового курсора. Команда в строке с курсором выполняться не будет. Она станет текущей (на нее теперь будет указывать желтая стрелка).

Пошаговый метод отладки удобен для отладки небольших несложных программ или отдельных участков большой программы. Но представьте себе, что ваша программа содержит цикл, который должен быть выполнен большое количество раз. Для того, чтобы проверить правильность выполнения всего этого цикла в пошаговом режиме, вам пришлось бы

Точка останова *— это специальная метка, которую в отладочном режиме программист может поставить против любой строки программы.*

очень долго щелкать мышкой! В подобных случаях применяются **точки останова (Breakpoint)**.

Затем программа запускается под управлением отладчика. Но это не реальная работа. Это лишь имитация работы микроконтроллера. Программа выполняется строка за строкой, пока в очередной строке не встретится точка останова. Обнаружив такую точку, отладчик приостанавливает выполнение программы.

Выглядит это таким образом, как-будто за один шаг вы выполнили большой кусок программы. Теперь вы можете снова просмотреть и (или) изменить содержимое любого регистра. А затем продолжить отладку. Причем, вы можете продолжить ее как в пошаговом режиме, так и запустить программу в режиме автоматического выполнения до следующей точки останова.

Для управления точками останова программа имеет несколько встроенных директив, которые показаны в **табл. 4.2.**

Директивы управления точками останова *Таблица 4.2*

Название	Пункт меню «Debug»	Кнопка	Горячая клавиша	Описание
Поставить точку останова	Toggle Breakpoint		F9	Поставить (снять) точку останова в строке, где находится курсор
Убрать все точки останова	Remove all Breakpoints		–	Убрать все поставленные ранее точки останова
Создать программную точку останова	New Breakpoints / Program Breakpoint	–	–	Создать точку останова путем задания программного условия
Создать точку останова по данным	New Breakpoints / Data Breakpoint	–	–	Создать точку останова путем задания условия по данным

Для того, **чтобы поставить точку останова** в какой-либо строке программы, нужно сначала поместить в эту строку текстовый курсор. Затем выбрать директиву «Поставить точку останова» (см. **табл. 4.2**). Точка останова выглядит как коричневый кружочек напротив выбранной строки программы на левой границе текстового окна.

Если поместить курсор в строку, где уже есть точка останова, и выполнить еще раз директиву «Поставить точку останова», то точка убирается. Убрать сразу все поставленные точки останова можно при помощи директивы «Убрать все точки останова».

Второй способ простановки точек останова — задание их через меню. Предназначенный для этого пункт «New Breakpoints» меню

«Debug» имеет два подпункта. При помощи подпункта «Program Breakpoint» можно устанавливать программные точки останова. То есть точно такие, какие мы ставили предыдущим способом.

Отличие способа постановки точек через меню в том, что их местоположение в программе вы определяете путем заполнения полей в специальной форме. В этой форме, кроме номера строки или адреса программы, где вы хотите поставить точку останова, вы можете указать количество проходов.

Для этого вам необходимо заполнить поле «Break execution after: — hits» («Остановить выполнение после: — проходов»). Если число в этом поле не равно нулю, то программа остановится в данной точке останова не с первого раза, а лишь тогда, когда пройдет через нее указанное количество раз.

Если вы установили вашу точку останова не через меню, а напрямую в тексте программы, вы все равно можете вызвать описанный выше диалог и изменить в нем количество проходов, щелкнув мышью по строке с описанием нужной точки останова во вкладке «Breakpoints and Tracepoints».

При помощи подпункта «Data Breakpoint» пункта «New Breakpoints» меню «Debug» можно задавать точки останова по данным. При выборе этого пункта меню открывается диалог, в котором вы можете выбрать любую из переменных вашей программы или любой ресурс микроконтроллера (из открывающегося списка) и поставить точку останова по обращению к этой переменной (ресурсу).

Программа позволяет выбрать целый **ряд условий**, при которых наступит останов программы. По умолчанию останов происходит при любом обращении к этой переменной как в режиме чтения, так и в режиме записи. Вы можете выбрать другое условие. **Например**, при равенстве переменной определенному значению. Выбор условия производится при помощи поля «Break when:» («Остановиться если:») и поля «Access type:» («Тип доступа»). Имя переменной выбирается при помощи поля «Location».

Диалог простановки точек останова обоих видов можно вызывать не только через меню. В верхней левой части вкладки «Breakpoints and Tracepoints» для этого имеется специальная кнопка.

После того, как вы проставили все точки останова, вы можете запускать программу в **режиме автоматического выполнения**. Для управления отладчиком в этом режиме программа AVR Studio также имеет несколько специальных директив (см. **табл. 4.3**). Запуск автоматического выполнения программы производится при помощи директивы «Пуск».

Пока программа находится в режиме автоматического выполнения, новое состояние регистров не отображается. Указатель текущей

Директивы управления процессом отладки *Таблица 4.3*

Название	Пункт меню «Debug»	Кнопка	Горячая клавиша	Описание
Запустить	Run		F5	Запуск автоматического выполнения программы с текущей команды
Остановить	Break		Ctrl+F5	Остановка автоматического выполнения программы
Сброс	Reset		Chift+F5	Исходное состояние (сброс микроконтроллера)
Закончить отладку	Stop Debugging		Ctrl+Chift+F5	Закончить отладку

команды также отсутствует. В нижней строке главной панели программы в правой ее стороне находится индикатор состояния. В режиме останова это желтый кружочек с минусом посередине. Слева от него находится слово «Stopped» (Остановлено). В режиме автоматического выполнения программы желтый кружочек превращается в зеленый с плюсом внутри. Вместо слова «Stopped» появляется слово «Running» (Запущено).

Если вы неправильно поставили точку останова либо и вовсе забыли ее поставить, программа будет находиться в режиме автоматического выполнения бесконечно долго. Для досрочной остановки программы используется директива «Остановить». Если в процессе отладки программы понадобится начать все сначала (сымитировать сброс микроконтроллера), это можно сделать при помощи директивы «Сброс». По окончании отладки программы необходимо перейти в режим редактирования. Для этого служит директива «Закончить отладку».

Для оперативного просмотра и изменения содержимого введенных вами переменных в процессе отладки можно открыть специальное окно. Для этого достаточно выбрать пункт «Watch» в меню «View». Окно имеет четыре вкладки. Поэтому можно иметь четыре разных набора переменных.

Для того чтобы включить какую-либо переменную в текущее окно «Watch», необходимо установить курсор мыши на имя этой переменной в тексте программы и нажать правую кнопку мыши. Допустим, вы установили курсор на переменную temp. Тогда в открывшемся меню вы увидите пункт «Add Watch»: «temp». Выберите этот пункт, и переменная будет включена в список «Watch.

Точно так же можно оперативно просматривать содержимое любого вида памяти. Для этого выберите пункт «Memory» в меню «View». Откроется новое окно под названием «Memory». По умолчанию в этом окне в виде дампа будет представлено содержимое программной памяти. При помощи выпадающего списка в левой верхней части этого окна можно выбрать другой вид памяти. Память данных (Data), EEPROM или даже содержимое РОН или портов вода/вывода. В процессе

отладки вы всегда будете видеть в этом окне все изменения выбранной части памяти. Если вы желаете видеть одновременно содержимое сразу нескольких видов памяти, то вы можете открыть второе и даже третье подобное окно. Для этого выберите пункт «Memory2» или «Memory3» в меню «View».

Исправление ошибок

Все программы, приведенные в данной книге, уже отлажены, и изменения в них не требуется. Однако в том случае, если вы захотите доработать программу либо написать новую, вам придется много раз переписывать ее, искать различные фрагменты, заменять их на другие и т. д. Редактор программы AVR Studio дает полный спектр стандартных средств редактирования. Одно из таких средств — это простановка закладок. Поставив закладку в любом месте в тексте программы, вы можете спокойно листать этот текст дальше. В случае необходимости вы можете в любой момент вернуться к закладке. В **табл. 4.4** приведены все директивы работы с закладками.

Для создания новой закладки нужно установить в нужной строке текстовый курсор и выбрать директиву «Поставить закладку». При повторном вызове этой директивы в той же **строке**, закладка убирается. Проставив несколько закладок, можно передвигаться по ним при помощи директив «Перейти к следующей закладке» и «Перейти к предыдущей закладке». При помощи соответствующей директивы можно убрать все закладки.

Директивы работы с закладками *Таблица 4.4*

Название	Пункт меню «Edit»	Кнопка	Горячая клавиша	Описание
Поставить закладку	Toggle Bookmark		Ctrl+F2	Поставить (снять) закладку в строке, где находится курсор
Перейти к следующей закладке	–		F2	Переместить курсор к следующей строке с закладкой
Перейти к предыдущей закладке	–		Chift+F2	Переместить курсор к предыдущей строке с закладкой
Убрать все закладки	Remove Bookmarks		Ctrl+Chift+F2	Удалить все поставленные ранее закладки

Создание проектов на языке СИ

Как уже упоминалось ранее, программа AVR Studio позволяет создавать, транслировать и отлаживать проекты на языке СИ. При этом для трансляции используется программный продукт стороннего производителя под названием **WinAVR**, который в случае установки на ваш компьютер автоматически интегрируется с программной средой AVR Studio.

Программа WinAVR представляет собой набор утилит, предназначенных для разработки программ для микроконтроллеров AVR. Она включает в себя простейшие средства разработки, в том числе компилятор с языков C и C++. Компилятор имеет открытую лицензию, так называемую GNU (General Public License). Открытая лицензия предполагает распространение программ в полностью доступном виде вместе с исходным текстом и разрешает не только любое некоммерческое использование программы, но и доработку текста программы по своему усмотрению. Отрицательным моментом такого способа предоставления лицензий является отсутствие каких-либо гарантий работоспособности программы. Все ошибки исправляйте сами!

Программу WinAVR можно найти на прилагаемом к книге диске или свободно скачать с сайта производителя (**www.atmel.com**). После установки этой программы на ваш компьютер программа AVR Studio приобретает возможность транслировать программы с языка СИ.

При этом процессы создания проекта, трансляции всех его программ, а также процесс их отладки будут выглядеть точно так же, как и в случае программ на Ассемблере. Отличие будет только в самом начале. При создании проекта вы должны выбрать другой тип проекта. Вместо пункта «Atmel AVR Assembler» нужно выбрать «AVR GCC».

Несмотря на очевидные преимущества бесплатных условий распространения данной программы, для начинающих программистов я бы не рекомендовал использовать WinAVR. Именно по этой причине все программные примеры в данной книге выполнены при помощи другой программной среды, которая называется **«Code Vision AVR»**. Подробнее о CodeVisionAVR речь пойдет в следующем разделе.

Система WinAVR и система Code Vision AVR поддерживают разные версии языка СИ. Поэтому, если все же вы решите попробовать применить WinAVR, то учтите, что программные примеры, приведенные в **главе 5** данной книги, не пригодны для системы WinAVR без определенной переработки. И хотя необходимая доработка не носит кардинального характера, без определенных знаний выполнить ее невозможно.

Для тех, кто хочет научиться этому самостоятельно, в файле программных примеров, который я уже рекомендовал вам скачать с моего сайта в Интернете, имеется несколько проектов, представляющих собой уже знакомые нам программы, переделанные под GCC.

Ниже описана другая система программирования, позволяющая создавать программы на языке СИ. Это программная среда Code Vision. В отличии от WinAVR, система Code Vision гораздо удобнее, компактнее и работает более устойчиво.

4.2. Система программирования Code Vision AVR

Общие сведения

Этот программный продукт разработан румынской фирмой «HP Infotech». Инсталляционный пакет свободно распространяемой версии программы (CodeVisionAVR Evaluation), рассчитанной на создание программ, результирующий код которых не превышает 4 Кбайта, можно найти на прилагаемом к книге диске, либо скачать из Интернет. Адрес страницы, для скачивания всех версий программы вы найдете в конце книги (ссылка 4.1). Там же можно скачать полную или облегченную коммерческие версии той же программы, защищенные паролем. Эти версии платные. Условия предоставления права на использование этих программ можно прочитать на той же самой странице. Если возникнут трудности с английским языком на сайте, копию страницы загрузки с текстами на русском языке вы найдете на официальном сайте книги (http://book.mirmk.ru).

Как по назначению, так и по структуре и устройству, программа Code Vision AVR очень напоминает AVR Studio. Главное отличие — отсутствие собственных средств отладки. Для отладки программ Code Vision AVR пользуется отладочными средствами системы AVR Studio. Система Code Vision AVR, так же как AVR Studio, работает не с программами, а с проектами. Позже, в **разделе 5.2** в **пятой главе** нашей книги мы достаточно подробно рассмотрим процесс создания проекта, формирования заготовки будущей программы и превращения этой заготовки в законченную программу. Сейчас же ограничимся общим описанием процесса отладки. Сама системная среда CodeVisionAVR позволяет вводить и редактировать текст программы, проверять программу на наличие синтаксических ошибок, производить трансляцию программы и сохранение результатов трансляции в HEX-файле, а также производить так называемую расширенную трансляцию.

В процессе расширенной трансляции программы формируется не только HEX-файл, но и файл той же программы, переведенный на язык Ассемблер, а также специальный файл в COF-формате, предназначенный для передачи программы в AVR Studio для отладки. После создания и расширенной трансляции проекта его директория будет содержать файлы со следующими расширениями:

- **prj** — файл проекта Code Vision AVR;
- **txt** — файл комментариев. Это простой текстовый файл, который вы заполняете по своему усмотрению;
- **c** — текст программы на языке СИ;
- **asm** — текст программы на Ассемблере (сформирован Code Vision);
- **cof** — формат для передачи программы в другие системы для отладки;

eep — содержимое EEPROM (формируется одновременно с HEX-файлом);
hex — результат трансляции программы;
inc — файл-дополнение к программе на Ассемблере с описанием всех зарезервированных ячеек и определением констант;
lst — листинг трансляции программы на Ассемблере;
map — распределение памяти микроконтроллера для всех переменных программы на СИ;
obj — объектный файл (промежуточный файл, используемый при трансляции);
rom — описание содержимого программной памяти (та же информация, что и в HEX-файле, но в виде таблицы);
vec — еще одно дополнение к программе на Ассемблере, содержащее команды переопределения векторов прерываний;
cwp — файл построителя проекта. Содержит все параметры, которые вы ввели в построитель (см. **раздел 5.2**).

Все перечисленные выше файлы имеют одинаковые имена, соответствующее имени проекта. Кроме перечисленных выше файлов, директория проекта может содержать несколько файлов с расширением типа `c~`, `pr~` или `cw~`. Это страховочные копии соответственно файлов `c`, `prj` и `cwp`. То же самое, что файл `bak` для текстовых файлов.

Интерфейс системы Code Vision AVR

Интерфейс программы Code Vision AVR показан на **рис. 4.5**. Центральное место занимает окно программ. В этом окне может быть открыто сразу несколько программных модулей. Для каждого модуля появляется отдельная вкладка. Кроме того, там же появляется вкладка «Notes» — окно комментариев к текущему проекту. Все вкладки окна программ, обладают свойствами текстового редактора. Так же, как и в AVR Studio, здесь поддерживаются функции выделения фрагментов, их перетаскивания, копирования, вставки, поиска, поиска и замены и т. д. Так как система Code Vision AVR не поддерживает функцию отладки, в окне отсутствуют все команды, связанные с этим режимом. Отсутствует в ней также и механизм закладок.

Остальные окна имеют вспомогательное значение. Для работы с программой их наличие не обязательно. Каждое из них может быть закрыто или переведено во всплывающий режим. Для этого в заголовке каждого окна есть соответствующие инструменты. Рассмотрим назначение вспомогательных окон.

Окно «Code Navigator».

В этом окне показывается структура текущего открытого проекта. Структура включает в себя список всех файлов, из которых состоит

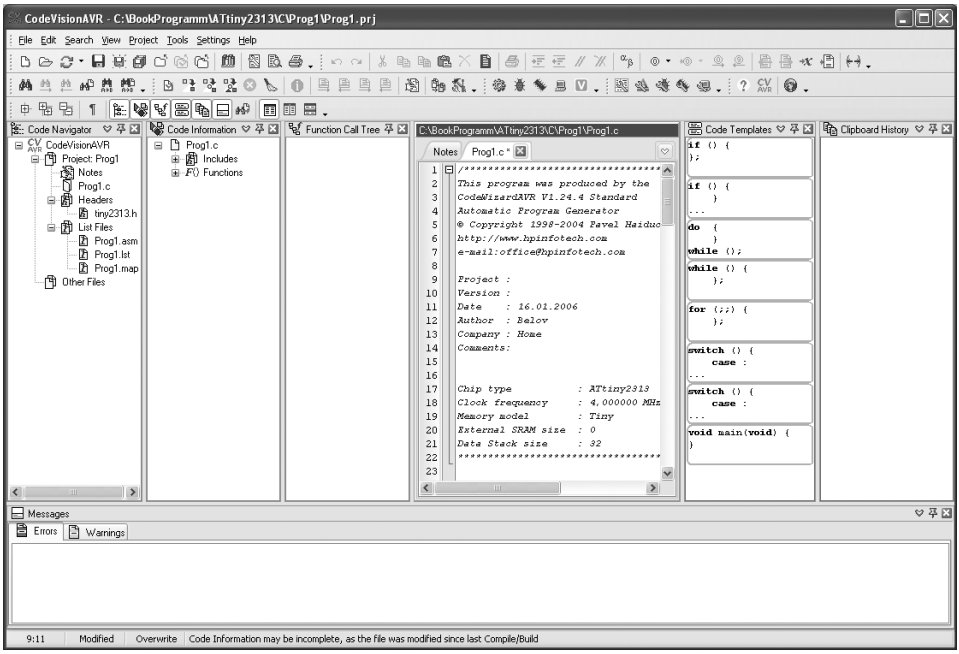


Рис. 4.5. *Основное окно программы Code Vision*

проект, а также список найденных ошибок и предупреждений, который появляется здесь после трансляции программы. В данном случае под файлами проекта понимаются не все те файлы, которые были перечислены в предыдущем разделе, а только исходные файлы (тексты программ на языке СИ плюс файл описания).

Окно «Code Information».

Содержит информацию о всех функциях программы и присоединенных библиотеках. Информация представлена в виде дерева. Присоединенные библиотеки вы найдете раскрыв ветвь «Includes», а дерево используемых функций, раскрыв «Functions». Содержимое всех этих ветвей появляется автоматически (формируется путем анализа текста программы). Такая структура очень удобна для навигации. Щелчок мышью по имени любой из функций в дереве проекта приведет к тому, что окно с текстом программы, содержащей эту функцию, переместится на передний план, и текстовый курсор установится в начало выбранной функции.

Окно «Function Call Tree».

Показывает последовательность вложенных вызовов функций. Но только тех, которые интенсивно используют стек. В наших примерах оно не задействовано и смело может быть закрыто.

Окно «Code Templates» (Шаблоны Кода).

Эта окно — помощь программисту. Она содержит шаблоны нескольких основных конструкций языка СИ. В частности, шаблоны операторов for, while, if и так далее. В любой момент программист может «перетащить» нужную структуру при помощи мыши в окно с основным текстом программы. При этом перетянется его копия, а оригинал останется в окне «Code Templates». Затем вам нужно лишь заполнить полученный таким образом шаблон командами, и фрагмент программы готов. При желании вы можете пополнить набор шаблонов. Для того чтобы ввести новый элемент в список шаблонов, сначала наберите его текст в текстовом окне программы, выделите его и «перетяните» при помощи мыши в окно шаблонов. Записанный таким образом шаблон останется в окне шаблонов даже после загрузки нового проекта. С этого момента вы всегда можете использовать его в любой другой программе. Лишние шаблоны можно удалить.

Для этого достаточно щелкнуть по ненужному шаблону правой кнопкой мыши и в появившемся меню выбрать пункт «Delete».

Окно «Clipboard History» (История значений буфера обмена).

Структура этого окна такая же, как структура окна «Code Templates». Но вместо шаблонов в окно в процессе редактирования помещается все, что попало в буфер обмена. Если вам снова понадобилось какое-либо из этих значений, вы в любой момент можете «перетянуть» его при помощи мышки в вашу программу. Если оно не нужно, можете его закрыть (как, впрочем, и предыдущее).

Окно «Messages» (Сообщения).

Сюда выводятся все сообщения об ошибках и предупреждения в процессе трансляции. Так же, как и в AVR Studio, вы можете быстро перейти к тому месту программы, где найдена ошибка. Для этого достаточно выполнить двойной щелчок мышкой по соответствующему сообщению об ошибке в окне.

Создание проекта без использования мастера

Программа Code Vision AVR так же, как и любая современная программа, работающая в среде Windows, имеет **строку меню** и **панель инструментов**. При помощи меню можно управлять всеми функциями системы. Кнопки панели инструментов дублируют самые важные команды меню.

Так как процесс создания проекта при помощи мастера подробно описан в **разделе 5.2**, рассмотрим случай создания проекта без использования мастера. Для того, чтобы **начать процесс создания проекта,** выбе-

рите пункт «New» меню «File» или нажмите кнопку на панели инструментов. На экране появится окно «Create new file» (**рис. 4.6**). В окне предлагается выбрать вид создаваемого файла: Исходный текст программы («Source») или файл проекта («Project»).

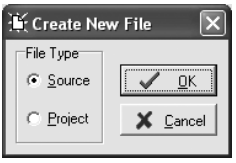


Рис. 4.6. *Создание проекта*

Сначала выберите пункт «Source» и нажмите кнопку «Ок». На экране появится текстовое окно. Внесите в это окно текст программы. Можно сначала внести лишь сокращенный вариант, если полного еще не существует. Теперь нужно записать набранный текст на жесткий диск. Для этого выберите пункт «File Save» меню «File» или нажмите кнопку . Откроется диалог записи файла. Вы должны набрать имя файла, выбрать директорию для проекта и нажать кнопку «Сохранить».

Следующий этап — создание самого проекта. Снова выберите «New» меню «File» или нажмите кнопку . Теперь вместо «Source» выберите «Project» (см. **рис. 4.6**). На вопрос, будете ли вы создавать новый проект при помощи построителя, ответьте «No». Откроется диалог записи файла. Вы должны выбрать имя для файла проекта и записать его в ту же директорию, что и файл с текстом программы. После нажатия кнопки «Сохранить» появляется окно проекта, в котором пока нет файлов с программой (**рис. 4.7**).

Для того, чтобы добавить созданный нами файл к нашему проекту, нажмите кнопку «Add». Откроется диалог, в котором вы должны найти и выбрать созданный ранее файл с программой. После нажатия кнопки «Открыть» файл присоединится к проекту. Теперь нажмите кнопку «Ок», и окно проекта закроется. С вновь созданным проектом вы можете работать так же, как и с проектами, созданными при помощи построителя.

Перед тем, как приступать к трансляции проекта, необходимо **настроить параметры компилятора**. Снова открываем окно «Configure project». Для этого нажимаем кнопку . В окне проекта (см. **рис. 4.7**) открываем вкладку «C Compiler». На этой вкладке имеется множество параметров, определяющих стратегию компиляции. Установим только главные из них.

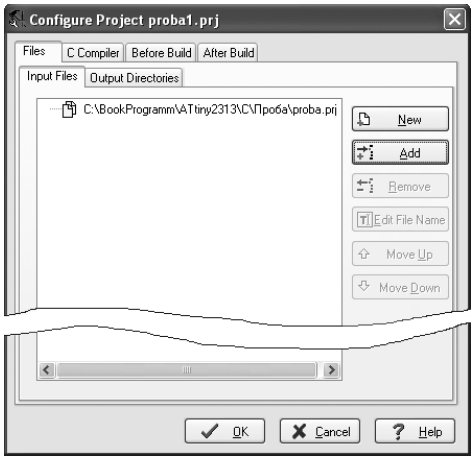


Рис. 4.7. *Окно конфигурации проекта*

Сначала нужно выбрать тип микросхемы. Для этого служит выпадающее меню с заголовком

«Chip:». Затем нужно выбрать частоту тактового генератора. Данные о частоте будут использованы транслятором при формировании процедур задержки. Выбор частоты производится при помощи другого выпадающего меню, озаглавленного «Clock:». На этом можно было бы и остановиться. Для остальных параметров можно оставить значения по умолчанию. Однако, при желании, вы можете **выбрать способ оптимизации**. Для выбора способа оптимизации служит поле «Optimize for:» («Оптимизировать по:»). Предлагаются два способа:

- оптимизация по размеру («Size»);
- оптимизация по скорости («Speed»).

Оптимизация по размеру заставляет компилятор создавать результирующий код программы, минимальный по длине. Оптимизация по скорости позволяет создать более длинную, но зато более быстродействующую программу.

После того, как все параметры установлены, запишите все эти изменения, нажав кнопку «Ok» в нижней части окна «Configure project». Окно проекта закроется. Теперь можно **приступать к компиляции**. Директивы режима компиляции приведены в **табл. 4.5.** Если программа достаточно большая, то перед компиляцией можно проверить ее на ошибки. Для этого служит директива «Проверка синтаксиса». При выборе директивы «Компиляция» проверка синтаксиса производится автоматически.

В процессе компиляции создается объектный файл в HEX-формате, а также файл, содержащий данные для EEPROM (файл с расширением `eep`). Директива «Построить проект» запускает процедуру полного построения проекта. Полное построение включает в себя проверку синтаксиса, компиляцию, создание файла программы на Ассемблере и файла в формате COF (для передачи в AVR Studio для отладки).

Директивы работы с программой Code Vision AVR *Таблица 4.5*

Название	Пункт меню «Project»	Кнопка	Горячая клавиша	Описание
Проверка синтаксиса	Check Sintax		–	Проверить синтаксис программы в текущем окне
Компиляция	Compile		F9	Скомпилировать программу
Построить	Make		Shift+F9	Построить проект
Перестроить все	Make All Files		Ctrl+F9	Перестроить все файлы проекта
Конфигурация	Configure		–	Открыть окно конфигурации проекта

Учтите, что файл COF создается только в том случае, если выставлен соответствующий параметр в окне проекта (окно «Configure project», вкладка «C Compiler», параметр «File output format(s):»). По умолчанию значение этого параметра равно «COF ROM HEX EEP». То есть то, что нам нужно. Однако не мешает все же лишний раз в этом убедиться.

Кроме основных перечисленных выше директив, программа «Code Vision AVR» имеет несколько дополнительных. Некоторые из них перечислены в **табл. 4.6**. **Директива «Запуск мастера»** позволяет в любой момент в процессе редактирования программы запускать мастер-построитель программ. Это может понадобиться для редактирования созданной ранее заготовки программы. Подробно процесс редактирования описан в **разделе 5.2**.

Дополнительные директивы Code Vision AVR *Таблица 4.6*

Название	Пункт меню «Tools»	Кнопка	Горячая клавиша	Описание
Запуск мастера	CodeWizardAVR		Shift+F2	Запуск мастера-построителя программ
Отладчик	Debugger		Shift+F3	Запуск внешней программы-отладчика
Программатор	Chip Programmer		Chift+F4	Запуск программатора микросхем

Директива «Отладчик» предназначена для вызова внешней программы-отладчика. При первой попытке вызова этой директивы программа запрашивает путь к исполняемому файлу отладчика. Если указать путь к программе AVR Studio, то в дальнейшем директива «Отладчик» может использоваться для вызова этой программы.

Особое значение имеет **директива «Программатор»**. Дело в том, что программа Code Vision AVR имеет в своем составе систему управления программатором. Поддерживается несколько видов программаторов. Благодаря этому программа Code Vision AVR является полнофункциональной. То есть позволяет проводить полный цикл работ по разработке программ для микроконтроллеров.

Действительно, при помощи этой системы можно написать программу, оттранслировать ее и прошить в программную память микроконтроллера. Отсутствует лишь возможность отладки. Но она реализуется при помощи AVR Studio. Перед тем, как пользоваться программатором, необходимо настроить его параметры. Окно настройки параметров программатора открывается при выборе пункта «Programmer» меню «Settings».

Прежде всего, нужно знать название платы программатора, которая подключена к вашему компьютеру. Кроме вида программатора, необхо-

димо выбрать имя порта, к которому он подключен, а также некоторые специальные параметры. Подробнее о программаторах мы поговорим в конце этой главы.

Отладка программы

После того, как программа полностью оттранслирована, можно производить ее отладку. Для этого, не закрывая CodeVisionAVR, необходимо открыть программу AVR Studio. Затем в AVR Studio необходимо выбрать пункт «Open File» в меню «File». Появится диалог открытия файла. В этом диалоге нужно изменить тип открываемого файла. В поле «Тип файла» нужно выбрать «Object Files (*.hex;*.d90;*.a90;*.r90;*.obj;*.cof;*.dbg;)». Затем нужно найти на вашем диске директорию с проектом на CodeVisionAVR.

Когда вы откроете эту директорию, то вы увидите три файла с одинаковым именем. Чтобы узнать, какой из этих файлов имеет расширение «cof», нужно подвести к каждому файлу курсор мыши и посмотреть его описание во всплывающем желтом окне подсказки. Обычно это самый первый (верхний) из файлов. Выберите его и нажмите кнопку «Открыть». Сразу после этого откроется другой диалог. На сей раз диалог записи файла.

Программа предложит записать на диск файл проекта в формате AVR Studio. Имя этого файла уже будет дано по умолчанию. Просто нажмите кнопку «Сохранить», и файл проекта сохранится в той же директории, что и проект на СИ. После этого откроется знакомое нам окно выбора микросхемы и отладочной платформы. Выберите платформу «AVR Simulator», а в качестве микросхемы — ту, которую вы используете в своей программе. Нажмите кнопку «Finish».

После нажатия этой кнопки начнется процесс подготовки к режиму отладки. Как только он закончится, программа AVR Studio перейдет в отладочный режим. При этом в окне программ вы увидите текст программы на языке СИ, а содержимое остальных окон будет точно такое же, как и при отладке программ на Ассемблере.

В процессе отладки вы можете пользоваться всеми удобствами и преимуществами программы AVR Studio:

- ставить точки останова любого типа;
- просматривать и изменять содержимое всех регистров;
- запускать программу в пошаговом режиме и в режиме выполнения под управлением отладчика.

Если в процессе отладки обнаружится ошибка, исправлять ее нужно следующим образом:

- не закрывая AVR Studio, перейдите в окно CodeVisionAVR и измените текст программы;

- запишите изменения и перетранслируйте программу;
- вернитесь в AVR Studio, где вы увидите сообщение о том, что программа изменилась, и предложение учесть изменения;
- ответьте «Yes» и продолжайте отладку;
- по окончании отладки закройте программу AVR Studio.

4.3. Программаторы

Общие сведения

Итак, мы узнали как происходит отладка и компиляция программ для микроконтроллеров. Теперь нам остается записать оттранслированную программу в программную память микроконтроллера и опробовать ее работу на практике. Для записи программного кода в память микроконтроллера используются специальные устройства — **программаторы**.

Программатор подключается к компьютеру и управляется специальной программой. Микросхема микроконтроллера, в свою очередь, подключается к программатору. Любой микроконтроллер имеет специальный режим — **режим программирования**. В этом режиме все или несколько выводов микросхемы меняют свои функции. В новом режиме они принимают данные и сигналы управления от программатора. Включение режима программирования производится при помощи входа Reset.

Как вы знаете, в рабочем состоянии на этом входе должна присутствовать логическая единица. Подача нулевого потенциала на этот вход приведет к сбросу микроконтроллера. Если точнее, для сброса нужно подать низкий логический уровень на вход Reset на короткое время, а затем опять перевести этот вход в единичное состояние. Если же подать и удерживать ноль, то микроконтроллер перейдет в режим программирования.

Подробнее о режимах программирования микросхем серии AVR вы можете узнать из специальной литературы. Но эти подробности могут пригодиться только в том случае, если вы желаете разработать свой собственный программатор. Однако нам вовсе не обязательно изобретать велосипед. В настоящее время разработано и успешно используется огромное количество различных программаторов. Достаточно выбрать из них подходящий и научиться использовать его для своей работы.

Как вы уже знаете, микросхемы AVR поддерживают несколько способов программирования. Основные из них:

- параллельное программирование;
- программирование по последовательному ISP-каналу.

Причем некоторые модели поддерживают лишь один из этих способов, но большинство поддерживают оба **[4]**. **При параллельном программировании** микросхему микроконтроллера обычно вынимают из панельки платы, где она должна работать, и вставляют в панельку программатора. После программирования ее необходимо извлечь из программатора и вставить в рабочую схему.

Последовательное программирование не требует обязательного извлечения микросхемы из отлаживаемой схемы. Канал ISP, используемый в этом случае, разработан таким образом, что позволяет производить так называемое **внутрисхемное программирование**, то есть прямо в схеме, не выключая питания.

При параллельном программировании данные передаются по байтам. Для передачи байта используется восемь ножек микросхемы, которые играют роль шины данных. При последовательном способе программирования для передачи данных используется всего три вывода микросхемы. Эти выводы имеют следующие названия MISO, MOSI, SCK. Разумеется, все эти выводы совмещены с выводами одного из портов. Расшифровка названий выводов следующая:

- **MISO** — Master Input, Slave Output (Ведущее работает на ввод, ведомое — на вывод);
- **MOSI** — Master Output, Slave Input (Ведущее работает на вывод, ведомое — на ввод);
- **SCK** — Synchronize Clock (Сигнал синхронизации).

При последовательном программировании байты передаются побитно. Очевидно, что при параллельном способе программирования микросхема будет запрограммирована быстрее, чем при последовательном способе. Однако параллельный способ не позволяет выполнять внутрисхемное программирование.

Параллельный способ программирования имеет **две модификации:**

- параллельное программирование с повышенным питанием;
- низковольтное параллельное программирование.

Повышенное питание (+12 В) подается на вывод Reset непосредственно в момент программирования. Низковольтное программирование не требует никаких дополнительных напряжений. Питание всех цепей осуществляется от напряжения +5 В.

Последовательное программирование бывает только низковольтным. Высоковольтный режим программирования обеспечивает самую высокую скорость «прошивки» микросхем.

Универсальные и специализированные программаторы

Как уже говорилось, в настоящее время разработано огромное множество различных схем программаторов. Их описание можно встре-

тить в различной литературе, а также скачать из Интернета. Все схемы можно классифицировать по ряду параметров.

Так, схемы бывают универсальные и специализированные. **Универсальные схемы** позволяют программировать не один вид микросхем, а несколько видов. Причем не только в пределах одной серии, но и даже микросхемы разных серий и разных производителей. Кроме того, универсальные программаторы обычно умеют программировать также и микросхемы других типов. **Например**, ПЗУ, EEPROM и т. д.

Специализированные программаторы предназначены для программирования микросхем одной серии или даже микросхем всего одного конкретного типа.

Если говорить о микросхемах серии AVR, то программаторы могут различаться по способу программирования. Существуют программаторы, которые поддерживают оба способа (параллельный и последовательный). Упрощенные программаторы умеют программировать только последовательным способом.

Способ подключения программатора к компьютеру

Следующий параметр, по которому можно классифицировать все без исключения программаторы, — способ подключения к компьютеру. Для подключения различных устройств к любому компьютеру обычно используется один (или несколько) стандартных интерфейсов. Еще недавно такими интерфейсами были: либо параллельный порт компьютера (так называемый LPT-порт), либо последовательный порт (COM-порт). В прошлом издании книги приводилась схема программатора, подключаемого к LPT порту. Это одна из реализаций популярного в Интернете программатора PonyProg. Преимущество такой схемы в ее простоте. Однако прогресс компьютерной техники привел к тому, что сначала вместо LPT, а позже и вместо COM портов все чаще стали использовать универсальный последовательный порт USB. В настоящее время в современных компьютерах практически уже никогда не применяется LPT порт и почти не применяется COM. Такая ситуация создает определенные трудности начинающим разработчикам микроконтроллерных устройств. Дело в том, что протокол работы USB очень сложный. Он не позволяет создать программатор, схема которого настолько же проста, как в случае LPT или COM порта. Типичная схема программатора для USB порта сама содержит микроконтроллер, который реализует USB протокол. Современная промышленность выпускает большой спектр переходников USB-COM. Такие переходники предназначены для подключения устройств, работающих с COM портом к USB интерфейсу компьютера. К сожалению, простейшие программаторы подобные программатору PonyProg практически никогда не работают с такими пере-

ходниками. Причина этого в том, что переходники эмулируют COM порт не в полном объеме. А программатор PonyProg использует COM интерфейс очень нестандартным образом. Как раз эти нестандартные свойства COM порта переходник либо не поддерживает, либо поддерживает, но программатор при этом работает недопустимо медленно.

Предлагаем читателям три простые схемы программаторов. Две схемы рассчитаны на старый тип интерфейса (один для LPT, второй для COM порта). Третий программатор рассчитан на работу с USB портом. Как уже говорилось выше, первые две схемы можно использовать только с устаревшими компьютерами, имеющими встроенные LPT или COM порт. Последняя схема может быть использована с любым современным компьютером. В том числе и с ноутбуком. Сейчас любой компьютер имеет USB интерфейс. Однако учтите, что приведенная нами схема программатора использует упрощенный USB интерфейс? Реадизующий функции USB интерфейса не в полном объеме. Тем не менее предложенная схема хорошо себя зарекомендовала в радиолюбительской практике.

Схема программатора для LPT порта

Схема программатора для параллельного LPT порта показана на **рис. 4.8**. Эта схема — один из вариантов программатора PonyProg и совместима с программатором системы STK200/300.

Предлагаемый программатор предназначен только для работы в последовательном режиме и предусматривает внутрисхемное програм-

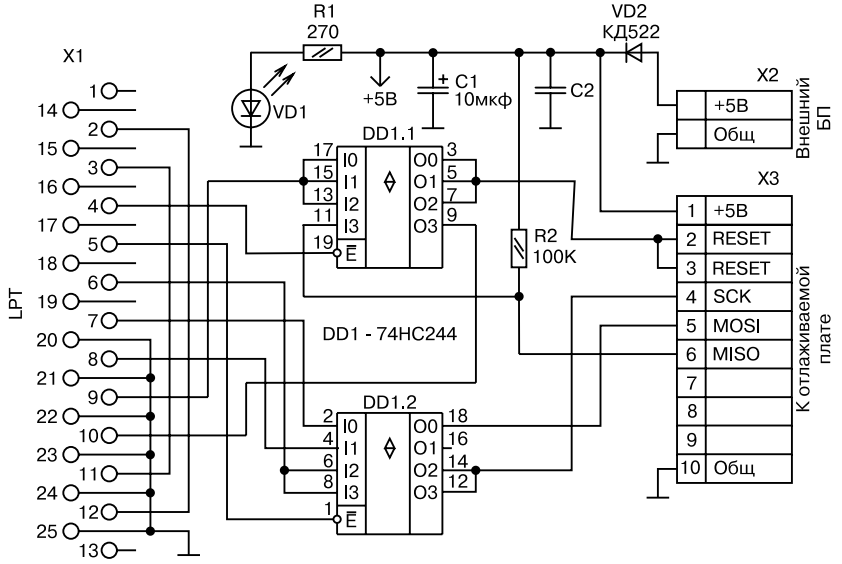


Рис. 4.8. *Схема программатора для LPT порта*

мирование. От простейшей схемы на трех резисторах данная схема отличается наличием защитного буфера на микросхеме 74HC244. Эта микросхема представляет собой два четырехканальных управляемых буфера.

Управление каждым из буферов производится при помощи входа $\overline{E}$. Сигнал логического нуля на этом входе открывает соответствующий буфер, и сигналы со входов буфера поступают на его выходы. Сигнал со входа I0 поступает на выход O0, сигнал со входа I1 — на выход O1 и так далее. Если на вход $\overline{E}$ подать логическую единицу, буфер закрывается, и все его выходы переходят в высокоимпендансное состояние.

Программатор подключается к LPT-порту компьютера при помощи разъема X1. Именно через этот разъем на схему поступают управляющие сигналы от компьютера. Через разъем X2 на программатор может подаваться внешнее питание. К отлаживаемой схеме программатор подключается при помощи разъема X3.

В процессе программирования программа управления программатором включает буферы DD1.1 и DD1.2, а затем организует обмен информацией с программируемой микросхемой посредством следующих сигналов:

- **MSIO, MOSI, SCK** (в соответствии с алгоритмом передачи данных);
- **RESET** (перевод микросхемы в режим программирования и сброс).

По окончании процесса программирования оба буфера закрываются. После этого программатор не мешает работе схемы, к которой он подключен.

Схема программатора для COM порта

Схема программатора для подключения через COM порт показана на **рис. 4.9**. Это вторая схема программатора PonyProg. Причем, вариант с COM портом разработчики программатора считают основным. Приведенная в книге схема немного упрощена и приближена к схеме для LPT интерфейса. Исходный вариант схемы отличается лишь тем, что в ней реализовано питание программируемой микросхемы непосредственно от COM порта. Такой способ получения напряжения питания плох тем, что способен повредить COM интерфейс вашего компьютера.

Как вы видите, схема очень простая. В схеме не применяются никакие буферные микросхемы. Защитой порта служат токоограничивающие резисторы. Стабилитроны VD3—VD5 служат для ограничения сигналов до TTL уровней на которых работает микроконтролер.

Именно эту схему программатора в последнее время радиолюбители пытаются приспособить для работы через USB интерфейс, подключая ее через различные модули USB-COM. В Интернете можно найти массу

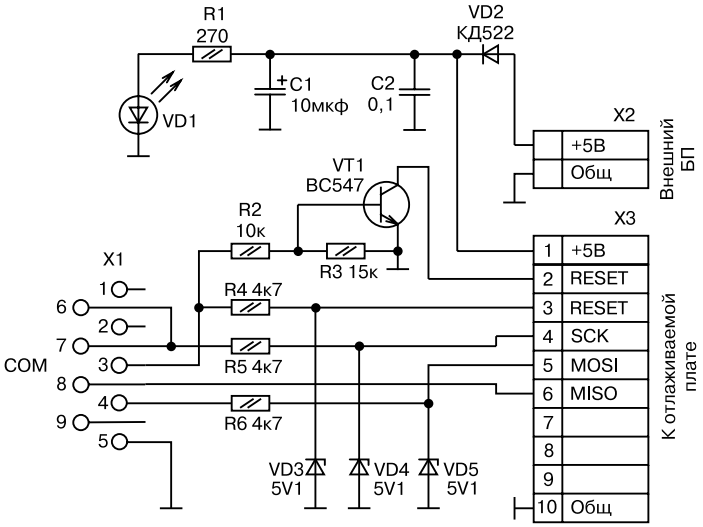


Рис. 4.9. *Схема программатора для COM порта*

статей и широкий обмен опытом. Обзор наиболее удачных вариантов из Интернета вы найдете на официальном сайте книги (ссылка 1 в конце книги). Здесь же мы приводим одну из таких схем, по общему мнению самую удачную.

Схема программатора для USB порта

Предлагаемая схема программатора разработана Томасом Фишелом (Thomas Fichl) из Германии. Программатор называется USBasp. Разработка опубликована на его сайте (www.fischl.de). Достоинство схемы — ее простота. Многие отечественные самодеятельные конструкторы уже опробовали эту схему. Отзывы положительные. Схема программатора приведена на **рис. 4.10**.

Основа схемы — микроконтроллер ATmega8 (DD1). Контроллер управляет всей работой программатора. Для связи с компьютером в схеме реализован упрощенный USB интерфейс. USB порт компьютера подключается к схеме программатора посредством разъема X1. Программа программатора, установленная на компьютере, управляет работой микроконтроллера, посылая по каналу USB команды и данные, необходимые для программирования. Программируемый микроконтроллер подключается к разъему X3. Распайка контактов и назначение сигналов разъема аналогично соответствующим сигналам аналогичного разъема в предыдущих двух схемах программатора. Управляющая программа, прошитая в программной памяти микроконтроллера DD1, реализует протокол USB интерфейса и протокол программирования на выходах MOSI, MISO, SCK.

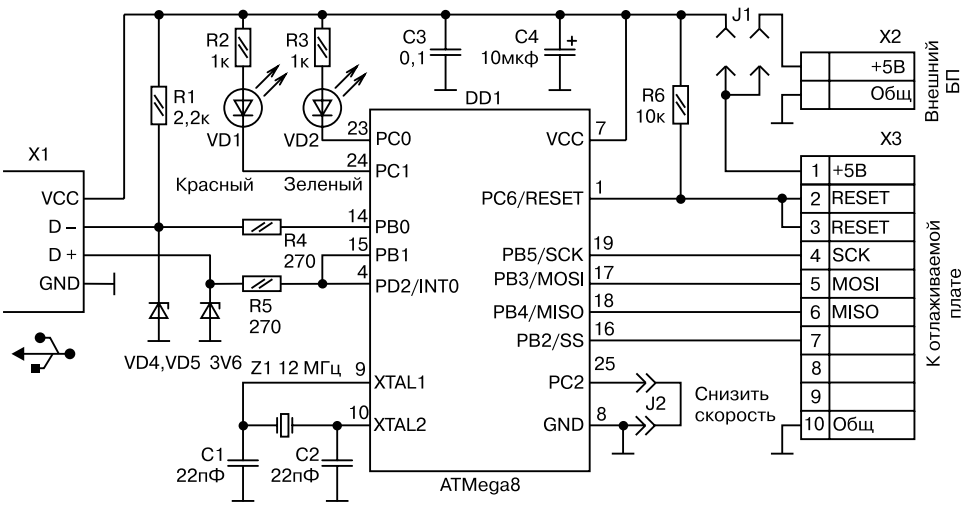


Рис. 4.10. *Схема программатора с USB интерфейсом*

На элементах R1, R4, R5, VD4, VD5 собрана схема согласования USB интерфейса с уровнями сигналов на выводах порта микроконтроллера DD1. Светодиоды VD1 и VD2 служат индикаторами режимов работы программатора. При помощи перемычки J1 можно выбирать источник питания программируемой микросхемы. Перемычка J2 служит для перевода программатора в режим пониженной тактовой частоты. Если перемычка отсутствует, программатор работает на повышенной тактовой частоте. При установке перемычки тактовая частота снижается. Этот режим полезен для программирования микросхем с низким быстродействием. Следует отметить, что время прошивки большинства микросхем достаточно небольшое, поэтому увеличение этого времени в режиме низкой частоты не вызывает никаких неудобств. Зато в режиме пониженной скорости можно прошивать как быстродействующие микроконтроллеры, так и низкоскоростные.

Прошивку для микроконтроллера DD1, входящего в схему программатора, вы можете найти на диске, прилагаемом к книге или на сайте разработчика. Для правильной работы микроконтроллера необходимо правильно установить значения его FUSE переключателей. Фьюзы должны быть запрограммированы следующим образом: WDTON, CKOPT, BOOTSZ1 и BOOTSZ0 должны быть установлены в 0. Остальные Фьюзы должны оставаться равными единице. Напоминаю, если для программирования Фьюзов использовать программатор PonyProg, в окне установки значений FUSE переключателей «галочка» напротив любого Фьюза соответствует его нулевому значению, а отсутствие галочки соответствует единичному значению.

Для работы программатора на компьютер нужно установить драйвер. Инсталляционный пакет драйвера USBasp программатора вы также найдете на прилагаемом диске. Можете также скачать его на сайте разработчика. Собранный программатор с прошитой в его память программой нужно просто подключить к USB порту компьютера. Компьютер обнаружит новое устройство и предложит найти для него драйвер. Выберите режим ручного поиска драйвера и укажите компьютеру путь к файлам драйвера на диске. После окончания процесса установки компьютер сообщит о том, что «устройство установлено и готово к работе». При этом в списке устройств компьютера появиться новое USB устройство. После этого можно устанавливать и запускать программу программатора. Об этом будет подробно рассказано ниже.

Внутрисхемное программирование

Для того, чтобы обеспечить возможность **внутрисхемного программирования,** необходимо при разработке схемы на микроконтроллере соблюдать следующее правило.

Правило

На все входы, используемые для последовательного программирования (MSIO, MOSI, SCK, RESET), не должны поступать никакие мешающие сигналы. Проще всего оставить эти входы свободными.

Если это невозможно, то старайтесь, чтобы к этим выводам были подключены только входы внешних микросхем, а не их выходы. Программные примеры в **главе 5** этой книги выполнены с учетом всех этих требований. Так, **в примерах с 1 по 9** выходы, предназначенные для последовательного программирования, оставлены свободными. В **примерах 10 и 11** с двумя из этих выводов пришлось совместить кнопку звонка и переключатель режимов работы. Поэтому для двух последних схем перед тем, как начать программирование, необходимо убедиться, что кнопка звонка отпущена, а переключатель режимов находится в положении «Работа» (контакты разомкнуты).

Питание программатора

Питание программатора для первых двух вариантов схемы происходит одинаково. Оно может осуществляться как от внешнего источника стабилизированного напряжения +5 В через разъем X2, так и от отлаживаемой схемы через контакт 7 разъема X3. В том случае, если отлаживаемая схема потребляет не слишком большое количество энергии, возможен вариант питания отлаживаемой схемы от внешнего источника питания через разъемы X2 и X3 программатора. Программатор для USB интерфейса питается через сам USB порт от блока питания компьютера. Поэтому при сборке и отладке программатора нужно особенно

тщательно проверять конструкцию на отсутствие коротких замыканий, что бы не вывести из строя компьютер. Если программируемая микросхема (или плата с программируемой микросхемой) потребляет не много энергии, то ее так же можно запитать от USB. Для этого нужно установить соответствующим образом перемычку J1.

Во всех трех схемах на шине питания установлены два конденсатора фильтра (10 мкф и 0,1 мкф). Светодиоды служат для индикации напряжения питания и режимов работы программатора.

Программа управления программатором PonyProg

Первые две схемы программаторов предназначены для работы с управляющей программой PonyProg.

Программа PonyProg — это открытый проект. Для распространения этой программы и еще нескольких проектов в Интернете создан специальный сайт (см. ссылку 5 в конце книги). Программа также распространяется с открытой лицензией (GNU), то есть вместе с текстом программы, который разрешается изменять по своему усмотрению. Однако в пакет программы входит специальная библиотека, которая содержит текст всех основных функций, обеспечивающих процесс программирования микросхем.

На библиотеку не распространяется открытая лицензия. Ее разрешается использовать, но не разрешается изменять входящие в нее процедуры. Изменения допускаются лишь в интерфейсе программы. Такое решение делает программу более надежной в работе.

На сайте вы можете загрузить не только инсталляционный пакет самой программы, но также исполняемый файл русифицированного или украинофицированного вариантов программы. Кроме этого, там еще имеется целый набор вариантов, поддерживающий множество других языков. После инсталляции программы вы просто меняете исполняемый файл в директории программы на новый, и программа полностью русифицируется. Однако учтите, что русифицированная версия программы — это устаревшая версия. Она может не поддерживать ряд микроконтроллеров. Поэтому, если вы не нашли в списке микросхем ту, что вам необходима, проинсталлируйте программу PonyProg заново и работайте с английской версией.

При запуске программы PonyProg открывается окно заставки и раздается фирменный звук — лошадиное ржание. Если вы не желаете слушать его каждый раз при запуске, поставьте галочку в поле «Disable Sound» (выключить звук). Нажмите «Ok». Рекламная заставка закроется, и откроется основная панель программы (см. **рис. 4.11**).

Главная панель содержит всего одно основное окно, где в свою очередь могут быть открыты одно или несколько окон с разными вариан-

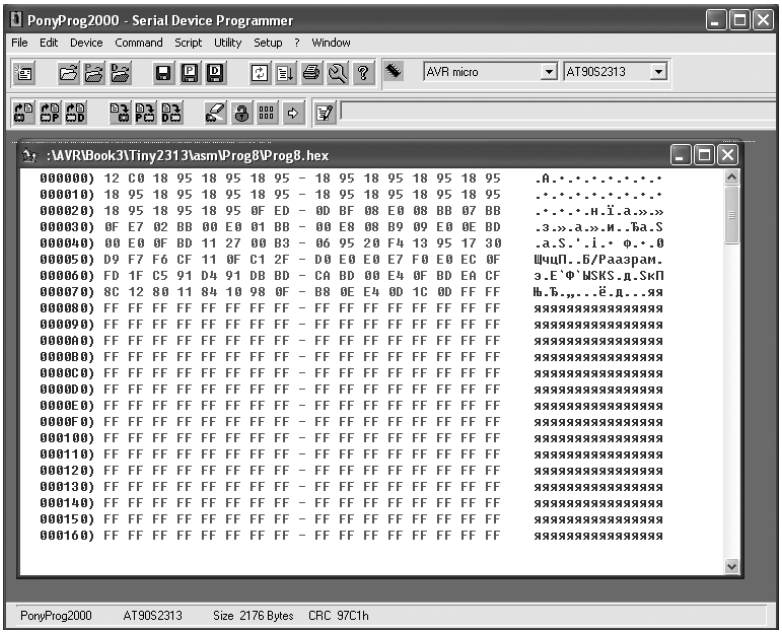


Рис. 4.11. *Основная панель программы PonyProg*

тами прошивок. В верхней части главной панели традиционно располагается меню и две панели инструментов.

Первое, что нужно сделать сразу после включения программы, — **выбрать тип микросхемы**, которую вы собираетесь программировать. Для этого в верхней панели инструментов имеются два выпадающих меню. Одно называется «Select device family» (Выбор семейства микросхем), а второе — «Select device type» (Выбор типа микросхем). Поля не подписаны. Названия появляются в виде всплывающей подсказки при наведении курсора мыши на соответствующее поле.

В поле выбора семейства выберите «AVR micro», а в поле выбора типа — требуемый тип микросхемы. Для всех наших примеров это будет ATtiny2313. Второй способ, при помощи которого также можно выбрать семейство и тип микросхемы, — воспользоваться меню «Device» (Устройство). Выбранный тип микросхемы автоматически сохраняется, и при повторном запуске программы вызывается снова.

Теперь вам необходимо **произвести настройку интерфейса** и **калибровку программы**. Эти две операции нужно выполнять только один раз, при первом запуске программы. Повторная настройка и калибровка может понадобиться лишь при сбое программы. Сначала выполним настройку интерфейса. Для этого нужно выбрать пункт «Interface Setup» меню «Setup» (см. **табл. 4.7**). Откроется окно настройки (см. **рис 4.12**).

Команды меню «Setup» (Установки) *Таблица 4.7*

Команда	Английский вариант	Кнопка	Описание
Настройка оборудования	Interface Setup		Открыть окно настройки интерфейса связи с компьютером
Калибровка	Calibration	–	Калибровка временных характеристик программатора

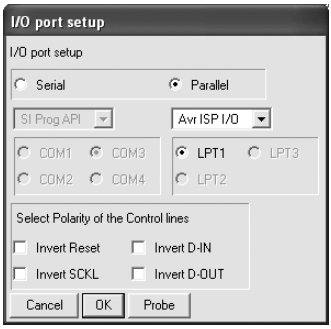


Рис. 4.12. *Окно настройки ввода-вывода*

В этом окне вы должны выбрать порт, к которому подключен ваш программатор. Кроме того, в этом окне можно проинвертировать любой из управляющих сигналов программатора, что бывает полезно при работе с нестандартными схемами. Для схемы, работающей с LPT интерфейсом выбираем параллельный порт (Parallel). Если ваш компьютер имеет несколько LPT-портов, выберите конкретный порт (обычно LPT1). В выпадающем списке в**ыберите способ программирования**. Это будет последовательное программирование по ISP-интерфейсу. Если в вашем компьютере имеется только один параллельный порт LPT1, то все установки должны выглядеть так, как это показано на **рис. 4.12**.

Если же вы выбрали схему, подключаемую при помощи COM порта, то в окне, показанном на **рис. 4.12** нужно выбрать режим «Serial» и вариант протокола «SI Prog API». Затем нужно выбрать конкретный номер COM порта (COM1, COM2, COM3 или COM4).

Вторая процедура, которую нужно выполнить хотя бы один раз при первом включении программы, — это **калибровка**. В процессе калибровки программа настраивает свои процедуры формирования временных интервалов под конкретный компьютер.

Цель калибровки — повышение точности формирования интервалов времени. При выполнении этой процедуры компьютер не должен выполнять никаких других программ. Закройте все открытые окна и выгрузите все программы, работающие в фоновом режиме. Затем выберите команду «Калибровка» (см. **табл. 4.7**). Появится окно предупреждения. Для старта процесса калибровки нажмите в этом окне кнопку «Ok». Процесс калибровки выполняется несколько секунд.

Программирование микросхем

После настройки и калибровки все готово для **программирования микросхем**. Для начала нам нужно загрузить данные из файлов, полу-

ченных в результате трансляции: файл программы для записи во Flash-память; файл данных для записи в EEPROM.

Для временного хранения этих данных программатор использует окно данных. Одно окно данных хранит один вариант задания (программа плюс данные). Запущенная программа Pony Prog обязательно содержит хотя бы одно такое окно. Пустое окно автоматически создается при запуске программы. После загрузки информации (программы или данных) в окне появляется дамп памяти.

***Дамп** – это широко распространенный способ представления цифровых данных. Он представляет собой таблицу шестнадцатиричных чисел, записанных рядами по 16 чисел в ряду (см. **рис. 4.11**).*

В начале каждого ряда записывается адрес первой его ячейки. Затем, правее, эти же шестнадцать чисел повторяются в символьном виде. То есть вместо каждого числа записывается соответствующий ему символ в кодировке ASCII.

В окно помещается сначала содержимое программной памяти микроконтроллера, а затем содержимое EEPROM. На **рис. 4.11** показан программатор с загруженной программой из примера номер 8. Из рисунка видно, что программа занимает первые восемь строчек. Причем занимает неполностью. С адреса 0x000000 по адрес 0x00007D. Дальше программная память пуста. Для того, чтобы зря не прошивать пустые ячейки, в них записан код 0xFF. Так как выбранная нами микросхема имеет объем программной памяти, равный 2 Кбайт, дамп программной памяти оканчивается ячейкой с адресом 0x0007FF. Но на этом дамп не заканчивается.

За содержимым программной памяти следует содержимое EEPROM. Причем адресация продолжается так, как будто содержимое обоих видов памяти составляет единое адресное пространство. Ячейка EEPROM с адресом 0x0000 в этой новой системе координат получает адрес 0x000800. И так далее, по нарастающей. Содержимое EEPROM на экране выделяется цветом. Пролистав дамп при помощи полосы прокрутки, вы сами можете в этом убедиться.

Размещение всей информации в едином адресном пространстве удобно, так как позволяет хранить программу и данные в одном файле. В процессе программирования микросхемы программатор автоматически отделяет программу от данных, используя информацию об объеме программной памяти данного конкретного микроконтроллера. Все, что выше этого объема, автоматически считается данными для EEPROM.

Для загрузки данных из файла, находящегося на жестком диске, в текущее окно программатора, а также для записи информации из окна программатора в файл, программа поддерживает ряд команд, объеди-

Команды меню «File» (Файл) *Таблица 4.8*

Команда	Английский вариант	Кнопка	Описание
Новое окно	New Window		Открыть новое пустое окно
Открыть файл с данными	Open Device file		Открыть полный файл с информацией для прошивки (программа и данные) и поместить ее в текущее окно
Открыть файл программы (Flash)	Open Program (Flash) File		Открыть файл с программой, предназначенной для прошивки, и поместить ее в текущее окно в область программы
Открыть файл данных (EEPROM)	Open Data (EEPROM) File		Открыть файл с данными для прошивки в EEPROM и поместить их в текущее окно в область данных
Сохранить файл с данными	Save Device File		Сохранить данные из текущего окна в файл (полный файл данных)
Сохранить файл с данными как —	Save Device File As—	-	Сохранить данные из текущего окна в виде нового файла с другим именем
Сохранить файл программ (Flash) как—	Save Program (Flash) File As—		Сохранить информацию из области программы текущего окна в файл (Информация для Flash)
Сохранить файл данных (EEPROM) как —	Save Data (EEPROM) File As—		Сохранить информацию из области данных текущего окна в файл (информация для EEPROM)
Открыть заново	Reload Files		Заново перечитать информацию текущего окна из файла
Печать	Print—		Открыть окно вывода на печать содержимого текущего окна
Закрыть	Close	–	Закрыть текущее окно. Если это последнее окно, то закрывается вся программа
Выход	Exit	–	Завершение работы программы

ненных в меню «File». Все эти и другие команды меню «File» приведены в **табл. 4.8**. В таблице приведены названия пунктов меню как на русском, так и на английском языках. А также показан внешний вид соответствующей этому пункту кнопки на панели инструментов.

Итак, з**агрузим программу и данные в программатор**. Если вы помните, все вышеперечисленные трансляторы создают отдельный файл для программы (файл с расширением hex) и отдельный файл для данных (файл с расширением eep). Поэтому для загрузки программы воспользуемся командой «Открыть файл программы (Flash)». При выборе этой команды появляется диалог «Открыть программу». Убедитесь, что в поле «Тип файла» выбрано «`*.hex`». Если это не так, выберите это значение из выпадающего списка.

Затем найдите на диске директорию вашего проекта, выберите файл и нажмите кнопку «Открыть». Загруженные данные появятся в текущем окне. Таким же образом загружается содержимое EEPROM. Только в этом случае нужно выбрать тип файла «`*.eep`».

После того, как программа и данные загружены, их можно просмотреть, при необходимости — подредактировать прямо в окне програм-

матора. А если нужно, то и записать обратно на диск. Если у вас есть принтер, можно распечатать дамп из текущего окна на бумаге.

Но основная функция — это, естественно, **запись программы и данных в память микроконтроллера**. Все команды, предназначенные для работы с микроконтроллером, сведены в меню «Command». Их описание приведено в **табл. 4.9**. При помощи этих команд вы можете отдельно запрограммировать память программ, отдельно — EEPROM. Команда «Записать все» позволяет запрограммировать программу и данные за одну операцию.

Команды меню «Command» (Команды) *Таблица 4.9*

Команда	Английский вариант	Кнопка	Описание
Считать все	Read All		Прочитать программу (из Flash) и данные (из EEPROM) из микросхемы и поместить в текущее окно
Считать программу (Flash)	Read Program (FLASH)		Прочитать программу (из Flash) из микросхемы и поместить в текущее окно в область программы
Считать данные (EEPROM)	Read DATA (EEPROM)		Прочитать данные (из EEPROM) из микросхемы и поместить в текущее окно в область данных
Записать все	Write All		Записать программу и данные из текущего окна в микросхему
Записать программу (Flash)	Write Program (FLASH)		Записать программу из текущего окна во Flash-память микросхемы
Записать данные (EEPROM)	Write DATA (EEPROM)		Записать данные из текущего окна в EEPROM-память микросхемы
Проверить все	Verify All	–	Сравнить программу и данные из текущего окна с содержимым соответственно Flash- и EEPROM-памяти микросхемы
Проверить программу (Flash)	Verify Program (FLASH)	–	Сравнить информацию из области программ текущего окна с информацией во Flash-памяти микросхемы
Проверить данные (EEPROM)	Verify DATA (EEPROM)	–	Сравнить информацию из области данных текущего окна с информацией в EEPROM-памяти микросхемы
Биты защиты и конфигурации	Security and Configuration Bits—		Открыть окно чтения/изменения битов защиты и битов конфигурации микросхемы
Стереть	Erase		Стереть Flash- и EEPROM-память микросхемы
Аппаратный сброс	Reset	–	Подать на микросхему сигнал аппаратного сброса
Программирование	Program		Выполнение последовательности команд, составляющих цикл программирования
Настройка программирования	Program Options—	–	Настройка цикла программирования (выбор команд)
Считать калибровочный бит ген.	Read Osc. Calibration Byte	–	Чтение значения калибровочного бита для дальнейшего использования в целях калибровки генератора
Настройка калибровки генератора	Osc. Calibration Options—	–	Настройка процедуры автоматической записи калибровочного значения в память данных или программ

Три команды считывания позволяют прочитать содержимое памяти программ и памяти данных микроконтроллера. Прочитанные данные помещаются в текущее окно программатора. Считанную из микросхемы информацию можно записать на диск при помощи команд, описанных в **табл. 4.8**. Группа команд проверки используется для сравнения информации, записанной в микросхему, и информации в текущем окне программатора.

Команда «Стереть» позволяет стереть память микросхемы. Команда стирает одновременно все виды памяти:

- память программ;
- память данных;
- ячейки защиты (если они были запрограммированы).

Однако здесь есть одно исключение. Некоторые микросхемы (в том числе и ATtiny2313) имеют бит конфигурации (fuse-переключатель), запрещающий стирание EEPROM. Если запрограммировать этот бит, то при стирании микросхемы EEPROM стираться не будет. Это позволяет не делать лишних циклов записи/стирания и сэкономить ресурс EEPROM в том случае, когда его содержимое менять не обязательно.

На **пункте меню «Биты защиты и конфигурации»** необходимо остановиться подробнее. Эта команда предназначена для чтения и изменения fuse-переключателей (битов конфигурации) и битов защиты микросхемы. В русскоязычном варианте программы этот пункт почему-то остался не переведенным.

При выборе этого пункта меню открывается окно, показанное на **рис. 4.13**. Набор элементов управления для каждого вида микросхем будет свой. Причем сразу после открытия окна все поля не будут выбраны (не будут содержать «галочек»). Это значит, что содержимое этих полей пока не соответствует реальному содержимому битов защиты и конфигурации микросхемы.

Для того, чтобы считать эти значения, необходимо нажать в том же окне кнопку «Read». На короткий момент появится окно, показываю-

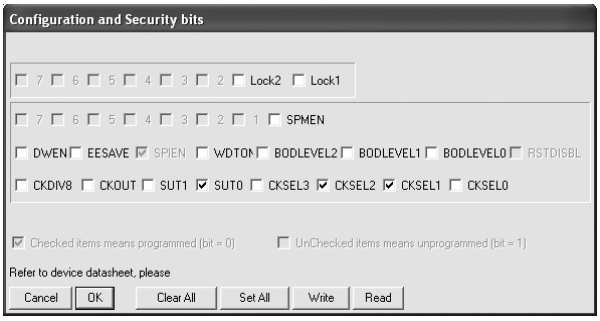


Рис. 4.13. *Окно установки битов конфигурации и защиты*

щее процесс считывания. Затем снова откроется окно битов защиты и конфигурации. Теперь уже все поля примут значения, считанные из микросхемы. Галочка в любом из полей означает, что данный бит запрограммирован.

> ***Напоминаю**, что запрограммированный бит содержит ноль, незапрограммированный — единицу.*

Теперь вы можете изменить значение любого бита. Но эти изменения будут только на экране. Для того, чтобы записать изменения в микросхему, нажмите кнопку «Write». Кнопки «Set All» и «Clear All» позволяют установить или сбросить значения сразу всех полей в данном окне.

Для всех примеров, приведенных в **главе 5**, необходимо установить такое значение битов, как показано на **рис. 4.13**. Если вы хотите обойтись без кварцевого резонатора, то достаточно поменять значения битов CKSEL3—CKSEL0 следующим образом:

- **CKSEL3** — 0 (есть галочка);
- **CKSEL2** — 0 (есть галочка);
- **CKSEL1** — 1 (нет галочки);
- **CKSEL0** — 0 (есть галочка).

Значения всех остальных битов нужно оставить без изменений. Когда вы запишите в микроконтроллер это измененное значение fuse-битов, то при использовании такой микросхемы в любом из наших программных примеров из схемы можно исключить кварцевый резонатор Q1 и конденсаторы C1 и C2.

Выводы XTAL0 и XTAL1 при этом останутся незадействованными. Все измененные таким образом схемы прекрасно работают. Единственный **недостаток внутреннего генератора** — низкая стабильность частоты. Однако ни один из наших примеров не критичен к стабильности частоты. Разве что тональность мелодий будет немного изменяться. Но на слух это обнаружить вряд ли удастся.

Две команды меню «Command» предназначены для калибровки внутреннего тактового генератора микроконтроллера. В микросхеме ATtiny2313 эта операция выполняется автоматически. Однако есть ряд микросхем, в которых значение калибровочного байта нужно вручную внести в одну из ячеек программной памяти процессора (или EEPROM). В программаторе эта операция автоматизирована.

Сначала вы должны определиться с адресом ячейки. Вы можете выбрать любую заведомо свободную ячейку в памяти программ или в EEPROM. Ваша программа должна быть составлена таким образом, чтобы в начале своей работы она читала содержимое этой ячейки и записывала его в регистр калибровки тактового генератора (OSCCAL).

Когда адрес ячейки известен, нужно настроить систему автоматического считывания байта калибровки в программаторе.

Для этого сначала выберите команду «Настройка калибровки генератора» (см. **табл. 4.9**). Откроется небольшое окно, куда вы должны ввести адрес ячейки. В том же окне имеется поле «Data memory offset» (Относительно памяти данных). Если поставить галочку в этом поле, то калибровочный байт будет записываться в EEPROM. Выбрав таким образом адрес и месторасположение ячейки, нажимаем кнопку «OK».

Теперь перед каждым программированием вам достаточно подать команду «Считать калибровочный байт ген.», и калибровочный байт будет прочитан и помещен по указанному вами адресу в текущее окно программатора. При записи информации в микроконтроллер калибровочное значение попадет в предназначенную ему ячейку автоматически.

Режимы работы программатора

Для удобства работы с программатором он имеет **режим группового выполнения команд**. Команды чтения информации из файла, чтения байта конфигурации, обновления серийного номера, стирания микросхемы и, наконец, программирования могут выполняться в пакете при нажатии всего одной кнопки. Для настройки пакета команд выберите пункт «Настройка программирования» меню «Command». Откроется окно «Program Options» (см. **рис. 4.14**). В этом окне отметьте галочками те операции, которые должны выполняться при запуске пакета, и нажмите кнопку «OK». Для запуска пакетной команды достаточно выбрать пункт «Программирование» (см. **табл. 4.9**). Или нажать кнопку на панели инструментов.

Пакетный режим очень удобен в процессе отладки программы. Если в пакет включена опция «Reload Files», то достаточно один раз вручную открыть нужный файл нужного проекта, а затем можно просто нажимать кнопку «Программирование» каждый раз, когда нужно перепрошить микросхему новой версией программы. Все остальное PoniProg сделает за вас. Новая версия программы сама загрузится в программатор, микросхема сотрется, а затем в нее запишется новая информация.

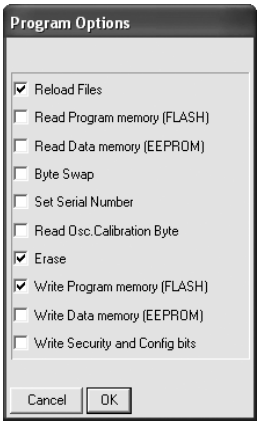


Рис. 4.14. *Настройка пакетного выполнения команд*

Только не забывайте поставить команду стирания, если вы собираетесь программировать. Помните, что при записи в нестертую микросхему результат непредсказуем. Каждый «про-

шитый» в процессе программирования бит может быть восстановлен только в результате стирания всей памяти.

И, в заключение, хочу рассказать об еще одной удобной функции программатора. Программатор имеет **встроенную систему автоматического формирования серийного номера программы**.

Серийный номер — это просто порядковый номер версии программы. Этот номер может автоматически записываться в выбранную вами ячейку памяти программ или памяти данных. Настройка данного режима производится при выборе пункта «SerialNumber Config...» (Установки серийного номера) меню «Utility» (Утилиты).

В открывшемся окне вы можете выбрать адрес ячейки для серийного номера, поставить галочку в поле «Data memory offset» (Относительно памяти данных), а также выбрать параметры его автоматического изменения. После настройки параметров изменение серийного номера и его запись в выбранную ячейку текущего окна программатора производится путем выбора пункта «Set Serial Number» (Установить серийный номер) меню «Utility» (Утилиты) или нажатием кнопки  .

Работа с управляющей программой USBasp

На сайте автора программатора (www.fischl.de) предлагается четыре варианта управляющей программы. Можете сами попробовать каждую из них. Я рекомендую использовать программу, которую автор назвал «Khazama AVR Programmer». Эта программа имеет минимальный интерфейс, но при этом работа с программой удобна и понятна. Программа имеет все необходимые функции. Особенностью программы является наличие фактически встроенного FUSE калькулятора.

Для выбора значений Фьюзов вы можете использовать набор выпадающих меню. Каждый пункт любого из этих меню — это описание словами одного из режимов работы программируемого микроконтроллера. Выбираешь нужные режимы, а программатор сам устанавливает значения Фьюзов.

Такой способ установки Фьюзов, кроме всего прочего, удобен тем, что в нем предусмотрен автоматический контроль за неправильным их выбором. Это гарантирует от «залочивания» программируемого микроконтроллера (эффект мертвой микросхемы). Однако, при необходимости, вы можете не пользоваться механизмом выбора режимов при помощи выпадающих списков и непосредственно выставлять значения Фьюзов. Это удобно, если вам уже заранее известны все эти значения.

Основное окно программы «Khazama AVR Programmer» показано на **рис. 4.15**.

Программа имеет меню и кнопки выбора функций управления, похожие на соответствующие кнопки программы PonyProg. Поэтому работа

с программой не должна вызвать затруднений.

Перед началом программирования необходимо выбрать тип программируемой микросхемы из выпадающего списка в центре основного окна программы (см. **рис. 4.15**). Все основные функции по программированию микросхем собраны в меню «Command» (см. **рис. 4.16**). Некоторые из них продублированы кнопками на панели управления программы.

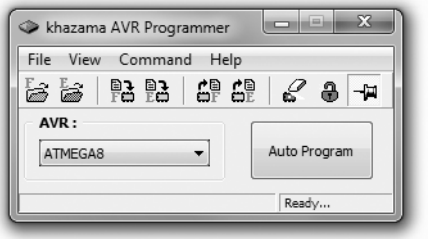


Рис. 4.15. *Основное окно управляющей программы*

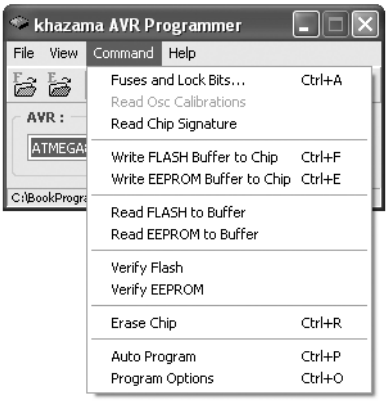


Рис. 4.16. *Меню «Comand»*

В **табл. 4.10** приведена расшифровка всех пунктов меню «Command».

Для того, что бы запрограммировать микросхему, нужно сначала загрузить данные для программирования из внешнего файла (HEX формат) в буфер программатора. Так же как и в программаторе PonyProg буфер программатора разбит на две части. Одна предназначена для хранения данных FLASH памяти (памяти программ), а вторая — для хранения данных EEPROM (памяти данных).

Для загрузки данных для FLASH памяти из файла в соответствующий буфер выполняется при помощи команды «Load FLASH file to Bufer» меню «File» или щелчком мыши по кнопке  . Для загрузки данных для EEPROM служит пункт «Load EEPROM file to Bufer» меню «File» или кнопка  .

Если память программируемого микроконтроллера не пуста, нужно выполнить очистку памяти (см. **табл. 4.10**). После этого можно прошивать память микроконтроллера соответствующими командами, перенося данные из буфера программатора в память микроконтроллера (см. описание команды «Erase Chip» в **табл. 4.10**).

Затем можно прошить микроконтроллер данными из буфера программатора. Для этого нужно воспользоваться командами «Write FLASH Buffer to Chip» и (или) «Write EEPROM Buffer to Chip» (см. **табл. 4.10**).

Так же, как и в PonyProg программатор USBasp имеет режим авто программирования. При нажатии кнопки «Auto Program» или выборе соответствующего пункта меню (см. **табл. 4.10**) программатор последовательно выполняет заранее заданные функции. Для настройки режима авто программирования служит пункт «Program Options» меню «Commsnd». При

Команды меню «Command» *Таблица 4.10*

Пункт меню	Расшифровка	Кнопка
Fuses and Lock Bits...	Работа с Фьюзами и битами защиты кристалла	
Read Osc Calibrations	Чтение байта калибровки тактового генератора	–
Read Chip Signature	Чтение сигнатуры микросхемы	–
Write FLASH Buffer to Chip	Запись данных из FLASH буфера программатора в программную память микроконтроллера	
Write EEPROM Buffer to Chip	Запись данных из EEPROM буфера программатора в память данных микроконтроллера	
Read FLASH to Buffer	Чтение из программной памяти программируемого микроконтроллера в буфер программатора.	
Read EEPROM to Buffer	Чтение из памяти данных программируемого микроконтроллера в буфер программатора.	
Verify Flash	Проверка Flash памяти контроллера (сравнение с буфером)	–
Verify EEPROM	Проверка EEPROM памяти контроллера (сравнение с буфером)	–
Erase Chip	Стирание памяти микроконтроллера	
Auto Program	Автопрограммирование	Одноименная кнопка
Program Options	Настройки команд автопрограммирования	–

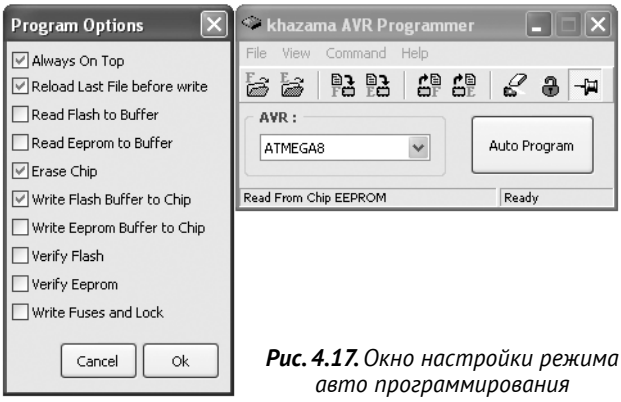


Рис. 4.17. *Окно настройки режима авто программирования*

выборе этого пункта открывается специальное окно со списком возможных команд (см. **рис 4.17**). Вы должны выбрать нужные и поставить галочку перед ними. Неотмеченные команды выполняться не будут.

Безусловным **преимуществом** программатора USBasp является развитый режим работы с битами защиты и программированием Фьюзов (FUSE). Для вызова этого режима служит пункт «Fuses and Lock Bits...» меню «Comand» и соответствующая кнопка на панели управления (см. **табл. 4.10**). При выборе этого режима открывается дополнительное окно. Как оно выглядит, вы можете видеть на **рис. 4.18**. В верхней части окна находится набор выпадающих списков при помощи которых, вы можете выбрать один из режимов микроконтроллера, определяющих способ защиты кристалла или его конфигурацию.

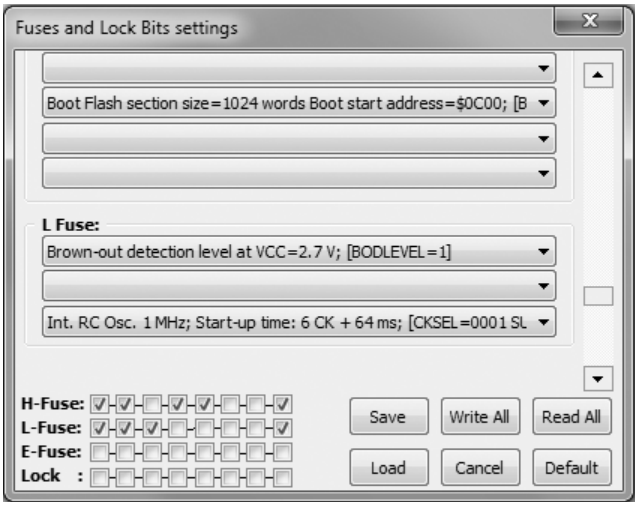


Рис. 4.18. *Режим работы с битами защиты и значениями FUSE переключателей*

Все выпадающие списки подразделены на три группы. Первая группа озаглавлена «Lock Bits» и служит для выбора режимов, связанных со значением битов защиты кристалла. Следующие две группы полей выбора озаглавлены «H Fuse» и «L Fuse». Они служат для определения режимов работы микроконтроллера, задаваемых, соответственно, старшим и младшим байтами FUSE переключателей.

> ***Обратите внимание:*** *в окне вы видите не весь список выпадающих меню. Для того, что бы увидеть остальные, первоначально скрытые органы настройки, нужно прокрутить список вниз. Для этого справа от набора полей имеется полоса прокрутки.*

В нижней части окна мы видим набор Чекбоксов, отражающих состояние каждого FUSE переключателя. При выборе режимов работы в верхней части окна они сразу же отображаются в нижней. И наоборот, если непосредственно задавать значения FUSE переключателей в нижней части окна, это тут же отображается в верхней части окна путем установки значений выпадающих списков.

При этом программа следит за корректностью выбора режимов. Поэтому, если в верхней части установлен один или несколько значений в выпадающих списках, то не для каждого фьюза можно изменить его значение. Некорректные значения не допускаются.

В нижней части окна так же расположены кнопки управления. Кнопка «Save» позволяет запомнить текущее состояние Фьюзов для последующего их восстановления. Кнопка «Load» восстанавливает запомненные значения фьюзов. Кнопка «Write All» служит для записи выбранных значений Фьюзов и битов защиты в микроконтроллер. Кнопка «Read All» служит для чтения состояния Фьюзов и битов защиты из микроконтроллера в это окно. Рекомендуется указанный ниже **порядок работы с фьюзами**.

При необходимости сменить значение одного или нескольких Фьюзов, сначала считайте значение всех Фьюзов из микроконтроллера.

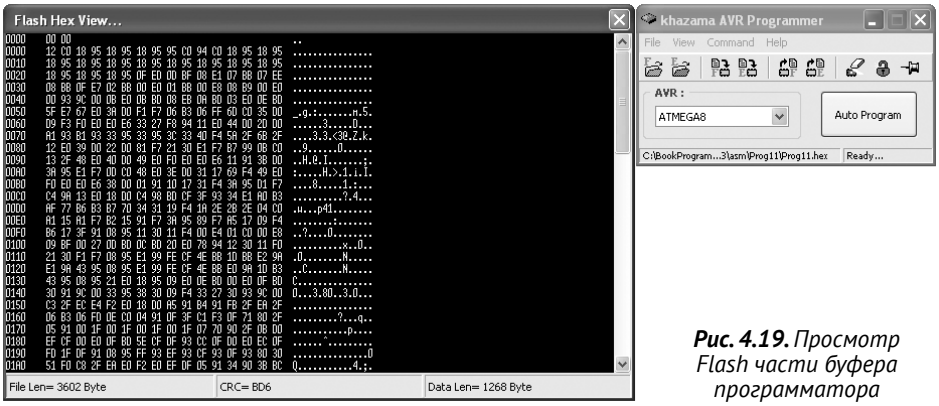


Рис. 4.19. *Просмотр Flash части буфера программатора*

Исправьте те значения, которые вы желаете поменять, и лишь затем запишите значения всех фьюзов обратно в микроконтроллер. Кнопка «Default» устанавливает все Фьюзы в положение по умолчанию (такое, в какое они устанавливаются производителем). Кнопка «Cancel» закрывает окно, не выполняя ни каких действий.

Кроме описанных выше возможностей программатор «Khazama AVR Programmer» имеет функцию просмотра содержимого своего буфера. Для просмотра FLASH-части буфера служит пункт «View Flash Hex Data» меню «View». При выборе этого пункта меню открывается специальное окно просмотра (см. **рис. 4.19**). Информация в окне просмотра представлена в HEX формате. Для просмотра EEPROM части буфера служит пункт «View EEPROM Hex Data» меню «View».

Кнопка служит для включения режима «Поверх всех окон». В этом режиме все окна программатора всегда будут находиться поверх любых окон Windows.

ГЛАВА 5

ПОЛЕЗНЫЕ УСТРОЙСТВА НА МИКРОКОНТРОЛЛЕРАХ СВОИМИ РУКАМИ

5.1. Общие положения

Главная задача этой книги — научиться создавать программы для микроконтроллеров. Как можно узнать из [4], программа для микроконтроллера — это набор кодов, который записывается в его специальную программную память. Программу должен написать программист, который разрабатывает ту или иную конкретную микропроцессорную систему.

Однако программист никогда не имеет дело с кодами. Часто программист даже и не задумывается о том, какой код соответствует той или иной команде. Дело в том, что для человека программирование в кодах очень неудобно. Человек же не компьютер.

Для человека удобнее оперировать с командами, каждая из которых имеет свое осмысленное название. Поэтому для написания программ человек использует языки программирования.

В качестве слов в языке программирования выступают:

- команды (операторы);
- специальные управляющие слова;
- названия регистров;
- числовые выражения.

Язык программирования *— это специально разработанный язык, служащий посредником между машиной и человеком. Как и обычный человеческий язык, любой язык программирования имеет свой словарь (набор слов) и правила их написания.*

Главная задача языка — однозначно описать последовательность действий, которую должен выполнить ваш микроконтроллер. В то же время язык должен быть удобен и понятен человеку.

В процессе создания программы программист просто пишет ее текст на компьютере точно так же, как он пишет любой другой текст. Затем программист запускает специальную программу — транслятор.

Написанный программистом текст программы называется **исходным** или **объектным кодом**. Код, полученный в результате трансляции, называется **результирующим** или **машинным кодом**. Именно

Транслятор *— это специальная программа, которая переводит текст, написанный программистом, в машинные коды, то есть в форму, понятную для микроконтроллера.*

этот код записывается в программную память микроконтроллера. Для записи результирующего кода в программную память применяются специальные устройства — **программаторы**. О программаторах мы подробно говорили ранее (см. **главу 4**).

Все языки программирования делятся на две группы: **языки низкого уровня** (машиноориентированные); **языки высокого уровня**.

Типичным примером машиноориентированного языка программирования является **язык Ассемблер**. Этот язык максимально приближен к системе команд микроконтроллера. Каждый оператор этого языка — это, по сути, словесное название какой-либо конкретной команды.

В процессе трансляции такая команда просто **заменяется кодом операции**. Составляя программу на языке Ассемблер, программист должен оперировать теми же видами данных, что и сам процессор, то есть байтами и битами.

Специфика языка Ассемблер состоит еще и в том, что набор операторов для этого языка напрямую зависит от системы команд конкретного микроконтроллера. Поэтому, если два микроконтроллера имеют разную систему команд, то и язык Ассемблер для каждого такого микроконтроллера будет свой. В данной книге мы будем изучать одну конкретную версию языка Ассемблер. А именно **Ассемблер для микроконтроллеров AVR**.

В недавнем прошлом язык Ассемблер был единственным языком программирования для микроконтроллеров. Только он позволял эффективно использовать скудные ресурсы самых первых микросхем. Однако в настоящее время, когда возможности современных микроконтроллеров значительно возросли, для составления программ все чаще используются языки высокого уровня, такие как **Бейсик**, **СИ** и т. п.

Эти языки в свое время были разработаны для больших настоящих компьютеров. Но сейчас широко используются также и для микроконтроллеров. Языки высокого уровня отличаются тем, что они гораздо больше ориентированы на человека. Большинство команд языков высокого уровня не связаны с конкретными командами микроконтроллера.

Такие языки оперируют уже не с байтами, а с привычными нам десятичными числами, а также с переменными, константами и другими элементами, знакомыми нам из математики. Константы и переменные могут принимать привычные для нас значения.

Например, положительные, отрицательные значения, вещественные значения (десятичные дроби) и т. п. Со всеми переменными и кон-

стантами можно выполнять знакомые нам арифметические операции и даже алгебраические функции.

Транслятор с языка высокого уровня производит более сложные преобразования, чем транслятор с Ассемблера. Но в результате тоже получается **программа в машинных кодах**. При этом транслятор использует все ресурсы микроконтроллера по своему усмотрению. В каких именно регистрах или ячейках памяти она будет хранить значения описанных вами переменных, по каким алгоритмам она будет вычислять математические функции, программист обычно не задумывается.

Программа-транслятор выбирает все это сама. Поэтому задача эффективности алгоритма полученной в результате трансляции программы целиком ложится на программу-транслятор. В целом, программы, написанные на языках высокого уровня, занимают в памяти микроконтроллера объем на 30—40 % больший, чем аналогичные программы, написанные на языке Ассемблер.

Однако если микроконтроллер имеет достаточно памяти и запас по быстродействию, то это увеличение программы — не проблема. **Преимуществом** же языков высокого уровня является существенное ускорение процесса разработки программы. Из всех языков высокого уровня **самым эффективным**, пожалуй, является язык СИ. Поэтому для иллюстрации языков высокого уровня мы выберем именно его.

Изучение приемов программирования мы будем осуществлять на ряде конкретных примеров:

- каждый пример будет начинаться с постановки задачи;
- затем мы научимся выбирать схемное решение;
- лишь после этого будут представлены примеры программ.

Для каждой задачи в книге приводятся два варианта программы. Одна на языке Ассемблер, вторая на языке СИ. В результате вы сможете не только научиться азам программирования на двух языках, но и понять все достоинства и недостатки каждого из языков программирования.

Все **примеры**, приведенные в моей книге, вы можете попробовать вживую на вашем компьютере. Причем текст программ не обязательно набирать вручную. Все приведенные в книге примеры вы можете скачать из Интернета с сайта **http://book.mirmk.net**. Кроме точной копии всех программ, приведенных в данной книге, на сайте вы найдете целый ряд дополнительных примеров. Все подробности смотрите на самом сайте.

5.2. Простейшая программа

Постановка задачи

Самая простая задача, которую можно придумать для микроконтроллера, может звучать следующим образом:

«Разработать устройство управления одним светодиодным индикатором при помощи одной кнопки. При нажатии кнопки светодиод должен зажечься, при отпускании — погаснуть».

С практической точки зрения это совершенно бессмысленная задача, так как для ее решения проще обойтись без микропроцессора. Но в качестве примера для обучения подойдет прекрасно.

Принципиальная электрическая схема

Попробуем разработать принципиальную электрическую схему, способную выполнять описанную выше задачу. Итак, к микроконтроллеру нам нужно подключить **светодиод** и **кнопку управления**. Как мы уже говорили, для подключения к микроконтроллеру AVR любых внешних устройств используются **порты ввода-вывода**. Причем каждый такой порт способен работать либо на ввод, либо и на вывод.

Удобнее всего светодиод подключить к одному из портов, а кнопку — к другому. В этом случае управляющая программа должна будет настроить порт, к которому подключен светодиод, на вывод, а порт, к которому подключена кнопка, на ввод. Других специальных требований к микроконтроллеру не имеется. Поэтому выберем микроконтроллер.

Очевидно, что нам нужен микроконтроллер, который имеет не менее двух портов. Данным условиям удовлетворяют многие микроконтроллеры AVR. Я предлагаю остановить свой выбор на довольно интересной **микросхеме ATtiny2313**. Эта микросхема, хотя и относится к семейству «Ttiny», на самом деле занимает некое промежуточное место между семейством «Ttiny» и семейством «Mega». Она не так перегружена внутренней периферией и не столь сложна, как микросхемы семейства «Mega». Но и не настолько примитивна, как все остальные контроллеры семейства «Ttiny».

Эта микросхема содержит два основных и один дополнительный порт ввода-вывода, имеет не только восьмиразрядный, но и шестнадцатиразрядный таймер/счетчик. Имеет оптимальные размеры (20-выводной корпус). И, по моему мнению, идеально подходит в качестве примера для изучения основ программирования. К тому же эта микросхема имеет и еще одну привлекательную особенность. По набору портов и расположению выводов она максимально приближена к микроконтроллеру AT89C2051, который был использован в качестве примера в первом моем Самоучителе (2005 г.).

Итак, если не считать порта A, который включается только в особом режиме, который мы пока рассматривать не будем, микроконтроллер имеет два основных порта ввода-вывода (порт B и порт D). Договоримся, что для управления светодиодом мы будем использовать младший разряд порта B (линия PB.0), а для считывания информации с кнопки управления используем младший разряд порта D (линия PD.0). Полная схема устройства, позволяющего решить поставленную выше задачу, приведена на **рис. 5.1**.

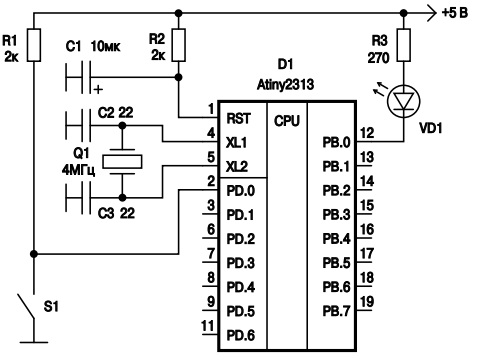


Рис. 5.1. *Принципиальная схема с одним светодиодом и одной кнопкой*

Для подключения кнопки S1 использована классическая схема. В исходном состоянии контакты кнопки разомкнуты. Через резистор R1 на вход PD.0 микроконтроллера подается «плюс» напряжения питания, что соответствует сигналу логической единицы.

При замыкании кнопки напряжение падает до нуля, что соответствует логическому нулю. Таким образом, считывая значение сигнала на соответствующем выводе порта, программа может определять момент нажатия кнопки. Несмотря на простоту данной схемы, микроконтроллер AVR позволяет ее упростить. А именно, предлагаю исключить резистор R1, заменив его внутренним нагрузочным резистором микроконтроллера. Как уже говорилось выше, микроконтроллеры серии AVR имеют встроенные нагрузочные резисторы для каждого разряда порта. Главное при написании программы — не забыть включить программным путем соответствующий резистор.

Подключение светодиода также выполнено по классической схеме. Это непосредственное подключение к выходу порта. Каждый выход микроконтроллера рассчитан на непосредственное управление светодиодом среднего размера с током потребления до 20 мА. В цепь светодиода включен токоограничивающий резистор R3.

Для того, чтобы зажечь светодиод, микроконтроллер должен подать на вывод PB.0 сигнал логического нуля. В этом случае напряжение, приложенное к цепочке R2, VD1, окажется равным напряжению питания, что вызовет ток через светодиод, и он загорится. Если же на вывод PD.0 подать сигнал логической единицы, падение напряжения на светодиоде и резисторе окажется равным нулю, и светодиод погаснет.

Кроме цепи подключения кнопки и цепи управления светодиодом, на схеме вы можете видеть еще несколько цепей. Это стандартные цепи,

обеспечивающие нормальную работу микроконтроллера. Кварцевый резонатор Q1 обеспечивает работу встроенного тактового генератора. Конденсаторы C2 и C3 — это цепи согласования кварцевого резонатора.

Элементы C1, R2 — это стандартная цепь начального сброса. Такая цепь обеспечивает сброс микроконтроллера в момент включения питания. Еще недавно подобная цепь была обязательным атрибутом любой микропроцессорной системы. Однако технология производства микроконтроллеров достигла такого уровня, что обе эти цепи (внешний кварц и цепь начального сброса) теперь можно исключить.

Большинство микроконтроллеров AVR, кроме тактового генератора с внешним кварцевым резонатором, содержат внутренний RC-генератор, не требующий никаких внешних цепей. Если вы не предъявляете высоких требований к точности и стабильности частоты задающего генератора, то микросхему можно перевести в режим внутреннего RC-генератора и отказаться как от внешнего кварца (Q1), так и от согласующих конденсаторов (C2 и C3).

Цепь начального сброса тоже можно исключить. Любой микроконтроллер AVR имеет внутреннюю систему сброса, которая в большинстве случаев прекрасно обеспечивает стабильный сброс при включении питания. Внешние цепи сброса применяются только при наличии особых требований к длительности импульса сброса. А это бывает лишь в тех случаях, когда микроконтроллер работает в условиях больших помех и нестабильного питания.

Все описанные выше переключения производятся при помощи соответствующих fuse-переключателей. Как это можно сделать, мы подробно описали в **предыдущей главе**. Три освободившихся вывода микроконтроллера могут быть использованы как дополнительный порт (порт A). Но в данном случае в этом нет необходимости.

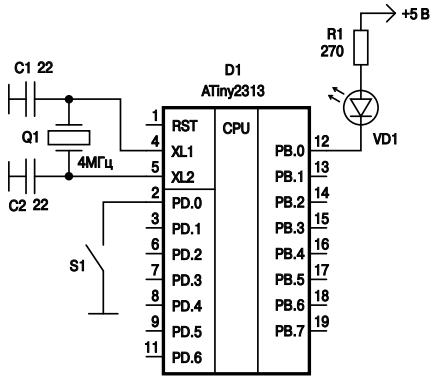


Рис. 5.2. *Усовершенствованная схема для первого задания*

Упростим схему, показанную на **рис. 5.1,** с учетом описанных выше возможностей. От внешнего кварца пока отказываться не будем. Он нам пригодиться чуть позже, когда мы начнем формировать временные интервалы. Доработанная схема изображена на **рис. 5.2**.

Алгоритм

Итак, схема у нас есть. Теперь нужно приступать к разработке программы. Разработка любой программы начинается с разработки алгоритма.

В нашем случае алгоритм таков: после операций начальной настройки портов микроконтроллер должен войти в непрерывный цикл, в процессе которого он должен опрашивать вход, подключенный к нашей кнопке, и в зависимости от ее состояния управлять светодиодом. Опишем это подробнее.

> ***Алгоритм*** *— это последовательность действий, которую должен произвести наш микроконтроллер, чтобы достичь требуемого результата. Для простых задач алгоритм можно просто описать словами. Для более сложных задач алгоритм рисуется в графическом виде.*

Операции начальной настройки:

- установить начальное значение для вершины стека микроконтроллера;
- настроить порт B на вывод информации;
- подать на выход PB.0 сигнал логической единицы (потушить светодиод);
- сконфигурировать порт D на ввод;
- включить внутренние нагрузочные резисторы порта D.

Операции, составляющее тело цикла:

- прочитать состояние младшего разряда порта PD (PD.0);
- если значение этого разряда равно единице, выключить светодиод;
- если значение разряда PD.0 равно нулю, включить светодиод;
- перейти на начало цикла.

Программа на Ассемблере

Для создания программ мы используем версию Ассемблера, предложенную разработчиком микроконтроллеров AVR — фирмой Atmel. А также воспользуемся программным комплексом «AVR Studio», разработанным той же фирмой и предназначенным для создания, редактирования, трансляции и отладки программ для AVR на Ассемблере. Подробнее о программной среде «AVR Studio» мы говорили в **предыдущей главе**.

А сейчас наша **задача** — научиться создавать программы. Изучение языка будет происходить следующим образом. Я буду приводить готовый текст программы для каждой конкретной задачи, а затем подробно описывать все его элементы и объяснять, как программа работает.

Текст возможного варианта программы, реализующий поставленную выше задачу, приведен в **листинге 5.1**. Прежде, чем мы приступим к описанию данного примера, я хотел бы дать несколько общих понятий о языке Ассемблер.

Программа на Ассемблере представляет собой набор команд и комментариев (иногда команды называют **инструкциями**). Каждая

Некоторые команды состоят только из одного оператора. Другие же команды имеют один или два операнда (параметра). Операнды записываются в той же строке сразу после оператора, через пробел. Если операндов два, их записывают через запятую.

команда занимает одну отдельную строку. Их допускается перемежать пустыми строками. Команда обязательно содержит оператор, который выглядит как имя выполняемой операции.

Так, в **строке 6** нашей программы записана команда загрузки константы в регистр общего назначения. Она состоит из оператора `ldi` и двух операндов `temp` и `RAMEND`.

В случае необходимости перед командой допускается ставить так называемую **метку**. Она состоит из имени метки, заканчивающимся двоеточием. Метка служит для именования данной строки программы. Затем это имя используется в различных командах для обращения к помеченной строке.

При выборе имени метки необходимо соблюдать следующие **правила**:

- имя должно состоять из одного слова, содержащего только латинские буквы и цифры;
- допускается также применять символ подчеркивания;
- первым символом метки обязательно должна быть буква или символ подчеркивания.

Строка 16 нашей программы содержит метку с именем `main`. Метка не обязательно должна стоять в строке с оператором. Допускается ставить метку в любой строке программы. Кроме команд и меток, программа содержит комментарии.

Если в уже готовой программе вы поставите точку с запятой в начале строки перед какой-либо командой, то данная строка для транслятора как бы исчезнет. С этого момента транслятор будет считать всю эту строку комментарием. Таким образом, можно временно отключать отдельные строки программы в процессе отладки (то есть при поиске ошибок в программе).

***Комментарий** — это специальная запись в теле программы, предназначенная для человека. Компьютер в процессе трансляции программы игнорирует все комментарии. Комментарий может занимать отдельную строку, а может стоять в той же строке, что и команда. Начинается комментарий с символа «точка с запятой». Все, что находится после точки с запятой до конца текущей строки программы, считается комментарием.*

Кроме операторов, в языке Ассемблер применяются **псевдооператоры** или **директивы**. Если оператор — это некий эквивалент реальной команды микроконтроллера и в процессе трансляции заменяется соответствующим

машинным кодом, который помещается в файл результата трансляции, то директива, хотя по форме и напоминает оператор, но не является командой процессора.

> ***Директивы*** *– это специальные вспомогательные команды для транслятора, определяющие режимы трансляции и реализующие различные вспомогательные функции.*

Далее из конкретных примеров вы поймете, о чем идет речь. В данной конкретной версии Ассемблера директивы выделяются особым образом. Имя каждой директивы начинается с точки. Смотри **листинг 5.1, строки с 1 по 5.**

При написании программ на Ассемблере принято соблюдать **особую форму записи:**

- программа записывается в несколько колонок (см. **листинг 5.1**);
- аналогичные элементы разных команд принято размещать друг под другом;
- самая первая (левая) колонка зарезервирована для меток;
- если метка отсутствует, место в колонке пустует;
- следующая колонка предназначена для записи операторов;
- затем идет колонка для операндов;
- оставшееся пространство (крайняя колонка справа) предназначено для комментариев.

В некоторых случаях, например , когда текст команды очень длинный, допускается нарушать этот порядок. Но, по возможности, нужно оформлять программу именно так. Оформленная подобным образом программа более наглядна и гораздо лучше читается. Поэтому привыкайте писать программы правильно.

Итак, мы рассмотрели общие принципы построения программы на Ассемблере. Теперь пора приступать к подробному описанию конкретной программы, приведенной в **листинге 5.1**. И начнем мы с описания входящих в нее команд.

Директивы

.include

Присоединение к текущему тексту программы другого программного текста. Подобный прием используется практически во всех существующих языках программирования. При составлении программ часто бывает как, что в совершенно разных программах приходится применять абсолютно одинаковые программные фрагменты. Для того, чтобы не переписывать эти фрагменты из программы в программу, их принято оформлять в виде отдельного файла с таким расчетом, чтобы этот файл могли использовать все программы, где этот фрагмент потребуется.

В языке Ассемблер для присоединения фрагмента к программе используется **псевдоопера тор `include`**. В качестве параметра для этой директивы должно быть указано имя присоединяемого файла. Если такой оператор поставить в любом месте программы, то содержащийся в присоединяемом файле фрагмент в процессе трансляции как бы вставляется в то самое место, где находится оператор. **Например**, в программе на **листинге 5.1** в **строке 1** в основной текст программы вставляется текст из файла `tn2313def.inc`.

Кстати, подробнее об этом файле. Файл `tn2313def.inc` — это файл описаний. Он содержит описание всех регистров и некоторых других параметров микроконтроллера ATtiny2313. Это описание понадобится нам для того, чтобы в программе мы могли обращаться к каждому регистру по его имени. О том, как делаются такие описания, мы поговорим при рассмотрении конкретных программ.

.list

Включение генерации листинга. В данном случае листинг — это специальный файл, в котором отражается весь ход трансляции программы. Такой листинг повторяет весь текст вашей программы, включая все присоединенные фрагменты. Против каждой строки программы, содержащей реальную команду, помещаются соответствующие ей машинные коды. Там же показываются все найденные в процессе трансляции ошибки. По умолчанию листинг не формируется. Если вам нужен листинг, включите данную команду в вашу программу.

.def

Макроопределение. Эта команда позволяет присваивать различным регистрам микроконтроллера любые осмысленные имена, упрощающие чтение и понимание текста программы. В нашем случае нам понадобится один регистр для временного хранения различных величин. Выберем для этой цели регистр `r16` и присвоим ему наименование `temp` от английского слова temporary — временный.

Данная команда выполняется в **строке 3** (см. **листинг 5.1**). Теперь в любом месте программы вместо имени r16 можно применять имя `temp`. Вы спросите: а зачем это нужно? Да для наглядности и читаемости программы. В данной программе мы будем использовать лишь один регистр, и преимущества такого переименования здесь не очень видны. Но представьте, что вы используете множество разных регистров для хранения самых разных величин. В этом случае присвоение осмысленного имени очень облегчает программирование. Скоро вы сами в этом убедитесь. Кстати, именно таким образом определены имена всех стандартных регистров в файле `tn2313def.inc`.

.cseg

Псевдооператор выбора программного сегмента памяти. О чем идет речь? Как уже говорилось, микроконтроллер для хранения данных имеет **три вида памяти**: память программ (Flash), оперативную память (SRAM) и энергонезависимую память данных (EEPROM). Программа на Ассемблере должна работать с любым из этих трех видов памяти. Для этого в Ассемблере существует понятие «сегмент памяти». Существуют директивы, объявляющие каждый такой сегмент:

- сегмент кода (памяти программ) . . . . . .cseg;
- сегмент данных (ОЗУ) . . . . . .dseg;
- сегмент EEPROM . . . . . .eseg.

После объявления каждого такого сегмента он становится **текущим**. Это значит, что все последующие операторы относятся исключительно к объявленному сегменту. Объявленный сегмент будет оставаться текущим до тех пор, пока не будет объявлен какой-либо другой сегмент.

Только в сегменте кода Ассемблер описывает команды, которые затем в виде кодов будут записаны в память программ. В остальных двух сегментах используются директивы распределения памяти и директивы описания данных. Ну, к сегментам `dseg` и `eseg` мы еще вернемся. Сейчас же подробнее рассмотрим сегмент `cseg`.

Так как команды в программной памяти должны располагаться по порядку, одна за другой, то их размещение удобно автоматизировать. Программист не указывает, по какому адресу в памяти должна быть расположена та либо иная команда. Программист просто последовательно пишет команды.

А уже транслятор автоматически размещает их в памяти. Для этого используется понятие **«указатель текущего адреса»**. Указатель текущего адреса не имеет отношения к регистру адреса микроконтроллера и вообще физически не существует. Это просто понятие, используемое в языке Ассемблер. Указатель помогает транслятору разместить все команды программы по ячейкам памяти.

По умолчанию считается, что в начале программы значение текущего указателя рано нулю. Поэтому первая же команда программы будет размещена по нулевому адресу. По мере трансляции программы указатель смещается в сторону увеличения адреса. Если команда имеет длину в один байт, то после ее трансляции указатель смещается на одну ячейку. Если команда состоит из двух байтов — на две. Таким образом, размещаются все команды программы.

.org

Принудительное позиционирование указателя текущего адреса. Иногда необходимо разместить какой-либо фрагмент программы в

программной памяти не сразу после предыдущего фрагмента, а в конкретном месте программной памяти. **Например**, начиная с какого-нибудь заранее определенного адреса. Для этого используют директиву `org`.

Она позволяет принудительно изменить значение указателя текущего адреса. Оператор `org` имеет всего один параметр — новое значение указателя адреса. **К примеру**, команда `.org 0x10` установит указатель на адрес 0x10. Транслятор автоматически следит, чтобы при перемещении указателя ваши фрагменты программы не налезали друг на друга. В случае несоблюдения этого условия транслятор выдает сообщение об ошибке.

В нашей программе команда позиционирования указателя применяется всего один раз. В **строке 5** указатель устанавливается на нулевой адрес. В данном случае директива `org` имеет чисто декларативное значение, так как в начале программы значение указателя и так равно нулю.

Операторы

ldi

Загрузка в РОН числовой константы. В **строке 6** программы (листинг 5.1) при помощи этой команды в регистр `temp` (`r16`) записывается числовая константа, равная максимальному адресу ОЗУ. Эта константа имеет имя RAMEND. Ее значение описано в файле `tn2313def.inc`. В нашем случае (для микроконтроллера ATtiny2313) значение RAMEND равно $DF.

Как можно видеть из **листинга 5.1**, оператор `ldi` имеет два параметра:

- первый параметр — это имя РОН, куда помещается наша константа;
- второй параметр — значение этой константы.

Обратите внимание, что в команде сначала записывается приемник информации, затем ее источник. Такой же порядок вы увидите в любой другой команде, имеющей два операнда. Это **общее правило** для языка Ассемблер.

out

Вывод содержимого РОН в регистр ввода-вывода. Команда также имеет два параметра:

- первый параметр — имя РВВ, являющегося приемником информации;
- второй параметр — имя РОН, являющегося источником.

В **строке 7** программы содержимое регистра `temp` выводится в РВВ с именем `SPL`.

in

Ввод информации из регистра ввода-вывода. Имеет два параметра. Параметры те же, что и в предыдущем случае, но источник и приемник меняются местами. В **строке 19** программы содержимое регистра `PORTD` помещается в регистр `temp`.

rjmp

Команда безусловного перехода. Команда имеет всего один параметр — адрес перехода. В **строке 18** программы оператор безусловного перехода передает управление на строку, помеченную меткой `main`. То есть на **строку 16.** Данная строка демонстрирует использование метки.

На самом деле в качестве параметра оператора rjmp должен выступать так называемый относительный адрес перехода. То есть число байт, на которое нужно сместиться вверх или вниз от текущего адреса. Направление смещения (вверх или вниз) — это знак числа. Он определяется старшим битом. Язык Ассемблера избавляет программиста от необходимости подсчета величины смещения. Достаточно в нужной строке программы поставить метку, а в качестве адреса перехода указать ее имя, и транслятор сам вычислит значение этого параметра.

При использовании команды `rjmp` существует одно **ограничение**. Соответствующая команда микроконтроллера кодируется при помощи одного шестнадцатиразрядного слова. Для указания величины смещения она использует всего двенадцать разрядов. Поэтому такая команда может вызвать переход в пределах ±2 Кбайт. Если вы расположите метку слишком далеко от оператора `rjmp`, то при трансляции программы это вызовет сообщение об ошибке.

Описание программы (листинг 5.1)

Листинг 5.1

```
;###############################################
;##               Пример 1                    ##
;##     Программа управления светодиодом      ##
;###############################################

;------------------------- Команды управления

1   .include "tn2313def.inc"                    ; Присоединение файла описаний
2   .list                                       ; Включение листинга

3   .def   temp = R16                           ; Определение главного рабочего регистра

;------------------------- Начало программного кода
```

```
 4            .cseg                          ; Выбор сегмента программного кода
 5            .org      0                    ; Установка текущего адреса на ноль

      ;-------------------------- Инициализация стека

 6            ldi       temp, RAMEND         ; Выбор адреса вершины стека
 7            out       SPL, temp            ; Запись его в регистр стека

      ;-------------------------- Инициализация портов ВВ

 8            ldi       temp, 0              ; Записываем 0 в регистр temp
 9            out       DDRD, temp           ; Записываем этот ноль в DDRD (порт PD на ввод)

10            ldi       temp, 0xFF           ; Записываем число $FF в регистр temp
11            out       DDRB, temp           ; Записываем temp в DDRB (порт PB на вывод)
12            out       PORTB, temp          ; Записываем temp в PORTB (потушить светодиод)
13            out       PORTD, temp          ; Записываем temp в PORTD (включаем внутр. резист.)

      ;-------------------------- Инициализация компаратора

14            ldi       temp, 0x80           ; Выключение компаратора
15            out       ACSR, temp

      ;-------------------------- Основной цикл

16    main:   in        temp, PIND           ; Читаем содержимое порта PD
17            out       PORTB, temp          ; Пересылаем в порт PB
18            rjmp      main                 ; К началу цикла
```

Текст программы **начинается** шапкой с названием программы. **Шапка** представляет собой несколько строк комментариев. Шапка в начале программы помогает отличать программы друг от друга. Кроме названия программы, в шапку можно поместить ее версию, а также дату написания.

Самая первая команда программы — это псевдокоманда `include`, которая присоединяет к основному тексту программы файл описаний (см. **листинг 5.1 строка 1**). В стандартном пакете AVR-Studio имеется целый набор подобных файлов описаний. Для каждого микроконтроллера серии AVR — свой отдельный файл. Все стандартные файлы описаний находятся в директории «C:\Program Files\Atmel\AVR Tools\AvrAssembler\Appnotes\». Программисту нужно лишь выбрать нужный файл и включить подобную **строку** в свою программу. Учтите, что без присоединения файла описаний дальнейшая программа работать не будет.

Для микроконтроллера ATtiny2313 **файл описаний** имеет название `tn2313def.inc`. Если файл описаний находится в указанной выше директории, то в команде `include` достаточно лишь указать его полное имя (с расширением). Указывать полный путь необязательно.

Назначение команды `.list` (**строка 2**), надеюсь, у вас уже не вызывает вопросов. Остановимся на **команде макроопределения (строка).** Эта команда, как уже говорилось, присваивает регистру `r16` имя `temp`. Дальше в программе регистр `temp` используется для временного хранения промежуточных величин. Уместно задаться **вопро-**

сом: почему выбран именно r16, а, к примеру, не r0? Это становится понятно, если вспомнить, что регистры, начиная с r0 и заканчивая r15, имеют меньше возможностей. **Например**, в **строке 14** программы регистр temp используется в команде ldi. Однако команда ldi не работает с регистрами r0—r15. Именно по этой причине мы и выбрали r16.

Следующие две команды (**строки 4, 5**) подробно описаны в начале этого раздела. Они служат для выбора программного сегмента памяти и установки начального значения указателя.

В **строках 6 и 7** производится **инициализация стека**. В регистр стека SPL записывается **адрес его вершины**. В качестве адреса выбран самый верхний адрес ОЗУ. Для обозначения этого адреса в данной версии Ассемблера существует специальная константа с именем RAMEND. Значение этой константы определяется в файле описаний (в нашем случае в файле tn2313def.inc). Для микроконтроллера ATtiny2313 константа RAMEND равна 0xDF.

Одной строкой записать константу в регистр стека невозможно, так как в системе команд микроконтроллеров AVR отсутствует подобная команда. Отсутствующую команду мы заменяем двумя другими. И тут нам пригодится **регистр temp**. Он послужит в данном случае передаточным звеном. Сначала **константа RAMEND** помещается в регистр temp (**строка 6**), а затем уже содержимое temp помещается в **регистр SPL** (**строка 7**).

В **строках 8—12** производится **настройка портов ввода-вывода**. Ранее мы уже договорились, что порт PD у нас будет работать на ввод, а порт PB — на вывод. Для выбора нужного направления передачи информации запишем управляющие коды в соответствующие регистры DDRx. Во все разряды **регистра DDRD** запишем нули (настройка порта PD на ввод), а во все разряды регистра DDRB запишем единицы (настройка порта PB на вывод). Кроме того, нам нужно включить внутренние нагрузочные резисторы порта PD. Для этого мы запишем единицы (то есть число 0xFF) во все разряды **регистра PORTD**. И, наконец, в момент старта программы желательно погасить светодиод. Для этого мы запишем единицы в разряды порта PB.

Все описанные выше действия по настройке порта также выполняются с использованием промежуточного **регистра temp**. Сначала в него помещается ноль (**строка 8**). Ноль записывается только в регистр DDRD (**строка 9**). Затем в регистр temp помещается число 0xFF (**строка 10**). Это число по очереди записывается в регистры DDRB, PORTB, PORTD (**строки 11, 12, 13**).

Строки 14 и 15 включены в программу для перестраховки. Дело в том, что встроенный компаратор микроконтроллера после системного сброса остается включен. И хотя прерывания при этом отключены и срабатывание компаратора не может повлиять на работу нашей про-

граммы, мы все же отключим компаратор. Именно это и делается в **строках 14** и **15**.

Здесь уже знакомым нам способом с использованием регистра `temp` производится запись константы 0x80 в регистр ACSR. **Регистр ACSR предназначен** для управления режимами работы компаратора, а константа 0x80, записанная в этот регистр, отключает компаратор.

Настройкой компаратора заканчивается подготовительная часть программы. Подготовительная часть занимает **строки 1—15** и выполняется всего один раз после включения питания или после системного сброса. **Строки 16—18** составляет **основной цикл программы**.

Основной цикл *— это часть программы, которая повторяется многократно и выполняет все основные действия.*

В нашем случае, согласно алгоритму, действия программы состоят в том, чтобы прочитать состояние кнопки и перенести его на светодиод. Есть много способов перенести содержимое младшего разряда порта PD в младший разряд порта PB. В нашем случае реализован самый простой вариант. Мы просто переносим одновременно все разряды. Для этого достаточно двух операторов.

Первый из них читает содержимое порта PD и запоминает это содержимое в **регистре `temp`** (**строка 16**). Следующий оператор записывает это число в **порт PB** (**строка 17**). Завершает основной цикл программы **оператор безусловного перехода** (**строка 18**). Он передает управление **по метке `main`**.

В результате три оператора, составляющие тело цикла, повторяются бесконечно. Благодаря этому бесконечному циклу все изменения порта PD тут же попадают в порт PB. По этой причине, если кнопка S1 не нажата, логическая единица со входа PD0 за один проход цикла передается на выход PB0. И светодиод не светится. При нажатии кнопки S1 логический ноль со входа PD0 поступает на выход PB0, и светодиод загорается.

Эта же самая программа без каких-либо изменений может обслуживать до семи кнопок и такое же количество светодиодов. Дополнительные кнопки подключаются к линиям PD1—PD6, а дополнительные светодиоды (каждый со своим токоограничивающим резистором) — к выходам PB1—PB7. При этом каждая кнопка будет управлять своим собственным светодиодом. Такое стало возможным потому, что все выводы каждого из двух портов мы настроили одинаково (смотри **строки 8—13**).

Программа на языке СИ

Для создания программ на языке СИ мы будем использовать **программную среду CodeVisionAVR**. Это среда специально предназна-

чена для разработки программ на языке СИ для микроконтроллеров серии AVR. Среда CodeVisionAVR не имеет своего отладчика, но позволяет отлаживать программы, используя возможности системы AVR Studio.

Отличительной **особенностью** системы CodeVisionAVR является наличие мастера-построителя программы. **Мастер-построитель** облегчает работу программисту. Он избавляет от необходимости листать справочник и выискивать информацию о том, какой регистр за что отвечает и какие коды нужно в него записать. **Результат** работы мастера — это заготовка будущей программы, в которую включены все команды предварительной настройки, а также заготовки всех процедур языка СИ. Поэтому давайте сначала научимся работать с мастером, создадим заготовку нашей программы на языке СИ, а затем разберем подробнее все ее элементы. И уже после этого создадим из заготовки готовую программу.

Для того, чтобы понять работу мастера, нам необходимо прояснить **несколько новых понятий**. Программа CodeVisionAVR, как и любая современная среда программирования, работает не просто с текстом программы, а с так называемым **Проектом.**

Задача программиста — лишь написать текст программы, для которого в проекте отводится отдельный файл с расширением «c». **Например**, «proba.c».

При помощи мастера мы будем создавать новый проект. В дальнейшем мы еще рассмотрим подробно процесс установки и работу с программной средой CodeVisionAVR. Сейчас же считаем, что она установлена и запущена.

Проект, кроме текста программы, содержит ряд вспомогательных файлов, содержащих дополнительную информацию, такую, как тип используемого процессора, тактовую частоту кварца и т. п. Эта информация необходима в процессе трансляции и отладки программ. За исключением текста программы, все остальные файлы проекта создаются и изменяются автоматически.

Для создания нового проекта выберем в меню «File» пункт «New». Откроется небольшой диалог, в котором вы должны выбрать тип создаваемого файла. Предлагается два варианта:

- «Source» (Исходный текст программы);
- «Project» (Проект).

Выбираем Project и нажимаем «Ok». Появляется окно с вопросом «You are about to create a new project. Do you want to use the CodeWizardAVR?». В переводе на русский это означает: «Вы создаете новый проект. Будете ли вы использовать построитель CodeWizardAVR?». Мы договорились, что будем, поэтому выбираем «Yes», выберите нужный тип чипа (AT90,

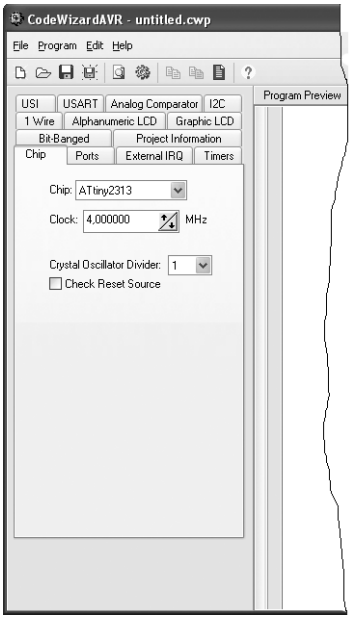


Рис. 5.3. *Управляющая панель окна мастера CodeVisionAVR*

ATiny, ATmega, FPSLIC), после чего открывается окно построителя (см. **рис. 5.3**). Как видите, это окно имеет множество вкладок, каждая из которых содержит элементы выбора режимов.

Все эти управляющие элементы позволяют **настроить параметры создаваемой заготовки программы**. Сразу после запуска мастера все параметры принимают значения по умолчанию (все внутренние устройства выключены, а все порты ввода-вывода настроены на ввод, внутренние нагрузочные резисторы отключены). Это соответствует начальному состоянию микроконтроллера непосредственно после системного сброса.

Пройдемся немного по вкладкам **мастера** и выберем необходимые нам параметры. Те же параметры, которые нам не нужны, мы трогать пока не будем (оставим значения по умолчанию).

Первая вкладка называется «Chip». На этой вкладке мы можем выбрать общие параметры проекта. Используя выпадающий список «Chip», выберем тип микроконтроллера. Для этого щелкаем мышью по окошку и в выпавшем списке выбираем ATtiny2313.

При помощи поля «Clock» выбираем частоту кварцевого резонатора. В нашем случае она должна быть равна 4 МГц. При помощи поля «Crystal Oscillator Divider» выбирается **коэффициент деления тактового генератора**. Этот параметр требует пояснений. Дело в том, что выбранный нами микроконтроллер имеет систему предварительного деления тактовых импульсов. Если частота тактового генератора нас не устраивает, мы можем поделить ее, и микроконтроллер будет работать на другой, более низкой частоте. Коэффициент деления может изменяться от 1 до 256. Мы выберем его равным единице (без деления). То есть оставим значение по умолчанию.

Без изменений оставим флажок «Check Reset Source» (Проверка источника сигнала сброса). Не будем углубляться в его назначение. Достаточно будет понять, что включение данного флажка добавляет к создаваемой программе процедуру, связанную с определением источника сигнала системного сброса.

Покончив с общими настройками, перейдем к **вкладке (Порты)**. Эта вкладка позволяет настроить все имеющиеся порты ввода-вывода. На

вкладке «Ports» мы видим еще три вкладки поменьше (см. **рис. 5.4**). По одной вкладке для каждого порта. Как уже говорилось выше, порт PA в данной схеме мы не применяем. Поэтому сразу выбираем вкладку «Port B».

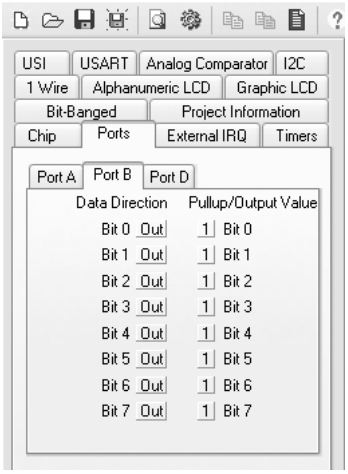


Рис. 5.4. *Настройка порта PB*

На вкладке мы видим два столбца с параметрами. Столбец **«Data direction» (Направление передачи данных)** позволяет настроить каждую линию порта либо на ввод, либо на вывод. По умолчанию каждый параметр имеет значение «In» (вход). Поменяем для каждого разряда это значение на «Out» (Выход).

Для того, чтобы поменять значение разряда, достаточно щелкнуть по полю с надписью «In» один раз мышью, и параметр сменится на «Out». Повторный щелчок заставит вернуться к «In». Каждое поле столбца «Data direction» определяет, какое значение будет присвоено соответствующему разряду регистра DDRB в нашей будущей программе.

Второй столбец на той же вкладке **называется «Pullup / Output Value» (Включение нагрузки / Выходное значение)**. Этот столбец определяет, какое значение будет присвоено каждому из разрядов регистра PORTB. В нашем случае порт PB работает на вывод. Поэтому содержимое регистра PORTB определяет выходное значение всех разрядов порта. По умолчанию все они имеют значение «0». Но по условиям нашей задачи они должны быть равны единице (при старте программы светодиод должен быть отключен). Поэтому изменим все нули на единицы. Для этого также достаточно простого щелчка мыши. В результате всех операций вкладка «Port B» будет выглядеть так, как это показано на **рис. 5.4**.

Теперь перейдем к **настройке последнего порта**. Для этого выберем вкладку «Potr D». На вкладке мы увидим такие же органы управления, как на вкладке «Port B» (см. **рис. 5.5**). По условиям задачи порт PD микроконтроллера должен работать на ввод. Поэтому состояние элементов первого столбца мы не меняем.

Однако не забывайте, что нам нужно включить внутренние нагрузочные резисторы для каждого из входов. Для этого изменим значения элементов второго столбца. Так как порт PD работает в режиме ввода, элементы в столбце «Pullup / Output Value» принимают значение «T» или «P».

«T» (Terminate) означает отсутствие внутренней нагрузки, а **«P» (Pull-up)** означает: нагрузка включена. Включим нагрузку для каждого

разряда порта PD, изменив при помощи мыши значение поля с «T» на «P». В результате элементы управления будут выглядеть так, как показано на **рис. 5.5**.

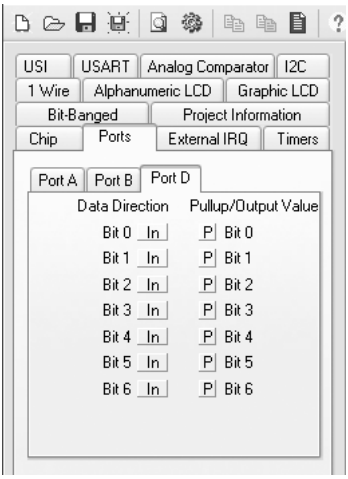


Рис. 5.5. *Настройка порта PD*

Для данной простейшей программы настройки можно считать оконченными. Остальные системы микроконтроллера нам пока не нужны. Оставим их настройки без изменений.

Воспользуемся еще одним **полезным свойством мастера программ**. Откроем вкладку «Project Information» (см. **рис. 5.6**). В поле «Project Name» вы можете занести название вашего проекта. **Поле «Version»** предназначено для номера версии. В **поле «Date»** помещают дату разработки программы. В **полях «Author»** и **«Company»** помещается, соответственно, имя автора и название компании. В **поле «Comments:»** можно поместить любые необходимые комментарии. Вся эта информация будет автоматически помещена в заголовок будущей программы.

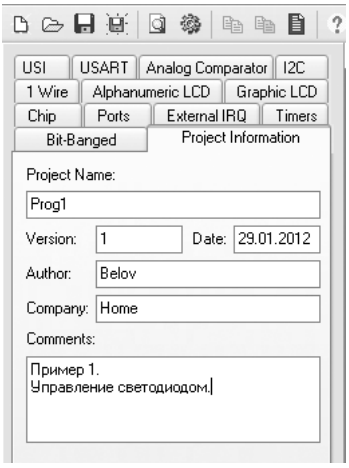


Рис. 5.6. *Занесение информации для заголовка программы*

После того, как все параметры выставлены, приступаем непосредственно **к процессу генерации программы**. Для этого выбираем в меню «File» нашего мастера пункт «Generate, Save and Exit», как это показано на **рис. 5.7**. Процесс генерации начнется с запроса имени файла будущей программы. Для этого откроется стандартный диалог сохранения файла, в котором вы сначала должны выбрать каталог, а затем указать имя файла программы. Что касается каталога, то вам нужно самостоятельно выбрать каталог, где будет размещен весь ваш проект.

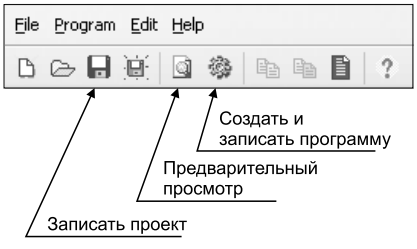


Рис. 5.7. *Запуск процесса генерации программы*

Рекомендуется для каждого проекта создавать свой отдельный каталог. В нашем случае самое простое — назвать каталог именем «Prog1». А вернее, выбрать что-то вроде

«C:\AVR\C\Prog1\». Если каталог еще не создан, то вы можете создать его прямо в диалоге создания файла. Файлу рекомендую присвоить то же самое имя: «Prog1».

Однако вы можете выбрать имя по своему усмотрению. Для завершения процесса создания файла нажмите кнопку «Сохранить». В результате на ваш жесткий диск запишется файл «Prog1.c», в который будет помещен созданный построителем текст заготовки вашей программы. Расширение «.c» набирать не нужно. Оно присваивается автоматически. Это стандартное расширение для программ на языке СИ.

Однако процесс генерации проекта на этом не заканчивается. Сразу после того, как файл программы будет записан, откроется новый диалог, который запросит имя для файла проекта. **Файл проекта** предназначен для хранения параметров конкретного проекта.

Кроме типа используемой микросхемы и частоты задающего генератора, файл проекта используется для хранения вспомогательной информации, такой как наличие и расположение точек останова, закладок и т. д.

Подробнее о закладках и точках останова мы говорили, когда изучали работу отладчика. В качестве имени файла проекта удобнее всего использовать то же самое имя, что и для текста программы. То есть «Prog1». Файлу проекта автоматически присваивается расширение «.prj».

Когда файл проекта будет записан, диалог записи файла откроется в третий раз. Теперь нам предложат **выбрать имя для файла данных построителя**. В этот файл будут записаны текущие значения всех параметров, мастера. То есть значения всех управляющих элементов со всех его вкладок. И те, которые мы изменили в процессе настройки, и те, которые остались без изменений (по умолчанию).

Эти данные могут понадобиться, если потребуется заново пересоздать проект. Используя файл данных построителя, вы можете в любой момент восстановить значения всех его элементов, немного подкорректировать их и создать новый проект. Файлу данных построителя присвоим то же самое имя, что обоим предыдущим («Prog1»). Новый файл получит расширение «.cwp» (Code Wizard Project).

После того, как и этот файл будет записан, процесс генерации проекта завершается. На экране появляются два новых окна. В одном окне открывается **содержимое файла «Prog1.c»**. Втрое окно — это **файл комментариев**. Сюда вы можете записать, а затем сохранить на диске любые замечания по вашей программе. В дальнейшем они всегда будут у вас перед глазами.

Посмотрим теперь, что же сформировал наш построитель. Текст программы, полученной описанным выше образом, дополненный русскоязычными комментариями, приведен в **листинге 5.2**.

Все русскоязычные комментарии дублируют соответствующие англоязычные комментарии, автоматически созданные в процессе генерации программы. Кроме новых комментариев, в программу добавлена только одна дополнительная строка (**строка 32**). Именно она превращает созданную построителем заготовку в законченную программу. **Итак, мы получили программу на языке СИ.**

Работа программы, написанной на языке СИ

Теперь наша задача — разобраться, как программа работает. Именно этим мы сейчас и займемся. Но сначала небольшое введение в новый для нас язык СИ.

Программа на языке **СИ, в отличие от Ассемблера,** гораздо более абстрагирована от системы команд микроконтроллера. Основные операторы языка СИ вовсе не привязаны к командам микроконтроллера. Для реализации всего одной команды на языке СИ на самом деле используется не одна, а несколько команд микроконтроллера. Иногда даже целая небольшая программа.

В результате облегчается труд программиста, так как он теперь работает с более крупными категориями. Ему не приходится вдаваться в мелкие подробности, и он может сосредоточиться на главном. Язык СИ так же, как и другие языки программирования, состоит из команд. Для записи каждой команды СИ использует свои операторы и псевдооператоры. Но форма написания команд в программе приближена к форме, принятой в математике. Сейчас вы в этом убедитесь сами.

В языке СИ для хранения различных данных используются **переменные**. Понятие «переменная» в языке СИ аналогично одноименному математическому понятию. Каждая переменная имеет свое имя, и ей можно присваивать различные значения. Используя переменные, можно строить различные выражения. Каждое выражения представляет собой одну или несколько переменных и числовых констант, связанных арифметическими и (или) логическими операциями. **Например:**

- **a*b** — произведение переменных a и b (символ * означает умножение);
- **k1/2** — переменная k1, деленная на два (символ «/» означает деление);
- **massa1 + massa2** — сумма двух переменных (massa1 и massa2);
- **tkon << 2** — циклический сдвиг содержимого переменной tkon на 2 разряда влево;
- **dat & mask** — логическое умножение (операция «И») между двумя переменными (dat и mask).

Приведенные примеры — это простейшие выражения, каждое из которых состоит всего их двух членов. Язык СИ допускает выражения практически любой сложности.

В языке СИ **переменные делятся на типы**. Переменная каждого типа может принимать значения из одного определенного диапазона (см. **табл. 5.1**). **Например**:

- переменная типа **char** — это только целые числа;
- переменная типа **float** — вещественные числа (десятичная дробь) и т. д.

Использование переменных нескольких фиксированных типов — это отличительная особенность любого языка высокого уровня. Разные версии языка СИ поддерживают различное количество типов переменных. Версия СИ, используемая в CodeVisionAVR, поддерживает тринадцать типов переменных (см. **табл. 5.1**).

В языке СИ любая переменная, прежде чем она будет использована, должна быть описана. При **описании** задается ее тип. В дальнейшем диапазон принимаемых значений должен строго соответствовать выбранному типу переменной. Описание переменной и задание ее типа необходимы потому, что оттранслированная с языка СИ программа выделяет для хранения значений каждой переменной определенные ресурсы памяти.

Типы данных языка СИ *Таблица 5.1*

Название	Количество бит	Значение
bit	1	0 или 1
char	8	–128 – 127
unsigned char	8	0 – 255
signed char	8	–128 – 127
int	16	–32768 – 32767
short int	16	–32768 – 32767
unsigned int	16	0 – 65535
signed int	16	–32768 – 32767
long int	32	–2147483648 – 2147483647
unsigned long int	32	0 – 4294967295
signed long int	32	–2147483648 – 2147483647
float	32	±1.175e–38 – ±3.402e38
double	32	±1.175e–38 – ±3.402e38

Листинг 5.2

```
/*****************************************************
This program was produced by the
CodeWizardAVR V1.24.4 Standard
Automatic Program Generator
© Copyright 1998-2004 Pavel Haiduc, HP InfoTech s.r.l
http://www.hpinfotech.com
e-mail:office@hpinfotech.com

Project : Prog1
Version : 1
Date    : 06.01.2006
Author  : Belov
Company : Home
Comments:
Пример 1
```

```
      Управление светодиодом


      Chip type           : ATtiny2313
      Clock frequency     : 4,000000 MHz
      Memory model        : Tiny
      External SRAM size  : 0
      Data Stack size     : 32
      *****************************************************/

1     #include <tiny2313.h>

      // Declare your global variables here (определение ваших глобальных переменных)

2     void main(void)
      {
      // Crystal Oscillator division factor: 1
      // Коэффициент деления частоты системного генератора: 1
3     CLKPR=0x80;
4     CLKPR=0x00;

      // Input/Output Ports initialization (Инициализация портов ввода-вывода)
      // Port A initialization  (Инициализация порта A)
      // Func2=In Func1=In Func0=In
      // State2=T State1=T State0=T
5     PORTA=0x00;
6     DDRA=0x00;

      // Port B initialization (Инициализация порта B)
      // Func7=Out Func6=Out Func5=Out Func4=Out Func3=Out Func2=Out Func1=Out Func0=Out
      // State7=1 State6=1 State5=1 State4=1 State3=1 State2=1 State1=1 State0=1
7     PORTB=0xFF;
8     DDRB=0xFF;

      // Port D initialization (Инициализация порта D)
      // Func6=In Func5=In Func4=In Func3=In Func2=In Func1=In Func0=In
      // State6=P State5=P State4=P State3=P State2=P State1=P State0=P
9     PORTD=0x7F;
10    DDRD=0x00;

      // Timer/Counter 0 initialization (Инициализация таймера/счетчика 0)
      // Clock source: System Clock  (Источник сигнала: системный генератор)
      // Clock value: Timer 0 Stopped  (Значение частоты: Таймер 0 остановлен)
      // Mode: Normal top=FFh   (Режим Normal  макс. значение FFH)
      // OC0A output: Disconnected   (Выход OC0A отключен)
      // OC0B output: Disconnected   (Выход OC0B отключен)
11    TCCR0A=0x00;
12    TCCR0B=0x00;
13    TCNT0=0x00;
14    OCR0A=0x00;
15    OCR0B=0x00;

      // Timer/Counter 1 initialization (Инициализация таймера/счетчика 1)
      // Clock source: System Clock   (Источник сигнала: системный генератор)
      // Clock value: Timer 1 Stopped  (Значение частоты: Таймер 1 остановлен)
      // Mode: Normal top=FFFFh   (Режим Normal  макс. значение FFFFH)
      // OC1A output: Discon.   (Выход OC0A отключен)
      // OC1B output: Discon.   (Выход OC0B отключен)
      // Noise Canceler: Off
      // Input Capture on Falling Edge
16    TCCR1A=0x00;
17    TCCR1B=0x00;
18    TCNT1H=0x00;
19    TCNT1L=0x00;
20    ICR1H=0x00;
21    ICR1L=0x00;
22    OCR1H=0x00;
23    OCR1L=0x00;
24    OCR1BH=0x00;
25    OCR1BL=0x00;

      // External Interrupt(s) initialization (Инициализация внешних прерываний)
      // INT0: Off   (Прерывание INT0 выключено)
      // INT1: Off   (Прерывание INT1 выключено)
26    GIMSK=0x00;
27    MCUCR=0x00;

      // Timer(s)/Counter(s) Interrupt(s) initialization
      // (Инициализация прерываний от таймеров)
28    TIMSK=0x00;

      // Universal Serial Interface initialization
      // (Инициализация универсального последовательного интерфейса)
      // Mode: Disabled   (Режим:  Выключен)
      // Clock source: Register & Counter=no clk.
```

```
      // USI Counter Overflow Interrupt: Off
29    USICR=0x00;

      // Analog Comparator initialization (Инициализация аналогового компаратора)
      // Analog Comparator: Off   (Аналоговый компаратор:  Выключен)
      // Analog Comparator Input Capture by Timer/Counter 1: Off
30    ACSR=0x80;

31    while (1)
            {
                  // Place your code here  (Пожалуйста вставьте ваш код)
32                PORTB=PIND;
            };
      }
```

Это могут быть ячейки ОЗУ, регистры общего назначения или даже ячейки EEPROM или Flash-памяти (памяти программ). В зависимости от заданного типа, выделяется различное количество ячеек для каждой конкретной переменной. Косвенно о количестве выделяемых ячеек можно судить по содержимому графы «Количество бит» **табл. 5.1**. Описывая переменную, мы сообщаем транслятору, сколько ячеек выделять и как затем интерпретировать их содержимое. Посмотрим, как выглядит строка описания переменной в программе. Она представляет собой запись следующего вида:

```
Тип Имя;
```

где «Тип» — это тип переменной, а «Имя» — ее имя.

Имя переменной выбирает программист. Принцип формирования имен в языке СИ не отличается от подобного принципа в языке Ассемблер. Допускается использование только латинских букв, цифр и символа подчеркивания. Начинаться имя должно с буквы или символа подчеркивания.

Кроме арифметических и логических выражений, язык СИ использует **функции**.

Правда, в отличие от математических функций, функции языка СИ не всегда имеют входные значения и даже не обязательно возвращают результат. Далее на конкретных примерах мы увидим, как и почему это происходит.

> ***Функция в языке СИ*** *— это аналог соответствующего математического понятия. Функция получает одно или несколько значений в качестве параметров, производит над ними некие вычисления и возвращает результат.*

Вообще, роль функций в языке СИ огромная. Программа на языке СИ просто-напросто состоит из одной или нескольких функций. Каждая функция имеет свое имя и описание. По имени производится обращение к функции. Описание определяет выполняемые функцией действия и преобразования. Вот как выглядит описание функции в программе СИ:

```
тип Name (список параметров)
    {
    тело функции
    }
```

Здесь **Name** — это **имя функции**. Имя для функции выбирается по тем же правилам, что и для переменной. При описании функции перед ее именем положено указать тип возвращаемого значения. Это необходимо транслятору, так как для возвращаемого значения он тоже резервирует ячейки.

Если перед именем функции вместо типа возвращаемого значения записать слово `void`, то это будет означать, что данная функция не возвращает никаких значений. В круглых скобках после имени функции записывается список передаваемых в нее параметров.

Функция может иметь любое количество параметров. Если параметров два и более, то они записываются через запятую. Перед именем каждого параметра также должен быть указан его **тип**. Если у функции нет параметров, то в скобках вместо списка параметров должно стоять **слово `void`**. В фигурных скобках размещается **тело функции**.

> ***Тело функции*** *— это набор операторов на языке СИ, выполняющих некие действия. В конце каждого оператора ставится точка с запятой. Если функция небольшая, то ее можно записать в одну строку.*

В этом случае операторы, составляющие тело функции, разделяет только точка с запятой. Вот **пример** такой записи:

```
тип Name (список параметров)
{тело функции}
```

Любая программа на языке СИ должна обязательно содержать **одну главную функцию**. Главная функция должна иметь **имя `main`**. Выполнение программы всегда начинается с выполнения функции `main`. В нашем случае (см. **листинг 5.2**) описание функции `main` начинается со **строки 2** и заканчивается в конце программы. Функция `main` в данной версии языка СИ никогда не имеет параметров и никогда не возвращает никакого значения.

Тело функции, кроме команд, может содержать описание переменных. Все переменные должны быть описаны в самом начале функции, до первого оператора. Такие переменные могут быть использованы только в той функции, в начале которой они описаны. Вне этой функции данной переменной как бы не существует.

Если вы объявите переменную в одной функции, а примените ее в другой, то транслятор выдаст **сообщение об ошибке**. Это дает возможность объявлять внутри разных функций переменные с одинаковыми именами и использовать их независимо друг от друга.

Глобальная переменная объявляется не внутри функций, а в начале программы, еще до описания самой первой функции. Не спешите без необходимости делать переменную глобальной. Если программа достаточно большая, то можно случайно присвоить двум разным переменным одно и то же имя, что приведет к ошибке. Такую ошибку очень трудно найти.

*Переменные, объявленные внутри функций, называются **локальными**. При написании программ иногда необходим другой порядок использования переменных. Иногда нужны переменные, которые могут работать сразу со всеми функциями. Такие переменные называются **глобальными переменными**.*

Для того, чтобы все вышесказанное было понятнее, обратимся к конкретному примеру — программе (**листинг 5.2**). А начнем мы изучение этой программы с описания нам пока неизвестных используемых там команд.

include

Оператор присоединения внешних файлов. Данный оператор выполняет точно такую же роль, что и аналогичный оператор в языке Ассемблер. В **строке 1** программы (**листинг 5.2**) этот оператор присоединяет к основному тексту программы стандартный текст описаний для микроконтроллера ATtiny2313.

while

Оператор цикла. Форма написания команды `while` очень похожа на форму описания функции. В общем случае команда while выглядит следующим образом:

```
while (условие)
    {
    тело цикла
    };
```

Перевод английского слова `while` — «пока». Эта команда организует цикл, многократно повторяя тело цикла до тех пор, пока выполняется «условие», то есть пока выражение в скобках является истинным. В языке СИ принято считать, что выражение истинно, если оно не равно нулю, и ложно, если равно.

***Тело цикла** — это ряд любых операторов языка СИ. Как и любая другая команда, вся конструкция while должна заканчиваться символом «точка с запятой».*

В программе на **листинге 5.2** оператор `while` вы можете видеть в **строке 31**. В качестве условия в этом операторе используется про-

сто число 1. Так как 1 — не ноль, то такое условие всегда истинно. Такой прием позволяет создавать бесконечные циклы. Это значит, что цикл, начинающийся в **строке 31**, будет выполняться бесконечно. Тело цикла составляет единственная команда (**строка 32**).

Комментарии

В программе на языке СИ так же, как и в Ассемблере, широко используются комментарии. В языке СИ принято два способа написания комментариев. **Первый способ** — использование специальных обозначений начала и конца комментария. Начало комментария помечается парой символов /*, а конец комментария символами */. Это выглядит следующим образом:

```
/* Комментарий */
```

Причем комментарий, выделенный таким образом, может занимать не одну, а несколько строк. В **листинге 5.2** шапка программы выполнена в виде комментария, который записан именно таким образом. **Второй способ** написания комментария — двойная наклонная черта (//).

В этом случае комментарий начинается сразу после двойной наклонной черты и заканчивается в конце текущей строки. В **листинге 5.2** такой способ применяется по всему тексту программы.

Описание программы (листинг 5.2)

Как уже говорилось, текст программы, который вы видите в **листинге 5.2**, в основном сформирован автоматически. Большую часть программы занимает **функция `main`**. Она начинается в **строке 2** и заканчивается в конце программы. Вся программа снабжена подробными комментариями, которые также сформированы автоматически.

Исключения составляют все русскоязычные комментарии, которые я добавил вручную, и одна **строка** в конце программы (**строка 32**). Начинается программа с заголовка. В начале заголовка мастер поместил информацию о том, что программа создана при помощи CodeWizardAVR. Далее в заголовок включен блок информации из вкладки «Project Information». Эту информацию мы с вами набирали собственноручно. Далее заголовок сообщает тип процессора, его тактовую частоту, модель памяти (Tiny — означает малая модель), размер используемой внешней памяти и размер стека.

В **строке 1** находится **команда include**, присоединяющая файл описаний. После команды include мастер поместил сообщение для программиста. Сообщение предупреждает о том, что именно в этом месте программисту нужно поместить описание всех глобальных

переменных (если, конечно, они вам понадобятся). В данном конкретном случае глобальные переменные нам не нужны. Поэтому мы добавлять их не будем.

Теперь перейдем к **функции `main`**. Функция `main` содержит в себе набор команд, настройки системы (**строки 3—30**) и заготовку главного цикла программы (**строка 31**).

В программе на Ассемблере (**листинг 5.1**) мы тоже производили подобную настройку. Однако там мы ограничились инициализацией портов ввода-вывода и отключением компаратора. Состояние всех остальных служебных регистров микроконтроллера программа на Ассемблере не меняла. То есть оставляла значения по умолчанию.

> ***Настройка системы*** *— это запись требуемых значений во все управляющие регистры микроконтроллера.*

На языке СИ можно было бы поступить точно так же. Но мастер-построитель программы поступает по-другому. Он присваивает значения всем без исключения служебным регистрам. И тем, значения которых должны отличаться от значений по умолчанию, и тем, значения которых не изменяются.

В последнем случае регистру присваивается то же самое значение, какое он и так имеет после системного сброса. Такие, на первый взгляд, избыточные действия имеют свой смысл. Они гарантируют правильную работу программы в том случае, если в результате ошибки в программе управление будет передано на ее начало.

Лишние команды при желании можно убрать. В нашем случае достаточно оставить лишь команды инициализации портов (**строки 7, 8** для порта PB и **строки 9, 10** для порта PD). А также команду инициализации компаратора (**строка 30**).

Теперь посмотрим, как же **происходит присвоение значений**. В **строке 3** регистру `CLKPR` присваивается значение `0x80`. Для присвоения значения используется хорошо знакомый нам символ «=» (равно). В языке СИ такой символ называется **оператором присвоения**. Таким же самым образом присваиваются значения и всем остальным регистрам.

Что касается настройки портов PB и PD, то в **строках 7, 8** регистрам PORTB и DDRB присваивается значение 0x7F. А в **строках 9, 10** в регистр PORTD записывается 0x7F, а в регистр DDRD — ноль. Если вы помните, те же значения мы присваивали тем же самым регистрам в программе на Ассемблере. Точнее, небольшое отличие все же есть.

В программе на Ассемблере в регистр PORTD мы записывали 0xFF (в двоичном варианте 0b11111111). В программе на СИ в тот же регистр мы записываем 0x7F (0b01111111). Эти два числа отличаются лишь зна-

чением старшего бита. Но для данного порта это несущественно, так как его старший разряд незадействован. Поэтому в каждой программе мы делаем так, как это удобнее.

После инициализации всех регистров начинается основной цикл программы (**строка 31**). **Основной цикл** — это обязательный элемент любой программы для микроконтроллера. Поэтому мастер всегда создает заготовку этого цикла. То есть создает цикл, тело которого пока не содержит не одной команды.

В том месте, где программист должен расположить команды, составляющие **тело этого цикла**, мастер помещает специальное сообщение, приглашающее вставить туда код программы. Оно гласит: Place your code here (Пожалуйста, вставьте ваш код). Мы последовали этому приглашению и вставили требуемый код (**строка 32**). В нашем случае он состоит всего из одной команды. Эта команда присваивает регистру PORTB значение регистра PORTD.

Наша программа на языке СИ готова. Выполняясь многократно в бесконечном цикле, команда присвоения постоянно переносит содержимое порта PD в порт PB, реализуя тем самым наш алгоритм.

5.3. Переключающийся светодиод

Постановка задачи

Как уже говорилось, предыдущая задача настолько проста, что решение ее средствами микропроцессорной техники лишено всякого смысла. Усложним немного задачу. **Заставим переключаться светодиод при нажатии кнопки**.

Новая задача может звучать так:

«Разработать устройство управления одним светодиодным индикатором при помощи одной кнопки. При каждом нажатии кнопки светодиод должен поочередно включаться и отключаться. При первом нажатии кнопки светодиод должен включиться, при следующем отключиться и т. д.».

Вы можете сказать, что и эта новая задача легко решаема при помощи простейшего D-триггера. Однако все же рассмотрим, как ее можно решить при помощи микроконтроллера.

Принципиальная схема

Так как для новой задачи, как и для предыдущей, нам необходима всего одна кнопка и всего один светодиод, то придумывать новую схему не имеет смысла. Применим для второй задачи уже знакомую нам схему, показанную на **рис. 5.2**.

Алгоритм

Алгоритм задачи номер два начинается так же, как алгоритм нашей первой задачи. То есть с набора команд, выполняющих инициализацию системы. Так как схема и принцип работы портов ввода-вывода не изменились, то алгоритм инициализации системы будет полностью повторять соответствующий алгоритм из предыдущего примера.

После команд инициализации начинается основной цикл программы. Однако действия, выполняемые основным циклом, будут немного другими. Попробуем, как и в предыдущем случае, описать эти действия словами.

1. Прочитать состояние младшего разряда порта PD (PD.0).
2. Если значение этого разряда равно единице, перейти к началу цикла.
3. Если значение разряда PD.0 равно нулю, изменить состояние выхода PB.0 на противоположное.
4. Перейти к началу цикла.

Итак, мы описали алгоритм словами. Причем это довольно общее описание. Реальный алгоритм немного сложнее. Словесное описание алгоритма не всегда удобно. Гораздо нагляднее графический способ описания алгоритма. На **рис. 5.3** алгоритм нашей работы будущей программы изображен **в графическом виде**.

Такой способ отображения информации называется **графом**. Прямоугольниками обозначаются различные действия, выполняемые программой. Суть выполняемого действия вписывается внутрь такого прямоугольника.

Допускается объединять несколько операций в один блок и обозначать одним прямоугольником. Последовательность выполнения действий показывается стрелками. Ромбик реализует разветвление программы. Он представляет собой операцию выбора. Условие выбора записывается внутри ромбика. Если условие истинно, то дальнейшее выполнение программы продолжится по пути, обозначенному словом «Да».

Если условие не выполнено, то программа пойдет по другому пути, обозначенному стрелкой с надписью «Нет». Прямоугольником со скругленными боками принято обозначать начало и конец алгоритма. В нашем случае программа не имеет конца. Основной цикл программы является бесконечным циклом.

Рассмотрим подробнее алгоритм, изображенный на **рис. 5.8**. Как видно из рисунка, сразу после старта программы выполняется установка вершины стека. Следующее действие — это программирование портов ввода-вывода. Затем начинается главный цикл программы (обведен пунктирной линией). Внутри цикла ход выполнения программы разветвляется.

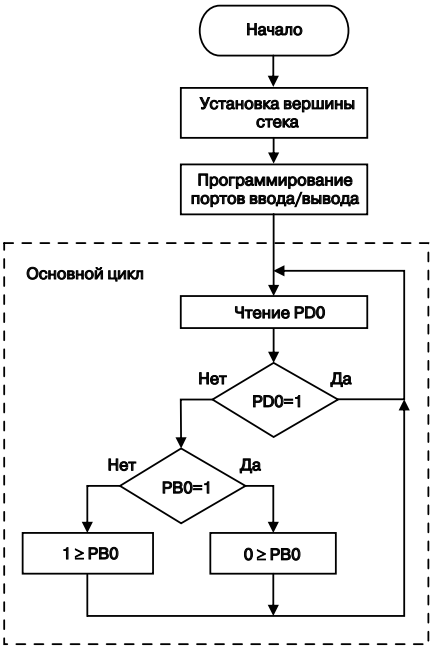


Рис. 5.8. *Алгоритм программы с переключающимся светодиодом*

Первой операцией цикла является проверка состояния младшего разряда порта PD (PD0). Программа сначала читает состояние этой линии, а затем выполняет операцию сравнения. В процессе сравнения значение разряда PD0 проверяется на равенство единице. Если условие выполняется, программа переходит к началу цикла (по стрелке «Да»).

Если условие не выполняется (PD0 не равен единице), выполнение программы продолжается по стрелке «Нет», где выполняется еще одна операции сравнения. Это сравнение является частью процедуры переключения светодиода. Для того, чтобы переключить светодиод, мы должны проверить его текущее состояние и перевести его в противоположное.

Как вы помните, светодиодом управляет младший разряд порта PB (PB0). Поэтому именно его мы будем проверять и изменять. **Работа алгоритма** переключения светодиода предельно проста. Сначала оператор сравнения проверяет разряд PB0 на равенство единице. Если результат проверки — истина (PB0=1), то разряд сбрасывается в ноль (0 => PB0). Если ложно, устанавливается в единицу (1 => PB0). **Сочетание символов «=>»** означает **операцию присвоения**. Такое обозначение иногда используется в программировании при написании алгоритмов. После переключения светодиода управление передается на начало главного цикла.

Итак, наш алгоритм готов, и можно **приступать к составлению программы**. Но не торопитесь. Все не так просто. Приведенный выше алгоритм хорош лишь для теоретического изучения приемов программирования. На практике же он работать не будет.

Дело в том, что микроконтроллер работает с такой скоростью, что за время, пока человек будет удерживать кнопку в нажатом состоянии, главный цикл программы успеет выполниться многократно (до сотни раз). Это произойдет даже в том случае, если человек постарается нажать и отпустить кнопку очень быстро. При каждом проходе главного цикла программа обнаружит факт нажатия кнопки и переключит светодиод.

В результате работа нашего устройства будет выглядеть следующим образом. Как только кнопка будет нажата, светодиод начнет быстро переключаться. Настолько быстро, что вы даже не увидите, как он мерцает. Это будет выглядеть как свечение в полнакала.

В момент отпускания кнопки процесс переключения остановится, и светодиод окажется в одном из своих состояний (засветится или потухнет). В каком именно состоянии он останется, зависит от момента отпускания кнопки. А это случайная величина. Как видите, описанный выше алгоритм не позволяет создать устройство, соответствующее нашему техническому заданию.

Для того, чтобы решить данную проблему, нам необходимо **усовершенствовать наш алгоритм**. Для этого в программу достаточно ввести **процедуру ожидания**. Процедура ожидания приостанавливает основной цикл программы сразу после того, как произойдет переключение светодиода. Теперь программа должна ожидать момента отпускания кнопки. Как только кнопка окажется отпущенной, выполнение главного цикла возобновляется.

Новый, доработанный алгоритм приведен на **рис. 5.9**. Как видно из рисунка, новый алгоритм дополнен всего двумя новыми операциями, которые и реализуют цикл ожидания. Цикл ожидания добавлен после процедуры переключения светодиода. Выполняя цикл ожидания, программа сначала читает значение бита PD0, а затем проверяет его на равенство единице. Если PD0 не равно единице (кнопка нажата), то цикл ожидания повторяется. Если PD0 равно единице (кнопка отпущена) то цикл ожидания прерывается, и управление перейдет на начало основного цикла.

Новый алгоритм вполне работоспособен и может стать основой реальной программы. Попробуем составить такую программу.

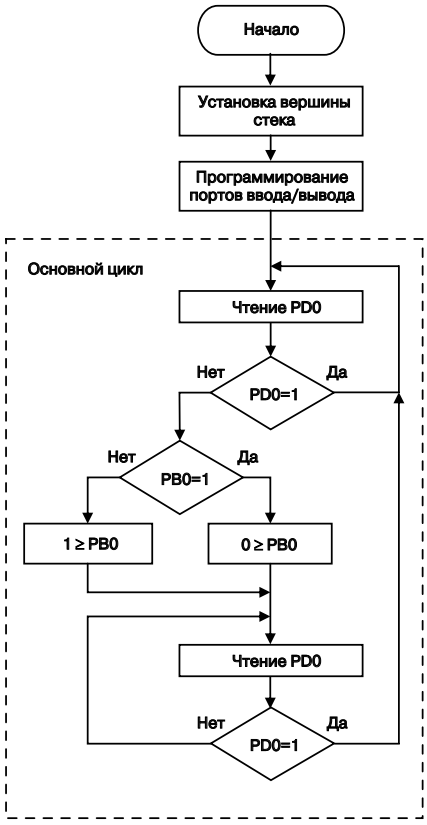


Рис. 5.9. *Усовершенствованный алгоритм программы с переключающимся светодиодом*

Программа на Ассемблере

Текст возможного варианта программы для второго примера приведен в **листинге 5.3.** В программе применены следующие новые для нас команды:

sbrc

Команда из группы условных переходов. Вызывает пропуск следующей за ней команды, если соответствующий разряд РОН сброшен. У команды два параметра. Первый параметр — **имя регистра общего назначения**, второй параметр — **номер проверяемого бита**. В **строке 17** программы (**листинг 5.3**) подобная команда проверяет нулевой разряд регистра `temp`. Если этот разряд равен нулю, то команда, записанная в **строке 18**, пропускается, и выполняется команда из **строки 19**. Если проверяемый бит равен единице, то пропуска не происходит, и выполняется команда в **строке 18**.

sbrs

Команда, обратная предыдущей. Пропускает следующую команду, если соответствующий разряд РОН установлен в единицу. Имеет те же два параметра, что и команда `sbrc`. В **строке 26** (**листинг 5.3**) подобная команда проверяет значение младшего разряда регистра `temp`. Если проверяемый бит равен единице, то команда в **строке 27** пропускается, и выполняется команда в **строке** 28. Если проверяемый разряд равен нулю, то выполняется **строка 27**.

sbi

Установка в единицу одного из разрядов порта ввода-вывода. Команда имеет два параметра: имя порта и номер устанавливаемого разряда. В **строке 22** (**листинг 5.3**) подобная команда выполняет установку младшего разряда порта `PORTB`.

cbi

Сброс в ноль одного из разрядов порта ввода-вывода. Имеет такие же два параметра, как и предыдущая команда. В **строке 24** (**листинг 5.3**) подобная команда сбрасывает младший разряд порта `PORTB` в ноль.

Листинг 5.3

```
;################################################
;##              Пример 2                      ##
;##     Программа переключения светодиода      ##
;################################################

;------------------------ Псевдокоманды управления
```

```
 1  .include "tn2313def.inc"                  ; Присоединение файла описаний
 2  .list                                     ; Включение листинга

 3  .def temp = R16                           ; Определение главного рабочего регистра

    ;------------------------- Начало программного кода

 4                .cseg                       ; Выбор сегмента программного кода
 5                .org       0                ; Установка текущего адреса на ноль

    ;------------------------- Инициализация стека

 6                ldi        temp, RAMEND     ; Выбор адреса вершины стека
 7                out        SPL, temp        ; Запись его в регистр стека

    ;------------------------- Инициализация портов ВВ

 8                ldi        temp, 0          ; Записываем ноль в регистр temp
 9                out        DDRD, temp       ; Записываем этот ноль в DDRD (порт PD на ввод)

10                ldi        temp, 0xFF       ; Записываем число $FF в регистр temp
11                out        DDRB, temp       ; Записываем temp в DDRB (порт PB на вывод)
12                out        PORTB, temp      ; Записываем temp в PORTB (потушить светодиод)
13                out        PORTD, temp      ; Записываем temp в PORTD (включаем внутр.резист.)

    ;------------------------- Инициализация компаратора

14                ldi        temp, 0x80       ; Выключение компаратора
15                out        ACSR, temp

    ;------------------------- Начало основного цикла

16  main:         in         temp, PIND       ; Читаем содержимое порта PD
17                sbrc       temp, 0          ; Проверка младшего разряда
18                rjmp       main             ; Если не ноль, переходим в начало

    ;------------------------------- Переключение светодиода

19                in         temp, PINB       ; Читаем содержимое порта PB
20                sbrc       temp, 0          ; Проверка младшего разряда
21                rjmp       m1
22                sbi        PORTB, 0         ; Установка выход PB0 в единицу
23                rjmp       m2
24  m1:           cbi        PORTB, 0         ; Сброс PB0 в ноль

    ;------------------------------ Цикл ожидания отпускания кнопки

25  m2:           in         temp, PIND       ; Читаем содержимое порта PD
26                sbrs       temp, 0          ; Проверка младшего разряда
27                rjmp       m2               ; Продолжить ожидание отпускания кнопки
28                rjmp       main             ; К началу цикла
```

Описание программы (листинг 5.3)

Первая часть программы (**строки 1—15**) полностью повторяет аналогичную часть программы из предыдущего примера (**листинг 5.1**). И это неудивительно, так как алгоритм инициализации не изменился. Зато значительно **усложнился основной цикл программы**. Теперь он значительно вырос по объему и занимает **строки 16—28**. В **строке 16** производится чтение порта `PORTD`. Число, прочитанное из порта, записывается в регистр `temp`.

В **строке 17** производится проверка младшего разряда прочитанного числа. Если младший бит равен единице (кнопка не нажата), то управление переходит к **строке 18**. В **строке 18** находится оператор безусловного перехода, который передает управление по метке `main`, то есть на начало цикла. Таким образом, пока кнопка не нажата, будет выполняться короткий цикл программы (**строки 16, 17** и **18**).

Если кнопка нажата, младший разряд числа в регистре `temp` окажется равным нулю. В этом случае оператор `sbrc` в **строке 17** передаст

управление к **строке 19**, где начинается модуль переключения светодиода. И начинается он с чтения состояния порта PB.

В **строке 20** производится проверка младшего бита считанного числа. Если этот бит равен нулю, то **строка 21** пропускается, и выполняется **строка 22**. Если младший бит равен единице, то выполняется **строка 21**. В **строке 22** оператор `sbi` устанавливает младший бит регистра `PORTB` в единицу.

А в **строке 21** находится оператор безусловного перехода, который передает управление по метке `m1` на **строку 24**. Там оператор `cbi` сбрасывает младший бит регистра `PORTB` в ноль. Таким образом, происходит переключение в младшем разряде порта PB. Ноль меняется на единицу, а единица на ноль.

После переключения светодиода управление передается на **строку 25**. Это происходит либо при помощи команды безусловного перехода (**строка 23**), либо естественным путем после **строки 24**.

Строки 25—27 содержат **цикл ожидания момента отпускания кнопки**. Цикл ожидания начинается с чтения содержимого порта `PORTD` (**строка 25**). Прочитанное значение записывается в регистр `temp`. Затем производится проверка младшего разряда прочитанного числа (**строка 26**). Если этот разряд равен нулю (кнопка еще не отпущена), выполняется **строка 27** (безусловный переход на метку `m2`), и цикл ожидания повторяется снова.

Когда при очередной проверке кнопка окажется отпущенной, повинуясь команде sbrc (в **строке 26**), микроконтроллер пропустит **строку 27** и перейдет к **строке 28**. Расположенный там безусловный переход передаст управление на начало основного цикла (по метке `main`).

Программа на языке СИ

Та же задача на языке СИ решается следующим образом. При помощи построителя создаем точно такую же заготовку программы с теми же параметрами, как и в предыдущем случае. **Доработка** программы также будет сводиться к вписыванию необходимых команд в основной цикл программы. Однако это будут другие команды, реализующие новый алгоритм.

Возможный вариант программы смотри в **листинге 5.4**. Прежде чем мы перейдем к изучению этой программы, необходимо остановиться на новом элементе языка СИ, который в ней применяется.

if else

Условный оператор. Этот оператор позволяет выполнять разные операции в зависимости от некоторого условия. В программе на языке СИ оператор записывается следующим образом:

```
if (условие)
{ Набор операторов 1 }
else
{ Набор операторов 2 }
```

Условие — это любое логическое выражение. Если результат этого выражения — истина (не равен 0), то выполняется «Набор операторов 1». В противном случае выполняется «Набор операторов 2».

Оба набора операторов — это любые допустимые операторы языка СИ. Каждый из операторов в наборе должен оканчиваться точкой с запятой. Добавочное слово `else` не обязательно. Его можно исключить вместе с набором операторов 2. Тогда, если условие ложно, оператор не будет выполнять никаких действий.

Описание программы (листинг 5.4)

Начало программы (**до строки 30**) сформировано автоматически и полностью соответствует соответствующей части предыдущей программы (**листинг 5.2**). Я лишь немного сократил комментарии для того, чтобы не перегружать текст программы лишней информацией.

Тело основного цикла претерпело значительные изменения. Теперь он занимает **строки 31—37**. В **строке 32** расположена процедура ожидания нажатия кнопки. Она представляет собой пустой цикл `while`. В теле цикла (две фигурные скобки) нет ни одного оператора.

Листинг 5.4

```
      /*****************************************************
      Project : Prog2
      Пример 2
      Управление светодиодом

      Chip type           : ATtiny2313
      Clock frequency     : 4,000000 MHz
      Data Stack size     : 32
      *****************************************************/

 1    #include <tiny2313.h>
 2    void main(void)
      {
 3    CLKPR=0x80;  // Отключить деление частоты системного генератора
 4    CLKPR=0x00;

 5    PORTA=0x00;  // Инициализация порта A
 6    DDRA=0x00;

 7    PORTB=0xFF;  // Инициализация порта B
 8    DDRB=0xFF;

 9    PORTD=0x7F;   // Инициализация порта D
10    DDRD=0x00;

11    TCCR0A=0x00;  // Инициализация таймера/счетчика 0
12    TCCR0B=0x00;
13    TCNT0=0x00;
14    OCR0A=0x00;
15    OCR0B=0x00;

16    TCCR1A=0x00;  // Инициализация таймера/счетчика 1
17    TCCR1B=0x00;
18    TCNT1H=0x00;
19    TCNT1L=0x00;
20    ICR1H=0x00;
```

```
21  ICR1L=0x00;
22  OCR1H=0x00;
23  OCR1L=0x00;
24  OCR1BH=0x00;
25  OCR1BL=0x00;

26  GIMSK=0x00;   // Инициализация внешних прерываний
27  MCUCR=0x00;

28  TIMSK=0x00;   // Инициализация прерываний от таймеров

29  USICR=0x00;   // Инициализация универсального последовательного интерфейса

30  ACSR=0x80;    // Инициализация аналогового компаратора

31  while (1)
        {
32          while (PIND.0==1) {}
33          if (PINB.0==1)
               { PORTB.0=0; }
35          else
36             { PORTB.0=1; }
37          while (PIND.0==0) {}
        };
    }
```

Цикл не выполняет никаких действий. Он будет выполняться, пока его условие истинно. В качестве условия выбрано равенство младшего разряда регистра PORTD нулю. На языке СИ это записывается следующим образом:

```
PORTD.0==1.
```

В языке СИ различают **оператор равенства** и **оператор присвоения**:

- один символ «=» означает присвоение,
- запись типа `A=5` означает присвоение переменной A значения 5;
- двойной символ «==» означает операцию сравнения,
- запись `A==5` означает проверку на равенство значений переменной `A` и константы `5`.

Результат такого сравнения равен единице в случае, если `A` равно пяти, и равен нулю, если это не так. Поэтому цикл `while` в **строке 32** программы продолжается до тех пор, пока значение разряда PORTD.0 равно единице. То есть до тех пор, пока кнопка, подключенная к этому разряду, остается ненажатой. Как только кнопка окажется нажатой, цикл (**строка 32**) заканчивается, и программа перейдет к **строке 33**.

В **строках 33—36** находится **оператор сравнения**. Он выполняет **задачу переключения светодиода**. Для этого в его условии записана проверка младшего разряда порта PB, то есть содержимого регистра PINB. Разряд проверяется на равенство единице (**строка 33**).

Если значение разряда равно единице, то выполняется **строка 34**, в которой младшему разряду регистра PORTB присваивается нулевое значение. Если условие не выполняется (значение PINB.0 не равно единице), выполняется **строка 36,** и младшему разряду PORTB присваивается единица. Значение, записанное в регистр PORTB, непосредственно поступает на выход порта PB. В результате состояние младшего разряда порта (PB0) меняется на противоположное.

В **строке 37** программы расположен **цикл ожидания отпускания кнопки**. Он аналогичен циклу в **строке 32**. Только условие теперь обратное. Цикл выполняется до ех пор, пока значение PORTD.0 равно нулю. То есть пока кнопка нажата.

5.4. Боремся с дребезгом контактов

Постановка задачи

Обратимся еще раз к схеме на **рис. 5.2**. В схеме используется кнопка, имеющая одну группу из двух нормально разомкнутых контактов. А если есть контакты, значит, есть и дребезг этих контактов. В первой главе нашей книги рассматривается способ борьбы с антидребезгом аппаратным способом. Теперь рассмотрим **способ борьбы с дребезгом контактов программным путем**.

Итак, новая задача будет сформулирована следующим образом:

«Разработать схему управления светодиодом при помощи одной кнопки. При нажатии кнопки светодиод должен изменять свое состояние на противоположное (включен или выключен). При разработке программы принять меры для борьбы с дребезгом контактов».

Схема

Как уже говорилось ранее, принципиальная схема остается прежней (см. **рис. 5.2**).

Алгоритм

Алгоритм нам придется доработать. Самый простой способ борьбы с дребезгом — введение в программу специальных задержек. Рассмотрим это подробнее. Начнем с исходного состояния, когда контакты кнопки разомкнуты. Программа ожидает их замыкания. В момент замыкания возникает дребезг контактов.

Дребезг приводит к тому, что на соответствующем разряде порта PD вместо простого перехода с единицы в ноль мы получим серию импульсов. Для того, чтобы избавится от их паразитного влияния, программа должна сработать следующим образом. Обнаружив первый же нулевой уровень на входе, программа должна перейти в режим ожидания. В режиме ожидания программа приостанавливает все свои действия и просто отрабатывает задержку.

Время задержки должно быть выбрано таким образом, чтобы оно превышало время дребезга контактов. Такую же процедуру задержки нужно ввести в том месте программы, где она ожидает отпускания кнопки. Для

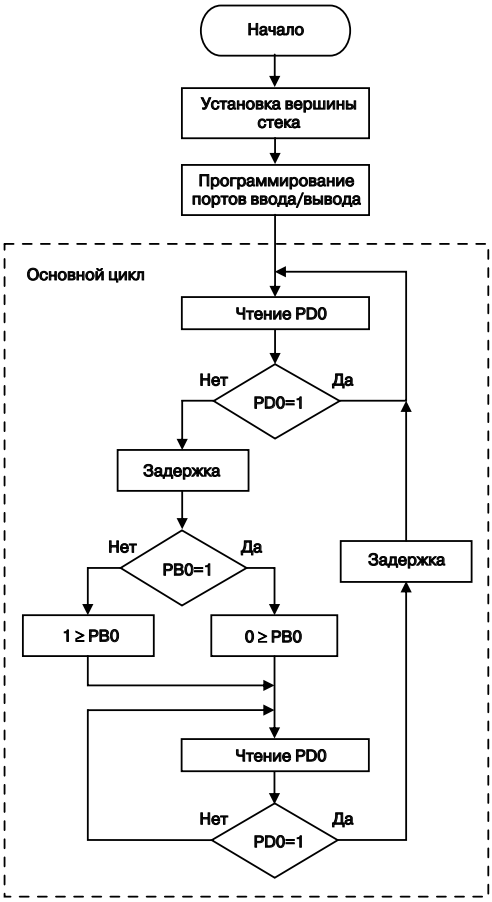


Рис. 5.10. *Алгоритм управления светодиодом с антидребезгом*

разработки нового алгоритма возьмем за основу предыдущий (см. **рис. 5.9**). Доработанный алгоритм с добавлением операций антидребезговой задержки приведен на **рис. 5.10**. Как вы можете видеть из рисунка, вся доработка свелась к включению двух процедур задержки. Одной — после обнаружения факта нажатия кнопки, а второй — после обнаружения факта ее отпускания.

Программа на Ассемблере

Для реализации нового алгоритма немного доработаем программу (**листинг 5.3**). Новый вариант программы приведен ниже (**листинг 5.5**). В этой программе используются следующие новые для нас операторы:

rcall

Переход к подпрограмме. У этого оператора всего один параметр — относительный адрес начала подпрограммы. Относительный адрес — это просто смещение относительно текущего адреса. Выполняя команду `rcall`, микроконтроллер запоминает в стеке текущий адрес программы из счетчика команд и переходит по адресу, определяемому смещением. Такой же принцип задания адреса для перехода мы уже встречали в команде `rjmp`. В **строке 20** программы (**листинг 5.5**) производится вызов подпрограммы задержки по адресу, соответствующему метке `wait`.

ret

Команда выхода из подпрограммы. По этой команде микроконтроллер извлекает из стека адрес, записанный туда при выполнении команды `rcall`, и осуществляет передачу управления по этому адресу. В **листинге 5.5** команду `ret` мы можем видеть в конце подпрограммы `wait` (см. **строку 37**).

push

Запись содержимого регистра общего назначения в стек. У данного оператора всего один операнд — имя регистра, содержимое которого нужно поместить в стек. В **строке 32** программы (**листинг 5.5**) в стек помещается содержимое регистра с именем `loop`.

pop

Извлечение информации из стека. У этого оператора тоже всего один операнд — имя регистра, в который помещается информация, извлекаемая из стека. В **строке 36** программы (**листинг 5.5**) информация извлекается из стека и помещается в регистр `loop`.

dec

Уменьшение содержимого РОН. У команды один параметр — имя регистра. Команда `dec` (декремент) уменьшает на единицу содержимое регистра, имя которого является ее параметром. В **строке 34** программы (**листинг 5.5**) уменьшается на единицу содержимое регистра `loop`.

brne

Оператор условного перехода (переход по условию). У этого оператора всего один параметр — относительный адрес перехода. Условие перехода звучит как «не равно». Попробуем разобраться, как проверяется это условие. И тут нам придется вспомнить о регистре состояния микроконтроллера (SREG).

Как уже говорилось ранее, каждый бит этого регистра представляет собой флаг. Все флаги регистра предназначены для управления работой микроконтроллера. Кроме уже известного нам флага I (глобальное разрешение прерываний), этот регистр имеет ряд флагов, отражающих результаты работы различных операций.

В данном случае нас интересует лишь один из таких флагов — **флаг нулевого результата (флаг Z)**. Этот флаг устанавливается в том случае, когда при выполнения очередной команды результат окажется равным нулю. **Например** при вычитании двух чисел, сдвиге разрядов числа или в результате операции сравнения. В нашем случае на значение флага влияет команда `dec` (**строка 34**). Если в результате действия этого оператора содержимое регистра окажется равным нулю, то флаг Z устанавливается в единицу. В противном случае он сбрасывается в ноль.

Флаг Z будет хранить записанное в него значение до тех пор, пока какая-нибудь другая команда его не изменит. Какие из команд оказывают влияние на флаг Z, а какие нет, можно узнать из приложения.

Команда `brne` использует флаг Z в качестве условия. Команда выполняет переход только в том случае, если флаг Z сброшен. То есть когда

результат предыдущей команды не равен нулю. В нашей программе (**листинг 5.5**) подобный оператор применяется в **строке 35**.

Листинг 5.5

```
        ;##############################################
        ;##                Пример 3                  ##
        ;##     Программа переключения светодиода    ##
        ;##       с использованием антидребезга      ##
        ;##############################################

        ;------------------------ Псевдокоманды управления

 1      .include "tn2313def.inc"        ; Присоединение файла описаний
 2      .list                           ; Включение листинга

 3      .def temp = R16                 ; Определение главного рабочего регистра
 4      .def loop = R17                 ; Определение регистра организации цикла

        ;------------------------ Начало программного кода

 5              .cseg                   ; Выбор сегмента программного кода
 6              .org    0               ; Установка текущего адреса на ноль

        ;------------------------ Инициализация стека

 7              ldi     temp, RAMEND    ; Выбор адреса вершины стека
 8              out     SPL, temp       ; Запись его в регистр стека

        ;------------------------ Инициализация портов ВВ

 9              ldi     temp, 0         ; Записываем ноль в регистр temp
10              out     DDRD, temp      ; Записываем этот ноль в DDRD (порт PD на ввод)

11              ldi     temp, 0xFF      ; Записываем число $FF в регистр temp
12              out     DDRB, temp      ; Записываем temp в DDRB (порт PB на вывод)
13              out     PORTB, temp     ; Записываем temp в PORTB (потушить светодиод)
14              out     PORTD, temp     ; Записываем temp в PORTD (включаем внутр.резист.)

        ;------------------------ Инициализация компаратора

15              ldi     temp, 0x80      ; Выключение компаратора
16              out     ACSR, temp

        ;------------------------ Начало основного цикла

17      main:   in      temp, PIND      ; Читаем содержимое порта PD
18              sbrc    temp, 0         ; Проверка младшего разряда
19              rjmp    main            ; Если не ноль, переходим в начало
20              rcall   wait            ; Вызов подпрограммы задержки

        ;-------------------------------- Переключение светодиода

21              in      temp, PINB      ; Читаем содержимое порта PB
22              sbrc    temp, 0         ; Проверка младшего разряда
23              rjmp    m1
24              sbi     PORTB, 0        ; Установка выход PB0 в единицу
25              rjmp    m2
26      m1:     cbi     PORTB, 0        ; Сброс PB0 в ноль

        ;-------------------------------- Цикл ожидания отпускания кнопки

27      m2:     in      temp, PIND      ; Читаем содержимое порта PD
28              sbrs    temp, 0         ; Проверка младшего разряда
29              rjmp    m2              ; Продолжить ожидание отпускания кнопки

30              rcall   wait            ; Вызов подпрограммы задержки
31              rjmp    main            ; К началу цикла

        ;-------------------------------- Подпрограмма задержки

32      wait:   push    loop            ; Сохраняем содержимое регистра loop
33              ldi     loop, 200       ; Помещаем в loop константу задержки
                                        ; Цикл задержки
34      wt1:    dec     loop            ; Уменьшаем значение регистра loop
35              brne    wt1             ; Если не ноль, продолжаем цикл

36              pop     loop            ; Восстанавливаем значение регистра loop
37              ret                     ; Выход из подпрограммы
```

Описание программы (листинг 5.5)

Новый вариант программы является полной копией старой (см. **листинг 5.3**), в которую добавлены новые элементы, **обеспечивающие антидребезговую задержку**. Так как задержка нужна в двух разных местах программы, она оформлена в виде подпрограммы. Для формирования задержки используется один **дополнительный регистр общего назначения**.

Поэтому в начале нашей новой программы (**строка 4**) добавлена **команда описания регистра**. Регистру `r17` и присваивается имя `loop`. По-английски слово `loop` означает **цикл**. Таким именем принято называть переменные, применяемые для организации циклов. Не удивляйтесь, что я употребил тут термин «переменная». В языке Ассемблер тоже используется понятие «переменные». Так наш регистр `loop` можно считать переменной с именем `loop`.

Запись значения в этот регистр эквивалентна присвоению значения переменной. Также можно интерпретировать и другие операции с регистром. Сложение содержимого двух регистров можно считать сложением переменных, вычитание — вычитанием, и так далее.

Подпрограмма задержки расположена в **строках 32—37**. Первой **строке** подпрограммы присвоена метка `wait`. Именно по этой метке и будет вызываться подпрограмма. Опустим пока назначение команд `hush` и `pop` (**строки 32 и 36**). Собственно процедура задержки расположена в **строках 33—35**. Формирование задержки производится путем многократного выполнения пустого цикла. Сначала в регистр `loop` записывается некое начальное значение (**строка** 33). В нашем случае оно равно 200.

Затем начинается цикл, который постепенно уменьшает значение регистра `loop` до нуля (**строки 34 и 35**). Происходит это следующим образом. В **строке 34** содержимое регистра уменьшается на единицу, а в **строке 35** происходит проверка содержимого на ноль. Если ноль не достигнут, то управление передается по метке `wt1`, и цикл повторяется. Когда же содержимое `loop` кажется равным нулю, очередного перехода не произойдет, и цикл задержки закончится.

Очевидно, что в нашем случае цикл задержки выполнится 200 раз. Если обратиться к приложению, то можно узнать, что команда `dec` выполняется за один такт системного генератора. Команда `brne` выполняется: за один такт, если не вызывает перехода; за два такта, если вызывает переход.

Поэтому один цикл задержки будет выполняться за 3 такта. Двести циклов за 3×200=600 тактов. Тактовая частота кварцевого резонатора у нас равна 4 МГц. Длительность одного колебания тактовой частоты равна 1/4 = 0,25 мкс. Поэтому время, за которое будут выполнены все 200

циклов задержки, равно 600×0,25=150 мкс. Добавьте сюда время выполнения остальных команд подпрограммы, включая команду вызова подпрограммы и команду возврата из подпрограммы, и вы получите общее время задержки (162 мкс).

Максимальная задержка, которую можно сформировать при помощи данной подпрограммы, равна (255×3×0,25)+12=203,25 мкс. Учтите, что в нашем случае не применяется предварительное деление частоты тактового генератора. Если это было бы не так, то длительность выполнения каждой команды нужно было бы умножать на коэффициент деления предварительного делителя.

Теперь вернемся **к двум командам работы со стеком**, которые мы не стали рассматривать вначале. Они предназначены для сохранения в стеке и последующего восстановления содержимого регистра `loop`. В начале подпрограммы (**строке 32**) значение `loop` сохраняется, а перед выходом их подпрограммы (**строка 36**) — восстанавливается.

Подобный прием придает программе одно **полезное свойство**. После окончания работы подпрограммы значения всех регистров микроконтроллера остается без изменений. В данном конкретном случае такое свойство ничего не дает, кроме, разве что, **дополнительной задержки**. Однако в сложных программах, имеющих не одну, а несколько подпрограмм, одни и те же регистры удобно использовать в разных подпрограммах.

Те же самые регистры может использовать и основная программа. В этом случае описанное выше полезное свойство просто необходимо для правильной работы всей программы. Зная эту особенность, программисты стараются применять подобный прием в каждой подпрограмме, независимо от того, полезен он в данном конкретном случае или нет.

Не исключена ситуация, когда в процессе доработки программы вам все же придется повторно использовать какие-либо регистры. Заранее обеспечить корректную работу вашей подпрограммы — это хороший стиль программирования.

В соответствии с алгоритмом (**рис. 5.5**) подпрограмма задержки в нашей программе вызывается два раза. **Первый раз** — после окончания цикла ожидания нажатия кнопки (**строка 20**). **Второй раз** — после окончания цикла ожидания отпускания (**строка 30**).

Программа на языке СИ

С программой на языке СИ мы поступим так же, как с программой на Ассемблере. Мы просто возьмем предыдущий вариант (**листинг 5.4**) и вставим в него *задержки*. Для языка СИ добавить задержку в программу гораздо проще, чем для Ассемблера. Для того, чтобы ввести задержку, мы воспользуемся стандартной библиотекой процедур задержки.

Эта библиотека входит в состав **программного комплекса CodeVisionAVR**. Название этой библиотеки `delay.h`. Посмотрите на новый вариант программы (**листинг 5.6**). В **строке 2** мы присоединяем библиотеку `delay.h` к тексту нашей программы. Так же, как в **строке 1** мы присоединили файл описания микросхемы. После присоединения библиотеки в нашем распоряжении появляется несколько функций, реализующих задержку. Воспользуемся одной из них. Имя этой функции `delay_us` (задержка в микросекундах). Она обеспечивает задержку в любое целое количество микросекунд.

Количество микросекунд задержки передается в функцию в качестве параметра. В **строке 34** программы (**листинг 5.6**) функция задержки вызывается первый раз. Она обеспечивает задержку на 200 мкс после окончания цикла ожидания нажатия кнопки. В **строке 40** такая же задержка вызывается после окончания цикла ожидания отпускания кнопки.

Теперь немного поговорим **о функции `delay_us`**. Данная функция обеспечивает формирование задержки при помощи бесконечного цикла. Такого же цикла, который мы применяли в программе на Ассемблере. Но теперь нам не нужно описывать цикл в подробностях. Достаточно применить готовую функцию.

В процессе трансляции пустой цикл формируется автоматически. Начальное значение высчитывается, исходя из заданной величины задержки и частоты тактового генератора, указанной при создании проекта.

Листинг 5.6

```
/*****************************************************
Project : Prog3
Пример 3
Управление светодиодом

Chip type           : ATtiny2313
Clock frequency     : 4,000000 MHz
*****************************************************/

1   #include <tiny2313.h>
2   #include <delay.h>

3   void main(void)
    {
4   PORTB=0xFF;  // Инициализация порта B
5   DDRB=0xFF;
6   PORTD=0x7F;   // Инициализация порта D
7   DDRD=0x00;

8   ACSR=0x80;   // Инициализация аналогового компаратора

9   while (1)
        {
10          while (PIND.0==1) {}
11          delay_us(200);
12          if (PINB.0==1)
13             { PORTB.0=0; }
14          else
15             { PORTB.0=1; }
16          while (PIND.0==0) {}
17          delay_us(200);
        };
    }
```

Кроме новой для нас функции, программа, показанная на **листинге 5.6**, имеет еще несколько отличий от оригинала:

- в программе существенно сокращен блок команд инициализации;
- удалены все команды, созданные построителем, которые дублируют запись в регистры значений по умолчанию.

Удаление лишних команд сокращает объем программы и облегчает ее понимание. Какие же команды были удалены? Это команды настройки тех систем, которые в данном случае не используются. Оставлены лишь команды настройки портов B и D. А также команда настройки аналогового компаратора.

5.5. Мигающий светодиод

Постановка задачи

Создадим **программу с мигающим светодиодом**. Сформулируем условие следующим образом:

«Создать устройство с одним светодиодом и одной управляющей кнопкой. Кнопка должна включать и выключать мигание светодиода. Пока кнопка отпущена, светодиод не должен светиться. Все время, пока кнопка нажата, светодиод должен мигать с частотой 5 Гц».

Схема

И в этом примере мы воспользуемся уже знакомой нам схемой, изображенной на **рис. 5.2**.

Алгоритм программы

Алгоритм такой программы тоже состоит из **алгоритма начальной установки** и **алгоритма основного цикла**. Начальная установка в данном случае не отличается от начальной установки всех предыдущих примеров. Алгоритм основного цикла программы можно описать следующим образом.

1. Произвести чтение порта PD.
2. Проверить младший разряд полученного числа (если его значение равно нулю, включить алгоритм мигания).
3. Если значение младшего разряда PD равно единице, выключить алгоритм мигания и потушить светодиод.
4. Перейти к началу основного цикла (пункт1).

Для того, чтобы выполнить все предыдущие пункты, нам нужно описать **алгоритм мигания светодиода**. Он будет выглядеть следующим образом.

5. Зажечь светодиод.
6. Выдержать паузу.
7. Потушить светодиод.
8. Выдержать паузу.
9. Перейти к началу алгоритма мигания (пункт 1).

Программа на Ассемблере

Возможный вариант программы приведен в **листинге 5.7**.

Листинг 5.7

```
        ;###############################################
        ;##               Пример 4                    ##
        ;##          Мигающий светодиод               ##
        ;###############################################

        ;------------------------- Псевдокоманды управления

 1      .include "tn2313def.inc"          ; Присоединение файла описаний
 2      .list                             ; Включение листинга
 3      .def temp = R16                   ; Определение главного рабочего регистра
 4      .def loop1 = R17                  ; Определение первого регистра организации цикла
 5      .def loop2 = R18                  ; Определение второго регистра организации цикла
 6      .def loop3 = R19                  ; Определение третьего регистра организации цикла

        ;------------------------- Начало программного кода

 7              .cseg                     ; Выбор сегмента программного кода
 8              .org    0                 ; Установка текущего адреса на ноль

        ;------------------------- Инициализация стека

 9              ldi     temp, RAMEND      ; Выбор адреса вершины стека
10              out     SPL, temp         ; Запись его в регистр стека

        ;------------------------- Инициализация портов ВВ

11              ldi     temp, 0           ; Записываем ноль в регистр temp
12              out     DDRD, temp        ; Записываем этот ноль в DDRD (порт PD на ввод)

13              ldi     temp, 0xFF        ; Записываем число $FF в регистр temp
14              out     DDRB, temp        ; Записываем temp в DDRB (порт PB на вывод)
15              out     PORTB, temp       ; Записываем temp в PORTB (потушить светодиод)
16              out     PORTD, temp       ; Записываем temp в PORTD (включаем внутр.резист.)

        ;------------------------- Инициализация компаратора

17              ldi     temp, 0x80        ; Выключение компаратора
18              out     ACSR, temp

        ;------------------------- Начало основного цикла

19      main:   sbi     PORTB, 0          ; Устанавливаем PB0 в единицу (тушим светодиод)
20              in      temp, PIND        ; Читаем содержимое порта PD
21              sbrc    temp, 0           ; Проверка младшего разряда
22              rjmp    main              ; Если не ноль, переходим в начало

        ;------------------------------- Мигание светодиода

23              cbi     PORTB, 0          ; Сброс PB0 в ноль (зажигаем светодиод)
24              rcall   wait1             ; Вызов подпрограммы задержки
25              sbi     PORTB, 0          ; Установка PB0 в единицу (тушим светодиод)
26              rcall   wait1             ; Вызов подпрограммы задержки
27              rjmp    main              ; К началу цикла

        ;--------------------------------- Подпрограмма задержки

28      wait1:  push    loop1             ; Сохраняем содержимое регистра loop1
29              push    loop2             ; Сохраняем содержимое регистра loop2
30              push    loop3             ; Сохраняем содержимое регистра loop3

31              ldi     loop3, 15         ; Помещаем в loop3 константу задержки
32      wt1:    dec     loop3             ; Уменьшаем значение регистра loop3
33              breq    wt4
34              ldi     loop2, 100        ; Помещаем в loop2 константу задержки
```

```
35    wt2:    dec     loop2          ; Уменьшаем значение регистра loop2
36            breq    wt1
37            ldi     loop1, 255     ; Помещаем в loop1 константу задержки
38    wt3:    dec     loop1          ; Уменьшаем значение регистра loop1
39            brne    wt3            ; Если не ноль, продолжаем цикл
40            rjmp    wt2

41    wt4:    pop     loop3          ; Восстанавливаем значение регистра loop3
42            pop     loop2          ; Восстанавливаем значение регистра loop2
43            pop     loop1          ; Восстанавливаем значение регистра loop1
44            ret                    ; Выход из подпрограммы
```

Программа содержит всего одну новую для нас команду.

breq

Оператор условного перехода по условию «равно». Этот оператор — полная противоположность оператору `brne`, описанному в предыдущем примере. Отличие этих двух операторов друг от друга в том, что `brne` вызывает переход в том случае, если флаг Z сброшен, а оператор `breq`, напротив, вызовет переход, если Z установлен.

Описание программы (листинг 5.7)

Для новой задачи нам пришлось создать **новую подпрограмму задержки**. Это произошло потому, что приведенная в предыдущем примере подпрограмма не способна обеспечить задержку достаточно большой длительности. Новая подпрограмма задержки использует не один, а целых три вложенных друг в друга цикла. По этой причине нам понадобится не один, а три вспомогательных регистра.

Поэтому в блок инициализации новой программы включены три оператора, определяющие три вспомогательные переменные `loop1`, `loop2` и `loop3` (**строки 4, 5, 6**). В остальном блок инициализации полностью соответствует аналогичному блоку из предыдущего примера. Теперь он занимает **строки 1—18**.

Основной цикл программы занимает **строки 19—27**. Он начинается с установки единицы в младшем разряде порта PB (**строка 19**). В результате светодиод выключается. Следующая команда читает содержимое порта PD и помещает его в регистр `temp` (**строка 20**).

В **строке 21** содержимое младшего разряда полученного числа проверяется на равенство единице. Если младший разряд равен единице (кнопка отпущена), то управление передается по метке `main`. И цикл замыкается. Так происходит все время, пока кнопка не нажата. При каждом проходе оператор `sbi` (**строка 19**) подтверждает единицу на выходе PB0. Светодиод остается незажженным.

Как только кнопка будет нажата, младший бит считанного из порта PD числа окажется равным нулю. Повинуясь команде сравнения в **строке 21**, микроконтроллер пропустит **строку 22**, и управление перейдет к **строке 23**. В **строке 23** начнется процедура мигания светодиода.

Она реализует один цикл мигания и работает следующим образом. Сначала оператор `cbi` (**строка 23**) устанавливает на выходе PB0 низкий логический уровень (зажигает светодиод). Затем происходит вызов подпрограммы задержки (**строка 24**). По окончании задержки команда `sbi` (**строка 25**) переводит разряд PB0 в единицу (тушит светодиод). И снова задержка (**строка 26**).

Оператор безусловного перехода (**строка 27**) передает управление на начало основного цикла программы. И вся процедура повторится сначала. Снова проверка нажатия кнопки. Если кнопка нажата, то цикл мигания повторяется. Если же кнопка окажется отпущенной, продолжения мигания не произойдет. Программа потушит светодиод и войдет в цикл ожидания нажатия кнопки (**строки 19—22**).

Итак, с миганием мы разобрались. Теперь **перейдем к подпрограмме формирования задержки**. Текст этой подпрограммы занимает **строки 28—44**. Так как требуемая частота мигания должна быть равна 5 Гц, подпрограмма должна обеспечивать время задержку 1/5 = 0,2 с (200 мс).

Как уже говорилось, подпрограмма представляет собой три **вложенных друг в друга цикла**. Самый внутренний цикл организован при помощи регистра `loop1` и включает в себя **строки 38 и 39**. Перед началом цикла в регистр `loop1` записывается число 255 (**строка 37**). Поэтому цикл повторяется 255 раз. Число 255 — это самое большое значение, которое можно записать в один восьмиразрядный регистр. Как уже говорилось, задержка, формируемая таким циклом, может быть лишь чуть больше, чем 200 мкс.

Для увеличения задержки организован **второй цикл**. Второй цикл использует регистр `loop2`. Перед началом цикла в этот регистр записывается число 100 (**строка 34**). Цикл организован при помощи оператора `dec` (**строка 35**), который последовательно уменьшает содержимое регистра `loop2` оператора сравнения `breq` (**строка 36**), проверяет, не достигло ли значение регистра нуля.

Пока в `loop2` не равно нулю, выполняется тело цикла (**строки 37—40**). В тело второго цикла включен первый цикл, использующий регистр `loop1`. Таким образом, цикл `loop1` выполняется при каждом проходе цикла `loop2`. Общее суммарное количество проходов обоих циклов будет равно 255×100 = 25500.

Но и этого недостаточно для создания нужной задержки. Даже если начальное значение для `loop2` мы выберем равным 255, и тогда мы не получим искомые 200 мс. Поэтому вокруг первых двух циклов организован третий. **Третий цикл** использует регистр `loop3` и построен точно так же, как второй. Перед началом работы в регистр `loop3` записывается число 15 (**строка 31**). Выполнение цикла обеспечивают оператор

dec (**строка 32**) и оператор сравнения (**строка 33**). При каждом проходе цикла loop3 выполняются вложенные циклы loop1 и loop2. В результате общее количество проходов строенного цикла возрастает еще в 15 раз, что обеспечивает требуемую задержку.

Кроме построенного цикла, подпрограмма задержки содержит уже знакомые нам операторы сохранения и восстановления используемых регистров. В нашем случае подпрограмма использует три регистра. Поэтому в начале подпрограммы содержимое всех трех регистров сохраняется в стеке (**строки 28, 29, 30**).

Перед выходом из подпрограммы содержимое всех этих регистров восстанавливается (**строки 41, 42, 43**). Обратите внимание, что восстановление регистров происходит в порядке, обратном порядку их запоминания. Регистр, который был записан в стек последним, извлекается первым.

Программа на языке СИ

Листинг 5.8

```
        /*****************************************************
        Project : Prog4
        Пример 4
        Мигающий светодиод

        Chip type           : ATtiny2313
        Clock frequency     : 4,000000 MHz
        *****************************************************/

 1      #include <tiny2313.h>
 2      #include <delay.h>

 3      void main(void)
        {
 4      PORTB=0xFF;   // Инициализация порта B
 5      DDRB=0xFF;

 6      PORTD=0x7F;   // Инициализация порта D
 7      DDRD=0x00;

 8      ACSR=0x80;    // Инициализация аналогового компаратора

 9      while (1)
            {
10              if (PIND.0==1)     // Проверка нажатия кнопки
11                 { PORTB.0=1; }  // Тушим светодиод
12              else
                   {
13                    PORTB.1;        // Тушим светодиод
14                    delay_ms(200);   // Задержка
15                    PORTB.0=0;       // Зажигаем светодиод
16                    delay_ms(200);   // Задержка
                   }
            };
        }
```

Как вы уже наверно догадываетесь, осуществить задержку в программе, написанной на языке СИ, будет гораздо проще, чем на Ассемблере. **Листинг 5.8** содержит один из вариантов подобной программы. Программа не содержит новых для нас операторов, поэтому сразу перейдем к ее описанию. Для создания задержки используется та же самая библиотека подпрограмм, что и в предыдущем примере (**листинг 5.6**). Однако, в нашем случае, мы берем другую функцию из этой библиотеки. Функцию delay_ms (задержка в миллисекундах).

Рассмотрим подробнее **работу программы**. Все команды инициализации в новой программе взяты из предыдущего примера и перенесены оттуда без изменений. **Различия** начинаются в **главном цикле программы**.

Оператор `if` в **строке** 10 производит проверку младшего разряда регистра PD на равенство единице. Если разряд равен единице (кнопка не нажата), то выполняется **строка 11** (запись в PB0 единицы). Эта **строка** выполняется все время, пока кнопка не нажата. В этом случае светодиод остается потушенным. Если нажать кнопку, младший разряд PD окажется равным нулю. В этом случае вместо **строки 11** выполняются **строки 13—16**.

Они представляют собой **процедуру мигания светодиода**. Эта процедура работает следующим образом. В **строке 13** светодиод тушится. Затем осуществляется задержка на 200 мс (**срока 14**). В **строке 15** светодиод зажигается. После этого опять осуществляется задержка (**строка 16**). То есть **выполняется один цикл мигания**.

Так как вся конструкция if—else находится внутри основного цикла, после окончания цикла мигания все операции повторяются сначала. То есть снова выполняется проверка состояния кнопки (**строка 10**), а по результатам проверки — одно из вышеописанных действий. В случае, если кнопка все еще нажата, цикл мигания повторяется. Если кнопка отпущена — просто гасится светодиод.

5.6. Бегущие огни

Постановка задачи

В прежние времена очень популярны среди радиолюбителей были различные автоматы световых эффектов. Сейчас этим не удивить, и совсем недорого можно купить готовую мигающую световую гирлянду. Однако, как пример для программирования, такая задача вполне подойдет. Итак, **разрабатываем «Бегущие огни».**

Задание будет звучать следующим образом:

«Разработать автомат «Бегущие огни» для управления составной гирляндой из восьми отдельных гирлянд. Устройство должно обеспечивать «движение» огня в двух разных направлениях. Переключение направления «движения» должно осуществляться при помощи переключателя».

Схема

В соответствии с поставленной задачей наше устройство должно управлять восемью световыми гирляндами. Удобно задействовать для этого все восемь выходов одного из портов. Кроме того, нам придется

подключать переключатель направления. Для этого нам понадобится еще один порт. Очевидно, что для такой задачи вполне подойдет уже знакомый нам микроконтроллер ATtiny2313.

Для создания и отладки программы совсем не обязательно подключать к микроконтроллеру гирлянды лампочек. Для начала подключим просто восемь светодиодов. Для подключения настоящей гирлянды каждый светодиод нужно заменить ключевой схемой на тиристоре, к которой уже подключить гирлянду. Примеры ключевых схем легко найти в радиолюбительской литературе. Схема бегущих огней со светодиодами приведена на **рис. 5.11**.

Как видно из **рис. 5.11**, схема представляет собой доработанный вариант схемы управления светодиодом (см. **рис. 5.2**). К предыдущей схеме просто добавлены еще семь дополнительных светодиодов, включенных таким же образом, как и светодиод VD1.

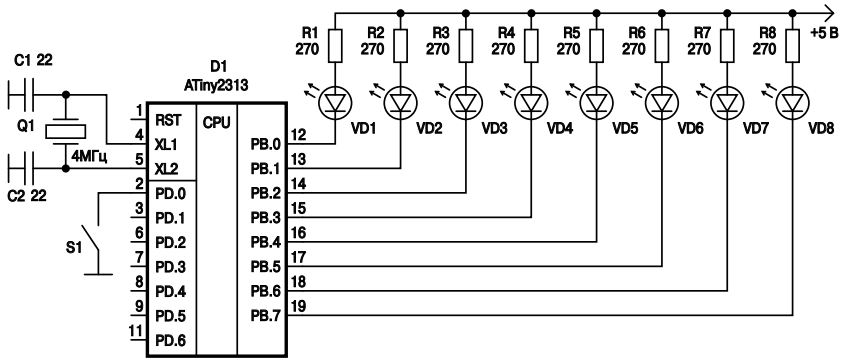


Рис. 5.11. *Схема автомата «Бегущие огни»*

Алгоритм

Для создания эффекта «бегущих огней» удобнее всего воспользоваться операторами сдвига, которые имеются в системе команд любого микроконтроллера. Такие операторы сдвигают содержимое одного из регистров микроконтроллера на один разряд влево или вправо. Если сдвигать содержимое регистра и после каждого сдвига выводить новое содержимое в порт PB, подключенные к нему светодиоды будут загораться поочередно, имитируя бегущий огонь. Алгоритм работы бегущих огней может быть разный. Один из возможных алгоритмов в самых общих чертах будет выглядеть следующим образом.

1. Считать состояние переключателя управления.
2. Если контакты переключателя разомкнуты, перейти к процедуре сдвига вправо.
3. Если контакты замкнуты, перейти к процедуре сдвига влево.

4. После окончания полного цикла сдвига (восемь последовательных сдвигов) перейти к началу алгоритма, то есть к пункту 1.

Таким образом, все время, пока контакты переключателя разомкнуты, программа будет выполнять сдвиг вправо. Если состояние переключателя не изменилось, сдвиг в прежнем направлении продолжается. Если замкнуть контакты переключателя, то все время, пока они замкнуты, будет выполняться сдвиг влево. Как при сдвиге вправо, так и при сдвиге влево после каждого полного цикла сдвига (8 шагов) происходит проверка переключателя. Если его состояние не такое же, как и прежде, то направление сдвига не изменяется. В противном случае программа меняет направление сдвига.

Выполнение алгоритма сдвига

Посмотрим теперь, как выполняется сам алгоритм сдвига. Сдвиг влево и сдвиг вправо выполняются аналогично. Ниже приводится обобщенный алгоритм для сдвига влево и сдвига вправо, снабженный комментариями.

1. Записать в рабочий регистр начальное значение. В качестве начального значения используется двоичное число, у которого один из разрядов равен единице, а остальные разряды равны нулю. Для сдвига вправо нам нужно число с единицей в самом старшем разряде (0b10000000). Для сдвига влево в единицу устанавливается младший разряд (0b00000001).
2. Вывести значение рабочего регистра в порт PB.
3. Вызвать подпрограмму задержки. Задержка нужна для того, чтобы скорость «бега» огней была нормальная для глаз наблюдателя. Если бы не было задержки, то скорость «бега» была бы столь велика, что мы бы и не увидели движения огней. С точки зрения наблюдателя мерцание огней выглядело бы как слабое свечение всех светодиодов.
4. Сдвинуть содержимое рабочего регистра вправо (влево) на один разряд.
5. Проверить, не окончился ли полный цикл сдвига (8 шагов).
6. Если полный цикл сдвига не закончен, перейти к пункту 2 данного алгоритма. Это приведет к тому, что пункты 2, 3, 4, 5 и 6 повторятся 8 раз, и лишь затем завершится полный цикл сдвига.

Программа на Ассемблере

Возможный вариант программы приведен ниже (см. **листинг 5.9**). В программе встречается несколько новых операторов. Кроме того, мы будем иметь дело с новым для нас флагом. Этот флаг также является одним из разрядов регистра SREG и называется **флагом переноса**.

Содержимое флага переноса так же, как и содержимое флага нулевого результата Z, может служить условием для оператора условного перехода. Кроме своего основного предназначения, флаг переноса ино-

Флаг переноса *— это разряд, куда помещается бит переноса при выполнении операций сложения двух чисел или бит заема при операциях вычитания.*

гда выполняет и вспомогательные функции. **Например**, он участвует во всех операциях сдвига в качестве дополнительного разряда. Теперь рассмотрим подробнее все новые операторы.

lsr

Логический сдвиг вправо. Этот оператор имеет всего один параметр — имя регистра, содержимое которого сдвигается. Схематически данная операция выглядит следующим образом:

$$0 \rightarrow d7 \rightarrow d6 \rightarrow d5 \rightarrow d4 \rightarrow d3 \rightarrow d2 \rightarrow d1 \rightarrow d0 \rightarrow C.$$

То есть содержимое младшего разряда переносится в флаг переноса C, на его место поступает содержимое разряда 1, в разряд 1 попадает содержимое разряда 2, и так далее. В самый старший разряд записывается ноль.

lsl

Логический сдвиг влево. Действие этого оператора обратно действию предыдущего. Схема такого сдвига выглядит следующим образом:

$$C \leftarrow d7 \leftarrow d6 \leftarrow d5 \leftarrow d4 \leftarrow d3 \leftarrow d2 \leftarrow d1 \leftarrow d0 \leftarrow 0.$$

То есть в данном случае в C попадает содержимое старшего разряда. Содержимое всех остальных разрядов сдвигается на один шаг влево. В самый младший разряд записывается ноль.

brcc

Переход по условию «нет переноса». Данный оператор проверяет содержимое флага переноса C и осуществляет переход по относительному адресу в том случае, если флаг C не установлен (равен нулю).

eor

Оператор «Исключающее ИЛИ». Этот оператор имеет два параметра. В качестве параметров выступают имена двух регистров, с содержимым которых производится данная операция. Оператор производит поразрядную операцию «Исключающее ИЛИ» между содержимым обоих регистров. Результат помещается в тот регистр, имя которого указано в качестве первого параметра.

Описание программы (листинг 5.9)

Как уже говорилось ранее, модуль инициализации новой программы остался таким же, как в предыдущих примерах. В новой программе он

занимает **строки 1—19**. Дополнен лишь блок описания переменных. Кроме уже знакомых нам регистров `loop1`, `loop2` и `loop3`, нам понадобится еще один дополнительный регистр. Этот регистр мы будем использовать как рабочий в операциях сдвига. В **строке 7** в качестве такого регистра выбран регистр `r20`, которому присваивается имя `rab`.

Листинг 5.9

```
        ;###############################################
        ;##               Пример 5                    ##
        ;##            "Бегущие огни"                 ##
        ;###############################################

        ;------------------------- Псевдокоманды управления

 1      .include "tn2313def.inc"                ; Присоединение файла описаний
 2      .list                                   ; Включение листинга

 3      .def  temp = R16                        ; Определение главного рабочего регистра
 4      .def  loop1 = R17                       ; Определение первого регистра организации цикла
 5      .def  loop2 = R18                       ; Определение второго регистра организации цикла
 6      .def  loop3 = R19                       ; Определение третьего регистра организации цикла
 7      .def  rab = R20                         ; Определение рабочего регистра для команд сдвига

        ;------------------------- Начало программного кода

 8              .cseg                           ; Выбор сегмента программного кода
 9              .org    0                       ; Установка текущего адреса генерации кода

        ;------------------------- Инициализация стека

10              ldi     temp, RAMEND            ; Выбор адреса вершины стека
11              out     SPL, temp               ; Запись его в регистр стека

        ;------------------------- Инициализация портов ВВ

12              ldi     temp, 0                 ; Записываем ноль в регистр temp
13              out     DDRD, temp              ; Записываем этот ноль в DDRD (порт PD на ввод)

14              ldi     temp, 0xFF              ; Записываем число $FF в регистр temp
15              out     DDRB, temp              ; Записываем temp в DDRB (порт PB на вывод)
16              out     PORTB, temp             ; Записываем temp в PORTB (потушить светодиод)
17              out     PORTD, temp             ; Записываем temp в PORTD (включаем внутр.резист.)

        ;------------------------- Инициализация компаратора

18              ldi     temp, 0x80              ; Выключение компаратора
19              out     ACSR, temp

        ;------------------------- Начало основного цикла

20      main:   in      temp, PIND              ; Читаем содержимое порта PD
21              sbrc    temp, 0                 ; Проверка младшего разряда
22              rjmp    m3                      ; Если не ноль, переходим к метке m3

        ;------------------------- Сдвиг вправо

23      m1:     ldi     rab, 0b10000000         ; Запись начального значения
24      m2:     ldi     temp, 0xFF
25              eor     temp, rab               ; Инверсия содержимого регистра rab
26              out     PORTB, temp             ; Вывод текущего значения в порт PB
27              rcall   wait1                   ; Задержка
28              lsr     rab                     ; Сдвиг содержимого рабочего регистра
29              brcc    m2                      ; Если не дошло до конца регистра продолжить
30              rjmp    main                    ; На начало

        ;------------------------- Сдвиг влево

31      m3:     ldi     rab, 0b00000001         ; Запись начального значения
32      m4:     ldi     temp, 0xFF
33              eor     temp, rab               ; Инверсия содержимого регистра rab
34              out     PORTB, temp             ; Вывод текущего значения в порт PB
35              rcall   wait1                   ; Задержка
36              lsl     rab                     ; Сдвиг содержимого рабочего регистра
37              brcc    m4                      ; Если не дошло до конца регистра продолжить
38              rjmp    main                    ; На начало
```

```
                ;-------------------------------- Подпрограмма задержки

39      wait1:  push    loop1                   ; Сохраняем содержимое регистра loop1
40              push    loop2                   ; Сохраняем содержимое регистра loop2
41              push    loop3                   ; Сохраняем содержимое регистра loop3

42              ldi     loop3, 15               ; Помещаем в loop3 константу задержки
43      wt1:    dec     loop3                   ; Уменьшаем значение регистра loop3
44              breq    wt4
45              ldi     loop2, 100              ; Помещаем в loop2 константу задержки
46      wt2:    dec     loop2                   ; Уменьшаем значение регистра loop2
47              breq    wt1
48              ldi     loop1, 255              ; Помещаем в loop1 константу задержки
49      wt3:    dec     loop1                   ; Уменьшаем значение регистра loop1
50              brne    wt3                     ; Если не ноль, продолжаем цикл
51              rjmp    wt2

52      wt4:    pop     loop3                   ; Восстанавливаем значение регистра loop3
53              pop     loop2                   ; Восстанавливаем значение регистра loop2
54              pop     loop1                   ; Восстанавливаем значение регистра loop1
55              ret                             ; Выход из подпрограммы
```

В **строке 20** начинается **основной цикл программы**. И начинается он с чтения содержимого порта PD. Результат помещается в регистр `temp`. В **строке 21** происходит оценка младшего разряда прочитанного числа. Если этот разряд равен единице, то оператор безусловного перехода в **строке 22** пропускается, и программа переходит к выполнению процедуры сдвига вправо (начало процедуры — **строка 23**). Если младший разряд считанного из PD числа равен нулю, то оператор `rjmp` в **строке 22** передает управление по метке m3, и программа переходит к процедуре сдвига влево (начало процедуры — **строка 31**).

Процедура «сдвиг вправо» работает следующим образом. В **строке 23** рабочему регистру `rab` присваивается начальное значение. Для наглядности это число записано в двоичном формате. Затем начинается **цикл сдвига** (**строки 24—30**). Первой операцией цикла сдвига, в соответствии с алгоритмом, должна быть операция вывода содержимого регистра `rab` в порт PB. Однако существует одно небольшое препятствие.

Если просто вывести содержимое `rab` в порт PB, то мы получим картину, обратную той, которая нам необходима. Все светодиоды, кроме одного, будут светиться. Это произойдет потому, что ноль на выходе порта зажигает светодиод, а единица тушит. Если мы хотим получить бегущий огонь, а не бегущую тень, нам нужно проинвертировать содержимое регистра `rab` перед тем, как вывести в порт PB.

Для инвертирования содержимого регистра `rab` воспользуемся командой `eor` («Исключающее ИЛИ»). Операция «Исключающее ИЛИ» обладает способностью инвертирования чисел. Если вы вернетесь назад и посмотрите на таблицу истинности операции «Исключающее ИЛИ» (**рис. 5.8**), то вы можете заметить эту особенность.

Поэтому, если произвести операцию «Исключающее ИЛИ» между двумя байтами, значение одного из которых будет равно 0xFF, то в результате этой операции мы получим инверсное значение второго

байта. Для выполнения такой операции используется вспомогательный регистр `temp`. В **строке 24** в регистр `temp` записывается число 0xFF. В **строке 25** производится операция «Исключающее ИЛИ» между содержимым регистров `temp` и `rab`.

Правило

Для всех строк таблицы истинности справедливо правило: если один из операндов равен единице, то результат операции равен инверсному значению второго операнда.

Результат этой операции помещается в `temp`, так как именно он является первым параметром данной команды. Содержимое регистра `rab` при этом не изменяется. В **строке 26** содержимое регистра `temp` выводится в порт PB.

Следующий этап процедуры сдвига — **вызов подпрограммы задержки**. Вызов этой подпрограммы происходит в **строке 27**. В **строке 28** производится сдвиг содержимого регистра `rab` на один бит вправо. В **строке 29** оператор `brcc` проверяет состояние признака переноса. Эта проверка позволяет обнаружить момент, когда закончится полный цикл сдвига. Как это происходит, иллюстрирует **табл. 5.2**.

В таблице показаны значения всех разрядов вспомогательного регистра `rab` для каждого из восьми шагов, составляющих **полный цикл сдвига**. Разряды сдвигаемого регистра обозначены как b7—b0. Последний столбец показывает содержимое признака переноса. Как видно из таблицы, единица, которая в начале находится в самом старшем разряде, при каждом очередном шаге сдвигается в соседнюю позицию.

Сдвиг информации в рабочем регистре *Таблица 5.2*

Шаг	b7	b6	b5	b4	b3	b2	b1	b0	C
1	1	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0
5	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	1

В результате, после восьмого шага она оказывается в ячейке признака переноса. Пока C равно нулю, оператор `brcc` в **строке 29** передает управление по метке m2, и цикл сдвига продолжается. После восьмого шага признак переноса C станет равен единице. Поэтому перехода на начало цикла в **строке 29** не произойдет, и управление перейдет к **строке 30**. В результате очередного девятого цикла сдвига не произой-

дет. Оператор безусловного перехода в **строке 30** передаст управление на начало основного цикла, и программа снова приступит к проверке состояния кнопки.

Процедура сдвига влево занимает **строки 31—38**. Эта процедура работает точно так же, как и процедура сдвига вправо. **Отличия:**

- начальное значение, записываемое в регистр `rab` (см. **строку 31**), равно 0b00000001;
- вместо оператора `lsr` (сдвиг вправо) в **строке 36** использован оператор `lsl` (сдвиг влево).

В качестве подпрограммы задержки применена уже известная нам **подпрограмма с тремя вложенными циклами**. Текст этой подпрограммы полностью скопирован из предыдущего примера (**листинг 5.7**) и расположен в **строках 39—55**.

Программа на языке СИ

Возможный вариант той же программы, но на языке СИ, приведен в **листинге 5.10**. В этой программе **впервые** мы будем использовать **переменную**. До сих пор мы не применяли переменные лишь благодаря предельной простоте предыдущих программ. Теперь же переменная нам понадобится для того, чтобы осуществлять операции сдвига.

Переменная будет хранить текущее значение всех сдвигаемых битов так же, как в программе на Ассемблере их хранил регистр `rab`. Назовем переменную тем же именем, каким мы называли регистр. Описание переменной в нашей программе происходит в **строке 4**. Так как сдвигаемых битов должно быть всего восемь, то самый подходящий тип данных для нашей переменной — это `unsigned char`.

Листинг 5.10

```
    /*****************************************************
    Project : Prog5
    Пример 5
    Бегущие огни

    Chip type           : ATtiny2313
    Clock frequency     : 4,000000 MHz
    Data Stack size     : 32
    *****************************************************/

1   #include <tiny2313.h>
2   #include <delay.h>

3   void main(void)
    {
4   unsigned char rab;  // Вводим переменную rab

5   PORTB=0xFF;  // Инициализация порта B
6   DDRB=0xFF;

7   PORTD=0x7F;   // Инициализация порта D
8   DDRD=0x00;

9   ACSR=0x80;   // Инициализация аналогового компаратора
```

```
10  while (1)
        {
11      if (PIND.0==1)  // Проверка состояния переключателя
              {                    // Сдвиг вправо
12            rab = 0b10000000;    // Запись начального значения
13            while (rab!=0)
                    {
14                        PORTB=rab^0xFF;  // Запись в порт с инверсией
15                        rab = rab >> 1;  // Сдвиг разрядов
16                        delay_ms (200);  // Задержка в 200 мсек
                    }
              }
17      else
              {                    // Сдвиг влево
18            rab = 0b00000001;    // Запись начального значения
19            while (rab!=0)
                    {
20                        PORTB=rab^0xFF;  // Запись в порт с инверсией
21                        rab = rab << 1;  // Сдвиг разрядов
22                        delay_ms (200);  // Задержка в 200 мсек
                    }
              }
        };
    }
```

Переменная такого типа имеет длину в один байт. Второй однобайтовый тип (`char`) в данном случае нам не подходит, так как представляет собой число со знаком. У такого числа старший разряд интерпретируется как знак. И лишь семь младших разрядов используются непосредственно для хранения значений.

Строки 1—9 составляет **модуль инициализации программы**. В полном соответствии с алгоритмом модуль инициализации новой программы почти полностью повторяет соответствующий модуль из предыдущего примера. **Исключение** составляет лишь вновь добавленная **строка** с описанием переменной (**строка 4**).

Основной цикл программы занимает **строки 10—22**. **Телом цикла** является **оператор сравнения** (конструкция `if—else`), проверяющий состояние бита, связанного с переключателем. Собственно проверка происходит в **строке 11**. Здесь младший бит порта PD проверяется на равенство единице. Если он равен единице, то выполняется процедура сдвига вправо (**строки 12—16**). В противном случае выполняется процедура сдвига влево (**строки 18—22**).

Каждая из этих процедур выполняет цикл из восьми сдвигов в нужном направлении. Так как вся конструкция `if—else` находится внутри бесконечного цикла, то она многократно повторяется. То есть после проверки происходит восемь сдвигов в нужном направлении. Затем новая проверка, и так далее.

Обе процедуры сдвига построены одинаково. Рассмотрим подробнее **процедуру сдвига вправо.** В **строке 12** переменной `rab` присваивается начальное значение. Затем начинается цикл сдвига. **Цикл организован** при помощи оператора `while` (**строка 13**). Его тело составляют **строки 14—16**, которые реализуют уже знакомый нам алгоритм. Сначала происходит вывод значения всех разрядов переменной `rab` в

порт PB. Как и в предыдущем случае, выводимое значение нам нужно предварительно проинвертировать.

Для инвертирования числа мы снова используем прием, который мы применили в программе на Ассемблере. То есть воспользуемся оператором «Исключающее ИЛИ». Обратимся к **строке 14** нашей программы. В этой **строке** регистру PORTB присваивается значение выражения `rab^0xFF`. Символ «^» в языке СИ как раз и означает операцию «Исключающее ИЛИ». При помощи единственного выражения мы сразу и инвертируем, и присваиваем.

В **строке 15** производится **сдвиг разрядов.** Для сдвига используется оператор «>>». Результатом выражения `rab >> 1` является число, полученное путем сдвига всех разрядов переменной `rab` на одну позицию вправо. Число 1 справа от оператора сдвига означает количество разрядов, на которое нужно сдвинуть число. Таким образом, выражение

```
rab = rab >> 1;
```

означает: присвоить переменной `rab` значение этой же переменной, сдвинутое на один разряд вправо. Язык СИ допускает другую, сокращенную форму записи того же самого выражения:

```
rab >>= 1;
```

Новое выражение полностью эквивалентно предыдущему. Подобные изящные сокращения являются фирменной особенностью языка СИ. Благодаря ним программа на языке СИ получается короче и проще. В **строке 16** вызывается функция задержки. Время задержки составляет 200 мс.

Для того, чтобы цикл сдвига повторялся только восемь раз, используется оператор цикла (**строка 13**). В качестве условия, при котором цикл выполняется, используется выражение `rab !=0`. В языке СИ выражение «!=» означает «Не равно».

Таким образом, наш цикл сдвига (**строки 14—16**) будет выполняться до тех пор, пока `rab` не равен нулю. Это и будут наши восемь шагов сдвига. Чтобы убедиться в этом, еще раз посмотрите на **табл. 5.2**. Значение одного из разрядов b0—b7, а, значит, и всей переменной `rab`, во время первых восьми шагов не равно нулю. И только на девятом шаге все восемь рабочих разрядов обнулятся. Но так как при этом заложенное нами условие не выполняется, последнего девятого цикла не будет.

Процедура сдвига влево находится в **строках 18—22** программы и работает точно так же, как процедура сдвига вправо. Имеются лишь два отличия:

- другое начальное значение переменной `rab` (см. **строку 18**);
- применен другой оператор сдвига.

Для сдвига влево применяется оператор «<<» (см. **строку 21**). При желании выражение в **строке 21** тоже можно сократить. Вместо `rab = rab << 1;` можно записать `rab <<= 1;`

5.7. Использование таймера

Постановка задачи

В предыдущих примерах для формирования задержки мы использовали один или несколько вложенных программных циклов. Однако такой способ приемлем далеко не всегда. **Главный недостаток** подобного метода состоит в том, что он полностью загружает центральный процессор. Пока микроконтроллер занят формированием задержки, он не может выполнять никаких других задач.

Еще один недостаток — невозможно с достаточной точностью выбрать время задержки. Гораздо лучшие результаты дает другой способ — **формирование интервалов времени при помощи одного из встроенных таймеров/счетчиков микроконтроллера**. Любой из таймеров/счетчиков может работать как с использованием прерываний, так и без прерываний. Далее мы рассмотрим оба эти варианта. И начнем мы с более простого случая.

Итак, заново сформулируем нашу задачу:

«Доработать программу «Бегущие огни», изменив процедуру формирования задержки. Новая процедура должна использовать один из внутренних таймеров/счетчиков и не использовать прерывания.»

Схема

Так как мы разрабатываем не самостоятельное устройство, а лишь усовершенствуем управляющую программу, то схема устройства (**рис. 5.11**) не изменяется.

Алгоритм

Как известно, в микроконтроллере ATtiny2313 имеются два встроенных таймера-счетчика. Поэтому сначала нам нужно выбрать, какой из них мы будем использовать. Исходить будем из заданного времени задержки 200 мс. Как известно, для формирования временных интервалов таймер/счетчик просто подсчитывает тактовые импульсы от системного генератора.

Частота сигнала этого генератора в нашем случае равна 4 МГц. А период импульсов 1/4=0,25 мкс. Для того, чтобы получить на выходе 200 мс, необходимо иметь коэффициент деления, равный $200 \cdot 10^{-3} / 0{,}25 \cdot 10^{-6} = 800 \cdot 10^{3}$ (восемьсот тысяч раз).

Микросхема ATtiny2313 содержит два таймера. Один восьмиразрядный и один шестнадцати. **Восьмиразрядный таймер** имеет максимальный коэффициент пересчета $2^8=256$, а шестнадцатиразрядный — $2^{16}=65536$. То есть даже шестнадцатиразрядного таймера нам не хватит для формирования требуемой задержки. Придется воспользоваться

предварительным делителем. Этот делитель производит предварительное деление тактового сигнала перед тем, как тот поступит на вход таймера/счетчика.

Программным путем можно выбрать один из четырех фиксированных коэффициентов деления. Выберем самый большой возможный коэффициент деления предделителя (1024). Тогда на его выходе мы получим сигнал с частотой $4 \cdot 10^6/1024 = 3906$ Гц. Период такого сигнала будет равен $1/3906 \approx 0{,}256 \cdot 10^{-3}$ с или 0,256 мс. Именно этот сигнал поступает на вход нашего таймера, который обеспечивает окончательное деление. Посчитаем коэффициент деления, который наш таймер должен нам обеспечить: 200/0,256 ≈ 780. Такой коэффициент пересчета нам может обеспечить только таймер T1.

Итак, мы определились как с выбором таймера, так и с его настройками. Теперь можно приступать к созданию **новой подпрограммы задержки**. Прежде, чем это сделать, попробуем описать **алгоритм ее работы**. Данный алгоритм предполагает, что все необходимые настройки таймера предварительного делителя произведены до первого вызова подпрограммы, таймер запущен и находится в режиме непрерывного счета.

Алгоритм подпрограммы задержки представлен ниже.

1. Записать в счетный регистр таймера T1 нулевое значение.
2. Начать цикл проверки содержимого счетного регистра. В теле цикла программа должна многократно считывать содержимое счетного регистра таймера и проверять, не достигло ли оно своего конечного значения (то есть значения 780).
3. При достижении счетным регистром конечного значения, завершить цикл проверки.
4. Выйти из подпрограммы задержки.

Программа на Ассемблере

Программа «Бегущие огни» с новым вариантом подпрограммы задержки приведена в **листинге 5.11**. Новая подпрограмма задержки использует таймер T1 и описанный выше алгоритм работы. Рассмотрим подробнее, как работает такая программа. А начнем, как обычно, с описания новых для нас операторов.

.equ

Псевдооператор присвоения. Название оператора происходит от английского слова «эквивалентно» (equality). Используется для присвоения имен различным константам. В **строке 5** листинга 5.11 числу 780 присваивается имя `kdel`. Теперь в любом месте программы вместо числа 780 можно применять константу `kdel`. Имя для константы имеет

то же значение, что и имя для переменной. Во-первых, по осмысленному имени легко понять назначение константы. **Например**, `kdel` расшифровывается, как «коэффициент деления». А, во-вторых, это удобно при смене значения. Поменяйте в **строке 5** число 780, **к примеру**, на 800, и везде, где бы ни встретилась константа `kdel`, она уже будет иметь новое значение.

cpi

Сравнение содержимого РОН с константой. Эта команда имеет два параметра. Первый параметр — имя регистра общего назначения, содержимое которого подлежит сравнению. Второй параметр — некая константа, с которой сравнивается содержимое РОН. По результатам сравнения устанавливаются все флаги регистра `SREG`. Флаги устанавливаются точно так же, как если бы содержимое РОН вычиталось из константы. А именно: флаг переноса C устанавливается в том случае, если при вычитании данных чисел возникает перенос в старший разряд (содержимое регистра меньше константы), и сбрасывается, если нет переноса. Флаг Z (нулевой результат) устанавливается при равенстве содержимого РОН и константы и сбрасывается в случае их неравенства. Все остальные флаги устанавливаются в соответствии со своим назначением. Подробнее об этом вы можете узнать из **приложения 1**. После того, как значения флагов определены, они могут быть использованы различными условными операторами. В **строке 44** программы (**листинг 5.11**) оператор `cpi` производит сравнение содержимого регистра `temp` с числом 0xD0.

brlo

Переход по условию «меньше». Имеется в виду, что в предыдущей команде, в результате сравнения (или вычитания) двух операндов, первый операнд оказался меньше, чем второй. Для проверки этого условия оператор использует флаг переноса C. Переход происходит лишь в том случае, если C = 1. В **строке 45** программы условный переход используется для того, чтобы передать управление на строку с меткой `wt1` в том случае, если по результатам предыдущего сравнения (**строка 44**) оказалось, что содержимое регистра `temp` меньше, чем число 0xD0.

Описание программы (листинг 5.11)

Как уже говорилось, данная программа является модификацией программы из предыдущего примера (см. **листинг 5.9**). **Основное отличие** новой программы от старой — полная переработка подпрограммы задержки. В связи с тем, что новая подпрограмма задержки использует таймер, для ее нормальной работы пришлось также доработать модуль инициализации основной программы.

Листинг 5.11

```
;###############################################
;##                Пример 6                   ##
;##            "Бегущие огни"                 ##
;##       с использованием таймера            ##
;###############################################

;------------------------- Псевдокоманды управления

1   .include "tn2313def.inc"                    ; Присоединение файла описаний
2   .list                                       ; Включение листинга

3   .def  temp = R16                            ; Определение главного рабочего регистра
4   .def  rab = R20                             ; Определение рабочего регистра для команд сдвига

5   .equ  kdel = 780

;------------------------- Начало программного кода

6           .cseg                               ; Выбор сегмента программного кода
7           .org            0                   ; Установка текущего адреса на ноль

;------------------------- Инициализация стека

8           ldi             temp, RAMEND        ; Выбор адреса вершины стека
9           out             SPL, temp           ; Запись его в регистр стека

;------------------------- Инициализация портов ВВ

12          ldi             temp, 0             ; Записываем ноль в регистр temp
11          out             DDRD, temp          ; Записываем этот ноль в DDRD (порт PD на ввод)

12          ldi             temp, 0xFF          ; Записываем число $FF в регистр temp
13          out             DDRB, temp          ; Записываем temp в DDRB (порт PB на вывод)
14          out             PORTB, temp         ; Записываем temp в PORTB (потушить светодиод)
15          out             PORTD, temp         ; Записываем temp в PORTD (включаем внутр.резист.)

;------------------------- Инициализация таймера T1

16          ldi             temp, 0x05          ; Код конфигурации записываем в temp
17          out             TCCR1B, temp        ; Переносим его в регистр конфигурации таймера

;------------------------- Инициализация компаратора

18          ldi             temp, 0x80          ; Выключение компаратора
19          out             ACSR, temp

;------------------------- Начало основного цикла

20  main:   in              temp, PIND          ; Читаем содержимое порта PD
21          sbrc            temp, 0             ; Проверка младшего разряда
22          rjmp            m3                  ; Если не ноль, переходим в начало

;------------------------- Сдвиг вправо

23  m1:     ldi             rab, 0b10000000     ; Запись начального значения
24  m2:     ldi             temp, 0xFF
25          eor             temp, rab           ; Инверсия содержимого регистра rab
26          out             PORTB, temp         ; Вывод текущего значения в порт PB
27          rcall           wait1               ; Задержка
28          lsr             rab                 ; Сдвиг содержимого рабочего регистра
29          brcc            m2                  ; Если не дошло до конца регистра продолжить
30          rjmp            main                ; На начало

;------------------------- Сдвиг влево

31  m3:     ldi             rab, 0b00000001     ; Запись начального значения
32  m4:     ldi             temp, 0xFF
33          eor             temp, rab           ; Инверсия содержимого регистра rab
34          out             PORTB, temp         ; Вывод текущего значения в порт PB
35          rcall           wait1               ; Задержка
36          lsl             rab                 ; Сдвиг содержимого рабочего регистра
37          brcc            m4                  ; Если не дошло до конца регистра продолжить
38          rjmp            main                ; На начало


;--------------------------------- Подпрограмма задержки

39  wait1:  push            temp                ; Сохраняем содержимое регистра temp

40          ldi             temp, 0             ; Помещаем temp ноль
```

```
41              out             TCNT1H, temp        ; Записываем этот ноль в старший регистр таймера
42              out             TCNT1L, temp        ; Записываем этот ноль в младший регистр таймера

43      wt1:    in              temp, TCNT1L        ; Чтение младшей части счетного регистра
44              cpi             temp, low(kdel)     ; Сравнение с числом $0C
45              brlo            wt1                 ; Переход, если temp меньше чем kdel
46              in              temp, TCNT1H        ; Чтение старшей части счетного регистра
47              cpi             temp, high(kdel)    ; Сравнение с числом $03
48              brlo            wt1                 ; Переход, если temp меньше чем $03

49              pop             temp                ; Восстанавливаем значение регистра temp
50              ret                                 ; Выход из подпрограммы
```

Первая доработка модуля инициализации — команда в **строке 5**. Эта команда описывает константу `kdel`, то есть коэффициент деления таймера. Обратите внимание, что значение этой константы равно 780. Если перевести это значение в двоичную форму, то количество разрядов такого числа будет больше восьми. А это значит, что для представления константы в двоичном виде потребуется не менее двух байтов.

Далее в программе с этой константой сравнивается содержимое счетного регистра таймера T1, который тоже имеет шестнадцать разрядов. Однако микроконтроллеры AVR работают лишь с восьмиразрядными величинами. Счетный регистр таймера T1 представляет собой два восьмиразрядных регистра TCNT1L и TCNT1H. Используемый для сравнения оператор `brlo` также работает с восьмиразрядными величинами. Как же выполняется такое сравнение?

Сравнение происходит в два этапа. Сначала сравниваются младшие разряды обеих величин, затем старшие. Младшее и старшее значение счетного регистра хранятся в двух соответствующих регистрах TCNT1L и TCNT1H. А для выделения младшего и старшего байта константы в языке Ассемблер существуют специальные функции `low` и `high`.

Рассмотрим действие этих функций на конкретном примере. Значение нашей константы `kdel` равно 780. В шестнадцатиричном виде это выглядит как 0x030C. Используя вышеописанные функции, мы можем найти старший и младший байты числа:

```
high(kdel) = 0x03 low(kdel) = 0x0C.
```

Данные функции используются в **строках** 44 и 47 программы.

Следующая доработка модуля инициализации — это две команды, выбирающие режим работы таймера (**строки 16, 17**). Эти команды записывают в регистр TCCR1B константу 0x05. В качестве вспомогательного регистра используется `temp`.

Регистр TCCR1B — это один из двух регистров выбора режимов работы таймера T1. При записи кода 0x05 в этот регистр устанавливается коэффициент предварительного деления 1/1024, и таймер переходит в режим счета. Второй регистр конфигурации таймера называется TCCR1A. Его значение нужно оставить по умолчанию.

Последние изменения основной части программы коснулись команд вызова подпрограммы задержки. Вызов задержки происходит в **стро-**

ках 27 и **35**. Других изменений основной части программы не потребовалось.

Новая подпрограмма задержки занимает **строки 39—50**. Начинается подпрограмма традиционно сохранением содержимого всех используемых ею регистров. В данном случае потребовалось сохранить лишь содержимое одного регистра `temp` (см. **строку 39**). Следующие три команды производят запись нулевого значения в счетный регистр таймера T1. Сначала ноль записывается в регистр `temp` (**строка 40**). А затем содержимое `temp` поочередно помещается в регистры TCNT1H и TCNT1L (**строки 41, 42**).

Порядок записи информации в пару регистров TCNT1H, TCNT1L неслучайный. Эти два регистра обладают **свойством** так называемой **двойной буферизации**. Правила работы с такими регистрами требуют, чтобы при записи значения в эти регистры сначала записывался старший регистр TCNT1H, а потом младший TCNT1L. Дело в том, что при записи старшего байта в регистр TCNT1H он не попадает сразу по назначению, а сохраняется в специальном внутреннем регистре. Когда же поступает команда записи младшего байта в регистр TCNT1L, оба байта записываются одновременно. В этом и состоит двойная буферизация. Использование двойной буферизации позволяет менять значение счетного регистра на ходу, не останавливая таймера.

После записи нулевого значения в счетный регистр начинается **цикл проверки**. Он занимает **строки 43—48** программы. Сравнение происходит в **два этапа**. В **строках 43—45** сравнивается младшая часть счетного регистра с младшим байтом коэффициента деления. В **строках 46—48** сравниваются старшие байты. Рассмотрим это подробнее.

В **строке 43** содержимое регистра TCNT1L помещается в регистр `temp`. В **строке 44** происходит сравнение содержимого регистра `temp` с младшим байтом константы. Команда условного перехода в **строке 45** передает управление на начало цикла сравнения только в том случае, если содержимое регистра еще не достигло требуемого значения.

В **строках 46—48** такие же операции сравнения производится для регистра TCNT1H. При этом используется старший байт константы `kdel`. Если старший разряд счетного регистра не достиг требуемого значения, то управление передается на метку `wt1`. То есть в этом случае программа опять повторяет сравнение младших разрядов.

Такой порядок также диктуется наличием двойной буферизации. При чтении младшей части регистра старшая его часть запоминается в специальном внутреннем буфере. Команда чтения старшей части регистра на самом деле читает содержимое этого буфера.

Пока происходит цикл сравнения, счетчик находится в режиме счета. Содержимое счетного регистра постепенно увеличивается и, в конце

концов, достигает требуемого значения. Пока происходит очередной цикл проверки, содержимое счетного регистра может даже превысить значение константы.

В этом случае переходов в **строке 45** и в **строке 48** не произойдет. В результате подпрограмма перейдет к своему завершению. В **строке 49** происходит восстановление содержимого регистра `temp`. А в **строке 50** — выход из подпрограммы.

Программа на языке СИ

Возможный вариант той же программы на языке СИ приведен в **листинге 5.12**. Эта программа представляет собой доработанный вариант программы из предыдущего примера (**листинг 5.10**). **Причем доработка свелась к созданию новой функции задержки**. Новая функция понадобилась нам потому, что использованная ранее библиотечная функция `delay_ms` в данном случае нам не подходит.

Для формирования задержки программа использует вложенные циклы. Готовой функции, удовлетворяющей нашим новым условиям, ни в одной из стандартных библиотек не существует. Поэтому нам пришлось создать ее самостоятельно. Текст новой функции задержки приводится в **строках 2—4**. Наша **новая функция задержки** получила имя `wait1`. **Обратите внимание**, что описание функции `wait1` расположено в нашей программе раньше, чем описание функции `main`. Такой порядок отнюдь не случаен.

Правило

В языке СИ действует правило: любая функция должна быть прежде описана и лишь затем в первый раз применена.

В связи с вводом новой функции нам пришлось **немного доработать и основную программу**. **Во-первых**, в модуль инициализации добавлены команды инициализации таймера (**строки 11, 12**). Причем команда в **строке 11** является избыточной. Нулевое значение в регистр TCCR1A можно и не записывать, так как там и так ноль по умолчанию.

Так как функция `main` *использует функцию* `wait1` *в качестве процедуры задержки, то описание* `wait1` *должно располагаться перед описанием* `main`

Листинг 5.12

```
/*****************************************************
Project : Prog6
Пример 6
Бегущие огни (задержка с использованием таймера без использования прерываний)

Chip type           : ATtiny2313
Clock frequency     : 4,000000 MHz
Data Stack size     : 32
*****************************************************/
```

```
1   #include <tiny2313.h>

2   void wait1 (void)    // --------------- Функция задержки
    {
3       TCNT1=0;
4       while (TCNT1<780) {};
    }

5   void main(void)      // ----------------- Главная функция программы
    {
6   unsigned char rab;   // Вводим переменную rab

7   PORTB=0xFF;   // Инициализация порта B
8   DDRB=0xFF;

9   PORTD=0x7F;   // Инициализация порта D
10  DDRD=0x00;

11  TCCR1A=0x00;  // Инициализация таймера/счетчика 1
12  TCCR1B=0x05;

13  ACSR=0x80;    // Инициализация аналогового компаратора

14  while (1)
        {
15      if (PIND.0==1)  // Проверка состояния переключателя
              {                     // Сдвиг вправо
16            rab = 0b10000000;     // Запись начального значения
17            while (rab!=0)
                    {
18                  PORTB=rab^0xFF;  // Запись в порт с инверсией
19                  rab = rab >> 1;  // Сдвиг разрядов
20                  wait1 ();        // Задержка в 200 мсек
                    }
              }
21      else
              {                     // Сдвиг влево
22            rab = 0b00000001;     // Запись начального значения
23            while (rab!=0)
                    {
24                  PORTB=rab^0xFF;  // Запись в порт с инверсией
25                  rab = rab << 1;  // Сдвиг разрядов
26                  wait1 ();        // Задержка в 200 мсек
                    }
              }
        };
    }
```

Второе изменение внесено в основной цикл программы. Оно очевидно. Вместо функции задержки `delay_ms` мы применим нашу новую функцию `wait1` (см. **строки 20 и 26**). Функция `wait1` не имеет параметров, так как предназначена для формирования фиксированного значения задержки.

С этим связана **и последняя доработка**. Так как библиотечная функция задержки нам больше не нужна, мы можем исключить из программы команду, присоединяющую библиотеку `delay.h`. Других доработок основная программа не потребовала.

Теперь разберем подробнее саму функцию `wait1`. Она **формирует задержку с использованием таймера/счетчика**. Подобный алгоритм мы уже реализовывали в подпрограмме `wait1` на Ассемблере (см. **листинг 5.11**).

Однако язык **СИ** значительно **упрощает задачу**. Во-первых, нам не обязательно работать с отдельными байтами. Теперь мы без труда можем оперировать шестнадцатиразрядными числами. В результате функция задержки предельно упрощается. Она занимает всего три **строки** (**строки 2—4**). Первая **строка** — это заголовок описания функции. Из

него видно, что функция `wait1` не использует параметров и не возвращает никаких величин. Тело функции составляют **строки 3 и 4**. В **строке 3** счетному регистру таймера T1 присваивается нулевое значение. Значение присваивается сразу всему шестнадцатиразрядному регистру TCNT1.

И неважно, что на самом деле микроконтроллер не имеет прямого доступа к этому регистру. После трансляции будет создана программа в машинных кодах, которая запишет сначала старшую часть (TCNT1H), а затем младшую часть (TCNT1L) регистра, строго соблюдая правила работы с регистрами, имеющими двойную буферизацию.

В **строке 4** расположен **цикл проверки**. Это пустой цикл, в качестве условия которого выступает выражение TCNT1<780. **Обратите внимание**, что в этом выражении мы тоже используем имя TCNT1. То есть проверяем значение всего шестнадцатиразрядного счетного регистра. Цикл проверки будет выполняться до тех пор, пока значение счетного регистра не будет превышать 780. Как только окажется, что это не так, цикл завершается, а с завершением цикла завершается и вся функция `wait1`.

5.8. Использование прерываний по таймеру

Постановка задачи

В **предыдущем примере** мы использовали таймер для формирования задержки, но не использовали его главного преимущества: способности вызывать прерывания. На практике подобным образом почти никогда не поступают. Чаще всего в подобных случаях применяют прерывания по таймеру. Это позволяет более точно формировать интервалы времени, но главное — позволяет разгрузить центральный процессор.

Пока таймер формирует задержку, программа может выполнять любые другие действия. В результате программу бегущих огней можно легко совместить, **например**, с программой генерации звуков. Но не будем усложнять нашу задачу и сформулируем ее следующим образом:

«Создать новую программу «бегущих огней» с использованием прерываний по таймеру.»

Схема

Схему оставим без изменений (см. **рис. 5.11**).

Алгоритм

Поставленная выше задача потребует **полной переделки всей нашей программы**. Ведь изменится режим работы таймера. В данном конкретном случае удобнее всего использовать режим совпадения. Точнее, его подрежим «сброс при совпадении». В этом режиме таймер

сам периодически вырабатывает запросы на прерывание с заранее заданным периодом.

Все функции управления «движением огней» выполняет процедура обработки прерывания. При каждом вызове прерывания процедура производит сдвиг «огней» на один шаг в нужном направлении.

Для того, чтобы обеспечить такую же скорость движения «огней», как в предыдущем примере, мы должны использовать те же самые коэффициенты деления. Для начала необходимо включить предварительный делитель и выбрать для него коэффициент деления 1/1024.

Второй коэффициент деления (780) мы помещаем в специальный системный регистр — **регистр совпадения**. Сравнение содержимого счетного регистра с содержимым регистра совпадения будет происходить на аппаратном уровне. В **режиме «сброс при совпадении»** таймер работает следующим образом. Сразу после запуска значение счетного регистра начнет увеличиваться. Когда это значение окажется равным значению регистра совпадения, таймер автоматически сбросится и продолжит работу с нуля. В момент сброса таймера формируется запрос на прерывание.

Для имитации бегущих огней, как и в предыдущих примерах, мы будем использовать **операции сдвига**. При этом нам также понадобится специальный рабочий регистр. То есть один из регистров общего назначения, в котором будет храниться текущее состояние наших «огней». В начале программы в рабочий регистр необходимо записать исходное значение. То есть число, один из разрядов которого равен единице, а остальные — нулю. В результате операций сдвига эта единица будет перемещаться вправо или влево, создавая эффект бегущего огня. Проверка состояния кнопки и сдвиг на один шаг будет производиться при каждом вызове процедуры обработки прерывания.

Исходя из вышесказанного, алгоритм работы программы состоит из двух независимых алгоритмов. **Во-первых**, это алгоритм основной программы, а **во-вторых**, алгоритм процедуры обработки прерывания. Рассмотрим их по порядку.

Алгоритм основной программы.

1. Настроить стек и порты ввода-вывода микроконтроллера.
2. Настроить таймер и систему прерываний.
3. Записать в рабочий регистр исходное значение.
4. Разрешить работу таймера.
5. Разрешить прерывания.
6. Перейти к выполнению основного цикла.

Так как все операции, связанные с движением огней, выполняет процедура обработки прерываний, в основном цикле программы нам ничего делать не нужно. Для простоты оставим основной цикл пустым.

Алгоритм процедуры обработки прерывания.

1. Проверить состояние переключателя режимов.
2. Если контакты переключателя разомкнуты, произвести сдвиг всех разрядов рабочего регистра на один разряд вправо. Если в результате этого сдвига единичный бит выйдет за пределы байта, создать новый единичный бит в крайней левой позиции.
3. Если контакты переключателя замкнуты, произвести сдвиг всех разрядов рабочего регистра на один разряд влево. Если в результате этого сдвига единичный бит выйдет за пределы байта, создать новый единичный бит в крайней правой позиции.
4. Вывести содержимое рабочего регистра в порт PB, предварительно проинвертировав его.
5. Закончить процедуру обработки прерывания.

Программа на Ассемблере

Текст возможного варианта программы на языке Ассемблер приведен в **листинге 5.13**. В программе встречаются несколько новых для нас операторов.

Используется новый для нас флаг — флаг глобального разрешения прерываний, который называется I.

Мы уже упоминали этот флаг в **главе 3**. Флаг I, так же, как флаги C и Z, является одним из разрядов регистра SREG. Однако управление флагом I происходит совсем по-другому. На него не влияют ни арифметические, ни логические операции, а тем более операции сравнения. Для установки и сброса этого флага в системе команд предусмотрены **две специальные команды** (описаны ниже). Если флаг I сброшен, то все прерывания в микроконтроллере запрещены. Если флаг установлен, работа системы прерываний разрешается. Рассмотрим теперь по порядку все новые для нас операторы.

.dseg

Оператор выбора сегмента памяти данных. До сих пор во всех предыдущих ассемблерных программах мы обязательно использовали оператор `.cseg`, который позволял нам выбирать программный сегмент памяти. Пора научиться работать и с другими сегментами. Следующий по значению после программного сегмента — это **сегмент памяти данных**, то есть сегмент ОЗУ. В программе (**листинг 5.13**) в **строке 6** производится выбор именно этого сегмента.

.byte

Оператор резервирования памяти. Это один из операторов, которые действуют в сегменте памяти данных. Оператор позволяет зарезерви-

ровать один или несколько байтов (ячеек ОЗУ) для того, чтобы затем использовать их в программе. Вы спросите: зачем это нужно? Основная цель резервирования — учет и распределение памяти.

Если программист будет произвольно, по своему усмотрению, выбирать адреса ячеек ОЗУ для той либо иной задачи, то ему придется внимательно следить за тем, чтобы не выбрать повторно одну и ту же ячейку для хранения разных значений. Иначе программа при записи одного значения испортит второе, что приведет к ошибке в ее работе.

Механизм резервирования памяти позволяет транслятору контролировать использование памяти и исключать двойное использование ячеек. Кроме того, подобный механизм вообще избавляет программиста от необходимости запоминать адреса. Все происходит автоматически.

Оператор `.byte` имеет всего один параметр — **количество ячеек, которые нужно зарезервировать**. В нашей программе применяется лишь одна команда, резервирующая память (**строка 8, листинг 5.13**). В данном случае резервируется всего одна ячейка памяти. Метка `buf`, поставленная перед оператором, используется для обращения к зарезервированной ячейке.

reti

Оператор завершения подпрограммы обработки прерывания. Действие этого оператора аналогично действию оператора `ret`. Он извлекает адрес из стека и передает управление по этому адресу. Различие состоит в том, что команда `reti` еще и устанавливает в единицу флаг глобального разрешения прерываний I.

sts

Команда записи содержимого РОН в ОЗУ. Имеет два параметра. Первый параметр — **адрес ячейки памяти**, куда записываются данные. Второй параметр — **имя регистра** источника данных. В **строке 49** программы содержимое регистра `rab` записывается в ОЗУ по адресу, определяемому меткой `buf`.

lds

Команда чтения информации из ячейки памяти. Прочитанная информация записывается в один из РОН. Команда также имеет два параметра. Первый параметр — **имя РОН**, куда записываются считанные данные. Второй параметр — **адрес ячейки памяти (источника данных)**.

sei

Команда разрешения прерываний. Эта команда устанавливает флаг I. То есть разрешает все прерывания.

Описание программы (листинг 5.13)

Начало программы (**строки 1—5**) у вас вызывать затруднений не должно. Здесь выполняется присоединение библиотечного файла, описание двух переменных (`temp` и `rab`) и описание константы `kdel`. Подобные операции мы уже выполняли в предыдущей программе. Различия начинаются в **строке 6**.

Тут мы впервые сталкиваемся с **процедурой резервирования ячеек ОЗУ**. Правда, зарезервируем мы для начала всего одну ячейку. Процесс резервирования похож на процесс автоматического размещения команд в программной памяти (см. **раздел 5.2**). Здесь также используется указатель текущего адреса. При резервировании ячеек указатель перемещается от нулевого адреса вверх, в сторону увеличения адресов. Если очередная директива `byte` резервирует N ячеек памяти, то и указатель перемещается на N позиций.

Листинг 5.13

```
        ;################################################
        ;##                Пример 7                    ##
        ;##             "Бегущие огни"                 ##
        ;##  с использованием прерываний от таймера  ##
        ;################################################

        ;------------------------- Псевдокоманды управления

 1      .include "tn2313def.inc"               ; Присоединение файла описаний
 2      .list                                  ; Включение листинга
 3      .def  temp = R16                       ; Определение главного рабочего регистра
 4      .def  rab = R17                        ; Определение рабочего регистра для команд сдвига
 5      .equ  kdel = 780

        ;------------------------- Резервирование ячеек памяти

 6              .dseg                          ; Выбираем сегмент ОЗУ
 7              .org    0x60                   ; Устанавливаем текущий адрес сегмента

 8      buf:    .byte   1                      ; Один байт для хранения рабочего значения

        ;------------------------- Начало программного кода

 9              .cseg                          ; Выбор сегмента программного кода
10              .org    0                      ; Установка текущего адреса на ноль

        ;------------------------- Переопределение векторов прерываний

11      start:  rjmp    init                   ; Переход на начало программы
12              reti                           ; Внешнее прерывание 0
13              reti                           ; Внешнее прерывание 1
14              reti                           ; Таймер/счетчик 1, захват
15              rjmp    prtim1                 ; Таймер/счетчик 1, совпадение, канал A
16              reti                           ; Таймер/счетчик 1, прерывание по переполнению
17              reti                           ; Таймер/счетчик 0, прерывание по переполнению
18              reti                           ; Прерывание UART прием завершен
19              reti                           ; Прерывание UART регистр данных пуст
20              reti                           ; Прерывание UART передача завершена
21              reti                           ; Прерывание по компаратору
22              reti                           ; Прерывание по изменению на любом контакте
23              reti                           ; Таймер/счетчик 1. Совпадение, канал B
24              reti                           ; Таймер/счетчик 0. Совпадение, канал B
25              reti                           ; Таймер/счетчик 0. Совпадение, канал A
26              reti                           ; USI готовность к старту
27              reti                           ; USI Переполнение
28              reti                           ; EEPROM Готовность
29              reti                           ; Переполнение охранного таймера

        ; ------------------- Модуль инициализации
        init:
        ;-------------------------- Инициализация стека
```

```
30              ldi     temp, RAMEND        ; Выбор адреса вершины стека
31              out     SPL, temp           ; Запись его в регистр стека

        ;-------------------------- Инициализация портов ВВ

32              ldi     temp, 0             ; Записываем ноль в регистр temp
33              out     DDRD, temp          ; Записываем этот ноль в DDRD (порт PD на ввод)

34              ldi     temp, 0xFF          ; Записываем число $FF в регистр temp
35              out     DDRB, temp          ; Записываем temp в DDRB (порт PB на вывод)
36              out     PORTB, temp         ; Записываем temp в PORTB (потушить светодиод)
37              out     PORTD, temp         ; Записываем temp в PORTD (включаем внутр.резист.)

        ;-------------------------- Инициализация таймера T1

38              ldi     temp, 0x0D          ; Выбор режима таймера
39              out     TCCR1B, temp
40              ldi     temp, high(kdel)    ; Старший полубайт кода совпадения
41              out     OCR1AH, temp        ; Запись в регистр совпадения старш. полубайта
42              ldi     temp, low(kdel)     ; Младший полубайт кода совпадения
43              out     OCR1AL, temp        ; Запись в регистр совпадения младш. полубайта

        ;-------------------------- Определение маски прерываний

44              ldi     temp, 0b01000000    ; Байт маски. Разрешено одно прерывание (№4)
45              out     TIMSK, temp         ; Записываем маску

        ;-------------------------- Инициализация компаратора

46              ldi     temp, 0x80          ; Выключение компаратора
47              out     ACSR, temp

        ;-------------------------- Начало основной программы

48      main:   ldi     rab, 0b00010000     ; Запись начального значения
49              sts     buf, rab            ; Запись содержимого регистра rab в ОЗУ

50              sei                         ; Разрешение прерываний
51      m1:     rjmp    m1                  ; Пустой бесконечный цикл

        ;=======================================================
        ;       Подпрограмма обработки прерываний
        ;=======================================================

52      prtim1: push    temp                ; Сохраняем регистр temp
53              push    rab                 ; Сохраняем регистр rab

54              lds     rab, buf            ; Читаем содержимое rab из ОЗУ
55              in      temp, PIND          ; Считываем содержимое порта PD
56              sbrs    temp, 0             ; Проверка младшего разряда
57              rjmp    p2                  ; Если не ноль, переходим к сдвигу влево

        ;------------------------ Сдвиг вправо

58      p1:     lsr     rab                 ; Сдвиг содержимого рабочего регистра
59              brcc    p3                  ; Если не дошло до конца регистра пропустить
60              ldi     rab, 0b10000000     ; Запись начального значения
61              rjmp    p3                  ; В конец

        ;------------------------ Сдвиг влево

62      p2:     lsl     rab                 ; Сдвиг содержимого рабочего регистра
63              brcc    p3                  ; Если не дошло до конца регистра пропустить
64              ldi     rab, 0b00000001     ; Запись начального значения

        ;------------------------ Конец процедуры обработки прерывания

65      p3:     ldi     temp, 0xFF          ; Запись в temp числа $FF
66              eor     temp, rab           ; Инверсия содержимого rab (исключающее ИЛИ)
67              out     PORTB, temp         ; Вывод текущего значения в порт PB

68              sts     buf, rab            ; Запись регистра rab в ОЗУ

69              pop     rab                 ; Восстанавливаем регистр rab
70              pop     temp                ; Восстанавливаем регистр temp

71              reti
```

В нашей программе весь процесс резервирования занимает всего три **строки** (**строки 6—8**):

- в **строке 6** выбирается нужный нам сегмент памяти (сегмент памяти данных);
- в **строке 7** выбирается новое значение для указателя в этом сегменте;
- в **строке 8** происходит собственно резервирование.

Так как в **строке 7** указателю присваивается значение 0x60, то именно по этому адресу будет располагаться ячейка памяти, резервируемая в **строке 8**. Почему же мы выбрали такой адрес?

Вспомните **схему распределения памяти микроконтроллера AVR [4]**. Ячейки ОЗУ с адресами от 0 до 0x1F совмещены с файлом регистров общего назначения, ячейки с адресами 0x20—0x5F совмещены с регистрами ввода-вывода. Ячейка с адресом 0x60 — это первая ячейка ОЗУ, предназначенная исключительно для хранения данных.

Зарезервированная нами ячейка далее в программе будет использоваться в качестве буфера для хранения содержимого рабочего регистра `rab` в промежутке между двумя вызовами прерывания. Именно из этих соображений для нее выбрано имя `buf`.

Резервированием памяти заканчивается модуль определений. Далее начинается **непосредственно программный код**, то есть код, помещаемый в программную память. Поэтому мы выбираем программный сегмент памяти (**строка 9**).

В **строке 10** устанавливается начальное значение указателя для этого сегмента. Далее начинается код самой нашей программы. Но начинается код программы совсем не так, как мы уже привыкли во всех предыдущих примерах. **Строки 11—29** занимает **блок команд переопределения векторов прерываний**. До сих пор в наших программах мы не имели подобного блока команд, потому что до сих пор мы не использовали прерываний.

Напомню **определение**: ***векторами прерываний*** *называется несколько специально зарезервированных адресов в начале программной памяти, предназначенных для обслуживания прерываний.*

Микроконтроллер ATtiny2313 имеет таблицу векторов прерываний, состоящую из 19 адресов (с адреса 0x0000 по адрес 0x0012). Каждый из этих адресов, по сути, является адресом начала процедуры обработки одного из видов прерываний. Переопределение векторов состоит в том, что в каждую такую ячейку мы можем поместить команду безусловного перехода, передающую управление на адрес в программной памяти, где уже действительно начинается соответствующая процедура.

Обычно программа не использует сразу все заложенные в микропроцессор прерывания. **Например**, в нашем случае используется лишь одно прерывание — **прерывание по совпадению таймера**. Поэтому переопределение производят только для тех векторов, которые исполь-

зуются в данной программе. Однако и все остальные векторы принято не оставлять без внимания. По всем остальным адресам таблицы принято ставить команды-заглушки.

Назначение команды-заглушки: предотвратить негативные последствия в случае ошибочного вызова незадействованного прерывания. Иногда в качестве такой заглушки применяют безусловный переход по нулевому адресу. Но удобнее всего использовать команду завершения процедуры обработки прерывания (`reti`). Если ненужное нам прерывание все же сработает, то оно тут же завершится, не нанеся никакого урона.

Какие же вектора прерываний переопределяются в нашей программе? **Во-первых**, **вектор нулевого адреса**. По адресу 0x0000 (**строка 11** программы) помещается команда безусловного перехода по метке `init`. В **строке** с этой меткой начинается основная процедура нашей программы. Как известно, ***нулевой адрес*** *— это вектор начального сброса микроконтроллера*. Именно с этого адреса начинается выполнение программы после системного сброса. Безусловный переход с нулевого адреса позволяет «перепрыгнуть» таблицу векторов прерываний и разместить основную программу за пределами этой таблицы.

Второй переопределяемый вектор — это вектор прерываний по совпадению таймера/счетчика T1. Его адрес равен 0x0004. Сюда мы помещаем команду безусловного перехода на метку `prtim1` (**строка 15** программы). Именно с этой метки начинается процедура обработки данного прерывания.

По всем остальным адресам таблицы помещены команды `reti`.

Сразу за таблицей векторов прерываний начинается **модуль инициализации**. Подобный модуль нам не в новинку. Модуль инициализации обязательно входит в любую программу. Наша программа — это всего лишь новый вариант программы для уже знакомой нам схемы бегущих огней. Режимы работы большинства систем микроконтроллера не изменяются. Поэтому модуль инициализации новой программы почти полностью повторяет соответствующий модуль из предыдущего примера. В предыдущей программе (**листинг 5.11**) подобный модуль занимал **строки 8—19**.

Но есть и **отличия**. В новой программе немного по-другому происходит инициализация таймера/счетчика. Теперь таймер должен быть переведен в режим сброса при совпадении. Возможны два варианта реализации такого режима:

- сброс при совпадении в канале A;
- сброс при совпадении в канале B.

Для каждого из каналов имеется свой собственный регистр совпадения. Не буду вдаваться в подробности. Просто скажу, что **мы выберем**

канал А. Для того, чтобы перевести наш таймер/счетчик в выбранный нами режим, достаточно в регистр конфигурации таймера TCCR1B записать код 0x0D (**строки 38, 39**). Этот код не только переводит таймер в выбранный нами режим, но и устанавливает коэффициент предварительного деления, равный 1/1024.

После того, как режим таймер выбран, нужно **записать код совпадения** в соответствующий регистр. Для канала А этот регистр называется OCR1A. Он имеет шестнадцать разрядов и физически состоит из двух отдельных регистров OCR1AH и OCR1AL. В каждую из этих половинок регистра записывается своя часть кода совпадения. В регистр OCR1AH записывается старший байт (**строки 40, 41**), а в регистр OCR1AL — младший байт (**строки 42, 43**) кода. Данный регистр совпадения обладает **свойством двойной буферизации**. Поэтому и здесь важен порядок записи двух его половинок. Сначала нужно записывать старший байт кода, а затем младший.

После инициализации таймера необходимо **инициализировать систему прерываний**. Инициализация системы прерываний сводится к выбору нового значения маски прерываний по таймеру. Значение маски записывается в регистр TIMSK. В данном случае нам нужно разрешить лишь один вид прерываний: прерывания по совпадению в канале А. Для этого соответствующий выбранному прерыванию бит в байте маски должен быть установлен в единицу.

Остальные биты должны оставаться равными нулю. Запись маски производится в **строках 44 и 45**. Во всем остальном новый модуль инициализации полностью соответствуют аналогичному модулю в программе из предыдущего примера (**листинг 5.11**).

За модулем инициализации начинается **основная программа**. В нашем случае она занимает всего четыре **строки** (**строки 48—51**). В **строках 48, 49** происходит присвоение начального значения рабочему регистру `rab` и сохранение этого значения в буфере `buf`. Как и в предыдущих примерах, рабочий регистр будет использоваться для операций сдвига, имитирующих движение нашего «огня».

Начальное значение должно представлять собой двоичное число, один двоичный разряд которого равен единице, а все остальные — нулю. Затем процедура обработки прерывания будет двигать этот разряд вправо и влево. Поэтому будет логично, если первоначально нашу единичку мы расположим где-то посередине. То есть выберем в качестве начального значения, **например**, число 0b00010000. Что и сделано в **строке 48**.

Так как мы договорились, что в промежутках между двумя прерываниями эта величина будет храниться в буфере `buf`, в **строке 49** содержимое `rab` помещается в этот буфер. Теперь все готово к запуску

системы прерываний. В **строке 50** находится команда, разрешающая все прерывания. **Обратите внимание**, что к этому моменту наш таймер/счетчик уже находится в режиме счета. Он начал работать сразу после записи значения в регистр TCCR1B. Однако все прерывания до сих пор были запрещены. Теперь, когда прерывания мы разрешили, система бегущих огней сразу начинает работать.

В **строке 51** основная программа завершается. Так как все операции по управлению движением «огней» выполняет процедура обработки прерывания, то основной программе больше ничего делать не нужно. Поэтому в **строке 51** организован бесконечный цикл. Он представляет собой безусловный переход сам на себя. Попав в такой цикл, программа будет бесконечно выполнять один и тот же оператор.

Немного о том, **как происходит вызов прерывания**. Таймер/счетчик непрерывно производит подсчет тактовых импульсов системного генератора. В момент, когда содержимое счетного регистра совпадет с содержимым регистра OCR1A, счетчик сбрасывается и начинает счет сначала. При очередном совпадении все повторяется. В момент сброса счетчика вызывается прерывание. Таким образом, процедура обработки прерывания выполняется периодически, каждый раз, когда счетчик досчитает до момента совпадения.

Коэффициент предварительного деления и величину кода совпадения мы выбрали таким образом, что период, с которым происходит вызов прерывания, равен 200 мс. То есть соответствует нашему техническому заданию. Процедура обработки прерывания заканчивается гораздо быстрее. Время выполнения этой процедуры примерно равно 6 мкс. Поэтому к тому времени, когда прерывание будет вызвано повторно, процедура обработки предыдущего прерывания уже давно закончится.

Теперь **перейдем к самой процедуре обработки прерывания**. Текст этой процедуры занимает **строки 52—71**. Начинается процедура с сохранения всех регистров, которые она в дальнейшем будет использовать (**строки 52, 53**). Как видите, мы сохраняем даже регистр `rab`. Теперь, в случае необходимости, наша основная программа сможет использовать этот регистр для своих целей. Так как в промежутке между двумя прерываниями содержимое `rab` хранится в буфере ОЗУ, то в **строке 54** мы извлекаем это значение из буфера и помещаем в `rab`.

Теперь все готово к **операции сдвига**. Но сначала нам нужно определить направление этого сдвига. Для этого достаточно проверить состояние контактов переключателя. Проверка производится в **строках 55—57**. В **строке 55** читается содержимое порта PD и записывается в регистр `temp`. В **строке 56** проверяется значение младшего разряда считанного значения. Если значение этого разряда равно единице

(контакты переключателя разомкнуты), то **строка 57** пропускается, и программа переходит к процедуре сдвига вправо, которая начинается в **строке 58**. Если контакты замкнуты, то выполняется безусловный переход в **строке 57**, и управление передается по метке `p2`, где начинается процедура сдвига влево.

Процедура сдвига вправо занимает **строки 58—61**. Собственно сдвиг происходит в **строке 58**. В **строке 59** происходит проверка, не дошла ли в результате этого сдвига сдвигаемая единица до последнего разряда. Так же, как и в предыдущих примерах, признаком достижения конечной позиции служит появление единицы в флаге переноса (вспомните **табл. 5.2**).

Проверка флага переноса производится в **строке 59**. Если значение флага равно нулю, **строка 60** программы пропускается. Если же значение флага окажется равным единице, то команда в **строке 60** записывает в регистр `rab` новое значение. После записи этого значения единица окажется в самом старшем разряде. Таким образом организуется движение единицы по кругу (дойдя до крайней правой позиции, единица появляется слева).

В **строке 61** процедура сдвига вправо завершается. Управление передается по метке `p3`. То есть к процедуре вывода сдвинутого значения в порт.

Процедура сдвига влево (**строки 62—64**) работает аналогично предыдущей процедуре. Отличие состоит лишь в том, что здесь применяется другая команда сдвига (**строка 62**). Кроме того, при достижении крайней позиции регистру присваивается другое начальное значение. Теперь единица окажется в самом младшем разряде. Таким образом организуется кольцевое движение, но в другую сторону.

В **строке 65** начинается процедура вывода содержимого `rab` в порт PB. Процедура занимает **строки 65—67**. Точно такая же процедура применялась и в двух предыдущих версиях программы бегущих огней.

В **строках 68—70** происходит подготовка к завершению процедуры обработки прерывания. Сначала содержимое `rab` сохраняется в буфере ОЗУ (**строка 68**). Затем в **строках 69, 70** восстанавливаются значения регистров `temp` и `rab`. И, наконец, в **строке 71** процедура обработки прерывания завершается.

Программа на языке СИ

Как мы убедились на примере Ассемблера, для нашей новой задачи потребуются довольно **значительные изменения программы**. При разработке программы средствами системы CodeVisionAVR в подобной ситуации целесообразнее воспользоваться **мастером**, при помощи которого удобно создать новую заготовку программы.

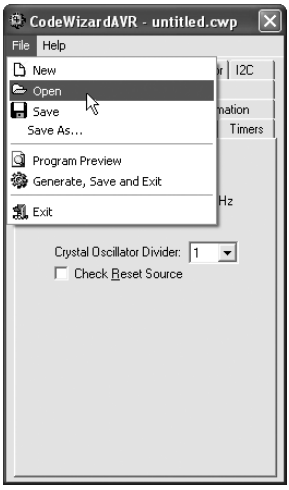


Рис. 5.12. *Загрузка параметров мастера*

Для создания новой заготовки программы удобно восстановить настройки из созданного ранее примера, подкорректировать их в соответствии с новыми требованиями и записать под новым именем. Для этого зайдем в программу CodeVisionAVR и запустим мастер. Для запуска мастера достаточно нажать на панели инструментов кнопку .

После запуска мастера все его управляющие элементы будут находиться в исходном состоянии. То есть иметь **значения по умолчанию**. Теперь нам нужно восстановить настройки из самого первого примера (см. **раздел 5.2**).

Для этого в меню «File» мастера выберем пункт «Open», как показано на **рис. 5.12**. Появится стандартное окно открытия файла. Найдите на вашем диске файл Prog1.cwp и откройте его. После того, как вы его откроете, все органы управления во всех вкладках мастера примут те значения, какие они имели при создании программы Prog1 (см. **рис. 5.3—5.6**).

Теперь нужно **сохранить этот пример с другим именем**. Для этого при помощи пункта меню «File / Save As» сделайте копию только что загруженного файла. Копию поместите в новую директорию с именем Prog7. В этой директории будет храниться наш новый проект. Теперь можно приступать к изменениям настроек под новые требования.

Исходя из технического задания, нам придется **изменить лишь настройки нашего таймера**. Для этого сначала откройте вкладку «Timers». На этой вкладке вы увидите еще три вкладки поменьше (см. **рис. 5.13**). Эти вкладки предназначены для настройки двух системных таймеров (Timer0 и Timer1), а также охранного таймера (Watchdog). Открываем вкладку «Timer1», как показано на **рис. 5.13**. При помощи расположенных там элементов управления выставляем следующие настройки.

- **Поле «Clock Source» (Источник Тактового Сигнала)** оставляем в положении «System Clock» (Тактовый сигнал от системного генератора).
- В **поле «Clock Value» (Значение тактовой частоты)** выбираем значение 3,906 кГц. На самом деле мы выбираем коэффициент деления предварительного делителя. Но для удобства программиста показаны получаемые при этом частоты сигнала. Значения этих частот вычисляются исходя из выбранной частоты тактового генератора (см. вкладку «Chip»). Как вы помните, тактовая частота в нашем случае равна 4 МГц.

- В **поле «Mode»** выбираем режим работы таймера. В нашем случае мы должны выбрать режим «CTC top=OCR1A» (режим сброса при совпадении с регистром OCR1A). Два поля «Out. A» и «Out. B» оставляем без изменений. При помощи этих полей можно включить и настроить параметры вывода сигнала совпадения на внешние контакты микроконтроллера. В данном случае нам это не нужно.
- **Поле «Input Capt.» (Вход Захвата)** используется при работе в режиме захвата. Мы не используем режим захвата, поэтому и это поле тоже оставим без изменения.
- **Поле «Interrupt On» (Прерывание От)** нам нужно изменить. Это поле предназначено для разрешения или запрета различных видов прерываний от таймера. В правой части поля имеются две маленькие кнопочки, при помощи которых вы можете листать его содержимое. В процессе перелистывания в окошке появляются названия видов прерываний.
- Слева от названия каждого вида прерывания имеется **поле выбора («Check Box»)**, при помощи которого вы можете включить либо выключить данное прерывание. Путем перелистывания найдите прерывание, которое называется «Compare A Mach» (при совпадении с A). Поставьте «птичку» в соответствующем поле «Check Box». Теперь нужное прерывание будет включено.
- **Поля «Value» и «Inp. Capture»** оставим без изменений. Эти поля предназначены для задания начальных значений счетного регистра и содержимого регистра захвата.
- И, наконец, **два поля под общим названием «Comp.»** позволяют выставить значения, записываемые в регистры совпадения A и B. Мы будем использовать регистр A. Запишем в поле «A:» значение нужного нам коэффициента деления (3D0). В данном случае допускается только шестнадцатиричный формат.

В результате всех этих манипуляций окно настройки таймера должно выглядеть так, как показано на **рис. 5.13**. Сохраните новое значение параметров при помощи меню «File / Save» мастера.

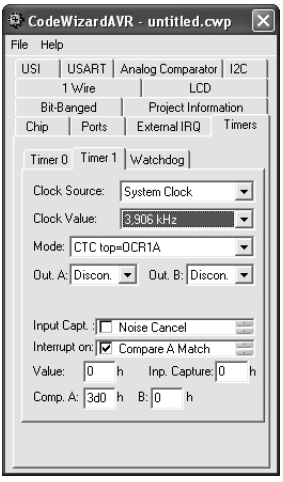


Рис. 5.13. *Настройка таймера*

Теперь можно **приступать к процессу формирования нового проекта**. Этот процесс был подробно описан в **разделе 5.1**, поэтому я повторяться не буду.

После того, как проект сформирован, приступаем **к созданию новой программы**.

Возможный вариант такой программы приведен в **листинге 5.14**. В новой программе в основном использованы уже знакомые нам операторы. Из новых операторов появился лишь один.

#asm()

Это специальная системная функция, позволяющая в программе на СИ выполнять команды Ассемблера. Параметр функции — это строковая переменная или строковая константа, значение которой представляет собой текст команды на Ассемблере. Подобная функция добавляет программе значительную гибкость. В **строке 31** программы (**листинг 5.14**) при помощи функции `#asm()` выполняется ассемблерная команда разрешения прерывания `sei`. Команды такого уровня не имеют аналогов в традиционном синтаксисе языка СИ. Да и нецелесообразно выдумывать новые команды, когда удобнее просто выполнить команду Ассемблера.

Описание программы (листинг 5.14)

Текст программы, сформированный мастером, содержит две функции. Вернее, еще не функции, а их заготовки. **Во-первых**, это главная функция `main`, которая в уже готовой доработанной программе занимает **строки 11—42.**

Листинг 5.14

```
        /*****************************************************
        This program was produced by the
        CodeWizardAVR V1.24.4 Standard
        Automatic Program Generator
        © Copyright 1998-2004 Pavel Haiduc, HP InfoTech s.r.l.
        Project : Prog 7
        Comments: Бегущие огни с использованием прерываний
        Chip type           : ATtiny2313
        Clock frequency     : 4,000000 MHz
        Memory model        : Tiny
        External SRAM size  : 0
        Data Stack size     : 32
        *****************************************************/

1       #include <tiny2313.h>
        // Описание глобальных переменных
2       unsigned char rab;

        // Прерывание по совпадению таймера T1

3       interrupt [TIM1_COMP] void timer1_comp_isr(void)
4           {
            if (PIND.0==1)  // Проверка состояния переключателя
5               {
6               rab = rab >> 1;  // Сдвиг разрядов
                if (rab==0) rab = 0b10000000;  // Если дошло до конца
                }
7           else
8               {
9               rab = rab << 1;  // Сдвиг разрядов
                if (rab==0) rab = 0b00000001;  // Сдвиг влево
                }
10          PORTB=rab^0xFF;  // Запись в порт с инверсией
            }
11      void main(void)
```

```
    {
12      CLKPR=0x80; // Отключить деление частоты системного генератора
13      CLKPR=0x00;

14      PORTA=0x00;  // Инициализация порта A
15      DDRA=0x00;

16      PORTB=0xFF;  // Инициализация порта B
17      DDRB=0xFF;

18      PORTD=0x7F;  // Инициализация порта D
19      DDRD=0x00;

20      TCCR0A=0x00; // Инициализация таймера T0
21      TCCR0B=0x00;
22      TCNT0=0x00;
23      OCR0A=0x00;
24      OCR0B=0x00;

25      TCCR1A=0x00; // Инициализация таймера T1
26      TCCR1B=0x0D;
27      TCNT1H=0x00;
28      TCNT1L=0x00;
29      ICR1H=0x00;
30      ICR1L=0x00;
31      OCR1AH=0x03;  // Наш коэффициент деления (030CH = 780)
32      OCR1AL=0x0C;  //
33      OCR1BH=0x00;
34      OCR1BL=0x00;
35      GIMSK=0x00;   // Инициализация внешних прерываний
36      MCUCR=0x00;

37      TIMSK=0x40;   // Запись маски прерываний

38      USICR=0x00;   // Инициализация универсального последовательного интерфейса

39      ACSR=0x80;    // Инициализация аналогового компаратора

40      rab=0b00010000;  // Присвоение начального значение переменной rab
41      #asm("sei")      // Команде разрешения прерываний
42      while (1) {};
    }
```

Кроме главной функции, первоначальная автоматически сформированная программа содержит заготовку **еще одной функции — функции обработки прерывания**. В **строках 3—10** мы можем видеть ее в уже доработанном виде. В первоначальном виде функция `main` содержит только **строки** инициализации (**строки 12—39**), а функция обработки (функция «`timer1_comp_isr`») вообще не содержит ни одного оператора.

Посмотрите внимательно на текст программы. Описание функции `timer1_comp_isr` (**строка 3** программы) отличается от всех встречавшихся нам до сих пор описаний. Слева от стандартного описания функции добавлены еще два дополнительных элемента. **Первый —** это слово `Interrupt` (прерывание). Это управляющее слово указывает транслятору на то, что данная функция является процедурой обработки прерывания. Вид прерывания, которое будет вызывать данную функцию, указывается сразу за словом `Interrupt` в квадратных скобках. Выражение `Interrupt [TIM1_COMP]` означает, что данная функция является процедурой обработки прерывания по совпадению таймера T1.

Имя функции обработки прерывания, автоматически данное мастером (`timer1_comp_isr`), не является обязательным. Если вы пожелаете разработать программу без участия мастера, вы можете выбрать имя

для вашей функции по своему усмотрению. Вы можете также поменять имя и в нашей автоматически сформированной программе.

Функция обработки прерывания, как и функция `main`, не должна иметь параметров и не возвращает никаких значений. Поэтому перед именем функции и в круглых скобках всегда стоит слово `void`.

Теперь посмотрим, **что же мы изменили в программе вручную**:

- из автоматически сформированной программы были удалены все лишние комментарии, а вместо них были добавлены другие, на русском языке;
- в программу были внесены новые команды и операторы, реализующие нужные нам алгоритмы.

И первое, что нам пришлось сделать, — это создать переменную `rab`. Эта переменная нам будет нужна для операций сдвига, и использоваться она будет как в функции `main`, так и в функции обработки прерывания. То есть переменная должна быть глобальной. Поэтому описание этой переменной мы поместили вне обеих функций в самом начале программы (**строка 2**).

Функция `main` претерпела наименьшие изменения. Потребовалось лишь добавить команду присвоения начального значения переменной `rab` (**строка 40**) и команду разрешения прерываний (**строка 41**). Главный же цикл программы оставлен пустым, как и в программе на Ассемблере.

Доработанная подобным образом функция `main` работает точно так же, как основная программа в программе на Ассемблере. То есть после выполнения команд конфигурации и разрешения прерывания таймер/счетчик будет запущен в режиме сброса при совпадении. Каждые 200 мс он будет вызывать процедуру обработки прерывания, то есть функцию `timer1_comp_isr`.

Теперь посмотрим, **как доработана функция `timer1_comp_isr`.** Основная часть доработок пришлась именно на нее. Новые команды, составляющие тело функции, занимают в программе **строки 4—10**. Функция обработки прерывания так же, как и соответствующая процедура в программе на Ассемблере, занимается сдвигом содержимого переменной `rab` на один шаг влево или вправо и выводом полученного значения в порт PB. Но перед тем, как выполнить сдвиг, нужно определить направление этого сдвига.

Для этого нужно проверить состояние переключателя. Для проверки состояния переключателя служит команда `if` (**строка 4**). Эта команда проверяет значение младшего разряда порта PD. Если значение разряда равно единице (контакты переключателя разомкнуты), то выполняется процедура сдвига на один бит вправо (**строки 5, 6**). Если младший бит PD равен нулю (контакты переключателя замкнуты), то выполняется процедура сдвига на один бит влево (**строки 8, 9**).

Рассмотрим подробнее процедуры сдвига. Сдвиг на один бит вправо выполняется в **строке 5**. Оператор `if` в **строке 6** проверяет, не дошла ли сдвигаемая единица до конца байта. Признаком того, что единица уже дошла до конца, является равенство переменной `rab` нулю. Если условие выполняется, то переменной `rab` будет присвоено значение 0b10000000. То есть дойдя до правого края, единица появляется слева. Таким образом реализуется эффект кругового движения единичного бита.

Обратите внимание, что в команде `if` в **строке 6** не используются фигурные скобки. Язык СИ допускает опускать фигурные скобки только в том случае, когда в них заключен всего один оператор.

Процедура сдвига на один бит влево (**строки 8, 9**) выполнен аналогичным образом. Я думаю, что тут вы легко разберетесь сами.

После выполнения одной из вышеописанных процедур сдвига производится запись содержимого переменной `rab` в порт PB с одновременным инвертированием этого содержимого (**строка 10**). Записанное в этой **строке** выражение нам уже хорошо знакомо. Оно применялось нами в обоих предыдущих примерах.

5.9. Формирование звука

Постановка задачи

В общем случае задача формирования звука не составляет большого труда. Достаточно взять за основу схему с мигающим светодиодом (см. **раздел 5.5**), подключить вместо светодиода звуковой излучатель (например, телефонный капсюль), а в соответствующей программе (**листинг 5.7**) поменять константу задержки таким образом, чтобы частота «мигания» повысилась и достигла звукового диапазона.

Диапазон частот, которые может услышать человек, лежит в пределах примерно от 50 Гц до 15 кГц. Светодиод в упомянутой выше программе мигает с частотой 4 Гц. Если уменьшить время задержки в 1000 раз, то можно получить частоту сигнала на выходе, равную 4 кГц. Эта частота как раз входит в звуковой диапазон.

Предлагаемый выше способ формирования звукового сигнала реализует эту задачу программным путем. Однако для формирования звука гораздо удобнее использовать таймеры/счетчики микроконтроллера. **Попробуем создать простейшее сигнальное устройство, которое при нажатии разных клавиш будет издавать звуки разной частоты.**

Допустим, мы имеем семь кнопок (датчиков). Сформулируем задачу следующим образом:

«Разработать электронное устройство, имеющее семь входов и один звуковой выход. К каждому из входов подключен датчик, состоящий их двух нормально разомкнутых контактов. При замыкании контактов любого из датчиков устройство должно вырабатывать звуковой сигнал определенной частоты. Каждому датчику должна соответствовать своя собственная частота звукового сигнала. Если контакты всех датчиков разомкнуты, звуковой сигнал на выходе должен отсутствовать. Назовем наше устройство Сигнализатор «Семь нот».»

Схема

Поставленная выше задача прекрасно решается при помощи уже известного нам микроконтроллера ATtiny2313. Выберем его и на этот раз. Микроконтроллер имеет два встроенных таймера/счетчика. Какой же из таймеров использовать нам? Для формирования звука лучше подходит шестнадцатиразрядный таймер. Чем больше разрядов, тем с большей точностью можно выбирать его коэффициент деления.

Это очень **важно для создания нотного стана**. Поэтому для формирования звука выберем шестнадцатиразрядный таймер T1. Теперь определимся **с режимом работы** нашего таймера. Как и в случае с бегущими огнями, для генерации звука удобнее всего использовать режим CTC (сброс по совпадению). Нам просто нужно выбрать такой коэффициент деления, чтобы на выходе таймера получить колебания в звуковом диапазоне частот.

Прежде всего, нам нужно отказаться от предварительного деления. Если частота кварцевого генератора и код, помещаемый в регистр совпадения, останутся такими же, как в предыдущем примере (в программе «Бегущие огни»), то в новом варианте частота повысится более чем в тысячу раз и как раз попадет в нужный нам диапазон.

Теперь определимся с тем, как наш **сигнал будет попадать на внешний вывод микроконтроллера**. Конечно, это можно сделать программно, при помощи процедуры обработки соответствующего прерывания. Но микроконтроллер предусматривает прямой вывод сигнала на один из своих выходов. Причем предусмотрены отдельные выходы для каждого из каналов совпадения. Для канала A подобный выход называется OC1A. Он совмещен с третьим разрядом порта PB и является альтернативной функцией данного контакта.

Подключение и отключение сигнала совпадения к внешнему выводу OC1A производится программным путем. Это позволяет программе в нужный момент включать или выключать звук. Так как для вывода звука мы будем использовать один из разрядов порта PB, то для подключения датчиков воспользуемся другим портом. А именно портом

PD. Вариант принципиальной схемы описанного выше устройства показан на **рис. 5.14**.

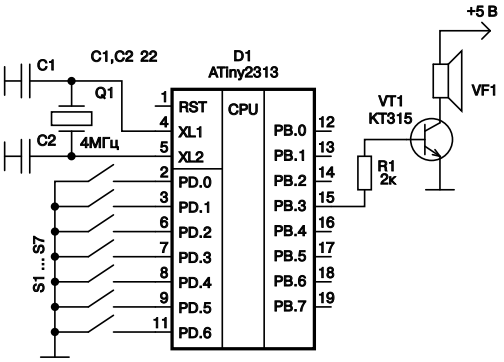


Рис. 5.14. *Схема сигнализатора «Семь нот»*

Как видно из рисунка, мы снова применили внешний кварцевый резонатор (Q1), естественно, не забыв при этом цепи согласования (C1, C2). При подключении датчиков используется та же схема, что использовалась до сих пор для подключения контактов переключателя. Датчики подключаются ко всем разрядам порта PD. При этом для правильной работы датчиков для каждого разряда порта PD должны быть активизированы встроенные резисторы нагрузки.

Для подключения звукоизлучателя (динамика) применяется ключевой каскад на транзисторе VT1. Это самый простой способ получить звук достаточной громкости, учитывая, что наш сигнал — это прямоугольные импульсы с амплитудой, почти равной напряжению питания. Транзисторный каскад нужен лишь для повышения нагрузочной способности.

Однако подобная схема имеет и свой **недостаток**. В отсутствие звукового сигнала на выходе 15 микроконтроллера обязательно нужно установить низкий логический уровень. Высокий логический уровень приведет к тому, что транзистор VT1 будет постоянно открыт. Это вызовет недопустимо большой ток через головку VF1. Постоянно протекающий ток через обмотку динамика вызовет излишнюю потерю мощности и может даже вызвать выход из строя как транзистора, так и динамика. При составлении программы мы должны учесть этот момент.

Алгоритм

На первый взгляд алгоритм такого устройства очень простой. При замыкании контактов любого из датчиков микроконтроллер должен загрузить в регистр совпадения нужный коэффициент и подключить выход таймера к выводу OC1B. При размыкании контактов датчика микроконтроллер должен отключить сигнал от внешнего вывода OC1B и подать на него низкий логический уровень. Если контакты всех датчиков разомкнуты, то внешний вывод должен оставаться отключенным.

Однако схема построена таким образом, что ничто не мешает одновременно замкнуться сразу нескольким контактам. Возникает **вопрос**: что делать в этом случае? Самый правильный **ответ** — обеспечить систему приоритетов. При замыкании нескольких контактов про-

грамма должна реагировать лишь на один из них. На тот, приоритет которого выше.

Обычно в таких случаях используется следующий прием. Программа поочередно проверяет состояние всех датчиков, **например**, справа налево. Обнаружив первый же замкнутый контакт, программа прекращает сканирование и выдает звуковой сигнал, соответствующий этому датчику.

Договоримся, что датчику, подключенному к входу PD.0, будет соответствовать нота «До». Следующему датчику — нота «Ре», и так далее до ноты «Си». Коэффициенты деления для каждой из нот выбираются по законам музыкального ряда. Подробнее о выборе коэффициентов деления для синтезатора музыкального ряда можно узнать из **[6]**.

Программа на Ассемблере

Возможный вариант программы на Ассемблере показан на **листинге 5.15**. В программе использованы следующие **новые для нас команды.**

brcs

Условный переход по признаку переноса. Выполняет передачу управления в случае, если признак переноса C равен единице. Данная команда является полной противоположностью уже знакомой нам команды `brcc`, которая, напротив, вызывает переход при отсутствии переноса.

add

Арифметическая операция сложения. Производит сложение содержимого двух РОН. Эта команда имеет два операнда, в качестве которых выступают имена складываемых регистров. В **строке 54** программы (**листинг 5.15**) к содержимому регистра `count` прибавляется содержимое регистра ZL. Результат помещается в ZL.

adc

Сложение с переносом. Этот оператор тоже выполняет сложение. Но в процессе сложения он учитывает перенос, возникший в предыдущей операции сложения. Команда сложения с учетом переноса используется при составлении программ, позволяющих складывать большие числа. Если каждое из слагаемых занимает больше, чем один байт, то входящие в них байты складывают в несколько этапов.

Сначала складывают младшие байты, а затем старшие. При сложении младших байтов может возникнуть бит переноса (если результат оказался больше, чем 0xFF). Этот перенос и нужно учесть при сложении старших байтов. Команда `adc` производит сложение содержимого двух регистров, имена которых указаны в качестве ее операндов.

К полученной сумме добавляется значение признака переноса. В **строке 56** программы к содержимому регистра `temp` прибавляется содержимое регистра `ZH` с учетом значения признака переноса. Результат помещается в `ZH`.

clr

Сброс всех разрядов РОН. Команда имеет один параметр — имя РОН, разряды которого нужно сбросить. Действие этой команды равносильно записи в РОН числа 0x00.

mov

Передача данных между двумя РОН. Эта команда имеет два операнда. Первый операнд — имя регистра, получателя данных. Второй операнд — имя регистра-источника. Команда копирует содержимое одного регистра в другой.

lpm

Чтение байта данных из программной памяти. Микроконтроллеры AVR имеют отдельную память данных и отдельную память программ. Однако некоторые виды данных удобно хранить в памяти программ. К таким данным относятся наборы различных констант. В нашем случае в памяти программ удобно хранить набор коэффициентов деления для всех наших нот. Для извлечения данных из памяти программ используется команда `lpm`.

Хранение данных в программной памяти имеет свои **особенности**. Дело в том, что память программ состоит из набора шестнадцатиразрядных ячеек. Коды команд также имеют шестнадцать разрядов. Данные же нужно хранить в виде отдельных байтов. То есть в виде восьмиразрядных двоичных чисел.

Для того, чтобы эффективнее использовать программную память, она организована таким образом, что в каждой шестнадцатиразрядной ячейке программной памяти можно хранить два разных байта данных. Команда `lpm` может читать каждый такой байт по отдельности. Для этого используется альтернативная адресация. Благодаря альтернативной адресации, программная память в режиме чтении данных имеет в два раза больше ячеек, чем при чтении кодов команд.

Это нужно учитывать при использовании команды `lpm`. Если вы знаете адрес размещения данных согласно основного способа адресации, прежде, чем использовать его в команде `lpm`, ее значение необходимо умножить на два, чтобы получить адрес того же байта в альтернативной адресации. В команде `lpm` нет операнда, определяющего адрес ячейки, содержимое которой требуется прочитать. Этот адрес предварительно должен быть записан в регистровую пару Z.

Команда `lpm` имеет три модификации. Ниже приведен формат всех трех модификаций этой команды:

```
lpm
lpm  Rd, Z
lpm  Rd, Z+
```

Первая версия команды не имеет никаких операндов. Выполняя эту команду, микроконтроллер читает содержимое ячейки программной памяти, адрес которой записан в регистровой паре Z, и помещает прочитанную информацию в регистр R0. Напоминаю, что в Z нужно помещать адрес ячейки в альтернативной адресации.

Вторая версия команды имеет два операнда:

- первый операнд — это имя регистра, куда будет помещен считанный байт;
- второй операнд всегда равен Z.

Новая версия команды работает так же, как предыдущая. Различие только в том, что прочитанная этой командой информация помещается в РОН, имя которого указано в качестве первого параметра.

Третья версия команды является модификацией второй. Она тоже имеет два операнда:

- первый операнд — это имя регистра, куда помещается прочитанный байт;
- второй операнд всегда равен Z+.

От второго варианта третий отличается тем, что сразу после чтения байта происходит автоматическое увеличение содержимого регистровой пары Z на единицу. Данную команду удобно использовать для последовательного чтения ряда констант из программной памяти. При каждом последующем вызове команда будет читать следующую константу.

Первая модификация команды **использовалась** в старых версиях микроконтроллеров и оставлена для совместимости. В новых микроконтроллерах гораздо удобнее пользоваться второй и третьей модификациями.

.dw

Директива описания данных. При помощи этой директивы описываются данные, помещаемые в память программ или в энергонезависимую память. Оператор `.dw` описывает «слова» данных. То есть шестнадцатиричные числа, каждое из которых записывается в память в виде пары байтов. **В правой части**, сразу после оператора, помещается список чисел через запятую. При трансляции программы эти числа помещаются в программную память (или в EEPROM) одно за другим так же, как туда помещаются команды.

В **строке 63** программы (**листинг 5.15**) в программную память помещается **набор коэффициентов деления**. Создается своеобразная **таблица** коэффициентов деления. Адрес начала таблицы соответствует метке `tabkd`. С точки зрения основной адресации, каждый коэффициент занимает одну шестнадцатиразрядную ячейку памяти.

С точки зрения альтернативной адресации, ***каждый коэффициент*** — *это два байта данных, записываемых в две соседние ячейки*. Причем сначала записывается младший байт, а затем старший. При чтении этих данных используется альтернативная адресация. Если мы будем последовательно читать таблицу, начиная с адреса `tabkd*2`, мы получим следующую цепочку данных:

```
0x8C, 0x12, 0x84, 0x10, 0xB8, 0x0E, 0xE4,
0x0D, 0x60, 0x0C, 0x36, 0x0B, 0xD2, 0x09.
```

Если вам не понятно, почему мы получим именно такую цепочку, вспомните, что первое число таблицы 4748 в шестнадцатиричном виде выглядит как 0x128C. То есть его старший байт равен 0x12, а младший — 0x8C. Шестнадцатиричное значение второго члена таблицы 4228 равно 0x1084. И так далее.

inc

Инкремент. Это очень простая команда. Она увеличивает содержимое одного из регистров общего назначения на единицу. У этой команды всего один параметр — имя регистра, содержимое которого нужно увеличить.

Листинг 5.15

```
        ;################################################
        ;##                Пример 8                    ##
        ;##    Сигнальное устройство "Семь нот"        ##
        ;################################################

        ;------------------------ Псевдокоманды управления

 1      .include "tn2313def.inc"                ; Присоединение файла описаний
 2      .list                                   ; Включение листинга

 3      .def        temp = R16                  ; Определение регистра передачи данных
 4      .def        count = R17                 ; Определение регистра счетчика опроса клавиш

        ;------------------------ Начало программного кода

 5              .cseg                       ; Выбор сегмента программного кода
 6              .org      0                 ; Установка текущего адреса на ноль

 7      start:  rjmp      init              ; Переход на начало программы
 8              reti                        ; Внешнее прерывание 0
 9              reti                        ; Внешнее прерывание 1
10              reti                        ; Таймер/счетчик 1, захват
11              reti                        ; Таймер/счетчик 1, совпадение, канал A
12              reti                        ; Таймер/счетчик 1, прерывание по переполнению
13              reti                        ; Таймер/счетчик 0, прерывание по переполнению
14              reti                        ; Прерывание UART прием завершен
15              reti                        ; Прерывание UART регистр данных пуст
16              reti                        ; Прерывание UART передача завершена
17              reti                        ; Прерывание по компаратору
```

```
18              reti                    ; Прерывание по изменению на любом контакте
19              reti                    ; Таймер/счетчик 1. Совпадение, канал B
20              reti                    ; Таймер/счетчик 0. Совпадение, канал B
21              reti                    ; Таймер/счетчик 0. Совпадение, канал A
22              reti                    ; USI готовность к старту
23              reti                    ; USI Переполнение
24              reti                    ; EEPROM Готовность
25              reti                    ; Переполнение охранного таймера


        ; ------------------- Модуль инициализации

26      init:   ldi     temp, RAMEND    ; Инициализация стека
27              out     SPL, temp

        ;-------------------------- Инициализация портов B/B

28              ldi     temp, 0x08      ; Инициализация порта PB
29              out     DDRB, temp

30              ldi     temp, 0x00
31              out     DDRD, temp      ; Инициализация порта PD
32              out     PORTB, temp     ; Выходное значение порта PB

33              ldi     temp, 0x7F
34              out     PORTD, temp     ; Включение внутренних резисторов

        ;-------------------------- Инициализация компаратора
35              ldi     temp, 0x80
36              out     ACSR, temp

        ;-------------------------- Инициализация таймера/счетчика
37              ldi     temp, 0x09
38              out     TCCR1B, temp    ; Выбор режима
39      m1:     ldi     temp, 0x00
40              out     TCCR1A, temp    ; Отключение звука


        ;-------------------------- Начало основного цикла программы

41      main:   clr     count                   ; Обнуление счетчика опроса клавиш
42              in      temp, PIND              ; Чтение порта D
43      m2:     lsr     temp                    ; Сдвигаем входной байт
44              brcc    m3                      ; Если текущий разряд был равен 0
45              inc     count                   ; Увеличиваем показание счетчика
46              cpi     count, 7                ; Сравнение (7 – конец сканирования)
47              brne    m2                      ; Если не конец, продолжить
48              rjmp    m1                      ; Если не одна клавиша не нажата

49      m3:     lsl     count                   ; Умножение номера кнопки на 2
50              mov     YL, count               ; Создаем первое слагаемое
51              ldi     YH, 0                   ; Старший его байт равен нулю
52              ldi     ZL, low(tabkd*2)        ; Второе слагаемое – начало таблицы tabkd
53              ldi     ZH, high(tabkd*2)
54              add     ZL, YL                  ; Складываем два 16-разр. слагаемых
55              adc     ZH, YH

56              lpm     YL, Z+                  ; Читаем младший байт коэфф. деления
57              lpm     YH, Z                   ; Читаем младший байт коэфф. деления
58              out     OCR1AH, YH              ; Вспомогательное значение
59              out     OCR1AL, YL              ; Записать в старш. часть регистра совпадения

60              ldi     temp, 0x40              ; Включить звук
61              out     TCCR1A, temp
62              rjmp    main

        ; ***********************************************************
        ; **           Таблица коэффициентов деления               **
        ; ***********************************************************

63      tabkd:  .dw     4748, 4228, 3768, 3556, 3168, 2822, 2514
```

Описание программы (листинг 5.15)

Четыре первые строки программы (**строки 1—4**), я думаю, пояснений не требуют. Все эти команды знакомы нам по предыдущему

примеру. **Строки 5—25** занимает таблица переопределения векторов прерываний. Мы применили эту таблицу, несмотря на то, что данная программа не использует прерываний. Именно поэтому во всех ячейках таблицы, кроме ячейки с нулевым адресом, поставлены **команды-заглушки**. Для серьезных проектов применение таблицы считается обязательным. Это повышает надежность работы программы, а также повышает ее наглядность.

Строки 26—40 занимает **модуль инициализации**. Начинается он с инициализации стека (**строки 26, 27**). Затем производится инициализация портов ввода-вывода. Команды инициализации портов занимают **строки 28—34**.

Порт PB настраивается таким образом, что линия PB.3 работает в режиме вывода информации, а остальные линии — в режиме ввода. Все разряды порта PD настраиваются на ввод. В регистр PORTB записывается нулевое значение. При этом на выходе PB.3 появляется низкий логический уровень, закрывающий ключ VT1. В регистр PORTD записывается код 0x7F, который включает внутренние резисторы нагрузки.

Код 0x7F в двоичном виде выглядит как 0x01111111, поэтому он включает нагрузки для семи младших разрядов порта. Для старшего, восьмого разряда нагрузку включить невозможно, так как этот разряд просто отсутствует.

В **строках 35, 36** производится **инициализация компаратора**. И, наконец, в **строках 37—40** производится инициализация таймера T1. Сначала настраиваются его **режимы работы**. Для этого в регистр TCCR1B записывается код 0x09 (**строки 37, 38**). Этот код переводит таймер в режим CTC и выбирает коэффициент предварительного деления равным единице.

Затем в регистр TCCR1A записывается ноль (**строки 39, 40**). Как видите, в комментариях к этому действию написано: «выключение звука». В данном случае подобное утверждение справедливо. Одна из функций регистра TCCR1A — управление подключением сигнала от таймера на внешний выход OC1A, который нашем случае служит выходом звука.

Включением и отключением выхода OC1A управляет разряд номер 6 регистра TCCR1A. Единица в шестом разряде подключает таймер к выходу OC1A (включает звук). При нулевом значении шестого разряда выход OC1A отключается, а соответствующему контакту микросхемы возвращается его основная функция.

Он становится просто выводом порта PB, в который, как вы помните, записан логический ноль. Этот ноль появляется на выходе, закрывая ключ. Таким образом, при выключенном звуке на выходе всегда будет ноль. Из всего вышеизложенного вы уже поняли, что запись в регистр TCCR1A кода 0x00 равносильна отключению звука.

В **строке 41** начинается **основной цикл нашей программы**. В первой части цикла (**строки 41—48**) расположена процедура опроса датчиков. Для работы этой процедуры используется вспомогательный регистр `count`. Программа сканирует датчики один за другим, а регистр `count` используется для подсчета уже отсканированных датчиков. Сканирование заканчивается тогда, когда обнаружится первый же датчик с замкнутыми контактами. При этом в регистре `count` останется номер этого датчика.

Рассмотрим работу процедуры сканирования подробнее. Перед началом сканирования содержимое регистра `count` обнуляется (**строка 41**). Затем производится чтение сигнала с контактов порта PD (**строка 42**). Считанный код помещается в регистр `temp`. Теперь код, находящийся в регистре `temp`, содержит полную информацию о состоянии всех семи датчиков.

Каждому из семи датчиков будет соответствовать один из семи младших разрядов кода в регистре `temp`:

- если в момент считывания контакты датчика были разомкнуты, то соответствующий разряд будет равен единице;
- если контакты датчика были замкнуты, соответствующий разряд будет равен нулю.

Сканирование датчиков сводится к проверке семи младших разрядов кода в регистре `temp`. Цикл сканирования составляют **строки 43—48**. В процессе сканирования программа просто сдвигает содержимое регистра `temp` вправо. В результате каждого сдвига содержимое очередного разряда попадает в флаг признака переноса.

По значению этого флага и определяется состояние датчика. Как только очередной разряд окажется равным нулю, это значит, что контакты соответствующего датчика были замкнуты. Поэтому цикл сканирования прекращается, и программа **приступает к процедуре формирования звука**.

Цикл сканирования повторяется семь раз (по количеству датчиков). Если после семи сдвигов нулевой бит не обнаружен, значит, контакты всех семи датчиков были незамкнуты. В этом случае управление передается по метке `m1`, где происходит выключение звука. Затем снова считывается состояние порта, и весь цикл сканирования повторяется сначала.

Логический сдвиг вправо разрядов регистра `temp` выполняется в **строке 43**. В **строке 44** производится проверка признака переноса. Если он равен нулю (контакты датчика замкнуты), то происходит переход к **строке 49** по метке `m3`.

Там **начинается вычисление параметров для формирования звука**. Если признак переноса равен единице (контакты датчика не замкнуты), цикл сканирования продолжается. Следующая команда (**строка 45**) уве-

личивает содержимое регистра `count`, осуществляя подсчет датчиков. В **строке 46** содержимое `count` сравнивается с числом 7. Таким образом ограничивается количество проходов цикла сканирования.

Если содержимое `count` еще не достигло семи, то выполняется переход по метке `m2`, и цикл сканирования продолжается. Если же содержимое `count` окажется равным семи, то это означает, что все семь датчиков мы уже перебрали. В этом случае выполняется оператор безусловного перехода в **строке 48**, который передает управление по метке `m1`.

Теперь посмотрим, что же происходит, когда **процедура сканирования обнаружит сработавший датчик**. В этом случае управление передается к **строке 49** (метка `m3`). Регистр `count` к этому моменту содержит номер сработавшего датчика. Для датчика, подключенного к входу PD.0, этот код будет равен нулю. Для PD.1 код будет равен единице. И так далее.

Теперь нам нужно **сгенерировать звук**, соответствующий коду датчика. Для этого мы должны извлечь соответствующий коэффициент деления из программной памяти, поместить его в регистр совпадения и подключить сигнал с таймера на внешний выход.

Сначала займемся **извлечением коэффициента деления**. Как уже говорилось выше, все коэффициенты записаны в программную память и составляют таблицу коэффициентов деления (см. **строку 63**). Для извлечения нужного нам коэффициента воспользуемся командой `lpm`. Но сначала нам необходимо вычислить адрес соответствующей ячейки памяти.

Для этого вспомним, что *каждый коэффициент представляет собой два байта, записанные в две соседние ячейки памяти (по альтернативной адресации)*. Если адрес начала таблицы равен `tabkd`, то в альтернативной адресации он будет равен `tabkd`×2. Очевидно, что коэффициент деления для датчика номер ноль будет занимать ячейки с адресами `tabkd`×2 и `tabkd`×2+1. Коэффициент деления для датчика номер один мы найдем в ячейках `tabkd`×2+2 и `tabkd`×2+3. И так далее. В общем случае адрес ячейки, содержащей первый байт нужного нам коэффициента мы найдем по формуле

`tabkd`×2+Nd ×2,

где Nd — это номер датчика.

Отсюда наша **задача**: используя номер, записанный в регистре `count`, вычислить адрес ячейки, где хранится нужный коэффициент деления. Так как любой ***адрес*** *— это шестнадцатиразрядное двоичное число*, нам придется производить операцию шестнадцатиразрядного сложения. В системе команд микроконтроллеров AVR такая команда отсутствует. Поэтому мы будем складывать шестнадцатиразрядные числа побайтно.

Команды вычисления адреса занимают **строки 49—55**. В **строке 49** происходит удвоение содержимого регистра `count` (умножение кода датчика на два). Для удвоения используется команда логического сдвига `lsl`.

Дело в том, что в двоичной системе логический сдвиг на один бит влево эквивалентен умножению на два. Теперь полученное в результате удвоения число необходимо прибавить к адресу начала таблицы. Для этого сформируем два шестнадцатиразрядных слагаемых. Первое слагаемое мы запишем в регистровую пару Y, а второе слагаемое — в регистровую пару Z. **Младший байт первого слагаемого** (удвоенный код датчика) мы берем из регистра `count` и помещаем в YL (**строка** 50). Так как датчиков всего семь (максимальное значение удвоенного кода равно 14), то **старший байт первого слагаемого** всегда будет равен нулю.

Запишем этот ноль в регистр YH (**строка 51**). В качестве второго слагаемого мы будем использовать число, равное удвоенному значению метки `tabkd`. Запишем младший и старший байты этого числа в регистровую пару Z (**строки 52, 53**). После того, как оба слагаемых сформированы, приступаем к процессу сложения. Сначала складываем младшие байты (**строка 54**). Затем складываем старшие байты с учетом переноса (**строка 55**). В результате сложения в регистре Z мы получим искомый адрес.

Теперь, используя этот адрес, приступим **к извлечению требуемого коэффициента деления**. В **строке 56** извлекается первый байт коэффициента. Причем используется версия команды `lpm`, которая автоматически увеличивает содержимое регистровой пары Z. Извлеченный байт помещается в младшую часть регистровой пары Y (регистр YL). Так как в содержимое Z стало на единицу большим, то очередная команда в **строке 57** извлекает очередной байт коэффициента деления. Извлеченный байт помещается в старшую часть регистровой пары Y (регистр YH).

В **строках 58 и 59** прочитанный только что коэффициент деления записывается в регистр совпадения OCR1A. При этом соблюдается **правило записи для регистров с двойной буферизацией:**

- сначала записывается старшая часть регистра (**строка 58**);
- затем записывается младшая (**строка 59**).

Сразу после записи коэффициента деления таймер начнет вырабатывать сигнал с нужной нам частотой. Теперь остается лишь подключить этот сигнал на выход (выполнить включение звука). Включение звука происходит в **строках 60, 61**.

Для этого в регистр TCCR1A записывается код 0x40. Этот код имеет единицу в шестом разряде, которая и подключает сигнал от таймера к выводу OC1A. В результате на выходе появляется звуковой сигнал,

который через транзисторный ключ VT1 поступает на звуковой излучатель VF1.

Команда безусловного перехода в **строке 62** завершает основной цикл программы. Она передает управление по метке `main`. В результате весь описанный выше процесс повторяется. Если в результате нового опроса датчиков обнаружится, что их состояние не изменилось, программа просто подтвердит все настройки таймера и генерация звука не прервется. Если состояние датчиков изменилось, то изменится и частота звука. Если же при очередном сканировании контакты всех датчиков окажутся незамкнутыми, управление перейдет по метке `m1`, и звук прекратится.

Программа на языке СИ

Возможный вариант той же программы на языке СИ приведен в **листинге 5.16**. В программе применяются следующие новые для нас операторы.

for

Оператор цикла. Мы уже привыкли к оператору цикла `while`. Оператор `for` — это альтернативный способ организации циклов. В общем случае оператор `for` записывается следующим образом:

```
for (Выр1; Выр2; Выр3)
    {
          Тело цикла;
    }
```

Выр1, Выр2 и Выр3 — любые корректные выражения языка СИ. Каждое из этих выражений имеет свое определенное **назначение**.

Выр1 — это команда начальной установки. Она выполняется один раз перед началом цикла.

Выр2 — условие выполнения цикла. Обычно это логическое выражение. Значение Выр2 проверяется в начале каждого прохода. Пока результат этого выражения «Истина» (не равен нулю), цикл продолжается. Как только результат Выр2 примет значение «Ложь» (станет равен нулю), цикл прекращается.

Выр3 — операция, выполняемая с параметром цикла. Это выражение выполняется в конце каждого прохода. Обычно в качестве Выр3 ставится команда, увеличивающая параметр цикла на единицу. Вот пример применения оператора `for`:

```
for (i=0; i<10; i++)
    {
          Тело цикла;
    }
```

Это простейший цикл с параметром `i`. Перед началом цикла параметру присваивается нулевое значение. Цикл выполняется до тех пор, пока `i` меньше десяти. Каждый раз после выполнения команд, составляющих тело цикла, оператор `i++` увеличивает значение переменной `i` на единицу. Выражение `i++` представляет собой одно из сокращений языка СИ. В развернутом виде та же команда выглядит так: `i=i+1`.

goto

Команда безусловного перехода. Тот, кто знаком с языком программирования Basic, хорошо знает эту команду. Команда `goto` в языке СИ то же самое, что `rjmp` на Ассемблере. Она имеет всего один параметр — имя метки. В **строке 19** нашей программы (**листинг 5.16**) команда `goto` передает управление к **строке 14** (по метке `m1`).

***Массив** — это набор элементов, каждый из которых может иметь свое собственное значение. Все элементы массива всегда имеют один тип.*

Кроме двух новых операторов, в нашей программе появляется новое для нас понятие: массив.

Перед тем, как использовать массив, его, как и переменную, нужно описать. Описание массива очень похоже на описание переменной. В общем виде это выглядит следующим образом:

```
Тип Имя [Разм];
```

Как и в случае с переменной, сначала указывается тип массива, затем его имя:

- тип массива может принимать те же значения, что и тип переменной (см. **табл. 5.1**);
- имя массива выбирается в соответствии со стандартными правилами выбора имен.

В квадратных скобках указывается размер массива, то есть количество его элементов. В языке СИ при описании массивов необходимо обязательно указывать их размер, чтобы транслятор мог зарезервировать память для их размещения. Однако в любом месте программы размер массива допускается изменять. Это делается при помощи повторного описания массива, но уже с другим значением размера. Если новый размер окажется больше старого, массив просто пополнится новыми членами. При этом значения старых членов сохраняются. Если новый размер окажется меньше старого, все лишние члены удаляются. Значения удаленных членов массива будут утеряны.

Приведем **пример** описания массива:

```
int rabtab[5];
```

Описанный выше массив имеет имя `rabtab`, тип `int` и количество членов, равное пяти. Теперь посмотрим, как **применяются массивы**. С любым элементом описанного выше массива можно работать как с отдельной переменной. **Например**, можно присвоить ему значение:

```
rabtab[1] = 231; //Первому члену массива присваивается
                 //значение 231.
```

Можно, наоборот, значение одного из элементов массива присвоить переменной:

```
x = rabtab[8];    //Переменной x присваивается
                  //значение восьмого элемента массива.
```

Если размер массива равен N, то указатель массива может принимать значения от 0 до N–1. Если окажется, что значение указателя выходит за указанные пределы, то транслятор выдаст сообщение об ошибке. В качестве указателя массива можно использовать не только числовую константу. В квадратные скобки вы можете вписать любую переменную или любое выражение. Главное, чтобы значение этого выражения входило в область допустимых значений.

*При использовании массива в тексте программы число в квадратных скобках уже означает не размер массива, а номер элемента, с которым производится данная операция. Поэтому такое число теперь называется **указателем массива**.*

*При описании массива можно одновременно производить его инициализацию. Под **инициализацией** понимается присвоение начальных значений всем элементам массива.*

Строка описания массива, производящая его инициализацию, выглядит примерно следующим образом:

```
int rabtab[5] = {23, 41, 52, 287, 40, 51};
char txt1[] = “Simferopol”;
```

Начальные значения всех элементов записываются после знака «=» в фигурных скобках через запятую (числовые значения) либо в кавычках в виде символьной строки. В последнем случае в каждый элемент массива записывается код одного из символов.

Если при описании массива используется инициализация, допускается не указывать его размер. В этом случае размер определяется автоматически по количеству значений в фигурных скобках или по количеству символов.

Еще один аспект программирования на языке СИ, который мы будем впервые использовать в нашей программе, — это **директивы локали-**

зации переменных и массивов. Так как наша версия СИ специально предназначена для составления программ для микроконтроллеров, она просто обязана иметь возможность явно указывать вид памяти, где нужно хранить ту либо иную переменную или массив.

По умолчанию транслятор сам решает эту задачу и размещает переменные и массивы по своему усмотрению. Но иногда вопрос размещения имеет принципиальное значение. В этом случае применяются **специальные директивы**. Эти директивы используются в процессе описания и указывают транслятору, в каком из видов памяти разместить данную конкретную переменную либо массив. **Например**, если вы желаете, чтобы значение вашей переменной хранилось в одном из регистров общего назначения, то при описании этой переменной вы должны использовать директиву `register`. Вот несколько **примеров** таких описаний:

```
register int alpha;   // Определение регистровой
                      // переменной alpha
register unsigned int beta  // Определение регистровой
                            // переменной beta
register int gamma @10;  // Определение регистровой
                      // переменной gamma с конкретным
                      // указанием регистра (R10),
                      // где она должна храниться
```

Второй тип памяти, где могут храниться переменные, — это **энергонезависимая память данных (EEPROM)**. Для размещения переменных в этой памяти используется директива `eeprom`. Причем в EEPROM могут храниться как переменные, так и массивы. Команда описания в этом случае будет выглядеть следующим образом:

```
eeprom char beta;        // Определение переменной beta
                         // с размещением в EEPROM
eeprom long array1[5];   // Определение массива array1
                         // в EEPROM
eeprom char string[]="Hello" // Определение
                         // текстового массива
```

При размещении массива в EEPROM необходимо учитывать, что указатель массива всегда должен иметь не менее 16 битов. То есть иметь тип `int` либо `unsigned int`.

Еще один тип памяти, где могут храниться данные, — это **программная память.** Но в этом случае мы уже не сможем в программе изменять значения переменных и элементов массива. Эти значения определяются один раз и только при инициализации. Для указания того факта, что массив или переменная должны храниться в программной памяти, используется директива `flesh`.

В качестве **примера** описания массива в программной памяти обратимся к **строке 2** нашей программы (**листинг 5.16**). Массив или переменная с индексом `flesh` могут быть только глобальными. Поэтому их описание всегда располагается в начале программы перед описанием всех функций.

Описание программы (листинг 5.16)

Программа содержит только одну главную функцию `main`. Перед началом этой функции происходит определение и инициализация массива (**строка 2**). Этот массив предназначен для хранения коэффициентов деления и является аналогом таблицы коэффициентов деления в программе на Ассемблере.

При инициализации элементам массива присваиваются уже знакомые нам значения. Те же значения мы помещали в таблицу коэффициентов.

Главная функция `main` программы занимает **строки 4—21**. В **строках 4** и **5** создаются две переменные:

- переменная `count`, которая будет использоваться для определения кода нажатого датчика;
- переменная `temp`, предназначенная для вспомогательных целей.

Листинг 5.16

```
     /*****************************************************
     Project : Prog 8
     Version : 1
     Date    : 31.01.2006
     Author  : Belov
     Comments: Сигнализатор «Семь нот»
     Chip type           : ATtiny2313
     Clock frequency     : 4,000000 MHz
     *****************************************************/

1    #include <tiny2313.h>
     // Объявление и инициализация массива коэффициентов деления
2    flash unsigned int tabkd[7] = {4748,4480,4228,3992,3768,3556,3356};

3    void main(void)
     {
4    unsigned char count;  // Определяем переменную count
5    unsigned char temp;  // Определяем переменную temp

6    PORTB=0x00;   // Инициализация порта PB
7    DDRB=0x08;

8    PORTD=0x7F;   // Инициализация порта PD
9    DDRD=0x00;

10   ACSR=0x80;   // Инициализация (отключение) компаратора

11   TCCR1A=0x00;   // Инициализация таймера счетчика T1
12   TCCR1B=0x09;

13   while (1)
          {
14   m1:       temp=PIND;
15             for (count=0; count<7; count++)  // Цикл сканирования датчиков
                 {
16                  if ((temp&1)==0) goto m2;   // Проверка младшего бита переменной temp
```

```
17                  temp >>= 1;                 // Сдвиг содержимого temp
                  }
18              TCCR1A=0x00;            // Выключение звука
19              goto m1;                // Переход на начало

20    m2:       OCR1A=tabkd[count];  // Запись коэффициента деления таймера.
21              TCCR1A=0x40;            // Включение звука
        };
    }
```

Строки 6—12 занимает **модуль инициализации**. Здесь настраиваются порты ввода вывода (**строки 6—9**), компаратор (**строка 10**) и таймер T1 (**строки 11, 12**). При настройке в управляющие регистры микроконтроллера записываются те же самые коды, которые мы записывали в те же регистры в программе на Ассемблере (см. **листинг 5.15**).

Основной цикл программы занимает **строки 13—21**. В **строке 14** программа читает содержимое порта PD и помещает прочитанный байт в переменную `temp`. Этот байт, как вы понимаете, содержит информацию о состоянии датчиков.

В **строках 15—17** находится **цикл сканирования датчиков**. В качестве параметра цикла используется переменная `count`. Цикл выполняется, пока значение переменной `count` меньше семи. В теле цикла происходит проверка младшего бита переменной `temp` (**строка 16**).

Для проверки значения бита используется выражение `temp&1`. Оператор «&» выполняет операцию побитового «И» между значением переменной `temp` и числом 1 (0x01). При этом все старшие биты обнуляются, и результат выражения становится равным значению младшего бита переменной `temp`.

Оператор `if` в **строке 16** проверяет результат выражения. Если он равен нулю (контакты очередного датчика замкнуты), управление передается по метке m2, и цикл прекращается досрочно. В противном случае (контакты датчика разомкнуты) цикл выполняется дальше.

В **строке 17** производится **логический сдвиг** содержимого переменной `temp`. А затем цикл сканирования начинается сначала. Таким образом, за семь проходов цикла проверяются все семь датчиков.

Если контакты всех датчиков окажутся незамкнутыми, то цикл `for` (**строки 15—17**) завершается естественным образом, а управление переходит к **строке 18**. Здесь происходит выключение звука: регистру TCCR1A присваивается нулевое значение.

Затем в **строке 19** управление передается в начало процедуры. Для передачи управления используется оператор безусловного перехода.

Если цикл завершился досрочно переходом к **строке 20** (по метке m2), начинается **процесс формирования звука**. Так как процесс сканирования датчиков закончился досрочно, в переменной `count` находится номер датчика, контакты которого оказались замкнутыми. Теперь нам остается лишь извлечь коэффициент деления, соответствующий этому номеру датчика, и записать его в регистр совпадения таймера.

На СИ это делается элементарно. Регистру OPCR1A просто присваивается значение соответствующего элемента массива `tabkd` (**строка 20**). В **строке 21** производится включение звука. Как и в программе на Ассемблере, звук включается путем записи в регистр TCCR1A кода 0x40. На этом **завершается тело основного цикла программы**. Так как основной цикл программы выполнен в виде бесконечного цикла, то после завершения его последней команды весь цикл повторяется сначала.

5.10. Музыкальная шкатулка

Постановка задачи

Все предыдущие примеры не имели практического значения. Их можно рассматривать как простейшие задачи для тренировки. Надеюсь, к данному моменту вы достигли такого уровня знаний, что вам по плечу настоящая программа, имеющая практическое значение.

Итак, попробуем создать музыкальное устройство, автоматически воспроизводящее разные мелодии. В предыдущем примере мы научились издавать различные музыкальные звуки. Теперь нужно заставить микроконтроллер составлять из этих звуков мелодии. Причем эти мелодии должны быть жестко записаны в память микроконтроллера.

Сформулируем задачу следующим образом:

«Разработать устройство, предназначенное для воспроизведения простых одноголосых мелодий, записанных в память программ на этапе программирования. Устройство должно иметь семь управляющих кнопок. Каждой из кнопок должна соответствовать своя мелодия. Мелодия воспроизводится при нажатии и удержании кнопки. При отпускании всех кнопок воспроизведение мелодий прекращается».

Схема

Неслучайно в условии задачи заложено именно семь кнопок и одноголосые мелодии. Это позволяет использовать для новой задачи схему из предыдущего примера (**рис. 5.14**). Представим, что контакты, подключенные к порту PD, — это не датчики, а кнопки управления. Каждая кнопка будет включать одну из семи заложенных в программу мелодий. Воспроизведение мелодий будет происходить при помощи звуковой части схемы (R1, VT1, VF1).

Алгоритм

Для начала нам нужно придумать, как мы будем хранить мелодии в памяти. Для того, чтобы в памяти можно было что-либо хранить, нужно

сначала это что-то каким-либо способом закодировать. Любая мелодия состоит из нот. Каждая нота имеет свой тон (частоту) и длительность звучания. Для того, чтобы закодировать тон ноты, можно просто все ноты пронумеровать по порядку. Удобнее нумеровать, начиная с самого низкого тона. На клавиатуре клавишного инструмента это будет слева направо.

Известно, что весь музыкальный ряд делится на **октавы**. Если вы думаете, что в каждой октаве семь нот, то вы плохо знаете физические основы музыкального ряда. На самом деле в современном музыкальном ряду каждая октава делится на 12 нот. Семь основных нот (белые клавиши) и пять дополнительных (черные клавиши).

Деление на основные и дополнительные ноты сложилось исторически. В настоящее время используется музыкальный строй, в котором все 12 нот одной октавы равнозначны. Частоты любых двух соседних нот отличаются друг от друга в одинаковое количество раз. При этом частоты одноименных нот в двух соседних октавах отличаются ровно в два раза. Более подробно об этом вы можете прочитать в **[6]**.

Для нас же важно то, что коды всем этим нотам мы должны присваивать в порядке возрастания частоты. И начнем мы с ноты «До» первой октавы. Для музыкальной шкатулки более низкие ноты не нужны. В **табл. 5.3** показаны коды для всей первой октавы. Следующая, вторая октава продолжает первую и по кодировке, и по набору частот. Так нота «До» второй октавы будет иметь код 13, а частоту $f_{12}=f_0\times2$. А нота «Ре» второй октавы будет иметь код 14 и частоту $f_{13}=f_1\times2$. И так далее.

Кодировка нот первой октавы *Таблица 5.3*

Код	Нота	Частота	Код	Нота	Частота
1	До	f_0	7	Фа#	$f_6=f_5/K$
2	До#	$f_1=f_0/K$	8	Соль	$f_7=f_6/K$
3	Ре	$f_2=f_1/K$	9	Соль#	$f_8=f_7/K$
4	Ре#	$f_3=f_2/K$	10	Ля	$f_9=f_8/K$
5	Ми	$f_4=f_3/K$	11	Ля#	$f_{10}=f_9/K$
6	Фа	$f_5=f_4/K$	12	Си	$f_{11}=f_{10}/K$

Для справки: $K=\sqrt[12]{2}$.

Музыкальная длительность тоже легко кодируется. В музыке применяют не произвольную длительность, а длительность, выраженную долями от целой (см. **табл. 5.4**). В зависимости от темпа реальная длительность целой ноты меняется. Для сохранения мелодии необходимо соблюдать лишь соотношения между длительностями. Поэтому нам необходимо закодировать лишь семь вариантов длительности. Присвоим им коды от 0 до 6. Например так, как это показано в графе «Код» **табл. 5.5**. Назначение графы «Коэффициент деления» мы пока опустим.

Кодирование музыкальных длительностей *Таблица 5.4*

Код	Длительность	Коэффициент деления
0	1 (целая)	64
1	1/2 (половинная)	128
2	1/4 (четверть)	256
3	1/8 (восьмая)	512
4	1/16 (шестнадцатая)	1024
5	1/32 (тридцать вторая)	2048
6	1/64 (шестьдесят четвертая)	4096

Кроме нот, любая мелодия обязательно содержит **музыкальные паузы**.

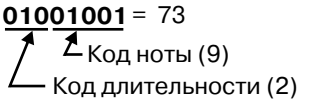
В связи с этим удобно представить паузу как еще одну ноту. Ноту без звука. Такой ноте логично присвоить нулевой код.

***Паузы** — это промежутки времени, когда ни один звук не звучит. Длительность музыкальных пауз принимает точно такие же значения, как и длительность нот.*

Кодируем мелодии

Для экономии памяти удобнее каждую **ноту** кодировать одним байтом. Договоримся, что три старших бита мы будем использовать для кодирования длительности ноты, а оставшиеся пять битов — для кодирования ее тона. Пятью битами можно закодировать до 32 разных нот, что вполне хватит для музыкальной шкатулки.

Итак, если использовать приведенный выше способ кодирования, то код ноты ля первой октавы длительностью 1/4 в двоичном виде будет равен



Теперь мы можем приступать к **кодированию мелодий**. Для того, чтобы закодировать мелодию, нам нужна ее **нотная запись**. Используя нотную запись, мы должны присвоить каждой ноте и каждой музыкальной паузе свой **код**.

Цепочка таких кодов и будет представлять собой закодированную мелодию. По условиям задачи наша музыкальная шкатулка должна уметь воспроизводить семь разных мелодий. Коды всех семи мелодий мы разместим в программной памяти микроконтроллера.

Как определить **конец каждой мелодии**? Для того, чтобы компьютер знал, где заканчивается каждая мелодия, используем код 255 в качестве признака конца.

Теперь нам нужно придумать, как микроконтроллер будет находить **начало каждой мелодии.** Все мелодии имеют разную длину, а в памяти они будут записаны одна за другой. Поэтому адрес начала каждой мелодии зависит от длины всех предыдущих. Удобнее всего просто по факту определить адрес начала каждой мелодии и поместить все семь адресов в специальную таблицу.

Кроме этой таблицы, нам еще понадобится таблица коэффициентов деления для всех 32 нот и таблица, хранящая константы задержки для всех используемых нами музыкальных длительностей.

Алгоритм работы музыкальной шкатулки

Теперь мы можем сформулировать алгоритм работы музыкальной шкатулки.

1. Просканировать клавиатуру и определить номер самой первой нажатой клавиши.
2. Извлечь из таблицы начал мелодий значение элемента, номер которого соответствует только что определенному номеру нажатой клавиши. Это значение будет равно адресу в программной памяти, где начинается нужная нам мелодия.
3. Начать цикл воспроизведения мелодии. Для этого поочередно извлекать коды нот из памяти, начиная с адреса, который мы определили в п. 2 алгоритма.
4. Каждый код ноты разложить на код тона и код длительности.
5. Если код тона равен нулю, отключить звук и перейти к формированию задержки (к п. 9 настоящего алгоритма).
6. Если код тона не равен нулю, извлечь из таблицы коэффициентов деления значение элемента с номером, равным коду тона.
7. Записать коэффициент деления, который мы нашли в пункте 6 настоящего алгоритма, в регистр совпадения таймера T1.
8. Включить звук (подключить вывод OC1A к выходу таймера T1).
9. Извлечь из таблицы длительностей задержки значение элемента с номером, равным коду длительности.
10. Сформировать паузу с использованием константы задержки, которую мы нашли в пункте 9 настоящего алгоритма.
11. По окончании паузы выключить звук (отключить OC1A от выхода таймера).
12. Повторять цикл (пункты 4—11 настоящего алгоритма) до тех пор, пока нажата соответствующая кнопка.
13. Если очередной код ноты окажется равным 255, перейти к началу текущей мелодии, то есть вернуться к п. 3 настоящего алгоритма.

Программа на Ассемблере

Возможный вариант программы на языке Ассемблер приведен в **листинге 5.17**. В программе используются следующие новые для нас операторы.

andi

Логическое «И» содержимого РОН и константы. Команда имеет следующий формат:

```
andi Rd, K,
```

где Rd — имя регистра общего назначения, а K — некая числовая константа.

Команда `andi` выполняет операцию побитового «И» между содержимым регистра и константой. Результат помещается в регистр Rd.

Команда `andi` часто используется как *операция наложения маски*. Что в данном случае означает понятие «маска»? Для того, чтобы это объяснить, я хочу немного отвлечься и обратиться к детективному жанру. Если точнее, я предлагаю вспомнить один из методов шифрования различных посланий.

Помните заставку к фильму о Шерлоке Холмсе? На лист бумаги, на котором в хаотическом на первый взгляд порядке записаны случайные знаки, накладывается второй лист, в котором в определенных местах проделаны дырочки. При совмещении этих двух листов лишние знаки оказываются невидимыми, а сквозь дырочки мы видим символы, составляющие зашифрованное сообщение.

Что-то подобное происходит при наложении маски с использованием двух двоичных чисел. Одно из этих чисел является маской для второго числа. При наложении маски лишние биты обнуляются, а остаются лишь те, которые несут нужную для нас информацию.

В нашей программе наложение маски используется для того, чтобы из **кода ноты** выделить **код тона** или **код длительности**. Код тона занимает пять младших разрядов общего кода. Для выделения кода тона нужно произвести операцию побитового «И» между кодом ноты и маской. Маска в данном случае должна быть равна 00011111B (или в шестнадцатиричном виде — 1FH). Выделение кода тона происходит в **строке 67** программы (см. **листинг 5.17**). В результате наложения маски три старших бита обнуляются, а пять младших остаются без изменений. Для выделения кода длительности выбирается другое значение маски (см. **строку 74**).

adiw

Шестнадцатиразрядное сложение. Позволяет прибавить к шестнадцатиразрядному числу некоторую числовую константу.

Шестнадцатиразрядное значение помещается в регистровую пару. При этом может использоваться одна из следующих пар: R24:R25, R26:R27, R28:R29 и R30:R31.

Команда имеет ограничение на величину константы. Константа может иметь значение 0 до 63. Команда имеет два параметра:

- первый параметр — это имя первого из регистров регистровой пары;
- второй параметр — это прибавляемая константа.

По странной прихоти разработчиков данной версии Ассемблера, в качестве первого параметра нельзя использовать имена регистровых пар X, Y и Z, а также их половинок (например, XL или YH).

В **строке 126** нашей программы (**листинг 5.17**) к содержимому регистровой пары R30:R31 прибавляется единица. По сути, в данном случае мы добавляем константу к содержимому регистровой пары Z.

cp

Сравнение двух РОН. Команда имеет два параметра. Оба параметра — это имена регистров общего назначения. Команда не изменяет содержимого самих регистров. Она просто производит их сравнение. По результатам сравнения устанавливаются флаги в регистре SREG. После команды сравнения обычно применяется одна из команд условного перехода.

rol

Циклический сдвиг содержимого регистра влево через признак переноса. Это еще один оператор сдвига. В данном случае сдвиг битов производится по кругу, как показано на **рис. 5.15**. В этот круг включена ячейка признака переноса C.

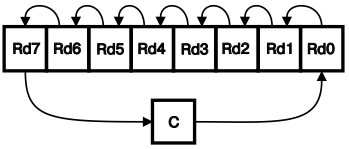


Рис. 5.15. *Схема выполнения операции циклического сдвига влево*

.db

Директива описания данных. Эта директива так же, как и уже известная нам директива `dw` предназначена для описания данных, помещаемых в память программ или в EEPROM. Директива `db`, в отличие от `dw`, описывает не двухбайтовые слова данных, а отдельные байты. Значения, которые нужно поместить в память, записываются справа после оператора `db` через запятую. Каждое такое значение не должно превышать 255 (максимальное число, которое можно записать при помощи одного байта).

Если данных очень много, для их записи можно применять несколько операторов `db`, расположенных непосредственно друг за другом. Однако

при этом нужно учитывать один важный момент. Сколько бы значений ни было записано в правой части директивы db, она всегда записывает четное их количество. Это связано с указателем текущего адреса в сегменте cseg.

Указатель работает с основной адресацией и должен переместиться на целое количество ячеек памяти в соответствии с этой адресацией. А в этой адресации каждая ячейка вмещает в себя два байта данных. Поэтому в том случае, если программист все же укажет нечетное количество параметров, то директива db в конце припишет еще один, который будет равен нулю. Поэтому, если вам необходимо записать в память большое количество значений и вы собираетесь разбить все параметры на несколько строк и записать каждую строку при помощи директивы db, то во всех строках, кроме последней, вы должны оставить четное количество параметров.

Листинг 5.17

```
;################################################
;##               Пример 9                     ##
;##          Музыкальная шкатулка              ##
;################################################

;-------------------------- Псевдокоманды управления

.include "tn2313def.inc"                    ; Присоединение файла описаний
.list                                       ; Включение листинга

.def        loop1 = R0                      ; Три ячейки для процедуры задержки
.def        loop2 = R1
.def        loop3 = R21
.def        temp = R16                      ; Вспомогательный регистр
.def        temp1 = R17                     ; Второй вспомогательный регистр
.def        count = R17                     ; Определение регистра счетчика опроса клавиш
.def        fnota = R19                     ; Частота текущей ноты
.def        dnota = R20                     ; Длительность текущей ноты

;-------------------------- Начало программного кода

            .cseg                           ; Выбор сегмента программного кода
            .org        0                   ; Установка текущего адреса на ноль

start:      rjmp        init                ; Переход на начало программы
            reti                            ; Внешнее прерывание 0
            reti                            ; Внешнее прерывание 1
            reti                            ; Таймер/счетчик 1, захват
            reti                            ; Таймер/счетчик 1, совпадение, канал A
            reti                            ; Таймер/счетчик 1, прерывание по переполнению
            reti                            ; Таймер/счетчик 0, прерывание по переполнению
            reti                            ; Прерывание UART прием завершен
            reti                            ; Прерывание UART регистр данных пуст
            reti                            ; Прерывание UART передача завершена
            reti                            ; Прерывание по компаратору
            reti                            ; Прерывание по изменению на любом контакте
            reti                            ; Таймер/счетчик 1. Совпадение, канал B
            reti                            ; Таймер/счетчик 0. Совпадение, канал B
            reti                            ; Таймер/счетчик 0. Совпадение, канал A
            reti                            ; USI готовность к старту
            reti                            ; USI Переполнение
            reti                            ; EEPROM Готовность
            reti                            ; Переполнение охранного таймера

;*****************************************************
;*              Модуль инициализации                 *
;*****************************************************
init:
;-------------------------- Инициализация стека
```

```
32                  ldi         temp, RAMEND        ; Инициализация стека
33                  out         SPL, temp

          ;--------------------------- Инициализация портов B/B

34                  ldi         temp, 0x08          ; Инициализация порта PB
35                  out         PORTB, temp
36                  out         DDRB, temp

35                  ldi         temp, 0x7F          ; Инициализация порта PD
36                  out         PORTD, temp
37                  ldi         temp, 0x00
38                  out         DDRD, temp

          ;--------------------------- Инициализация (выключение) компаратора

39                  ldi         temp, 0x80
40                  out         ACSR, temp

          ;--------------------------- Инициализация таймера T1

41                  ldi         temp, 0x09          ; Включаем режим CTC
42                  out         TCCR1B, temp
43        m1:       ldi         temp, 0x00          ; Выключаем звук
44                  out         TCCR1A, temp


          ;*****************************************************
          ;*             Начало основной программы             *
          ;*****************************************************
          main:
          ;--------------------------- Вычисление номера нажатой кнопки

45                  clr         count               ; Обнуление счетчика опроса клавиш
46                  in          temp, PIND          ; Чтение порта D
47        m2:       lsr         temp                ; Сдвигаем входной байт
48                  brcc        m3                  ; Если текущий разряд был равен 0
49                  inc         count               ; Увеличиваем показание счетчика
50                  cpi         count, 7            ; Сравнение (7 – конец сканирования)
51                  brne        m2                  ; Если не конец, продолжить
52                  rjmp        m1                  ; Если не одна клавиша не нажата

          ;--------------------------- Выбор мелодии

53        m3:       mov         YL, count           ; Вычисляем адрес, где
54                  ldi         ZL, low(tabm*2)     ; хранится начало мелодии
55                  ldi         ZH, high(tabm*2)
56                  rcall       addw                ; К подпрограмме 16-разрядного сложения

57                  lpm         XL, Z+              ; Извлекаем адреса из таблицы
58                  lpm         XH, Z               ; и помещаем в X

          ;--------------------------- Воспроизведение мелодии

59        m4:       mov         ZH, XH              ; Записываем в Z начало мелодии
60                  mov         ZL, XL

61        m5:       in          temp, PIND          ; Читаем содержимое порт D
62                  cpi         temp, 0x7F          ; Проверяем на равенство 7FH
63                  breq        m1                  ; Если равно (кнопки отпущены) в начало

64                  lpm         temp, Z             ; Извлекаем код ноты
65                  cpi         temp, 0xFF          ; Проверяем, не конец ли мелодии
66                  breq        m4                  ; Если конец, начинаем мелодию сначала

67                  andi        temp, 0x1F          ; Выделяем код тона из кода ноты
68                  mov         fnota, temp         ; Записываем в регистр кода тона
69                  lpm         temp, Z+            ; Еще раз берем код ноты
70                  rol         temp                ; Производим четырехкратный сдвиг кода ноты
71                  rol         temp
72                  rol         temp
73                  rol         temp
74                  andi        temp, 0x07          ; Выделяем код длительности
75                  mov         dnota, temp         ; Помещаем ее в регистр длительности

76                  rcall       nota                ; К подпрограмме воспроизведения ноты

77                  rjmp        m5                  ; В начало цикла (следующая нота)
```

```
        ;*****************************************************
        ;*          Вспомогательные подпрограммы             *
        ;*****************************************************

        ; ---------------------- Подпрограмма 16-ти разрядного сложения
 78     addw:       push    YH

 79                 lsl     YL                  ; Умножение первого слагаемого на 2
 80                 ldi     YH, 0               ; Второй байт первого слагаемого = 0
 81                 add     ZL, YL              ; Складываем два слагаемых
 82                 adc     ZH, YH

 83                 pop     YH
 84                 ret

        ;----------------------- Подпрограмма исполнения одной ноты
 85     nota:       push    ZH
 86                 push    ZL
 87                 push    YL
 88                 push    temp

 89                 cpi     fnota, 0x00         ; Проверка, не пауза ли
 90                 breq    nt1                 ; Если пауза, переходим сразу к задержке

 91                 mov     YL, fnota           ; Вычисляем адрес, где хранится
 92                 ldi     ZL, low(tabkd*2)    ; коэффициент деления для текущей ноты
 93                 ldi     ZH, high(tabkd*2)
 94                 rcall   addw                ; К подпрограмме 16-разрядного сложения

 95                 lpm     temp, Z+            ; Извлекаем мл. разряд КД для текущей ноты
 96                 lpm     temp1, Z            ; Извлекаем ст. разряд КД для текущей ноты
 97                 out     OCR1AH, temp1       ; Записать в старш. часть регистра совпадения
 98                 out     OCR1AL, temp        ; Записать в младш. часть регистра совпадения

 99                 ldi     temp, 0x40          ; Включить звук
100                 out     TCCR1A, temp

101     nt1:        rcall   wait                ; К подпрограмме задержки

102                 ldi     temp, 0x00          ; Выключить звук
103                 out     TCCR1A, temp

104                 ldi     dnota,0             ; Сбрасываем задержку для паузы между нотами
105                 rcall   wait                ; Пауза между нотами

106                 pop     temp                ; Завершение подпрограммы
107                 pop     YL
108                 pop     ZL
109                 pop     ZH
110                 ret

        ;----------------------- Подпрограмма формирования задержки
111     wait:       push    ZH
112                 push    ZL
113                 push    YH
114                 push    YL

115                 mov     YL, dnota           ; Вычисляем адрес, где хранится
116                 ldi     ZL, low(tabz*2)     ; нужный коэффициент задержки
117                 ldi     ZH, high(tabz*2)
118                 rcall   addw                ; К подпрограмме 16-разрядного сложения

119                 lpm     YL, Z+              ; Читаем первый байт коэффициента задержки
120                 lpm     YH, Z               ; Читаем второй байт коэффициента задержки

121                 clr     ZL                  ; Обнуляем регистровую пару Z
122                 clr     ZH

                    ; Цикл задержки
123     w1:         ldi     loop,255            ; Пустой внутренний цикл
124     w2:         dec     loop
125                 brne    w2
126                 adiw    R30, 1              ; Увеличение регистровой пары Z на единицу
127                 cp      YL, ZL              ; Проверка младшего разряда
128                 brne    w1
129                 cp      YH, ZH              ; Проверка старшего разряда
130                 brne    w1

131                 pop     YL                  ; Завершение подпрограммы
```

```
132                 pop         YH
133                 pop         ZL
134                 pop         ZH
135                 ret


                ;*****************************************************
                ;*          Таблица длительности задержек            *
                ;*****************************************************

136             tabz:       .dw         128,256,512,1024,2048,4096, 8192

                ;*****************************************************
                ;*          Таблица коэффициентов деления            *
                ;*****************************************************

137             tabkd:      .dw         0
138                         .dw         4748,4480,4228,3992,3768,3556,3356,3168,2990,2822,2664, 2514
139                         .dw         2374,2240,2114,1996,1884,1778,1678,1584,1495,1411,1332, 1257
140                         .dw         1187,1120,1057, 998, 942, 889, 839, 792

                ;*****************************************************
                ;*           Таблица начал всех мелодий              *
                ;*****************************************************

141             tabm:       .dw         mel1*2,mel2*2,mel3*2,mel4*2
142                         .dw         mel5*2,mel6*2,mel7*2

                ;*****************************************************
                ;*                 Таблица мелодий                   *
                ;*****************************************************

                ;           В траве сидел кузнечик
143             mel1:       .db         109,104,109,104,109,108,108, 96,108,104
144                         .db         108,104,108,109,109, 96,109,104,109,104
145                         .db         109,108,108, 96,108,104,108,104,108,141
146                         .db          96,109,111, 79, 79,111,111,112, 80, 80
147                         .db         112,112,112,111,109,108,109,109, 96,109
148                         .db         111, 79, 79,111,111,112, 80, 80,112,112
149                         .db         112,111,109,108,141,128, 96,255

                ;           Песенка крокодила Гены
150             mel2:       .db         109,110,141,102,104,105,102,109,110,141
151                         .db         104,105,107,104,109,110,141,104,105,139
152                         .db         109,110,173, 96,114,115,146,109,110,112
153                         .db         109,114,115,146,107,109,110,114,112,110
154                         .db         146,109,105,136,107,105,134,128,128,102
155                         .db         105,137,136,128,104,107,139,137,128,105
156                         .db         109,141,139,128,110,109,176,112,108,109
157                         .db         112,144,142,128,107,110,142,141,128,105
158                         .db         109,139,128,173,134,128,128,109,112,144
159                         .db         142,128,107,110,142,141,128,105,109,139
160                         .db         128,173,146,128, 96,255

                ;           В лесу родилась елочка
161             mel3:       .db         132,141,141,139,141,137,132,132,132,141
162                         .db         141,142,139,176,128,144,146,146,154,154
163                         .db         153,151,149,144,153,153,151,153,181,128
164                         .db         96,255

                ;           Happy births to you
165             mel4:       .db         107,107,141,139,144,143,128,107,107,141
166                         .db         139,146,144,128,107,107,151,148,146,112
167                         .db         111,149,117,117,148,144,146,144,128,255

                ;           С чего начинается родина
168             mel5:       .db          99,175,109,107,106,102, 99,144,111,175
169                         .db          96, 99,107,107,107,107,102,104,170, 96
170                         .db         99,109,109,109,109,107,106,143,109,141
171                         .db         99,109,109,109,109,104,106,171, 96, 99
172                         .db         111,109,107,106,102, 99,144,111,143,104
173                         .db         114,114,114,114,109,111,176, 96,104,116
174                         .db         112,109,107,106, 64, 73,143,107,131, 99
175                         .db         144, 80, 80,112,111, 64, 75,173,128,255

                ;           Песня из кинофильма "Веселые ребята"
176             mel6:       .db         105,109,112,149,116, 64, 80,148,114, 64
```

```
177                 .db          78,146,112,  96,105,105,109,144,111,  64
178                 .db          80,145,112,  64,  81,178,  96,117,117,117
179                 .db         149,116,  64,  82,146,112,  64,  79,146,144
180                 .db          96,105,105,107,141,108,109,112,110,102
181                 .db         104,137,128,  96,105,105,105,137,102,  64
182                 .db          73,142,105,107,109,  64,  75,137,  96,105
183                 .db         105,105,137,102,105,142,112,  64,  82,180
184                 .db          96,116,116,116,148,114,112,142,109,  64
185                 .db          78,146,144,  96,105,105,107,141,108,109
186                 .db         112,110,102,104,169,  96,  96,255

                    ;           Улыбка
187     mel7:       .db         107,104,141,139,102,105,104,102,164,128
188                 .db         104,107,109,109,109,111,114,112,111,109
189                 .db         144,139,128,109,111,144,  96,111,109,104
190                 .db         107,105,173,128,111,109,112,107,111,109
191                 .db         109,107,102,104,134,132,128,100,103,107
192                 .db         107,107,107,139,112,100,103,102,102,102
193                 .db         134,102,103,107,105,107,108,108,108,108
194                 .db         107,105,107,108,144,142,128,112,107,110
195                 .db         140,112,105,108,107,107,107,105,140,139
196                 .db         139,112,103,102,103,105,108,107,105,103
197                 .db         128,112,107,110,108,108,108,140,112,105
198                 .db         108,107,107,107,139,112,103,102,103,105
199                 .db         108,107,105,103,105,139,132,128,  96,  96
200                 .db          96,255
```

Описание программы (листинг 5.17)

Благодаря тому, что принципиальная схема, назначение выводов и режимы работы портов с предыдущего раза остались без изменений, то и новая наша программа во многом похожа на предыдущую.

Главное **отличие** новой программы — наличие не одной, а нескольких таблиц в памяти программ. Все таблицы помещаются в конце программы. Для описания данных, размещаемых в этих таблицах, применяются как директивы `db`, так и директивы `dw`.

Первая таблица содержит коэффициенты задержки для формирования всех вариантов музыкальной длительности. Таблица начинается с адреса, соответствующего метке `tabz`. Вся таблица занимает одну строку программы (**строка 136**). Так как в нашей программе мы будем применять лишь семь вариантов длительности, таблица имеет 7 элементов. Каждый элемент записывается в память как двухбайтовое слово.

В **строках 137—140** описывается **таблица коэффициентов деления для всех нот**. Начало таблицы соответствует метке `tabkd`. Каждый элемент этой таблицы также имеет размер в два байта. Первый элемент таблицы равен нулю. Это неиспользуемый элемент. Ноты номер ноль у нас не существует. Ноль мы использовали для кодирования паузы.

В паузе не формируется звуковой сигнал, поэтому и коэффициент деления там не имеет смысла. Поэтому значение нулевого элемента массива несущественно. Описание таблицы разбито на строки. Для удобства каждая строка описывает коэффициенты деления для одной октавы. Нулевая нота выделена в отдельную строку. Последняя октава неполная, так как наша шкатулка будет использовать всего 32 ноты.

В **строках 143—200** описана **таблица мелодий**. Вернее, это не одна таблица, а семь таблиц (своя таблица для каждой из мелодий). Каждая таблица помечена своей отдельной меткой (`mel1`, `mel2` — `mel7`). Значение каждой метки — это адрес начала соответствующей мелодии. Каждое значение таблицы мелодий записывается в память в виде одного байта. Поэтому все строки, кроме последней, для каждой таблицы имеют четное число значений.

В **строках 141, 142** описана **таблица начал всех мелодий**. Начало этой таблицы отмечено меткой `tabm`. Таблица используется для того, чтобы программа могла найти адрес начала нужной мелодии по ее номеру. В качестве элементов массива выступают удвоенные значения меток `mel1`, `mel2` — `mel7`. Применение удвоенных значений обусловлено необходимостью перевода адресов из основной адресации в альтернативную. При трансляции программы вместо меток в память будут записаны конкретные адреса.

Процедура вычисления адреса

Большое количество таблиц в нашей программе заставляет позаботиться о процедуре вычисления адреса. Как нам известно из предыдущей программы, для извлечения элемента из таблицы нам необходимо произвести вычисление его адреса. В новой программе подобные вычисления нам придется выполнять для каждой таблицы.

Однотипные вычисления удобно оформить в виде подпрограммы. Эта подпрограмма занимает **строки 78—84**. Вызов подпрограммы производится по имени `addw`. Подпрограмма получает номер элемента таблицы и адрес ее начала. Номер элемента передается в подпрограмму при помощи регистра YL, а адрес — через регистровую пару Z.

Используя эти данные, подпрограмма вычисляет **адрес нужного элемента**. Для этого она сначала удваивает номер элемента (**строка 79**). Затем дополняет полученное значение до шестнадцатиразрядного путем записи в YH нулевого байта (**строка 80**). И, наконец, производит сложение двух шестнадцатиразрядных величин, находящихся к этому моменту в регистровых парах Y и Z (**строки 81, 82**). Результат вычислений при этом попадает в регистровую пару Z. Назначение команд `push` и `pop` (**строки 78, 83**), по-видимому, уже объяснять не нужно.

Текст программы «шаг за шагом»

Теперь вернемся к самому началу и рассмотрим текст программы по порядку. **Начало программы** практически полностью соответствует предыдущему примеру (см. **листинг 5.15**). Небольшое отличие лишь в модуле описания переменных (в новой программе это **строки 3—10**). Теперь там описывается гораздо больше переменных (рабочих регистров).

Название и назначение новых переменных прекрасно видны из текста программы. Без изменений остался модуль переопределения векторов прерываний (**строки 13—31**) и модуль команд инициализации (**строки 32—44**). Не изменилась даже процедура сканирования управляющих кнопок (**строки 45—52**), которая так же, как и в предыдущем примере, определяет номер первого из входов, у которого оказались замкнуты контакты. На этом сходство двух программ заканчивается. Начиная со **строки 53**, мы видим абсолютно новую программу. Рассмотрим ее подробнее.

Особенности программы

Итак, процедура, расположенная в **строках 45—52** программы, просканировала клавиатуру и нашла код первой из нажатых кнопок. Искомый код, если вы не забыли, находится в регистре `count`. Затем управление переходит к **строке 53**. С этого места начинается процедура выбора мелодии (**строки 53—58**). Суть процедуры — прочитать из таблицы `tabm` значение адреса начала этой мелодии. То есть прочитать элемент таблицы, номер которого равен коду нажатой клавиши.

Прежде чем прочитать элемент, необходимо **найти его адрес**. Для вычисления адреса используем подпрограмму `addw`. Перед тем, как вызвать подпрограмму, подготовим все данные. Номер нажатой клавиши помещаем в регистр YL (**строка 53**). Адрес начала таблицы записываем в регистровую пару Z (**строки 54, 55**). И лишь затем в **строке 56** вызывается подпрограмма `addw`.

После выхода из подпрограммы в регистровой паре Z находится результат вычислений — адрес нужного нам элемента таблицы `tabm`. Следующие две команды (**строки 57** и **58**) извлекают тот элемент (адрес начала мелодии) и помещают его в регистровую пару X. Там этот адрес будет храниться все время, пока воспроизводится именно эта мелодия.

Следующий этап — воспроизведение мелодии. Воспроизведением мелодии занимается процедура, расположенная в **строках 59—77**. Для последовательного воспроизведения нот нам понадобится указатель текущей ноты. В качестве указателя текущей ноты используется регистровая пара Z. В самом начале процедуры воспроизведения мелодии в регистровую пару Z помещается адрес начала мелодии их регистровой пары X (**строки 59, 60**).

Затем начинается цикл воспроизведения (**строки 61—77**). В этом цикле программа извлекает код ноты по адресу, на который указывает наш указатель, выделяет из кода ноты код тона и код длительности, воспроизводит ноту, а затем увеличивает значение указателя на единицу. Затем весь цикл повторяется.

Этот процесс происходит до тех пор, пока код очередной ноты не окажется равным 255 (метка конца мелодии). Прочитав этот код, программа передает управление на **строку 62**, где в регистр Z снова записывается адрес начала мелодии. Воспроизведение мелодии начнется сначала. Этот процесс должен прерваться лишь в одном случае — при отпускании управляющей кнопки.

Для проверки состояния кнопок в цикл воспроизведения мелодии включена специальная процедура (**строки 61—63**). Процедура упрощенно проверяет состояние сразу всех кнопок. Она считывает содержимое порта PD (**строка 61**) и сравнивает его с кодом 0x7F (**строка 62**). Прочитанное из порта значение может быть равно 0x7F только в одном случае — если все кнопки отпущены. Если хотя бы одна кнопка нажата, то при чтении порта мы получим другое значение.

Проверкой вышеописанного условия занимается оператор `breq` в **строке 63**. Если все кнопки оказались отпущены, этот оператор завершает цикл воспроизведения мелодии и передает управление на метку `m1`, то есть на самое начало основного цикла программы. Там происходит выключение звука, а затем новое сканирование клавиатуры.

Если хотя бы одна кнопка окажется нажатой, то цикл воспроизведения звука продолжается дальше, и управление переходит к **строке 64**, где происходит извлечение кода ноты. Так как адрес этой ноты находится в регистровой паре Z (указатель текущей ноты), то для извлечения ноты просто используется команда `lpm`.

В **строке 65** происходит проверка признака конца мелодии. Только что прочитанный код ноты сравнивается с кодом 0xFF. Оператор `breq` в **строке 66** передает управление по метке `m4`, если мелодия действительно закончилась (условие выполняется). Если код ноты не равен 0xFF, перехода не происходит, и управление переходит к **строке 67**.

В **строках 67—75** происходит обработка кода ноты. То есть из кода ноты выделяется код тона и код длительности. Сначала на код ноты накладывается маска, которая оставляет пять младших разрядов, а три старших сбрасывает (**строка 67**). Под действием маски в регистре `temp` остается код тона, который затем помещается в регистр `fnota` (**строка 68**).

Теперь нам нужно найти **код длительности ноты**. Для этого нам заново придется извлечь код ноты из памяти программ. Так как до этого момента мы не изменяли положение указателя текущей ноты, то для извлечения нет никаких препятствий. В **строке 69** мы повторно извлекаем код ноты из памяти программ. Но на этот раз значение указателя увеличивается. Теперь можно приступать к выделению кода длительности. Как вы помните, длительность кодируется тремя младшими битами кода ноты. Для выделения этих битов нам также нужно использовать **маску**. Но одной маской нам не обойтись. Нам нужно не просто

выделить три старших разряда, а сделать их младшими, как это показано на **рис. 5.16**.

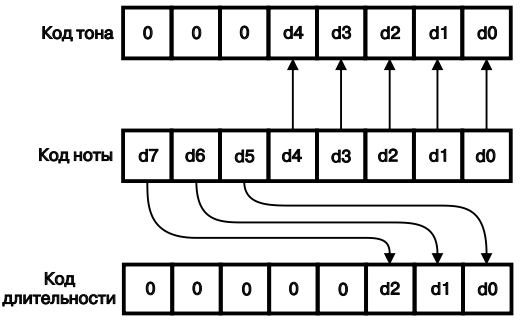


Рис. 5.16. *Разложение кода ноты*

Процедура выделения кода длительности занимает **строки 70—74**. Сначала программа производит многократный циклический сдвиг кода ноты до тех пор, пока три старших разряда не станут тремя младшими. Для сдвига используется команда `rol`. Так как сдвиг происходит через ячейку признака переноса (см. **рис. 5.15**), то нам понадобится четыре команды сдвига. Эти команды занимают в программе **строки 70—73**.

Затем в **строке 74** на полученное в результате сдвигов число накладывается маска, которая выделяет три младшие бита, а пять старших сбрасывает в ноль. Полученный таким образом код длительности записывается в регистр `dnota` (**строка 75**).

Когда **код тона** и код длительности определены, производится вызов подпрограммы воспроизведения ноты (**строка 76**). Оператор `rjmp` в **строке 77** передает управление на начало цикла воспроизведения мелодии, и цикл повторяется для следующей ноты.

Подпрограмма воспроизведения ноты занимает **строки 85—110**. Она выполняет следующие действия:

- извлекает из таблицы `tabkd` коэффициент деления, соответствующий коду ноты;
- программирует таймер и включает звук;
- затем выдерживает паузу и звук выключает.

Если код тона равен нулю (нужно воспроизвести паузу без звука), извлечение коэффициента деления и включение звука не выполняются. Подпрограмма сразу переходит к формированию паузы.

Начинается подпрограмма воспроизведения ноты с сохранения всех используемых регистров (**строки 85—88**). Затем производится проверка кода ноты на равенство нулю (**строка 89**). Если код ноты равен нулю, то оператор `breq` в **строке 90** передает управление по метке `nt1`, то есть к строке, где происходит вызов процедуры формирования задержки.

Если код ноты не равен нулю, то программа приступает к извлечению коэффициента деления. Для вычисления адреса элемента таблицы `tabkd`, где находится этот коэффициент, снова используется подпрограмма `addw`.

Код тона помещается в регистр YL (**строка 91**), а адрес начала таблицы — в регистровую пару Z (**строки 92, 93**). Вызов подпрограммы

`addw` производится в **строке 94**. В регистровой паре Z подпрограмма возвращает адрес элемента таблицы, где находится нужный нам коэффициент деления. В **строках 95, 96** из таблицы извлекается этот коэффициент. А в **строках 97, 98** он помещается в регистр совпадения таймера. В **строках 99, 100** включается звук.

В **строке 104** вызывается специальная подпрограмма, предназначенная для формирования задержки. Подпрограмма называется `wait` и формирует задержку с переменной длительностью. Длительность задержки зависит от значения регистра `dnota`. По окончании задержки звук выключается (**строки 102, 103**).

На этом можно было бы закончить процесс воспроизведения ноты. Однако это еще не все. Для правильного звучания мелодии между двумя соседними нотами необходимо обеспечить хотя бы **небольшую паузу**. Если такой паузы не будет, ноты будут звучать слитно. Это исказит мелодию, особенно если подряд идет несколько нот с одинаковым тоном. Формирование паузы между нотами происходит в **строках 104, 105**.

Вспомогательная пауза формируется при помощи уже знакомой нам подпрограммы задержки. В **строке 104** коду паузы присваивается нулевое значение (выбирается самая минимальная пауза). Затем в **строке 105** вызывается подпрограмма `wait`. После окончания паузы остается только восстановить содержимое всех со храненных регистров из стека (**строки 106—109**) и выйти из подпрограммы (**строка 110**).

Подпрограмма формирования задержки

И последнее, что нам осталось рассмотреть, — это подпрограмма формирования задержки. Текст подпрограммы занимает **строки 111—135**. Как и любая другая подпрограмма, подпрограмма `wait` в начале сохраняет (**строки 111—114**), а в конце — восстанавливает (**строки 131—134**) все используемые регистры.

Рассмотрим, как работает эта подпрограмма. Сначала определяется длительность задержки. Для этого извлекается соответствующий элемент из таблицы `tabz`. Номер элемента соответствует коду задержки, находящемуся в регистре `dnota`. Извлечение значения из таблицы производится уже знакомым нам образом. Команды, реализующие вычисление адреса нужного элемента таблицы, находятся в **строках 115—118**. Затем в **строках 119** и **120** производится чтение элемента таблицы. Прочитанный код задержки помещается в регистровую пару Y.

Теперь **наша задача**: *сформировать задержку, пропорциональную содержимому регистровой пары Y*. Так как микроконтроллер ATtiny2313 имеет только один шестнадцатиразрядный таймер, который уже занят формированием звука, будем формировать задержку программным путем. Ранее мы уже применяли один вариант такой подпрограммы

(см. **раздел 5.5**). Но в данном случае цикл формирования задержки построен немного по-другому.

Вообще-то, способов построения подобных подпрограмм может быть бесконечное множество. Все зависит от изобретательности программиста. Использованный в данном примере способ более удобен для формирования задержки переменной длительности, пропорциональной заданному коэффициенту. Главной **особенностью** нового способа является шестнадцатиразрядный параметр цикла.

Для хранения этого параметра используется регистровая пара Z. Перед началом цикла задержки в нее записывается ноль. Затем начинается цикл, на каждом проходе которого содержимое регистровой пары Z увеличивается на единицу. После каждого такого увеличения производится сравнение нового значения Z с содержимым регистровой пары Y.

Заканчивается цикл тогда, когда содержимое Z и содержимое Y окажутся равны. В результате число, записанное в регистровой паре Y, будет определять количество проходов цикла. Поэтому и время задержки, формируемое этим циклом, будет пропорционально константе задержки. Однако это время будет слишком мало для получения приемлемого темпа воспроизведения мелодий. Для того, чтобы увеличить время до нужной нам величины, внутрь главного цикла задержки помещен еще один цикл, имеющий фиксированное количество проходов.

Описанная выше процедура задержки занимает **строки 121—135**. В **строках 121, 122** производится запись нулевого значения в регистровую пару Z. Большой цикл задержки занимает **строки 123—130**. Малый внутренний цикл занимает **строки 124—125**. Для хранения параметра малого цикла используется регистр `loop`. В **строке 123** в него записывается начальное значение. **Строки 124, 125** выполняются до тех пор, пока содержимое `loop` не окажется равным нулю.

В **строке 126** содержимое регистровой пары Z увеличивается на единицу. В **строках 127—130** производится сравнение содержимого двух регистровых пар Y и Z. Сравнение производится побайтно. Сначала сравниваются младшие байты (**строка 127**). Если они не равны, оператор условного перехода в **строке 128** передает управление на начало цикла.

Если младшие байты равны, сравниваются старшие байты (**строка 129**). Если старшие байты неодинаковы, оператор `brne` в **строке 130** опять заставляет цикл начинаться с начала. И только когда оба оператора сравнения дадут положительный результат (не вызовут перехода), цикл заканчивается, и подпрограмма формирования задержки переходит к завершающей фазе (к **строкам 131—135**).

Программа на языке СИ

Возможный вариант программы на языке СИ приведен в **листинге 5.18**. Прежде чем начинать рассмотрение текста этой программы, необходимо разобраться с одним важным вопросом. В новой программе используется такой новый для нас элемент, как ссылочная (индексная) переменная.

Ссылочная переменная — это еще одна фирменная особенность классического СИ, которая широко применяется и позволяет сделать программы проще и эффективнее. В то же время никакой другой элемент не вызывает столько сложностей у начинающих программистов, как ссылочные переменные. Попробую объяснить этот вопрос как можно понятнее.

Итак, ссылочная переменная — это переменная, которая хранит указатель на другой объект. Таким объектом может быть либо другая переменная, либо элемент массива. При описании ссылочной переменной перед ее именем ставится символ * (звездочка). Вот как выглядит типичное описание ссылочной переменной:

```
int *sper;
```

В приведенном примере описывается ссылочная `sper` переменная, которая может хранить указатель на любую другую переменную или на любой элемент любого массива, но только если эта переменная или массив имеют тип `int`. Рассмотрим подробнее случай, когда **ссылочная переменная указывает на элемент массива**. Для того, чтобы поместить в ссылочную переменную указатель, указывающий на нулевой элемент массива `mass`, достаточно выполнить следующую команду:

```
sper = &mass[0];
```

Символ «&», поставленный перед переменной или массивом, — это операция определения ссылки на объект. Выражение `&mass[0]` возвращает указатель на нулевой элемент массива `mass`. Выражение `&mass[5]` возвращает указатель на пятый элемент. И так далее. Если нас интересует только начало массива, возможна и более простая запись. Допустим, ссылочной переменной `sper` нужно присвоить значение указателя на начало массива `mass` (то есть на его нулевой элемент). В этом случае можно записать

```
sper=mass;
```

Другими словами, в языке СИ имя массива является одновременно и указателем на его начало.

Допустим, переменная `sper` содержит указатель на нулевой элемент массива. Тогда `sper`+1 будет указывать на первый элемент того же мас-

сива, sper+2 на второй, и так далее. Если теперь переменной sper присвоить значение указателя — другой массив, то указанным выше способом мы получим доступ к новому массиву. Таким образом, используя одну и ту же ссылочную переменную, можно обращаться к любому элементу любого массива.

Обращаться к разным элементам одного массива можно и по-другому. Можно увеличивать значение самой ссылочной переменной. **Например**, если увеличить значение переменной sper на единицу (sper=sper+1), то после этого она будет указывать уже на следующий элемент массива.

Хочу обратить ваше внимание на то, что ссылочная переменная хранит не адрес в памяти, где хранится элемент массива, а именно указатель. **Указатель** — это не совсем адрес. Если переменная имеет тип char, то она занимает в памяти один байт. Переменная типа int занимает два байта.

Указатель автоматически учитывает этот факт. При увеличении значения указателя на единицу, он всегда указывает на следующий элемент массива, независимо от того, сколько байтов в памяти занимает каждый такой элемент.

Теперь посмотрим, как используются **указатели в программе**. Допустим, у нас есть ссылочная переменная sper, которая содержит указатель на начало массива mass. Тогда вместо выражения

```
x = mass[1];
```

можно записать

```
x = *sper;
```

В данном случае символ * означает операцию обращения к содержимому объекта, на который ссылается переменная sper. Если теперь увеличить значение переменной sper (например, при помощи команды sper++), то после этого выражение x=*sper присвоит переменной x значение уже следующего элемента массива (mass[2]).

Как видите, применение ссылочных переменных позволяет обратиться к любому элементу массива альтернативным способом. Все удобство такого способа иллюстрирует как раз наш программный пример. Как вы помните, музыкальная шкатулка должна воспроизводить одну из семи мелодий. Разумеется, для каждой из мелодий мы создадим свой отдельный массив, куда поместим все коды нот.

Однако программа должна выбрать одну из мелодий, то есть один из массивов, а затем именно из него извлекать ноты. При обычном способе доступа мы вынуждены конкретно указывать имя массива, с которым мы работаем. И мы не можем для разных мелодий указывать раз-

Двухмерный массив *— это массив, состоящий из нескольких наборов элементов (строк).*

ные массивы. В других языках программирования в таких случаях используют двухмерный массив.

Например, массив `mass[5,10]` будет состоять из пяти строк по десять элементов в каждой. Язык СИ тоже поддерживает двухмерные массивы. Для нашей задачи можно создать массив, имеющий семь строк, в каждой из которых будет храниться одна мелодия. Но в данном случае подобное решение будет иметь один **недостаток**. В двухмерном массиве все строки всегда имеют равную длину. А наши мелодии по длине разные. Конечно, можно выбрать длину для всех строк массива, равную длине самой длинной мелодии. Но это приведет к нерациональному использованию памяти.

Применение **ссылочной переменной** сразу решает все проблемы. Каждую мелодию мы помещаем в отдельный одномерный массив типа `unsigned char`. Затем мы создаем ссылочную переменную с именем `nota` (см. **строку 45** нашей программы).

Ну а дальше все просто. Для воспроизведения первой мелодии помещаем в переменную `nota` указатель на начало первого массива. Затем при помощи выражения `*nota` читаем по порядку все элементы массива и воспроизводим ноты, соответствующие прочитанным значениям. После чтения каждого очередного элемента увеличиваем содержимое переменной `nota` на единицу. Если нужно воспроизвести **вторую мелодию**, то записываем в переменную `nota` указатель на начало второго массива. Воспроизводим эту мелодию таким же способом, как и первую. Подобным же образом можно воспроизводить мелодии из любого массива.

Теперь рассмотрим подробнее программу с самого начала.

Описание программы (листинг 5.18)

Для формирования задержки мы будем использовать функцию из библиотеки delay.h. Поэтому в **строках 1, 2** программы, кроме файла описаний, мы присоединяем и эту библиотеку. Затем начинаются описания всех массивов. В **строке 3** описывается массив, содержащий величины всех музыкальных длительностей.

Так как для формирования длительности мы будем использовать функцию `delay_ms`, величина длительностей задана в миллисекундах. Как видно из текста программы, в данном случае мы используем массив типа `unsigned int`. Переменные этого типа имеют длину два байта, все 16 битов которых используются для хранения информации.

Именно такой тип наиболее подходит для хранения наших коэффициентов. Управляющее слово `fleash` перед описанием массива гарантирует, что эти данные будут размещены в программной памяти микроконтроллера.

В **строках 4, 5, 6** описывается **массив коэффициентов деления для всех нот**. В этом месте программы мы впервые используем перенос строки. Перенос строки применяется в том случае, когда текст команды не помещается в одной строке. Язык СИ разрешает свободно переносить текст на следующую строку. При этом не требуется никаких специальных директив и указателей.

Перенос допускается в том месте команды, где между двумя соседними элементами выражения можно поставить пробел. Тип массива, как и в предыдущем случае, — `usingnerd int`. Содержимое массива `tabkd` полностью соответствует содержимому таблицы с тем же названием из ассемблерного варианта программы.

В **строках 7—38** описываются **семь массивов для хранения семи мелодий**. Массивы имеют тип `unsigned char`. Переменные этого типа занимают в памяти один байт, и все восемь битов этого байта используются для хранения информации. Содержимое каждого из этих массивов полностью соответствует содержимому соответствующих таблиц в ассемблерной версии программы.

В **строке 39** описывается **массив, содержащий адрес начала каждой из семи мелодий**. Это не просто массив, а массив ссылок, на что указывает символ звездочки в тексте его описания. Так же, как и ссылочная переменная, каждый элемент массива ссылок предназначен для хранения ссылки. Данный массив тоже хранится в памяти программ, на что указывает управляющее слово `flesh` в его описании. Элементы этого массива хранят указатели на начало каждого из массивов мелодий, что указано при его инициализации (в фигурных скобках).

Строки 40—72 занимает функция `main`. Начинается функция с описания переменных (**строки 41—45**). Две рабочих переменных `count` и `temp`, а также переменная для хранения кода тона (`tnota`) и переменная для хранения кода длительности (`dnota`) нам уже знакомы. Мы использовали их в предыдущей программе.

Интерес представляет описание переменной `nota`. Это ссылочная переменная, которая предназначена для хранения указателей на объекты в программной памяти, имеющие тип `unsigned char`. Она будет использоваться нами для обращения к элементам массивов, хранящим коды нот. Эти массивы, как уже говорилось, расположены в программной памяти. Поэтому в описании переменной имеется слово `flash`, а перед именем переменной в ее описании стоит символ звездочки. То есть это ссылка на массивы типа `unsigned char`, расположенные во `flesh`.

В **строках 46—52** расположен **блок инициализации**. Эта часть программы полностью повторяет аналогичную часть программы из предыдущего примера (см. **листинг 5.16**).

Строки 53—72 занимает **основной цикл программы**. Цикл состоит всего из двух процедур. В начале цикла (**строки 54—59**) расположена **процедура сканирования клавиатуры**. Эта процедура один к одному скопирована из предыдущего примера (см. **листинг 5.16 строки 14—21**).

При обнаружении нажатой кнопки управление передается по метке `m3` (в новой программе это **строка 60**). Как вы помните, номер нажатой кнопки при выходе из процедуры сканирования содержится в переменной `count`.

Листинг 5.18

```
     /*****************************************************
     Project : Prog 9
     Version : 1
     Date    : 31.01.2006
     Author  : Belov
     Comments: Музыкальная шкатулка
     Chip type           : ATtiny2313
     Clock frequency     : 4,000000 MHz
     *****************************************************/

 1   #include <tiny2313.h>
 2   #include <delay.h>

     // Объявление и инициализация массивов

     // Таблица задержек
 3   flash unsigned int tabz[] =  {16,32,64,128,256,512,1024};

     // Массив коэффициентов деления
 4   flash unsigned int tabkd[] = {0,4748,4480,4228,3992,3768,3556,3356,3168,2990,2822,
 5         2664,2514,2374,2240,2114,1996,1884,1778,1678,1584,1495,1411,1332,1257,
 6         1187,1120,1057,  998,942,889,839,792};

     // Таблицы мелодий
     // В траве сидел кузнечик
 7   flash unsigned char mel1[] = {109,104,109,104,109,108,108,96,108,104,108,104,108,
 8         109,109,96,109,104,109,104,109,108,108,96,108,104,108,104,108,141,96,109,
 9         111,  79,  79,111,111,112,80,80,112,112,112,111,109,108,109,109,  96,109,111,
10         79,79,111,111,112,80,80,112,112,112,111,109,108,141,128,96,255};
     // Песенка крокодила Гены
11   flash unsigned char mel2[] = {109,110,141,102,104,105,102,109,110,141,104,105,107,
12         104,109,110,141,104,105,139,109,110,173,96,114,115,146,109,110,112,109,114,
13         115,146,107,109,110,114,112,110,146,109,105,136,107,105,134,128,128,102,105,
14         137,136,128,104,107,139,137,128,105,109,141,139,128,110,109,176,112,108,109,
15         112,144,142,128,107,110,142,141,128,105,109,139,128,173,134,128,128,109,112,
16         144,142,128,107,110,142,141,128,105,109,139,128,173,146,128,255};
     // В лесу родилась елочка
17   flash unsigned char mel3[] = {132,141,141,139,141,137,132,132,132,141,141,142,139,
18         176,128,144,146,146,154,154,153,151,149,144,153,153,151,153,181,128,96,255};
     // Happy births day to you
19   flash unsigned char mel4[] = {107,107,141,139,144,143,128,107,107,141,139,146,144,
20         128,107,107,151,148,146,112,111,149,117,117,148,144,146,144,128,255};
     // С чего начинается родина
21   flash unsigned char mel5[] = {99,175,109,107,106,102,99,144,111,175,96,99,107,107,
22         107,107,102,104,170,96,99,109,109,109,107,106,143,109,141,99,109,109,109,
23         109,104,106,171,96,99,111,109,107,106,102,99,144,111,143,104,114,114,114,114,
24         109,111,176,  96,104,116,112,109,107,106,64,73,143,107,131,99,144,80,80,112,
25         111,64,75,173,128,255};
     // Из кинофильма "Веселые ребята"
26   flash unsigned char mel6[] = {105,109,112,149,116,64,80,148,114,64,78,146,112,96,105,
27         105,109,144,111,64,80,145,112,64,81,178,96,117,117,117,149,116,64,82,146,112,
28         64,79,146,144,96,105,105,107,141,108,109,112,110,102,104,137,128,96,105,105,
29         105,137,102,64,73,142,105,107,109,64,75,137,96,105,105,105,137,102,105,142,112,
30         64,82,180,96,116,116,116,148,114,112,142,109,64,78,146,144,96,105,105,107,141,
31         108,109,112,110,102,104,169,96,96,255};
     // Улыбка
32   flash unsigned char mel7[] = {107,104,141,139,102,105,104,102,164,128,104,107,109,109,
33         109,111,114,112,111,109,144,139,128,109,111,144,  96,111,109,104,107,105,173,128,
34         111,109,112,107,111,109,109,107,102,104,134,132,128,100,103,107,107,107,107,139,
```

```
35          112,100,103,102,102,102,134,102,103,107,105,107,108,108,108,108,107,105,107,108,
36          144,142,128,112,107,110,140,112,105,108,107,107,107,105,140,139,139,112,103,102,
37          103,105,108,107,105,103,128,112,107,110,108,108,108,140,112,105,108,107,107,107,
38          139,112,103,102,103,105,108,107,105,103,105,139,132,128,96,96,96,255};

        // Таблица начал всех мелодий
39      flash unsigned char *tabm[] = {mel1, mel2, mel3, mel4, mel5, mel6, mel7};

40      void main(void)
        {
41      unsigned char count;    // Определяем переменную count
42      unsigned char temp;     // Определяем переменную temp
43      unsigned char fnota;    // Код тона ноты
44      unsigned char dnota;    // Код длительности ноты
45      flash unsigned char *nota;  // Ссылка на текущую ноту

46      PORTB=0x08;   // Инициализация порта PB
47      DDRB=0x08;

48      PORTD=0x7F;   // Инициализация порта PD
49      DDRD=0x00;

50      ACSR=0x80;   // Инициализация (отключение) компаратора

51      TCCR1A=0x00;   // Инициализация таймера счетчика T1
52      TCCR1B=0x09;

53      while (1)
            {
54      m1:   temp=PIND;
55            for (count=0; count<7; count++)  // Цикл сканирования датчиков
              {
56              if ((temp&1)==0) goto m3;   // Проверка младшего бита переменной temp
57              temp >>= 1;                 // Сдвиг содержимого temp
              }
58      m2:   TCCR1A=0x00;          // Выключение звука
59            goto m1;              // Переход на начало

              // Воспроизведение мелодии
60      m3:   nota = tabm[count];                       // Устанавливаем указатель на первую ноту
61      m4:   if (PIND==0x7F) goto m2;                  // Если ни одна кнопка не нажата, закончить
62            if (*nota==0xFF) goto m3;                 // Проверка на конец мелодии
63            fnota = (*nota)&0x1F;                     // Определяем код тона
64            dnota = ((*nota)>>5)&0x07;                // Определяем код длительности
65            if (fnota==0) goto m5;                    // Если пауза не воспроизводим звук
66            OCR1A=tabkd[fnota];                       // Программируем частоту звука
67            TCCR1A=0x40;                              // Включаем звук
68      m5:   delay_ms (tabz[dnota]);                   // Формируем задержку
69            TCCR1A=0x00;                              // Выключаем звук
70            delay_ms (tabz[0]);                       // Задержка между нотами
71            nota++;                                   // Перемещаем указатель на следующую ноту
72            goto m4;                                  // К началу цикла
            };
        }
```

Строки 60—72 занимает **процедура проигрывания мелодии**. Проигрывание начинается с того, что в переменную `nota` помещается указатель на массив, содержащий нужную нам мелодию (**строка 60**). А указатель — это элемент массива `tabm` с номером, равным коду нажатой кнопки. В **строках 61—72** находится цикл, который последовательно считывает мелодию нота за нотой и проигрывает прочитанные ноты. Цикл организован при помощи оператора безусловного перехода (**строка 72**).

Для перемещения вдоль массива содержимое переменной `nota` каждый раз увеличивается на единицу (**строка 71**). В этом же цикле производятся проверка состояния клавиатуры (нажата ли еще хоть одна кнопка) и проверка признака конца мелодии. Рассмотрим подробнее, как все это делается.

Проверка состояния клавиатуры происходит в **строке 61**. Если содержимое регистра PIND равно 0x7F, то воспроизведение мелодии прекращается. Управление передается по метке `m2`. Там происходит выклю-

чение звука, а затем переход по метке `m1`, то есть к началу основного цикла программы.

Если хоть одна кнопка еще нажата, перехода не происходит и воспроизведение мелодии продолжается. В **строке 62** производится проверка на конец мелодии. Содержимое элемента массива, на который указывает ссылочная переменная `nota` (код ноты), проверяется на равенство числу 0xFF. Если код ноты равен 0xFF, то управление передается по метке `m3`, где указатель снова устанавливается на начало мелодии.

В **строке 63** вычисляется значение **кода тона**. Для этого на код ноты, на который указывает переменная `nota`, накладывается маска. Наложение маски производится при помощи оператора «&». Полученный код тона записывается в переменную `fnota`.

В **строке 64** производится вычисление **кода длительности**. Для этого применяется составное математическое выражение. Операция `(*nota)>>5` сдвигает биты кода ноты на пять шагов вправо. При этом три старших разряда кода становятся тремя младшими. Мы применяем сдвиг вправо потому, что циклический сдвиг влево, использованный нами в Ассемблере, язык СИ не поддерживает. Язык СИ может выполнять только логический сдвиг, но не циклический. На полученное в результате сдвига число налагается маска 0x07. Полученный таким образом код длительности записывается в переменную `dnota`.

В **строке 65** происходит **проверка кода тона на равенство нулю**. Если код окажется равным нулю, то управление передается по метке `m5`, то есть к строке, где формируется пауза, обходя строки, где формируется звук.

Звук формируется в **строках 66, 67**. Сначала в регистр совпадения OCR1A помещается коэффициент деления из массива `tabkd`. Причем указатель массива равен коду тона. Затем в регистр управления TCCR1A записывается код, который подключает таймер к выводу OC1A и, тем самым, включает звук.

В **строке 68** происходит **вызов функции задержки**. В качестве параметра в эту функцию передается коэффициент, извлекаемый из массива `tabz`. Указатель массива при этом равен коду длительности. После выхода из функции задержки звук выключается.

Для этого в регистр TCCR1A записывается нулевое значение (**строка 69**). В **строке 70** формируется пауза между нотами. В качестве параметра для функции `delay_ms` в этом случае используется нулевой элемент массива `tabz`, то есть вырабатывается пауза минимальной длительности.

В **строке 71**, как уже говорилось, происходит **приращение содержимого указателя `nota`**. Оператор безусловного перехода в **строке 72** замыкает цикл воспроизведения мелодии.

5.11. Кодовый замок

Постановка задачи

Для завершения практикума я подбирал задачу достаточно сложную и интересную, способную как увлечь, так и научить работать с еще неохваченными элементами микроконтроллера. Самым удобным примером, на мой взгляд, является **кодовый замок**. Вообще, микроконтроллеры AVR с их встроенной энергонезависимой памятью (EEPROM) дают широкий простор для разработчика подобных конструкций. Память EEPROM идеально подходит для хранения кода. Причем такой код всегда легко поменять.

При разработке замка мне не хотелось быть тривиальным. Поэтому я предлагаю не совсем обычный замок. Представьте себе кодовый замок, который может воспринимать в качестве кодовой комбинации не только отдельно нажимаемые кнопки, но и любые их сочетания. Например, попарно нажимаемые кнопки, комбинации типа «Нажать кнопку 6 и, не отпуская, набрать код 257».

И вообще, выбрать любую комбинацию любых кнопок в любом сочетании. Мною был разработан такой замок. Его я и хочу предложить вашему вниманию.

Принцип действия замка следующий: *в режиме записи кода владелец нажимает кнопки набора кода в любом порядке и в любых комбинациях*. Микроконтроллер отслеживает все изменения на клавиатуре и записывает их в ОЗУ. Длина кодовой последовательности ограничена только размерами ОЗУ. Сигналом к окончанию ввода кода служит прекращение манипуляций с клавиатурой.

Считается, что манипуляции закончились, если состояние клавиатуры не изменилось в течение контрольного промежутка времени. Я выбрал его примерно равным одной секунде. Сразу по окончании процесса ввода кода (по окончании контрольного промежутка времени) микропроцессор записывает принятый таким образом код в EEPROM. Код представляет собой последовательность байтов, отражающих все состояния клавиатуры во время набора. После того, как коды будут записаны, замок можно перевести в рабочий режим. Для этого предусмотрен специальный тумблер выбора режимов.

В рабочем режиме замок ждет ввода кода. Для открывания двери необходимо повторить те же самые манипуляции с кнопками, которые вы делали в режиме записи. Микроконтроллер так же, как и в предыдущем случае, отслеживает эти манипуляции и записывает полученный таким образом код в ОЗУ. По окончании ввода кода (по истечении контрольного промежутка времени) программа переходит в режим сверки кода, находящегося в ОЗУ, и кода, записанного в EEPROM. Сначала

сравнивается длина обоих кодов. Затем коды сверяются побайтно. Если сравнение прошло успешно, микроконтроллер подает сигнал на механизм открывания замка.

Итак, **сформулируем задачу** следующим образом:

«Создать схему и программу электронного кодового замка, имеющего десять кнопок для ввода кода, обозначенных цифрами от «0» до «9». Замок должен иметь переключатель режимов «Запись/Работа». В случае правильного набора кода замок должен включать исполнительный механизм замка (соленоид или электромагнитную защелку). Ввод кода должен производиться описанным выше способом.»

Алгоритм

При составлении данного алгоритма нам не обойтись без такого понятия, как **«код состояния клавиатуры»**. Что такое код состояния? Все кнопки клавиатуры подключаются к микроконтроллеру при помощи портов ввода-вывода. Для подключения десяти кнопок (кнопки «0»—«9») одного порта недостаточно. Несколько кнопок придется подключить ко второму.

Контроллер читает содержимое этих портов и получает код, соответствующий их состоянию. Каждой кнопке клавиатуры в этом коде будет соответствовать свой отдельный бит. Когда кнопка нажата, соответствующий бит будет равен нулю. Когда отпущена — единице. Поэтому при разных сочетаниях нажатых и отпущенных кнопок код состояния клавиатуры будет иметь разные значения.

В момент включения питания все кнопки замка должны быть отпущены. Если это не так, то возникает неопределенность в работе замка. Поэтому наш **алгоритм должен начинаться** с процедуры ожидания отпускания всех кнопок. Как только все кнопки окажутся отпущенными или в случае, если они вообще не были нажаты, начинается другая **процедура ожидания**. На сей раз программа ожидает момента нажатия кнопок. Это как раз тот режим работы, в котором замок будет находиться большую часть времени. В момент нажатия любой из кнопок начинается цикл ввода ключевой комбинации.

Процедура ввода ключевой комбинации представляет собой многократно повторяющийся процесс, периодически считывающий код состояния клавиатуры. Каждый раз после очередного считывания кода программа проверяет, не изменился ли этот код. Как только код изменится, новое его значение записывается в очередную ячейку ОЗУ. В результате, пока состояние клавиатуры не изменяется, программа находится в режиме ожидания.

Как только состояние изменилось, происходит запись нового значения кода состояния в память. Поэтому ключевая комбинация, запи-

санная в ОЗУ, будет представлять собой перечисление всех значений кода состояния клавиатуры, которое он принимал в процессе ввода ключевой комбинации. Длительность же удержания кнопок в каждом из состояний в память не записывается.

Однако это лишь приблизительный алгоритм процедуры ввода ключевой комбинации. Так сказать, ее костяк. На самом деле алгоритм немного сложнее. Обнаружив изменение состояния клавиатуры, программа не сразу записывает новый код в ОЗУ. В целях борьбы с дребезгом контактов, а также для компенсации неточности одновременного нажатия нескольких кнопок, программа сначала выдерживает специальную защитную паузу, затем повторно считывает код состояния клавиатуры, и лишь после этого записывает новый код в ОЗУ.

Длительность защитной паузы выбрана равной 48 мс. Такая пауза особенно полезна в случае, если при наборе ключевой комбинации вы хотите использовать одновременное нажатие кнопок. Как бы вы не старались нажать кнопки одновременно, вам этого не удастся. Все равно будет какое-то расхождение в моменте замыкания контактов. Причем порядок замыкания контактов будет зависеть от многих факторов и практически является случайным.

Если не принять специальных мер, то в момент такого нажатия программа зафиксирует не одно, а несколько последовательных изменений кода состояния клавиатуры. Если полученная таким образом кодовая комбинация будет записана в EEPROM, то открыть такой замок будет практически невозможно.

При попытке повторить те же нажатия, замыкание контактов будут происходить в другом порядке. Программа воспримет его как совершенно другой код. Защитная пауза решает эту проблему. В каком бы порядке ни замыкались контакты при одновременном нажатии нескольких кнопок, после паузы все эти процессы закончатся. Повторное считывание даст уже устоявшийся код состояния клавиатуры. Повторить такую комбинацию не составит труда.

Кроме защитной паузы, для борьбы с дребезгом применяется **многократное считывание кода состояния**. То есть на самом деле каждый раз происходит не одно, а несколько последовательных операций по считыванию кода состояния. Считывание происходит до тех пор, пока несколько раз подряд будет получен один и тот же код.

Теперь поговорим о том, **как программа выходит из процедуры ввода ключевой комбинации**. Как уже говорилось ранее, для выхода из процедуры используется защитный промежуток времени. Для формирования этого промежутка применяется **таймер**. Таймер должен работать в режиме Normal. В этом режиме он просто считает тактовые импульсы.

Процедура ввода кодовой комбинации устроена таким образом, что при каждом нажатии или отпускании любой из кнопок таймер сбрасывается в ноль. В промежутке между нажатиями его показания увеличиваются под действием тактового сигнала. Если в течение защитного промежутка времени не будет нажата ни одна кнопка, показания таймера увеличатся до контрольного предела. Программа постоянно проверяет это условие. Как только показания счетчика превысят контрольный предел, процедура ввода кодовой комбинации завершается. Величина контрольного промежутка времени равна 1 с.

Дальнейшие действия после выхода из процедуры ввода кодовой комбинации определяются состоянием переключателя режимов работы. Если контакты переключателя замкнуты, программа переходит к процедуре записи кодовой комбинации в EEPROM.

Сначала в EEPROM записывается длина кодовой комбинации. А затем байт за байтом и сама комбинация. Если контакты переключателя режима работы разомкнуты, то программа переходит к процедуре проверки кода. Эта процедура сначала извлекает из EEPROM записанную ранее длину кодовой комбинации и сравнивает ее с длиной только что введенной комбинации.

Если две эти величины не равны, процедура проверки кода сразу же завершается с отрицательным результатом. Если длина обеих комбинаций одинакова, то программа приступает к побайтному их сравнению. Для этого она поочередно извлекает из EEPROM ранее записанные туда байты и сравнивает каждый из них с соответствующими байтами в ОЗУ. При первом же несовпадении процесс сравнения также завершается. И завершается отрицательно.

И только в том случае, если все байты в ОЗУ и в EEPROM окажутся одинаковыми, сравнение считается успешным. В случае успешного сравнения программа переходит к **процедуре открывания замка**. Процедура открывания начинается с выдачи открывающего сигнала на исполнительный механизм. Затем программа выдерживает паузу в 2 с и снимает сигнал. Этого времени достаточно для того, чтобы открыть дверь. Затем замок переходит в исходное состояние.

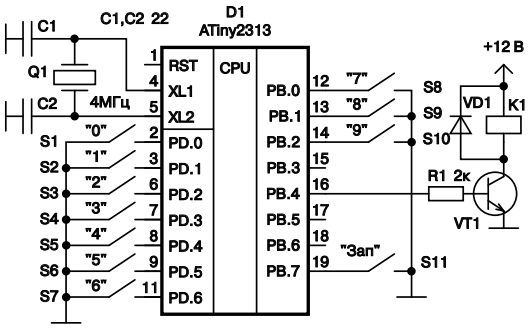


Рис. 5.17. *Схема кодового замка*

Схема

Возможный вариант схемы замка приведен на **рис. 5.17**. Кнопки S1—S10 служат для

набора кода. Переключатель S11 предназначен для выбора режима работы. Если контакт переключателя S11 замкнут, замок переходит в режим «Запись». Разомкнутые контакты соответствуют режиму «Работа».

Схема управления механизмом замка состоит из транзисторного ключа VT1 и электромагнитного реле K1. Резистор R1 ограничивает ток базы ключа. Диод VD1 служит для защиты от напряжения самоиндукции, возникающей на катушке реле. Питание реле осуществляется от отдельного источника +12 В (питание микроконтроллера +5 В). Если в качестве VT1 применять транзистор КТ315, то электромагнитное реле может иметь рабочее напряжение +12 В и рабочий ток не более 250 мА. Контакты реле должны быть рассчитаны на управление исполнительным механизмом (соленоидом).

Обратите внимание, что в данной схеме одни линии порта PB будут работать как входы, а другие (в частности линия PB.4) — как выходы. При распределении выводов порта между периферийными устройствами учитывалась возможность объединения замка с музыкальной шкатулкой. В этом случае шкатулка может управляться одной кнопкой и служить дверным звонком.

Программа на Ассемблере

Один из возможных вариантов программы на Ассемблере приведен в **листинге 5.19**. Прежде чем переходить к подробному описанию ее работы, разберемся в некоторых общих вопросах. Начнем с **кода состояния клавиатуры**. Как уже говорилось ранее, код состояния должен иметь отдельный бит для каждой кнопки клавиатуры. Итого, получается десять битов.

Одного байта явно мало. Значит, мы должны использовать двухбайтовый код состояния. Самый простой способ получить двухбайтовый код состояния — это прочитать сначала содержимое порта PD, а затем — содержимое порта PB. Затем нужно наложить на каждый из полученных байтов маску.

Маска должна обнулить все ненужные нам разряды и оставить разряды, к которым подключены наши кнопки. Для числа, прочитанного из порта PD, маска должна обнулить самый старший разряд и оставить все остальные. Для числа, прочитанного из порта PB, нужно, напротив, оставить три младших разряда и обнулить все остальные. Полученные таким образом два байта мы и будем считать кодом состояния клавиатуры.

В том случае, если все десять кнопок (S1—S10) отпущены, код состояния клавиатуры равен 0x7F, 0x07 (0b01111111, 0b00000111). В таком коде значащие биты, отражающие состояние той или иной кнопки, равны единице, а все остальные биты равны нулю. Если нажать любую кнопку, то код состояния изменится. Соответствующий этой кнопке бит примет

нулевое значение. Таким образом, любое изменение состояния клавиатуры вызовет соответствующее изменение кода.

Теперь **вернемся к тексту программы**. В программе применяются следующие новые для нас операторы.

cli

Общий запрет прерываний. Действие данной команды обратно действию уже знакомой нам команды `sei`. Команда не имеет параметров и служит для сброса флага I в регистре SREG.

st

Косвенная запись в память. Команда имеет три модификации:

```
st W,Rd        st W+,Rd        st -W,Rd,
```

где W — это одна из регистровых пар (X, Y или Z). Rd — имя одного из регистров общего назначения. Независимо от модификации команда выполняет запись содержимого регистра Rd в ОЗУ по адресу, который хранится в регистровой паре W.

При этом **первая модификация команды** не изменяет содержимое регистровой пары W. **Вторая модификация** увеличивает содержимое регистровой пары на единицу после того, как произойдет запись. А **третья модификация** команды уменьшает на единицу содержимое регистровой пары перед тем, как произойдет запись в ОЗУ.

В **строке 75** нашей программы (см. **листинг 5.19**) содержимое регистра `XL` записывается в ОЗУ по адресу, который хранится в регистровой паре Z. После этого содержимое регистровой пары Z увеличивается на единицу.

ld

Косвенное чтение из памяти. Данная операция является обратной по отношению к предыдущей. Она тоже имеет три модификации:

```
ld Rd,W    ld Rd,W+    ld Rd,-W
```

Операция производит чтение байта из ячейки ОЗУ, адрес которой хранится в регистровой паре W (то есть X, Y или Z) и записывает прочитанный байт в регистр общего назначения Rd. Содержимое регистровой пары так же, как и в предыдущем случае, ведет себя по-разному, в зависимости от модификации команды. То есть оно либо не изменяется, либо увеличивается после чтения, либо уменьшается прежде, чем байт будет прочитан.

В **строке 97** нашей программы (**листинг 5.19**) читается байт из ячейки ОЗУ, адрес которой хранится в регистровой паре Z, и записыва-

ется в регистр `data`. Затем содержимое регистровой пары Z увеличивается на единицу.

brsh

Переход по условию «больше или равно». В качестве условия для перехода выступает содержимое флага переноса C. Флаг переноса устанавливается по результатам операции сравнения или вычитания. Команда имеет всего один параметр — **относительный адрес перехода**. Переход выполняется в том случае, если флаг переноса равен нулю. А это происходит только тогда, когда в предшествующей операции сравнения (вычитания) второй операнд окажется больше или равен первому.

sbic

Оператор типа «проверить — пропустить». Общая форма записи команды:

```
sbic A,n,
```

где `A` — номер регистра ввода-вывода; `n` — номер разряда.

Вместо номера регистра и номера разряда может использоваться имя регистра и имя разряда. Обычно используются стандартные имена от фирмы Atmel. Команда проверяет содержимое разряда номер `n` регистра A. Если разряд сброшен, то очередная команда программы не выполняется.

Пример использования данной команды — **строка 159** нашей программы (см. **листинг 5.19**). В этой строке команда `sbic` проверяет бит `EEWE` регистра `EECR`. Если этот бит сброшен, то команда в **строке 160** не выполняется, а управление передается к **строке 161**. Если бит установлен, то выполняется команда в **строке** 160. Команда `sbic` имеет одно ограничение. Она работает с регистрами ввода-вывода с адресами в диапазоне от 0 до 31.

cbr

Сброс разрядов РОН. Данный оператор предназначен для одновременного сброса нескольких разрядов. Оператор имеет два параметра. **Первый параметр** — это имя регистра общего назначения, разряды которого должны быть сброшены. **Второй параметр** — это маска сброса разрядов. В данном случае маска — это двоичное число, у которого в единицу установлены те разряды, которые должны быть сброшены. Например, в **строке 129** программы (**листинг 5.19**) сбрасываются разряды регистра XH. Значение маски равно 0xF8. В двоичном виде число 0xF8 выглядит так: 0b11111000. Поэтому в результате действия команды `cbr` в **строке 129** пять старших разрядов числа, находящегося в реги-

стре XH, будут сброшены в ноль, а три младшие останутся без изменений. Это альтернативный способ наложения маски.

Описание программы (листинг 5.19)

Строки 1, 2 программы, я думаю, вопросов не вызывают. В **строках 3—11** происходит описание всех используемых в программе переменных. Назначение каждой из этих переменных мы рассмотрим в ходе описания принципов работы программы. Далее, в **строках 12—14** происходит описание констант. Каждая из трех констант определяет один из параметров нашего устройства. Остановимся на этом подробнее.

Константа `bsize` (**строка 12**) определяет **размер буфера для хранения кодовой комбинации**. Этот размер выбран равным 60 ячейкам. Учитывая, что в буфер будут записываться коды состояния клавиатуры, а каждый такой код состоит из двух байтов, в буфер указанного размера можно записать последовательность из 30 кодов.

Листинг 5.19

```
        ;###############################################
        ;##              Пример 10                    ##
        ;##            Кодовый замок                  ##
        ;###############################################

        ;------------------------- Псевдокоманды управления

1       .include "tn2313def.inc"            ; Присоединение файла описаний
2       .list                               ; Включение листинга

        ;-------------------------Модуль описаний

3       .def    drebL = R1                  ; Буфер антидребезга младший байт
4       .def    drebH = R2                  ; Буфер антидребезга старший байт
5       .def    temp = R16                  ; Вспомогательный регистр
6       .def    data = R17                  ; Регистр передачи данных
7       .def    flz = R18                   ; Флаг задержки
8       .def    count = R19                 ; Регистр передачи данных
9       .def    addre = R20                 ; Указатель адреса в EEPROM
10      .def    codL = R21                  ; Временный буфер кода младший байт
11      .def    codH = R22                  ; Временный буфер кода старший байт

        ;------------------------- Определение констант

12      .equ    bsize = 60                  ; Размер буфера для хранения кода
13      .equ    kzad  = 3000                ; Константа, определяющая длительность защитной паузы
14      .equ    kandr = 20                  ; Константа антидребезга

        ;------------------------- Резервирование ячеек памяти (SRAM)

15              .dseg                       ; Выбираем сегмент ОЗУ
16              .org            0x60        ; Устанавливаем текущий адрес сегмента

17      bufr:   .byte           bsize       ; Буфер для приема кода

        ;------------------------- Резервирование ячеек памяти  (EEPROM)

18              .eseg                       ; Выбираем сегмент EEPROM
19              .org            0x08        ; Устанавливаем текущий адрес сегмента

20      klen:   .byte           1           ; Ячейка для хранения длины кода
21      bufe:   .byte           bsize       ; Буфер для хранения кода

        ;------------------------- Начало программного кода

22              .cseg                       ; Выбор сегмента программного кода
23              .org            0           ; Установка текущего адреса на ноль
```

```
start:  rjmp    init        ; Переход на начало программы
        reti                ; Внешнее прерывание 0
        reti                ; Внешнее прерывание 1
        reti                ; Таймер/счетчик 1, захват
        rjmp    propr       ; Таймер/счетчик 1, совпадение, канал A
        rjmp    propr       ; Таймер/счетчик 1, прерывание по переполнению
        reti                ; Таймер/счетчик 0, прерывание по переполнению
        reti                ; Прерывание UART прием завершен
        reti                ; Прерывание UART регистр данных пуст
        reti                ; Прерывание UART передача завершена
        reti                ; Прерывание по компаратору
        reti                ; Прерывание по изменению на любом контакте
        reti                ; Таймер/счетчик 1. Совпадение, канал B
        reti                ; Таймер/счетчик 0. Совпадение, канал B
        reti                ; Таймер/счетчик 0. Совпадение, канал A
        reti                ; USI готовность к старту
        reti                ; USI Переполнение
        reti                ; EEPROM Готовность
        reti                ; Переполнение охранного таймера

;*****************************************************
;*                                                   *
;*              Модуль инициализации                 *
;*                                                   *
;*****************************************************

init:
;-------------------------- Инициализация стека

        ldi         temp, RAMEND    ; Выбор адреса вершины стека
        out         SPL, temp       ; Запись его в регистр стека

;-------------------------- Инициализация портов B/B

        ldi         temp,0x18       ; Инициализация порта PB
        out         DDRB, temp
        ldi         temp,0xE7
        out         PORTB, temp

        ldi         temp, 0x7F      ; Инициализация порта PD
        out         PORTD, temp
        ldi         temp, 0
        out         DDRD, temp

;-------------------------- Инициализация (выключение) компаратора

        ldi         temp, 0x80
        out         ACSR, temp

;-------------------------- Инициализация таймера

        ldi         temp, high(kzad )       ; Записываем коэффициент задержки
        out         OCR1AH, temp
        ldi         temp, low(kzad )
        out         OCR1AL, temp
        ldi         temp, 0x03              ; Выбор режима работы таймера
        out         TCCR1B, temp

;*****************************************************
;*                                                   *
;*            Начало основной программы              *
;*                                                   *
;*****************************************************

main:   ldi         codL,0x7F       ; Код для сравнения (младший байт)
        ldi         codH,0x07       ; Код для сравнения (старший байт)

m0:     rcall       incod           ; Ввод и проверка кода клавиш
        brne        m0              ; Если хоть одна не нажата, продолжаем

m1:     rcall       incod           ; Ввод и проверка кода клавиш
        breq        m1              ; Если не одна не нажата, продолжаем

m2:     ldi         ZH, high(bufr)    ; Установка указателя на начало буфера
        ldi         ZL, low(bufr)     ;
        clr         count             ; Сброс счетчика байт

;---------------------- Цикл ввода кода

m3:     cli                         ; Запрет всех прерываний
        ldi         data,1          ; Вызываем задержку первого типа
        rcall       wait            ; К подпрограмме задержки
```

```
 74   m5:     rcall       incod           ; Ввод и проверка кода кнопок
 75           st          Z+,XL           ; Записываем его в буфер
 76           st          Z+,XH
 77           inc         count           ; Увеличение счетчика байтов
 78           inc         count
 79           cpi         count, bsize    ; Проверяем, не конец ли буфера
 80           brsh        m7              ; Если конец, завершаем ввод кода

 81           mov         codL, XL        ; Записываем код как старый
 82           mov         codH, XH

 83           ldi         data,2          ; Вызываем задержку третьего типа
 84           rcall       wait            ; К подпрограмме задержки

 85   m6:     rcall       incod           ; Ввод и проверка кода кнопок
 86           brne        m3              ; Если изменилось, записываем в буфер
 87           cpi         flz,1           ; Проверка окончания фазы ввода кода
 88           brne        m6

      ; ------------------------------ Опрос состояния тумблера

 89   m7:     sbic        PINB,7          ; Проверка состояния тумблера
 90           rjmp        m9              ; К процедуре проверки кода

      ; ------------------------------ Процедура записи кода

 91           mov         data, count     ; Помещаем длину кода в data
 92           ldi         addre, klen     ; Адрес в EEPROM для хранения длины кода
 93           rcall       eewr            ; Записываем в длину кода EEPROM

 94           ldi         addre, bufe       ; В регистр адреса начало буфера в EEPROM
 95           ldi         ZH, high(bufr)    ; В регистровую пару Z записываем
 96           ldi         ZL, low(bufr)     ; адрес начала буфера в ОЗУ

 97   m8:     ld          data, Z+        ; Читаем очередной байт из ОЗУ
 98           rcall       eewr            ; Записываем в длину кода EEPROM
 99           dec         count           ; Декремент счетчика байтов
100           brne        m8              ; Если не конец, продолжаем запись

101           rjmp        m11             ; К процедуре открывания замка

      ; ------------------------------ Процедура проверки кода

102   m9:     ldi         addre, klen     ; Адрес хранения длины кода
103           rcall       eerd            ; Чтение длины кода из EEPROM
104           cp          count, data     ; Сравнение с новым значением
105           brne        m13             ; Если не равны, к началу

106           ldi         addre, bufe       ; В YL начало буфера в EEPROM
107           ldi         ZH, high(bufr)    ; В регистровую пару Z записываем
108           ldi         ZL, low(bufr)     ; адрес начала буфера в ОЗУ

109   m10:    rcall       eerd            ; Читаем очередной байт из EEPROM
110           ld          temp, Z+        ; Читаем очередной байт из ОЗУ

111           cp          data, temp      ; Сравниваем два байта разных кодов
112           brne        m13             ; Если не равны, переходим к началу

113           dec         count           ; Уменьшаем содержимое счетчика байтов
114           brne        m10             ; Если не конец, продолжаем проверку

      ; ------------------------------ Процедура открывания замка

115   m11:    sbi         PORTB,4         ; Команда "Открыть замок"
116           ldi         data,3          ; Вызываем задержку третьего типа
117           rcall       wait
118           cbi         PORTB,4         ; Команда "Закрыть замок"

119   m13:    rjmp        main            ; Перейти к началу


      ;*****************************************************
      ;*                                                   *
      ;*             Вспомогательные процедуры             *
      ;*                                                   *
      ;*****************************************************

      ; ------------------------------ Ввод и проверка 2 байтов с клавиатуры

120   incod:  push        count
```

```
121             ldi     XL,0            ; Обнуление регистровой пары X
122             ldi     XH,0

123     ic1:    ldi     count, kandr    ; Константа антидребезга
124             mov     drebL, XL       ; Старое значение младший байт
125             mov     drebH, XH       ; Старое значение старший байт

126     ic2:    in      XL,PIND         ; Вводим код (младший байт)
127             cbr     XL,0x80         ; Накладываем маску
128             in      XH,PINB         ; Вводим код (старший байт)
129             cbr     XH,0xF8         ; Накладываем маску

130     ic3:    cp      XL, drebL       ; Сверяем младший байт
131             brne    ic1             ; Если не равен, начинаем с начала
132             cp      XH, drebH       ; Сверяем старший байт
133             brne    ic1             ; Если не равен, начинаем с начала

134     ic4:    dec     count           ; Уменьшаем счетчик антидребезга
135             brne    ic2             ; Если еще не конец, продолжаем

136             cp      XL, codL        ; Сравнение с временным буфером
137             brne    ic5             ; Если не равно, заканчиваем сравнение
138             cp      XL, codH        ; Сравниваем старшие байты

139     ic5:    pop     count
140             ret


        ; ------------------------------ Подпрограмма задержки

141     wait:   cpi     data,1          ; Проверяем код задержки
142             brne    w1
143             ldi     temp,0x40       ; Разрешаем прерывание по совпадению
144             rjmp    w2
145     w1:     ldi     temp,0x80       ; Разрешаем прерывания по переполнению

146     w2:     out     TIMSK,temp      ; Записываем маску
147             clr     temp
148             out     TCNT1H,temp     ; Обнуляем таймер
149             out     TCNT1L,temp

150             ldi     flz,0           ; Сбрасываем флаг задержки
151             sei                     ; Разрешаем прерывания
152             cpi     data,2          ; Если это задержка 2-го типа
153             breq    w4              ; Переходим к концу подпрограммы

154     w3:     cpi     flz,1           ; Ожидание окончания задержки
155             brne    w3

156             cli                     ; Запрещаем прерывания

157     w4:     ret                     ; Завершаем подпрограмму

        ; ------------------------------ Запись байта в ячейку EEPROM

158     eewr:   cli                     ; Запрещаем прерывания
159             sbic    EECR, EEWE      ; Проверяем готовность EEPROM
160             rjmp    eewr            ; Если не готов ждем
161             out     EEAR, addre     ; Записываем адрес в регистр адреса
162             out     EEDR, data      ; Записываем данные в регистр данных
163             sbi     EECR, EEMWE     ; Устанавливаем бит разрешения записи
164             sbi     EECR, EEWE      ; Устанавливаем бит записи
165             inc     addre           ; Увеличиваем адрес в EEPROM
166             ret                     ; Выход из подпрограммы

        ; ------------------------------ Чтение байта из ячейки EEPROM

167     eerd:   cli                     ; Запрещаем прерывания
168             sbic    EECR, EEWE      ; Проверяем готовность EEPROM
169             rjmp    eerd            ; Если не готов ждем
170             out     EEAR, addre     ; Устанавливаем бит инициализации чтения
171             sbi     EECR, EERE      ; Устанавливаем бит инициализации чтения
172             in      data, EEDR      ; Помещаем прочитанный байт в data
173             inc     addre           ; Увеличиваем адрес в EEPROM
174             ret                     ; Выход из подпрограммы
```

```
        ;*****************************************************
        ;*                                                   *
        ;*          Процедура обработки прерываний           *
        ;*                                                   *
        ;*****************************************************

        ; --------------------------- Прерывание по совпадению
175     propr:  ldi             flz,1           ; Установка флага задержки
176             reti                            ; Завершаем обработку прерывания
```

Если учитывать, что записи подлежит каждое изменение кода состояния, а его изменение происходит как при нажатии кнопки, так и при ее отпускании, то после нажатия и отпускания одной из кнопок в буфер запишется два кода.

Значит, объема буфера нам хватит на 15 последовательных нажатий. Этого вполне достаточно, так как типичная кодовая комбинация состоит обычно из 4—5 цифр. Если вы считаете, что этого недостаточно, вы можете увеличить размер буфера, просто поменяв значение константы `bsize` в **строке 12**. Максимально возможный размер ограничен объемом ОЗУ и равен примерно 100 байтам (полный размер ОЗУ 128 байт).

Учтите, что буфер не может занимать весь объем ОЗУ, так как в верхних адресах необходимо обязательно оставить пространство, которое будет использовать стековая память. Обращаю ваше внимание, что константа `bsize` используется для задания размера не только буфера в ОЗУ, но и для задания размера буфера в EEPROM, который предназначен для долговременного хранения кодовой комбинации.

Константа `kzad` (**строка 13**) определяет **длительность защитной паузы**. Назначение защитной паузы подробно описано выше. Константа представляет собой коэффициент пересчета для таймера, при котором величина сформированной задержки будет равна 48 мс.

Константа `kandr` (**строка 14**) — это **константа антидребезга**. Она используется в специальной антидребезговой процедуре. Константа определяет, сколько раз подряд должен повториться один и тот же код состояния клавиатуры, чтобы программа прекратила цикл антидребезга и перешла к обработке считанного кода.

После определения констант начинается **блок резервирования оперативной памяти** (**строки 15—17**). В **строке 15** выбирается соответствующий сегмент памяти, в **строке 16** устанавливается указатель на адрес 0x60. Соображения для выбора именно этого адреса уже приводились в предыдущем примере.

Собственно резервирование ячеек производится в **строке 17**. Директива `byte` резервирует необходимое количество ячеек ОЗУ, начиная с адреса, определяемого меткой `bufr`. В данном случае `bufr` будет равен 0x60. Количество резервируемых ячеек определяется константой `bsize`.

Далее, в **строках 18—21** происходит резервирование ячеек в энергонезависимой памяти (EEPROM). В **строке 18** выбирается сегмент EEPROM. В **строке 19** устанавливается текущее значение указателя этого сегмента. Указателю присваивается значение 0x08. То есть размещение данных в памяти EEPROM будет начинаться с восьмой ячейки. Вообще-то в данном случае можно было начинать с нулевой ячейки. Это не сделано лишь из соображений надежности работы.

Фирма Atmel **не рекомендует** без особой необходимости использовать ячейку с нулевым адресом, так как именно она подвергается наибольшему риску потери информации при недопустимых перепадах напряжения питания, особенно если перепады напряжения возникают в момент записи информации в EEPROM. Так как EEPROM не работает со стеком, у нас есть запас по ячейкам. Поэтому мы отступили на целых восемь ячеек.

Собственно команды резервирования занимают **строки 20** и **21**. В **строке 20** резервируется одна ячейка, в которой будет храниться длина ключевой комбинации. В **строке 21** резервируется буфер длиной `bsize`, в котором будет храниться сама комбинация.

После резервирования ячеек мы переходим **в сегмент программного кода** (**строка 22**). И начинаем формирование программы с нулевого адреса (команда `org` в **строке 23**). Программный код начинается с **таблицы переопределения векторов прерываний** (**строки 24—42**). Как видите, в данном случае мы будем использовать два вида прерываний.

Это прерывание по совпадению в канале A таймера/счетчика 1 (**строка 28**) и прерывание по переполнению того же таймера (**строка 29**). Первое прерывание используется для формирования защитной задержки в 48 мс. А второе — для формирования контрольного промежутка времени в 1 с. Для обоих видов прерываний назначена одна и та же процедура обработки: `propr`. Почему одна и та же и как она работает, мы узнаем чуть позже.

В **строках 44—61** расположен **модуль инициализации**. Начинается инициализация с программирования портов ввода-вывода (**строки 46—53**). Все разряды порта PD и большая часть разрядов порта PB конфигурируются как входы. И лишь два разряда PB.3 и PB.4 конфигурируются как выходы.

Разряд PB.4 используется для управления механизмом замка. А **разряд PB.3** вообще пока не используется (зарезервирован). Его мы будем использовать как выход звука, когда будем объединять в одно устройство наш кодовый замок и музыкальную шкатулку. Для всех разрядов обоих портов, настроенных на ввод, включаются внутренние нагрузочные резисторы. На обоих выходах (PB.3 и PB.4) устанавливается низкий логический уровень.

В **строках 54, 55** программируется **аналоговый компаратор**. Программирование таймера/счетчика 1 производится в **строках 56—61**. Сначала в обе половинки регистра совпадения (OCR1AH, OCR1AL) записывается код, определяющий длительность защитной задержки (**строки 56—59**). Затем в регистр состояния TCCR1B записывается код 0x03 (**строки 60, 61**). При записи этого кода таймер/счетчик 1 переводится в режим Normal с использованием предварительного делителя. А коэффициент предварительного деления становится равным 1/64.

Строки 62—119 занимает **основной цикл программы**, **строки 120—174** — **набор вспомогательных подпрограмм**, а **строки 175 и 176** — **процедура обработки прерываний.** Рассмотрение программы удобнее начать со вспомогательных процедур.

И первая процедура, которую мы рассмотрим, — это **процедура ввода и предварительной обработки кода состояния клавиатуры**. Процедура представляет собой подпрограмму с именем `incod` и расположена в **строках 120—140**. В процессе работы процедура использует регистровую пару X и две вспомогательные регистровые пары: codH, codL (описаны в **строках 10, 11**) и drebH, drebL (описаны в **строках 3, 4**).

Главная задача данной процедуры — получить **код состояния клавиатуры**, используя алгоритм многократного считывания для борьбы с антидребезгом. Кроме того, процедура выполняет еще одну, вспомогательную функцию. Она сравнивает полученный описанным выше способом код состояния клавиатуры с другим кодом, который хранится в паре регистров codH—codL. Такое сравнение используется в основной программе для оценки значения кода состояния.

Таким образом, процедура `incod` по окончании своей работы возвращает два разных значения. Во-первых, код состояния клавиатуры (в регистровой паре X), а, во-вторых, результат сравнения кода с контрольным значением (используя флаг нулевого результата Z).

Рассмотрим работу процедуры подробнее. Подпрограмма использует традиционное сохранение и восстановление содержимого регистров в стеке. На этот раз сохранению подлежит всего один регистр — регистр `count`. В **строке 120** его значение сохраняется, а в **строке 139** — восстанавливается. Основной же текст подпрограммы состоит из двух частей:

- в первой части (**строки 121—133**) реализуется алгоритм ввода кода и борьбы с антидребезгом;
- во второй части (**строки 134—138**) производится сравнение полученного кода с числом в буфере codH-codL.

Начнем с **процедуры ввода кода и борьбы с антидребезгом**. Эта процедура представляет собой бесконечный цикл, который при каждом проходе производит формирование кода состояния клавиатуры и при этом подсчитывает, сколько раз подряд полученный код будет

иметь одинаковое значение. Для подсчета используется регистр `count`. Причем в начале в `count` записывается константа `kandr`, и каждый раз, когда новый код равен предыдущему, содержимое `count` уменьшается на единицу (используется обратный счет).

Если при очередном проходе код состояния изменит свое значение, то в регистр `count` снова записывается константа `kandr`, и подсчет начинается сначала. Заканчивается цикл считывания кодов лишь тогда, когда значение `count` достигнет нуля. Новое значение кода состояния клавиатуры формируется в регистровой паре X. Для того, чтобы новое значение можно было сравнивать со старым, старое записывается в буфер drebH—drebL.

Теперь **текст описанной выше программы рассмотрим по порядку**. Все начинается с подготовки регистровой пары X к принятию нового значения кода. В **строках 121, 122** содержимое X обнуляется. Бесконечный цикл многократного считывания занимает **строки 123—135**.

В **строке 123** в регистр `count` записывается начальное значение `kandr`. В **строках 124, 125** содержимое регистровой пары X сохраняется в буфере drebH—drebL. Таким образом запоминается старое значение кода состояния перед тем, как будет получено новое. В **строках 126—129** происходит считывание кодов из портов PD и PB и наложение масок. Полученный в результате этих операций код состояния клавиатуры окажется в регистровой паре X.

В **строках 130—133** производится сравнения старого и нового значений кодов. Сравнение происходит побайтно. Сначала сравниваются младшие байты (**строки 130, 131**), затем старшие (**строки 132, 133**). Оператор `brne` выполняет переход по условию «не равно». Поэтому, если хотя бы одна из операций сравнения даст положительный результат (коды окажутся неодинаковыми), то управление перейдет по метке `ic1`. То есть к тому месту программы, где счетчику `count` снова присваивается значение `kandr`.

Если же обе операции сравнения дадут отрицательные результаты, и коды окажутся равными, управление перейдет к **строке 134**. В **строке 134** происходит уменьшение содержимого регистра `count` на единицу. После этого производится проверка на ноль (**строка 135**). Если после уменьшения содержимого `count` оно еще не равно нулю, то оператор `brne` в **строке 135** передает управление по метке `ic2`, и цикл антидребезга продолжается. В противном случае цикл завершится, и управление переходит к **строке 136**.

В **строках 136—138** находится процедура сравнения только что полученного значения кода состояния клавиатуры с числом в буфере codH—codL. Сравнение проходит побайтно. Сначала в **строке 136** сравниваются младшие байты. Если они не равны, то дальнейшее сравнение

не имеет смысла. Поэтому управление передается по метке `ic5`, и подпрограмма завершается.

Если младшие байты сравниваемых величин оказались равны, то окончательный результат сравнения теперь можно получить, просто сравнив между собой старшие байты. Это сравнение производится в **строке 138**. В результате, при выходе из подпрограммы `incod`:

- флаг Z будет установлен, если сравниваемые коды равны между собой;
- флаг Z будет сброшен, если коды не равны.

Следующая дополнительная процедура, обеспечивающая работу основной части программы, — это **подпрограмма формирования задержки**. Для формирования временных интервалов эта процедура использует таймер. Мы уже рассматривали два варианта подобных процедур. Одна из них использовала прямое чтение содержимого таймера и цикл ожидания, а вторая использовала прерывания.

В данном случае разработан еще один вариант, который представляет собой нечто среднее между двумя предыдущими. В этом варианте будут использоваться и прерывание, и цикл ожидания. Новый алгоритм реализует подпрограмма `wait`, которая занимает **строки 141—157**. Подпрограмма имеет три режима работы. Номер режима передается в подпрограмму при ее вызове.

Для этого он записывается в регистр `data`. В **режиме номер 1** подпрограмма формирует задержку 48 мс. В **режиме номер 3** формируется задержка в 1 с. В **режиме номер 2** задержка не формируется. Подпрограмма просто настраивает таймер точно так же, как в режиме номер 3, разрешает прерывания и заканчивает свою работу. Контрольный интервал времени длительностью 1 с формируется уже вне подпрограммы `wait` (в основном тексте программы).

Основной **принцип формирования задержки** строится на использовании флага задержки. В качестве флага задержки применяется регистр `flz`. Описание этого регистра вы можете видеть в **строке 7** программы. Перед началом цикла задержки в регистр `flz` записывается ноль. Затем запускается таймер и разрешается работа одного из видов прерываний (прерывание по совпадению или прерывание по переполнению).

В определенный момент времени будет вызвана **процедура обработки прерывания**. Эта процедура (**строки 156, 176**) запишет в регистр флага (`flz`) единицу. Единица в регистре `flz` послужит индикатором того факта, что заданный промежуток времени закончился. Для обнаружения момента окончания задержки используется цикл ожидания.

В цикле ожидания программа постоянно проверяет содержимое регистра `flz`. Пока `flz` равно нулю, цикл продолжается. Заканчивается цикл в тот момент, когда `flz` будет равен единице.

Для формирования разных значений длительности задержки используются разные виды прерываний. Прерывание по совпадению используется для формирования задержки в 48 мс. Для этого значение регистра совпадения выбрано таким образом, чтобы содержимое счетного регистра достигло этого значения именно через 48 мс. Прерывание по переполнению таймера используется для формирования интервала времени, равного одной секунде. Благо, что при коэффициенте пересчета предварительного делителя 1/64 и тактовой частоте в 4 МГц переполнение таймера произойдет именно через 1 с.

Рассмотрим **текст подпрограммы `wait`** подробнее. Начинается подпрограмма с проверки значение регистра `data` (**строка 141**). Как уже говорилось ранее, при помощи этого регистра в подпрограмму передается код номера режима. В данном случае нужно определить длительность формируемой задержки.

От этого зависит, какой из видов прерываний будет активизирован. Для активизации того либо другого вида прерываний в регистр маски таймера (TIMSK) мы будем записывать разные значения маски. В **строках 141—145** как раз и выбирается это значение. Выбор сводится к проверке номера режима.

Если содержимое `data` равно 1, то выполняется **строка 143**, где в качестве маски выбирается число 0x40 (прерывание по совпадению). Выбранная маска записывается в регистр `temp`. Оператор безусловного перехода `rjmp` в **строке 144** передает управление по метке `w2` для того, чтобы «перепрыгнуть» **строку 145**, где выбирается другое значение маски.

Если код в регистре `data` не равен 1, то управление передается к **строке 145**, где в регистр `temp` записывается код 0x80. Маска 0x80 разрешает прерывание по переполнению. В **строке 146** выбранное значение маски записывается в регистр TIMSK.

В **строках 147—149** выполняется **сброс таймера** (в обе половины счетного регистра таймера записывается нулевое значение). С этого момента начинается отсчет времени задержки. В **строке 150** сбрасывается в ноль регистр флага задержки (`flz`). А в **строке 151** производится глобальное разрешение прерываний. На этом все настройки, необходимые для работы процедуры задержки, заканчиваются.

Остается лишь ждать окончания заданного промежутка времени. Но прежде, чем начинать цикл ожидания, программа производит еще одну проверку номера режима (**строки 152, 153**). Дело в том, что для режима номер 2 цикл ожидания организовывать не нужно. В этом случае цикл ожидания располагается вне подпрограммы `wait`, в теле самой программы. Поэтому, если был задан режим номер 2, то работа подпрограммы в этом месте должна заканчиваться.

Оператор `cpi` в **строке 152** проверяет содержимое регистра `data`, где хранится код номера режима на равенство цифре 2. Если код режима равен двум, то оператор условного перехода в **строке 153** передает управление на конец подпрограммы. В противном случае программа переходит к циклу ожидания.

Цикл ожидания занимает всего две **строки (154, 155)**. В **строке 154** производится проверка содержимого регистра флага (`flz`) на равенство единице. Пока счетчик находится в процессе счета и прерывание еще не сработало, то содержимое регистра `flz` равно нулю, оператор в **строке 155** передает управления назад на **строку 154**, и цикл продолжается. Как только процедура обработки прерывания запишет в `flz` единицу, цикл завершается, и управление переходит к **строке 156**. В этой строке происходит глобальный з.апрет всех прерываний. А в **строке 157** подпрограмма `wait` завершается.

Две оставшиеся, еще не описанные вспомогательные процедуры предназначены для работы с EEPROM. При записи в этот вид памяти и чтения из нее нужно соблюдать определенную последовательность действий. Эта последовательность подробно описана в документации на микроконтроллеры AVR **[5]**.

Там же приведены примеры процедур, рекомендованные производителем для этих целей. Описываемые ниже процедуры являются практически полной копией рекомендованных процедур, дополненные лишь командой автоматического увеличения адреса. Для управления памятью EEPROM используются специальные регистры ввода-вывода:

- **EEAR** — регистр адреса;
- **EEDR** — регистр данных;
- **EECR** — регистр управления.

Отдельные разряды регистра управления также имеют имена:

- **EEWE** — бит записи;
- **EEMWE** — бит разрешения записи;
- **EERE** — бит чтения.

Все названия введены фирмой-производителем и правильно понимаются транслятором, если вы не забыли присоединить в начале программы файл описаний.

Порядок записи байта в EEPROM следующий. Байт данных, предназначенный для записи, должен быть помещен в регистр EEDR, а байт адреса — в регистр в EEAR. Для того, чтобы разрешить запись, необходимо установить бит EEMWE. Затем в течение четырех машинных циклов (то есть следующей же командой) нужно установить бит EEWE. Сразу же после установки бита EEWE начинается процесс записи. Этот процесс занимает довольно продолжительное время. Все это время бит EEWE остается установленным. По окончании процесса записи он сам

сбрасывается в ноль. Такой многоступенчатый алгоритм придуман для предотвращения случайной записи.

Подпрограмма, реализующая описанный выше алгоритм записи байта, называется `eewr` и занимает **строки 158—166**. Байт данных, предназначенный для записи, передается в процедуру при помощи регистра `data`, а адрес ячейки, куда нужно записать данные, — через регистр `addre`. Работа подпрограммы начинается с глобального запрета всех прерываний (**строка 158**).

Это обязательное условие работы с EEPROM. Невовремя вызванное прерывание может помешать процессу записи. В данном случае запрет прерываний является избыточной мерой, так как программа построена таким образом, что при записи в EEPROM прерывания всегда запрещены.

В **строках 159, 160** расположен цикл проверки готовности EEPROM. Если бит EEWE установлен, это значит, что предыдущая операция записи еще не окончена. Поэтому в **строке 159** проверяется значение этого бита. Пока значение бита равно единице, выполняется команда безусловного перехода в **строке 160**, и проверка выполняется снова и снова.

Когда значение бита окажется равным нулю, **строка 160** будет пропущена (сработает команда `sbic` в **строке 159**), а цикл ожидания прервется. В **строке 161** происходит запись адреса из регистра `addre` в регистр EEAR. В **строке 162** в регистр EEDR записывается байт данных из регистра `data`.

В **строке 163** устанавливается бит разрешения записи. В **строке 164** — бит записи. После установки этого бита процесс записи будет запущен. Запись будет идти своим чередом, а программа может продолжать свою работу. Главное — не менять содержимое регистров EEDR и EEAR, пока процесс записи не закончится. В **строке 165** происходит приращение содержимого регистра `addre`.

И, наконец, в **строке 166** подпрограмма завершается. Команда в **строке 165** не относится к алгоритму записи в EEPROM. Но ее применение позволяет использовать подпрограмму `eewr` для последовательной записи цепочки байтов, в чем мы и убедимся дальше.

Порядок чтения байта гораздо проще. Достаточно в регистр EEAR записать адрес ячейки, содержимое которой нужно прочитать, а затем установить бит чтения (EERE). Прочитанный байт автоматически помещается в регистр EEDR.

Подпрограмма чтения байта из EEPROM называется `eerd` и занимает **строки 167—174**. Адрес ячейки, предназначенной для чтения, передается в подпрограмму через регистр `addre`. Прочитанный байт данных подпрограмма возвращает в регистре `data`. Начинается подпрограмма чтения, как и подпрограмма записи, с запрета прерываний (**строка 167**).

Так как чтению может предшествовать запись, прежде чем изменять значения регистра EEAR, нужно проверить, закончился ли процесс записи. Поэтому в **строках 168, 169** мы видим уже знакомый нам цикл ожидания готовности EEPROM. В **строке 170** в регистр EEAR записывается содержимое регистра `addre`.

В **строке 171** устанавливается бит чтения (EERE). Как только этот бит будет установлен, моментально происходит процесс чтения, и прочитанный байт данных появляется в регистре EEDR. В **строке 172** этот байт помещается в регистр `data`. В **строке 173** происходит приращение регистра адреса `addre`. Смысл этого приращения такой же, как и в предыдущем случае. Только теперь подобный прием позволяет читать из EEPROM цепочку байтов. В **строке 174** подпрограмма `eewr` завершается.

Теперь **перейдем к основной части программы**. Как уже говорилось, она занимает **строки 62—119**. И первое, что выполняет основная программа, — **цикл ожидания отпускания кнопок**. В цикле используется описанная выше подпрограмма `incod`. Она будет не только считывать код состояния клавиатуры, но и сразу же производить его сравнение.

Если вы не забыли, код состояния клавиатуры при полностью отпущенных кнопках равен 0x7F, 0x07. В **строках 62, 63** этот код записывается во вспомогательный буфер `codL+codH`. Цикл ожидания отпускания кнопок расположен в **строках 64, 65**. В **строке 64** вызывается подпрограмма `incod`. Она определяет код состояния клавиатуры и сравнивает полученный код с числом, записанным в буфере `codL+codH`.

Если код состояния клавиатуры равен коду в буфере, то после выхода из подпрограммы флаг Z будет установлен. В противном случае — сброшен. Равенство кодов означает, что кнопки отпущены. Поэтому оператор условного перехода в **строке 65** проверяет значение флага Z. Пока коды разные, управление передается по метке m0, и цикл ожидания продолжается. Как только коды окажутся равными, цикл прерывается, и управление переходит к **строке 66**.

В **строке 66** начинается **цикл ожидания нажатия кнопки**. Цикл занимает **строки 66, 67** и выглядит почти так же, как цикл ожидания отпускания. Различие состоит в операторе условного перехода. Вместо `brne` (переход по условию «не равно») применяется оператор `breq` (переход по условию «равно»).

В буфере `codL+codH` по-прежнему находится код состояния клавиатуры при полностью отпущенных кнопках. Поэтому выход из данного цикла произойдет тогда, когда будет нажата любая из кнопок (S1—S10).

Как только нажатие будет обнаружено, программа переходит в следующую стадию. Полученный код состояния клавиатуры должен стать первым кодом ключевой комбинации. Но прежде чем начинать цикл ввода этой комбинации, программа выполняет две очень важные опе-

рации. В **строках 68, 69** в регистровую пару Z записывается адрес начала буфера в ОЗУ, куда будет помещаться вводимая комбинация. Регистр Z будет хранить текущий указатель этого буфера.

Вторая важная операция производится в **строке 70**. Тут обнуляется счетчик байтов, записанных в буфер. После этого начинается цикл ввода кодовой комбинации. Цикл занимает **строки 71—88**. Начинается работа цикла с формирования защитной задержки. Почему начинается с задержки, если мы только что получили первое нажатие кнопки?

А согласно алгоритму, после нажатия положено формировать задержку. Для формирования защитной задержки используется подпрограмма `wait`, работающая в режиме 1. Сначала в **строке 72** в регистр `data` записывается номер режима. Затем в **строке 73** вызывается подпрограмма `wait`.

После окончания защитной задержки в **строке 74** снова производится ввод кода состояния. В **строках 75, 76** полученный код записывается в буфер. Для записи используются команды, увеличивающие значение указателя (Z). Поэтому после записи каждого очередного байта указатель передвигается в следующую позицию. Затем в **строках 77, 78** увеличивается значение счетчика принятых байтов. Так как мы записали два байта, то и значение счетчика увеличивается дважды.

В **строках 79, 80** производится оценка длины введенного кода. Если длина превысит размеры буфера, то цикл ввода кода досрочно прекращается. В **строке 79** производится сравнение текущего значения счетчика с размером буфера.

В **строке 80** находится оператор условного перехода, который передает управление по метке `m7` в случае, если длина кода превысит размер буфера. В **строках 81, 82** код состояния клавиатуры записывается в буфер codH—codL. Делается это для того, чтобы следующий введенный код состояния можно было сравнивать с текущим.

Дальше программа должна **ожидать очередное изменения кода состояния**. Но сначала нужно запустить таймер, чтобы он начал формирование защитного промежутка времени. Если в течение этого промежутка не будет нажата ни одна кнопка, то это должно послужить сигналом к выходу из цикла ввода ключевой комбинации. Запуск таймера производится при помощи подпрограммы `wait` в режиме номер два. В **строке 83** в регистр `data` записывается номер режима, а в **строке 84** вызывается сама подпрограмма.

В **строках 85—88** организован комбинированный цикл ожидания. В теле цикла происходит сразу несколько операций. Во-первых, вводится новое значение кода состояния клавиатуры (**строка 85**). В процессе ввода новый код сравнивается со старым, который хранится в буфере codH—codL. Если коды не равны, то оператор условного перехода в

строке 86 передает управление по метке `m3`, где происходит формирование защитной задержки, затем повторное считывание и запись кода в буфер и так далее.

Если новое значение кода равно старому, комбинированный цикл продолжается. В **строке 87** производится проверка флага задержки `flz`. Если флаг равен нулю, это значит, что защитный промежуток времени еще не закончился. В этом случае оператор условного перехода в **строке 88** передает управление по метке `m6`, и комбинированный цикл продолжается сначала. Если значение флага `flz` равно единице, то цикл завершается, и управление переходит к **строке 89**.

В **строках 89, 90** происходит проверка переключателя режимов работы (S11). В зависимости от состояния этого переключателя полученная только что кодовая комбинация либо записывается в EEPROM (режим «Запись»), либо поступает в процедуру проверки (режим «Работа»). Команда `sbic` в **строке 89** проверяет значение седьмого бита регистра PINB.

Если бит равен нулю (контакты тумблера замкнуты), то **строка 90** не выполняется, и управление переходит к **строке 91**, где начинается процедура записи в EEPROM. Если значение разряда равно единице (контакты тумблера не замкнуты), оператор условного перехода в **строке 90** передает управление по метке `m9` на начало процедуры сравнения.

Процедура записи ключевой комбинации в EEPROM

Эта процедура занимает **строки 91—101**. К началу этой процедуры кодовая комбинация уже находится в буфере ОЗУ. Длина комбинации содержится в переменной `count`. Нам остается только записать все это в EEPROM.

В **строках 91—93** в EEPROM записывается длина комбинации. В качестве адреса для записи используется метка `klen`. Эта метка указывает на ячейку, которая специально зарезервирована для этой цели (см. **строку 20**). Для записи байта в EEPROM используется подпрограмма `eewr`.

В **строке 91** длина комбинации помещается в регистр `data`. В **строке 92** адрес помещается в регистр `addre`. Затем вызывается подпрограмма `eewr` (**строка 93**).

В **строках 97—100** расположен цикл записи всех байтов ключевой комбинации. Перед началом цикла в регистр `addre` записывается адрес первой ячейки буфера-приемника, находящегося в EEPROM (**строка 94**). А в регистровую пару Z записывается адрес первой ячейки буфера-источника, находящегося в ОЗУ (**строки 95, 96**).

В процессе записи ключевой комбинации регистр `count` используется для подсчета записанных байтов. В начале, как уже говорилось, он

содержит длину комбинации. При записи каждого байта содержимое `count` уменьшается. Когда оно окажется равным нулю, цикл записи прекращается.

Цикл начинается с того, что очередной байт ключевой комбинации, находящейся в ОЗУ, помещается в регистр `data` (**строка 97**). Напомню, что адрес уже находится в регистре `addre`. В **строке 98** вызывается подпрограмма, которая записывает байт в EEPROM. Та же подпрограмма увеличивает значение `addre` на единицу. В **строке 99** уменьшается значение регистра `count`.

Проверку содержимого `count` на равенство нулю производит оператор `brne` в **строке 100**. Если содержимое не равно нулю, то оператор передает управление на метку `m8`, и цикл записи ключевой комбинации продолжается. В противном случае цикл завершается, и управление переходит к **строке 101**. То есть к процедуре открывания замка.

Теперь разберемся, зачем после записи кода вызывается процедура открывания замка. Это сделано для удобства. После ввода кодовой комбинации необходимо выдержать паузу в 1 с для того, чтобы введенная комбинация записалась в EEPROM. Для того, чтобы точно знать, когда заканчивается эта пауза, используется срабатывание замка. Как только щелкнет соленоид, можно считать, что ввод закончен.

Процедура проверки кода

Эта процедура занимает **строки 102—114**. Процедура проверки во многом похожа на процедуру записи. Для чтения байта из EEPROM используется подпрограмма `eerd`. Перед вызовом этой подпрограммы адрес ячейки, откуда будет прочитана информация, записывается в регистр `addre`. Прочитанное из EEPROM значение возвращается в регистре `data`.

В **строках 102—105** происходит считывание длины последовательности из EEPROM и сравнение ее с длиной новой последовательности. Сначала в **строке 102** адрес ячейки, где хранится длина кода, записывается в `addre`. Затем вызывается подпрограмма `eerd` (**строка 103**).

В **строке 104** сравнивается полученное из EEPROM значение (регистр `data`) и новое значение длины кода (регистр `count`). В случае неравенства этих двух величин оператор условного перехода в **строке 105** передает управление в начало программы. В этом случае дальнейшая проверка больше не производится. Дверь остается закрытой.

Если длина кода оказалась правильной, начинается цикл побайтной проверки кодов. Сначала в регистр `addre` записывается начальный адрес буфера в EEPROM (**строка 106**). В регистровую пару Z записывается начальный адрес буфера в ОЗУ (**строки 107, 108**). Сам цикл сравнения занимает **строки 109—114**.

В **строке 109** вызывается подпрограмма, которая читает очередной байт из EEPROM. Байт помещается в регистр `data`. В **строке 110** читается байт из ОЗУ. Этот байт помещается в регистр `temp`. В **строке 111** эти два байта сравниваются. Если байты не равны, оператор условного перехода в **строке 112** прерывает процесс сравнения и передает управление на начало программы. То есть дверь и в этом случае остается закрытой.

Если байты равны, выполняется уменьшение содержимого счетчика `count` (**строка 113**). Если после уменьшения содержимое `count` еще не достигло нуля, управление передается по метке `m10` (оператор `brne` в **строке 114**), и цикл сравнения продолжается. Когда содержимое `count` окажется равным нулю, процесс сравнения заканчивается. Это означает, что все байты обеих версий ключевой комбинации оказались равны. Поэтому программа плавно переходит к процедуре открывания замка (к **строке 115**).

Процедура открывания замка

Эта процедура занимает **строки 115—119**. Процедура очень проста. Для открывания замка на четвертый разряд порта PB подается единичный сигнал, который открывает транзистор ключа VT1 (см. **рис. 5.17**). Реле срабатывает, и замок открывается. Подав открывающий сигнал, программа выдерживает паузу, а затем сигнал снимает. После этого замок закрывается. Длительность паузы равна одной секунде. Этого времени достаточно, чтобы открыть дверь.

Подача открывающего сигнала на выход осуществляется в **строке 115**. В **строках 116, 117** производится вызов процедуры задержки. При этом выбирается режим номер 3. Сначала в регистр `data` помещается код режима задержки (**строка 116**). Затем вызывается подпрограмма `wait` (**строка 117**). В **строке 118** снимается сигнал открывания двери. В **строке 119** процедура открывания замка завершается. Оператор безусловного перехода, находящийся в этой строке, передает управление на начало программы. И весь процесс начинается сначала.

Программа на языке СИ

Возможный вариант программы кодового замка на языке СИ приведен в **листинге 5.20**. Программа реализует тот же самый алгоритм, что и приведенная выше программа на Ассемблере. В тексте этой программы были использованы несколько новых для нас элементов языка СИ. Рассмотрим их по порядку.

#define

Директива присвоения символьного имени любой константе. Это классический элемент языка СИ, который поддерживается любой вер-

сией языка. Директива имеет два параметра. Первый параметр — имя константы. Второй параметр — ее значение. В **строке 2** программы (**листинг 5.20**) числовой константе 0x77F присваивается имя `klfree`. После этого в любом месте программы, где нужно использовать число 0x77F, его можно заменить именем `klfree`.

В качестве значения константы может выступать не только число, но и любая комбинация чисел и букв. И даже комбинация, состоящая только из букв. Применение именованных констант делает программу более наглядной. Кроме того, изменять значение такой константы становится удобнее. Достаточно заменить значение константы в одном только месте (в строке описания). И сразу же новое значение будет учтено по всей программе.

#pragma

Директива, задающая специальные команды для компилятора. В качестве параметра в директиве указывается задаваемая команда. Ниже приведен ряд примеров использования этой директивы.

Включение/отключение сообщений об ошибках.

```
#pragma warn- // Отключает предупреждающие сообщения.
#pragma warn+ // Включает предупреждающие сообщения.
```

Включение/отключение оптимизации результирующего кода.

```
#pragma opt- // Отключает оптимизацию.
#pragma opt+ // Включает оптимизацию.
```

Включение/отключение оптимизации по минимальному размеру результирующего кода.

```
#pragma optsize- // Отключает оптимизацию по размеру.
#pragma optsize+ // Включает оптимизацию по размеру.
```

И так далее. Полный список всех команд, передаваемых посредством этой директивы, вы можете найти в файле помощи программы CodeVision в разделе «Препроцессор» («The Preprocessor»).

eeprom

Управляющее слово, используемое при описании переменных (массивов), которое указывает транслятору, что данная переменная (массив) будет располагаться в энергонезависимой памяти данных (EEPROM). **Например**, в **строке 9** программы находится описание переменной `klen`, предназначенной для хранения длины кодовой комбинации, а в **строке 10** описывается массив, в котором будет храниться сама комбинация. И переменная, и массив размещаются в EEPROM.

return

Команда возврата значения. Если функция языка СИ должна возвращать значение, последней командой в теле этой функции должна быть команда `return`. В качестве параметра этой команды указывается возвращаемое значение.

Примером может служить функция `incod()`, занимающая в нашей программе (**листинг 5.20**) **строки 16—26**. Функция производит определение кода состояния клавиатуры с использованием процедуры антидребезга. По результатам своей работы функция должна возвращать код состояния клавиатуры. Поэтому последняя команда в теле функции (**строка 26**) — это команда `return`. В качестве параметра эта команда использует переменную `cod1`, которая и содержит сформированный код состояния.

Кроме новых команд, в тексте программы (**листинг 5.20)** используется один, пока еще не знакомый нам интересный прием. Посмотрите, пожалуйста, на **строку 54** программы. В этой строке записано выражение, которое присваивает элементу массива `bufr` значение переменной `codS`. Однако в качестве номера элемента массива используется не просто переменная `ii`, а выражение `ii++`. Это и есть еще одна оригинальная особенность языка СИ.

Язык СИ допускает одновременно использовать переменную в любом выражении и изменять ее значение. Так, при вычислении выражения `bufr[ii++]=codS` сначала элементу массива с номером `ii` присваивается значение `codS`, а затем значение переменной `ii` увеличивается на единицу. Такой же прием допускается при использовании переменных в качестве параметров функций или в составе любых других выражений. Кроме команды увеличения, можно использовать команду уменьшения, а также менять порядок вычислений. Вот несколько примеров таких выражений:

```
a=MyBuffer[++ii]; b=MyFunction(ii--); c=85+(--ii)/2;
```

В любом случае, при использовании данного приема выполняются следующие правила:

- **`ii++`** означает: использовать значение, а затем увеличить его на единицу.
- **`++ii`** означает: увеличить значение на единицу, а затем использовать его.
- **`ii--`** означает: использовать значение, а затем уменьшить его на единицу.
- **`--ii`** означает: уменьшить значение на единицу, а затем использовать его.

Естественно, вместо переменной `ii` может использоваться любая другая переменная.

Описание программы (листинг 5.20)

Листинг 5.20

```
/*****************************************************
Project : Пример 10
Version : 1
Date    : 07.03.2006
Author  : Belov
Company : Home
Comments:
Кодовый замок
Chip type           : ATtiny2313
Clock frequency     : 4,000000 MHz
Memory model        : Tiny
Data Stack size     : 32
*****************************************************/

#include <tiny2313.h>

#define klfree 0x77F                  // Код состояния при полностью отпущенных кнопках
#define kzad 3000                     // Код задержки при сканировании
#define kandr 20                      // Константа антидребезга
#define bsize 30                      // Размер буфера для хранения кода

unsigned char flz;                          // Флаг задержки
unsigned int bufr[bsize];                   // Буфер в ОЗУ для хранения кода
#pragma warn-
eeprom unsigned char klen;                  // Ячейка для хранения длины кода
eeprom unsigned int bufe[bsize];            // Буфер в EEPROM для хранения кода
#pragma warn+

// Прерывание по переполнению Таймера 1
interrupt [TIM1_OVF] void timer1_ovf_isr(void)
{
  flz=1;
}

// Прерывание по совпадению в канале A Таймера 1
interrupt [TIM1_COMPA] void timer1_compa_isr(void)
{
  flz=1;
}

// Функция опроса клавиатуры и антидребезга
unsigned int incod (void)
{                                           // Создаем локальные переменные
  unsigned int cod0=0;                      // Вспомогательная переменная
  unsigned int cod1;                        // Еще одна вспомогательная переменная
  unsigned char k;                          // Параметр цикла антидребезга

  for (k=0; k<kandr; k++)                   // Цикл антидребезга
  {
            cod1=PINB&0x7;                  // Формируем первый байт кода
            cod1=(cod1<<8)+(PIND&0x7F);     // Формируем полный код состояния клавиатуры
            if (cod0!=cod1)                 // Сравниваем с первоначальным кодом
            {
                  k=0;                 // Если не равны, сбрасываем счетчик
                  cod0=cod1;           // Новое значение первоначального кода
            }
  }
  return cod1;
}

// Процедура формирования задержки
void wait (unsigned char kodz)
{
  if (kodz==1) TIMSK=0x40;              // Выбор маски прерываний по таймеру
  else TIMSK=0x80;
  TCNT1=0;                              // Обнуление таймера
  flz=0;                                // Сброс флага задержки
  #asm("sei");                          // Разрешаем прерывания
  if (kodz!=2) while(flz==0);           // Цикл задержки
}

// Основная функция
void main(void)
{
```

```
35      unsigned char ii;                   // Указатель массива
36      unsigned char i;                    // Вспомогательный указатель
37      unsigned int codS;                  // Старый код

38      PORTB=0xE7;                         // Порт B
39      DDRB=0x18;

40      PORTD=0x7F;                         // Порт D
41      DDRD=0x00;

42      TCCR1A=0x00;                        // Таймер/Счетчик 1
43      TCCR1B=0x03;
44      TCNT1=0;                            // Обнуление счетного регистра
45      OCR1A=kzad;                         // Инициализация регистра совпадения

46      ACSR=0x80;                          // Аналоговый компаратор

47      while (1)
        {
48      m1:                 while (incod() != klfree);    // Ожидание отпускания кнопок
49                          while (incod() == klfree);    // Ожидание нажатия кнопок
50                          ii=0;
51      m2:                 #asm("cli");                  // Запрещаем прерывания
52                          wait(1);                      // Задержка 1-го типа
53                          codS=incod();                 // Ввод кода и запись, как старого
54                          bufr[ii++]=codS;              // Запись очередного кода в буфер
55                          if (ii>=bsize) goto m4;       // Проверка конца буфера

56                          wait(2);                      // Задержка 2-го типа
57      m3:                 if (incod() != codS) goto m2; // Проверка, не изменилось ли состояние
58                          if (flz==0) goto m3;          // Проверка флага окончания задержки

59      m4:                 if (PINB.7==1) goto comp;     // Проверка переключателя режимов

        //------------------------------ Запись кода в EEPROM

60                          klen=ii;                              // Запись длины кода
61                          for (i=0; i<ii; i++) bufe[i]=bufr[i]; // Запись всех байтов кода
62                          goto zamok;                           // К процедуре открывания замка

        //------------------------------ Проверка кода

63      comp:               if (klen!=ii) goto m1;                // Проверка длины кода
64                          for (i=0; i<ii; i++)  if (bufe[i]!=bufr[i]) goto m1;    // Проверка кода

        //------------------------------ Открывание замка

65      zamok:              PORTB.4=1;        // Открываем замок
66                          wait(3);          // Задержка 3-го типа
67                          PORTB.4=0;        // Закрываем замок
        }
    }
```

Кардинальным **отличием** программы на языке СИ является тот факт, что все шестнадцатиразрядные значения, используемые в программе, теперь не нужно разбивать на отдельные байты. Для хранения каждой такой величины программа использует либо переменную, либо константу соответствующего типа.

Например, для хранения разных вариантов кода состояния клавиатуры в программе используется несколько переменных:

- `cod0` — **строка 17**;
- `cod1` — **строка 18**;
- `codS` — **строка 37**.

И все они имеют тип `unsigned char`. При этом сам этот код представляет собой шестнадцатиразрядное двоичное число, старшие восемь разрядов соответствуют содержимому порта PB, а младшие восемь разрядов — содержимому порта PD (на все это наложена маска).

Начинается наша программа, как и все предыдущие, с присоединения библиотечного файла (**строка 1**). Далее идет блок описания констант (**строки 2—5**). Первая константа имеет имя `klfree` и значение 0x77F. Это значение представляет собой код состояния клавиатуры при отпущенных кнопках. Константа используется в операциях сравнения.

Следующие три константы: `kzad` (код задержки), `kandr` (константа антидребезга) и `bsize` (размер буфера для кодовой комбинации) по своему назначению аналогичны соответствующим константам в программе на Ассемблере. У них даже значения одинаковы. Отличие только в значении константы `bsize`. В нашем случае она равна не 60, а 30.

Почему буфер стал вдвое короче? Дело в том, что на Ассемблере буфер представлял собой набор ячеек памяти размером в один байт каждая. Любое новое значение записывалось в буфер в виде двух отдельных байтов. В программе на СИ в качестве буфера будет использоваться массив, состоящий из шестнадцатиразрядных элементов. Каждое значение в такой буфер заносится как один отдельный элемент.

В **строках 6—11** находится блок описания переменных и массивов. Все переменные и массивы имеют свои аналоги в программе на Ассемблере. В **строке 6** описывается переменная `flz`, которая используется как флаг задержки. В **строке 7** описывается массив для оперативного хранения ключевой комбинации. Все значения этого массива будут размещены в ОЗУ (буфер в ОЗУ). Длина массива выбирается равной `bsize`.

В **строках 9, 10** описываются переменная и массив, которые будут храниться в EEPROM. Переменная `klen` предназначена для хранения длины кодовой комбинации. Массив `bufe` предназначен для хранения самой этой комбинации.

При описании переменных и массивов, которые будут размещаться в EEPROM, данная версия языка СИ требует обязательной их инициализации. То есть требует указать значение всех элементов по умолчанию. Указанные таким образом значения в процессе «прошивки» микроконтроллера попадают непосредственно в EEPROM. Если соблюдать указанные выше требования, то **строки 9 и 10** нашей программы должны выглядеть примерно так:

```
eeprom unsigned char klen=0x4;
eeprom unsigned int bufe[bsize]={0x76F, 0x77F, 0x7FE,
0x77F};
```

Если мы воспользуемся данной редакцией команд описания, то мы получим программу электронного кодового замка. В ней уже на стадии изготовления заложена некоторая начальная ключевая кодовая комбинация, которую, впрочем, в любой момент владелец замка может изменить на новую.

Но мне интересно показать, что можно сделать, если закладывать код заранее нежелательно. Если вы не стали указывать начальные значения для переменной и массива, то это не является критической ошибкой. Программа будет успешно оттранслирована, а результирующий код полностью работоспособен. Единственное неудобство — сообщение о некритичной ошибке. По-английски оно называется «Warning» (предупреждение). Оно будет возникать каждый раз при трансляции программы. Можно, конечно, просто не обращать на него внимание.

Однако более правильно будет временно отключить сообщение при помощи директивы `#pragma` и команды `warn`, как это и сделано в программе на **листинге 5.20**. В **строке 8** вывод предупреждений отключается, а в **строке 11** включается снова. Отключать предупреждения навсегда не рекомендуется. Так можно пропустить другие, более важные предупреждения.

Далее в программе начинается **описание всех составляющих ее функций**. Данная программа состоит из пяти функций:

- две функции обработки прерываний (**строки 12, 13** и **14, 15**);
- функция ввода кода состояния клавиатуры (**строки 16—26**);
- функция формирования задержки (**строки 27—33**);
- главная функция программы (**строки 34—67**).

Рассмотрение программы удобно начинать с главной функции `main`.

Функция `main` начинается с описания локальных переменных (**строки 35—37**). Кроме переменной `codS`, предназначенной для временного хранения кода состояния клавиатуры, здесь определяются еще две вспомогательные переменные с именами `i` и `ii`. После описания переменных начинается **блок инициализации** (**строки 38—46**). Блок инициализации данной программы по выполняемым действиям полностью повторяет аналогичный блок в программе на Ассемблере.

Эти действия сводятся к настройке портов ввода-вывода, таймера и компаратора. При настройке таймера не только выбирается его режим работы, но и обнуляется значение счетного регистра TCNT1 (**строка 44**), а в регистр совпадения OCR1A записывается код задержки kzad (**строка 45**).

Основной цикл программы занимает **строки 47—67**. Рассмотрим подробнее его работу. В **строке 48** находится цикл ожидания отпускания кнопок. Он представляет собой пустой цикл `while`. Тело цикла полностью отсутствует. За ненадобностью не поставлены даже фигурные скобки. Весь цикл состоит лишь из оператора `while` и выражения в круглых скобках, определяющего условие продолжения этого цикла.

Условие простое: цикл выполняется все время, пока код состояния клавиатуры и константа `klfree` не равны между собой. Значение константы равно коду состояния клавиатуры при всех отпущенных кноп-

ках. Для определения кода состояния клавиатуры используется функция `incod()`. Функция `incod()` выполняет те же самые действия, что одноименная процедура из программы на Ассемблере. То есть считывает состояние портов, накладывает маски и применяет при этом антидребезговый алгоритм. Подробнее работу функции мы рассмотрим в конце этого раздела.

Как только все кнопки будут отпущены, цикл в **строке 48** завершается, и программа переходит к **строке 49**, в которой находится цикл ожидания нажатия любой кнопки. Этот цикл очень похож на предыдущий. Изменилось только условие. Знак `!=` (не равно) заменен на `==` (равно). Как только будет нажата любая кнопка, цикл в **строке 49** заканчивается, и управление переходит к **строке 50**.

Теперь пора начинать **цикл ввода ключевой кодовой комбинации**. Но сначала нужно обнулить переменную `ii`, которая будет использоваться как счетчик принятых кодов и указатель текущего элемента в буфере `bufr`. Обнуление выполняется в **строке 50**. Цикл ввода кодовой комбинации занимает **строки 51—58**. Начинается цикл с глобального запрета всех прерываний (**строка 51**). Затем формируется защитная пауза. Для формирования паузы используется функция `wait()`.

Действие этой функции полностью аналогично действию одноименной процедуры из программы на Ассемблере. В **строке 52** формируется задержка первого вида (48 мс), то есть вызывается функция `wait()` с параметром, равным единице. По окончании задержки (**строка 53**) программа повторно считывает код состояния клавиатуры и записывает его в переменную `codS`.

В **строке 54** считанный код записывается в буфер ОЗУ (`bufr`). Одновременно указатель буфера увеличивается на единицу. В **строке 55** проверяется условие переполнения буфера. Такая проверка принудительно завершает работу цикла при попытке ввода слишком длинной кодовой комбинации. Если значение `ii` превысило величину константы `bsize`, значит, буфер уже полностью заполнен. В этом случае управление передается по метке `m4`, то есть на конец цикла.

Если значение `ii` не достигло конца буфера, то перехода не происходит, и управление переходит к **строке 56**. В этом месте запускается процесс формирования контрольного промежутка времени. Для запуска этого процесса используется функция `wait()`. В данном случае она формирует задержку второго типа, поэтому вызывается с параметром, равным двум.

Так же, как и в программе на Ассемблере, задержка второго типа лишь производит все настройки таймера, но не выполняет цикл ожидания. Комбинированный цикл ожидания находится вне функции задержки, а точнее в **строках 57, 58**. В **строке 57** считывается новый

код состояния клавиатуры и сравнивается со старым, который хранится в переменной codS.

Если коды не равны (состояние клавиатуры изменилось), комбинированный цикл ожидания прерывается, и управление передается по метке m2. То есть на начало цикла ввода кодовой комбинации. А уже там, в начале цикла, снова формируется защитная задержка, и очередной код состояния помещается в буфер.

Если при проверке в **строке 57** старый и новый коды оказались все же равны между собой, перехода не происходит, и выполняется **строка 58**. В этой строке проверяется значение флага flz. Если контрольный промежуток времени еще не истек, то значение флага равно нулю, и управление передается по метке m3. Комбинированный цикл продолжается.

Если же контрольный промежуток времени уже закончится, значение флага flz равно единице. Поэтому перехода не происходит, и управление переходит к **строке 59**. На этом и комбинированный цикл ожидания, и цикл ввода кодовой комбинации заканчиваются.

В **строке 59** проверяется состояние тумблера s11. В зависимости от этого состояния выполняется либо запись только что принятой кодовой комбинации в EEPROM, либо извлечение из EEPROM и сравнение двух кодовых комбинаций. Для проверки состояния переключателя оценивается значение седьмого разряда порта PB. Если контакты переключателя разомкнуты (PINB.7 равен единице), то управление передается по метке comp (к процедуре сравнения кодов). В противном случае выполняется процедура записи.

Процедура записи кода в EEPROM занимает **строки 60—62**. В **строке 60** длина кодовой комбинации записывается в переменную klen. Так как при описании переменной klen (см. **строку 9**) ее местом расположения выбран EEPROM, то записанное в переменную значение автоматически туда и попадает. Язык СИ сам выполняет все необходимые для этого процедуры. Как видите, в языке СИ запись в EEPROM происходит гораздо проще, чем на Ассемблере.

В **строке 61** находится цикл, который производит запись в EEPROM самой кодовой комбинации. Цикл просто по очереди записывает каждый элемент массива bufr в соответствующий элемент массива bufe. А bufe целиком находится в EEROM. В качестве параметра цикла используется переменная i. По ходу работы цикла значение этой переменной меняется от нуля до ii. То есть перебираются номера всех элементов кодовой комбинации (ii равно ее длине). Тело цикла составляет всего одно выражение. Это выражение записывает значение очередного элемента буфера bufr в буфер bufe.

Причем в качестве указателя для обоих массивов используется одна и та же переменная. Поэтому в буфере bufe элементы попадают в те

же позиции, какие они занимали в буфере `bufr`. По окончании цикла записи управление переходит к **строке 62**. В этой строке находится оператор безусловного перехода, который передает управление по метке `zamok`. То есть к процедуре открывания замка. Щелчок механизма замка оповещает об окончании процесса записи.

Процедура проверки занимает **строки 63, 64**. Напомним, что к началу этой процедуры переменная `ii` содержит длину только что введенной кодовой комбинации, а буфер `bufr` — саму эту комбинацию. Сначала, в **строке 63**, сравнивается значение переменной `klen` (длина, записанная ранее в EEPROM) и значение переменной `ii`. Язык СИ сам извлекает значение `klen` из EEPROM, используя все необходимые процедуры.

Если в результате проверки эти две длины окажутся не равными, то управление передается по метке m1. То есть к началу всей программы. Дальнейшее сравнение кодов не производится, и замок не открывается. Если оба значения одинаковы, то программа переходит к сравнению кодовых комбинаций. Цикл, производящий это сравнение, находится в **строке 64**. Этот цикл в качестве параметра тоже использует переменную `i`.

В процессе работы цикла значение этой переменной также меняется от нуля до `ii`. В теле цикла выполняется оператор сравнения `if`. Этот оператор сравнивает значения элементов двух массивов, один из которых (`bufr`) находится в EEPROM, а второй (`bufe`) расположен в ОЗУ. При первом же несовпадении кодов оператор безусловного перехода передает управление по метке m1. В этом случае цикл проверки досрочно прерывается, и замок не открывается. Если в процессе работы цикла проверки все коды оказались одинаковыми, то цикл завершается нормальным образом, и управление переходит к **строке 65**. То есть к процедуре открывания замка.

Процедура открывания замка очень проста. Она занимает **строки 65—67**. В **строке 65** подается команда, открывающая механизм замка (в четвертый разряд порта PB записывается единица). Затем вызывается задержка третьего типа (**строка 66**). По окончании задержки открывающий сигнал снимается (**строка 67**). С окончанием процедуры открывания замка заканчивается тело основного цикла программы. Так как основной цикл бесконечный, то управление передается на его начало. То есть на **строку 48**. Работа программы начинается сначала.

Теперь вернемся **к вспомогательным функциям программы**, которые мы пропустили в начале этого описания. Начнем по порядку. В **строках 12, 13** и **14, 15** размещены две разные по названию, но одинаковые по содержанию функции. Первая из них является процедурой обработки прерывания по переполнению таймера/счетчика1. А вторая — процедурой обработки прерывания по совпадению в канале A того же таймера.

В программе на Ассемблере оба вида прерываний вызывали одну и ту же общую процедуру. Данная версия языка СИ не позволяет использовать одну и ту же функцию в качестве процедуры обработки двух разных видов прерываний. В теле каждой из функций имеется всего одна строка. В этой строке присваивается единица переменной `flz`. То же самое делает процедура обработки прерывания в программе на Ассемблере.

Строки 16—26 занимает функция ввода состояния клавиатуры. Алгоритм работы этой функции немного отличается от алгоритма работы аналогичной процедуры на Ассемблере. Функция `incod()` в программе на языке СИ производит лишь считывание содержимого портов, наложение маски и антидребезговый алгоритм.

Операция сравнения в теле данной функции не выполняется. Так как функция `incod()` должна возвращать код состояния клавиатуры, тип возвращаемого значения определен как `unsigned int` (см. **строку 16**). Функция не имеет параметров, на что указывает слово `void`.

Рассмотрим подробнее, как работает функция `incod()`. В **строках 17—19** производится описание локальных переменных. Переменные `cod0` и `cod1` используются для хранения промежуточных значений кода состояния клавиатуры. Причем переменная `cod1` используется для хранения нового значения кода, а переменная `cod0` — для хранения старого значения (не путайте с буфером `codS` основной программы). При описании переменной `cod0` (**строка 17**) одновременно производится ее инициализация (присваивается нулевое значение). Этот ноль необходим для правильного начала цикла антидребезга. Для того чтобы при первом сравнении новый код не был равен старому. Еще одна переменная с именем `k` используется в качестве параметра цикла антидребезга.

Основу функции `incod()` составляет цикл антидребезга (**строки 20—25**). По сути, тело функции состоит только из этого цикла. Задача цикла — ввести код состояния клавиатуры заданное количество раз (определяется константой `kandr`). В данном случае используется стандартный цикл `for` (см. **строку 20**).

Цикл выполнится полностью (нужное количество раз) в том случае, если за время его работы код состояния клавиатуры не изменится. Если в процессе выполнения цикла состояние кнопок изменяется, то специальные команды внутри цикла перезапускают его работу сначала. Вычисление кода состояния производится в **строках 21** и **22**. Оператор `if` (**строка 23**) сравнивает старое и новое значения кодов. Если эти значения не равны, выполняются команды перезапуска цикла (**строки 24, 25**). По окончании работы цикла антидребезга команда `return` определяет возвращаемое значение (**строка 26**). На этом функция завершается.

Разберемся с отдельными элементами цикла антидребезга подробнее. И начнем со **строк 21** и **22**, где, как уже говорилось, происходит формирование кода состояния клавиатуры. Суть производимых вычислений наглядно проиллюстрирована на **рис. 5.18**. Источником информации для этих вычислений является содержимое регистров PINB и PIND, которые, как известно, непосредственно подключены к выводам портов PB и PD.

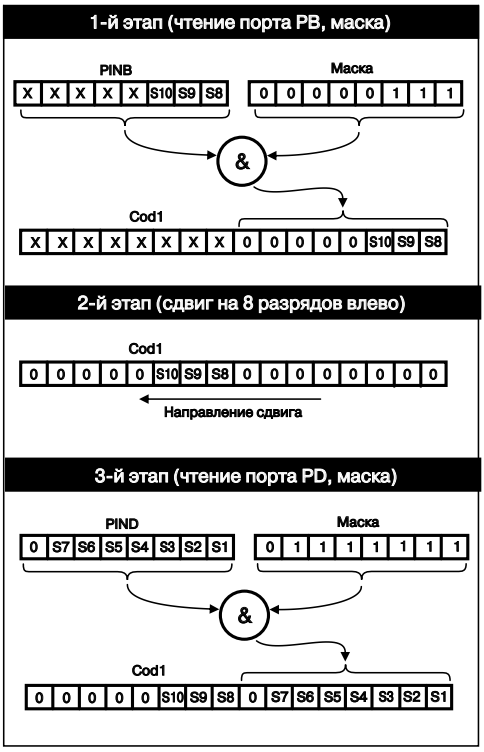


Рис. 5.18. *Формирование кода состояния клавиатуры*

На содержимое обоих портов накладываются соответствующие маски, а затем все это объединяется в одно шестнадцатиразрядное число и помещается в регистр `cod1`. Выражение в **строке 21** соответствует первому этапу на **рис. 5.18**. В правой части выражения выполняется операция логического умножения (операция «И») между содержимым порта PB и маской 0x07. Результат выражения записывается в переменную `cod1`.

Выражение в **строке 22** объединяет этапы 2 и 3. **Правая часть этого выражения** представляет собой сумму двух слагаемых. Первое слагаемое представляет собой операцию логического сдвига содержимого переменной `cod1` на восемь разрядов влево. Этот сдвиг соответствует этапу номер 2 на рисунке. В результате сдвига восемь младших разрядов числа становятся старшими.

Второе слагаемое — это еще одна операция логического умножения. На этот раз умножается содержимое порта PD и константа 0x7F, представляющая собой маску для этого порта. Результатом сложения двух этих слагаемых является искомый код состояния клавиатуры, который записывается в переменную `cod1`. Вычисление второго слагаемого и сложение их обоих и соответствует этапу номер 3 на **рис. 5.18**. Символом «X» на рисунке обозначаются разряды, значение которых не определено. Разряды, значение которых зависит от той либо иной кнопки клавиатуры, обозначается названием этой кнопки. Остальные разряды равны либо «0», либо «1».

Итак, в результате выполнения описанных выше операций переменная `cod1` содержит новое значение кода состояния клавиатуры. В **строке 23** этот код сравнивается со старым значением, которое хранится в переменной `cod0`. Если коды не равны, то выполняются команды перезапуска цикла (**строки 24, 25**). В **строке 24** переменной k (параметру цикла) присваивается нулевое значение. В **строке 25** значение переменной `cod1` записывается в переменную `cod0`. Теперь новое, только что полученное значение кода становится старым.

В **строках 27—33** расположена **функция формирования задержки**. Алгоритм работы и этой функции повторяет алгоритм работы аналогичной процедуры на Ассемблере. Функция не возвращает никаких значений, но зато имеет входной параметр: **код вида задержки**.

Как и процедура на Ассемблере, функция `wait()` формирует три вида задержки. Параметр, определяющий номер задержки, имеет имя `kodz` и тип `unsigned char`. Работа функции начинается с определения значения маски прерываний. Для этого в **строках 28, 29** оценивается значение переменной `kodz`. Если значение `kodz` равно 1, то в регистр маски прерываний TIMSK записывается код 0x40. Этот код разрешает прерывание по совпадению в канале A.

Если `kodz` не равен единице, то регистру TIMSK присваивается значение 0x80 (прерывание по переполнению). Таким образом, в режиме 1 будет работать прерывание по совпадению в канале A. При этом формируется задержка длительностью 48 мс. В остальных режимах (2, 3) используется прерывание по переполнению таймера. В этом случае формируется задержка длительностью в одну секунду.

В **строке 30** обнуляется значение счетного регистр таймера. С момента обнуления таймера начинается формирование заданного временного интервала. В **строке 31** обнуляется значение флага задержки. В **строке 32** выполняется команда глобального разрешения прерываний. На этом настройка таймера и системы прерываний заканчиваются. Теперь остается лишь организовать **цикл ожидания**.

Цикл ожидания, предназначенный для работы в режимах 1 и 3, организован в **строке 33**. Это пустой цикл, организованный при помощи оператора `while`. В качестве условия продолжения цикла выбрано равенство флага `flz` нулю. То есть пока `flz` равен нулю, цикл ожидания будет продолжаться. А закончится он в тот момент, когда процедура обработки прерывания изменит значение флага `flz` на единичное.

В режиме 2 используется другой цикл ожидания, который находится вне функции `wait()`. Поэтому в **строке 33**, кроме цикла ожидания, имеется оператор сравнения `if`. Он проверяет значение переменной `kodz`. Благодаря оператору сравнения, цикл ожидания в **строке 33** выполняется только в том случае, когда `kodz` не равен двум.

5.12. Кодовый замок с музыкальным звонком

Ну, и в заключение приведу пример, как можно объединить вместе две разные задачи. Посмотрим, как можно соединить описанный выше кодовый замок и музыкальную шкатулку. Музыкальная шкатулка с успехом может выполнять функцию дверного звонка. Нужно лишь немного изменить алгоритм запуска музыкальной программы и заставить шкатулку играть разные мелодии при нажатии одной кнопки.

Постановка задачи

Естественно, далеко не любые две задачи можно объединить в одном устройстве. Но данные две задачи прекрасно объединяются. Если немного доработать музыкальную шкатулку, то легко заставить ее работать от одной кнопки. Менять мелодии можно при каждом очередном нажатии этой кнопки.

Звучание мелодии должно начинаться при нажатии кнопки и продолжаться до ее отпускания. При следующем нажатии должна звучать другая мелодия. Доработанная таким образом программа будет использовать всего две линии порта. В схеме электронного замка как раз есть две свободные линии.

С точки зрения объединения двух программ тоже нет никаких проблем. Так как обе эти программы вполне могут работать по отдельности. Действительно, кто же будет одновременно звонить в звонок и набирать кодовую комбинацию?

Итак, мы можем **сформулировать последнюю нашу задачу** следующим образом:

«Объединить два разработанных ранее устройства: музыкальную шкатулку (пример 9) и электронный кодовый замок (пример 10) в одно универсальное устройство, обладающее обеими этими функциями. Устройство должно иметь десять кнопок набора кода, тумблер выбора режима, выходной каскад для управления электромагнитом замка, а также кнопку звонка и выходной каскад звука».

Алгоритм

Итак, нам нужно объединить два алгоритма. Как уже говорилось выше, эти два алгоритма не должны работать одновременно. Поэтому каждый алгоритм останется практически без изменений. Но их нужно все же соединить вместе. Как может выглядеть это соединение?

Для соединения алгоритмов достаточно объединить их процедуру ожидания. Такая процедура есть и в программе музыкальной шкатулки, и в программе замка. В первом случае такая процедура должна ожидать нажатия кнопки звонка, а во втором — нажатия одной из кнопок набора

кода. Обе эти процедуры нужно объединить в одну. Объединенная процедура ожидания должна сначала опрашивать кнопку звонка, а затем проверять, не нажаты ли кнопки набора кода.

Если обнаружится нажатие кнопки звонка, выполняется алгоритм музыкальной шкатулки. Если нажатыми окажутся кнопки набора кода, то выполняется алгоритм электронного замка. Если ничего не нажато, цикл ожидания продолжается.

В том случае, если начнется выполнение музыкального алгоритма, программа будет целиком занята этим алгоритмом, пока кнопка звонка не будет отпущена. Если начнется выполнение алгоритма замка, то программа будет выполнять его до конца. То есть пока набранная комбинация не запишется в память или пока не закончится процедура проверки. И только тогда, когда выбранный алгоритм закончит свою работу, управление передается к объединенной процедуре ожидания.

Схема

За основу схемы объединенного устройства удобнее взять **схему электронного замка** (см. **рис. 5.17**). Она потребует минимальной доработки. Новая схема приведена на **рис. 5.19**. К старой схеме добавлены всего два элемента. **Во-первых,** это выходной каскад звука, подключенный к выходу PB3. Именно этот выход используется в программе музыкальной шкатулки (**рис. 5.14**). **А во-вторых,** кнопка звонка, которую мы подключим к линии PB6.

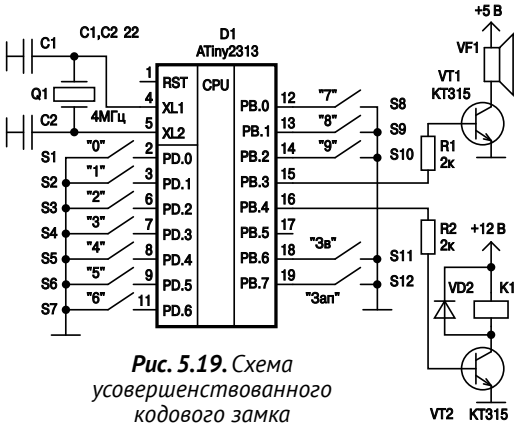


Рис. 5.19. *Схема усовершенствованного кодового замка*

Обратите внимание, что звуковая часть и сам микроконтроллер питаются от напряжения +5 В. А электромагнит замка питается от отдельного источника +12 В. Напряжение питания, подаваемое на микроконтроллер, обязательно должно быть стабилизированным. Питание на электромагнит стабилизировать совершенно необязательно. Для повышения громкости звонка и для защиты от помех питание для звуковой схемы можно осуществлять от напряжения, поступающего на вход стабилизатора.

Обычно напряжение на входе стабилизатора равно +7...+11 В. Поэтому можно электромагнит и звуковую схему запитывать от одного нестабилизированного источника +12 В. Те же 12 В можно подавать на вход стабилизатора, с выхода которого снимать напряжение +5 В для микроконтроллера.

Программа на Ассемблере

На **листинге 5.21** приведен возможный вариант объединения двух программ на Ассемблере. Новая программа, как и любая другая программа на Ассемблере, имеет свой модуль описания переменных и констант, свой модуль резервирования памяти, свою таблицу векторов прерывания и свою собственную основную часть.

Каждая из этих частей является результатом объединения аналогичных частей двух исходных программ. **Блок описаний переменных и констант** в новой программе занимает **строки 3—18**. При создании объединенного блока описаний учитывался тот факт, что в исходных программах широко используются одинаковые переменные. В объединенном блоке описаний каждая такая переменная описывается только один раз.

В **строках 19—22** находится **объединенный блок резервирования ОЗУ**. В **строке 21** резервируется буфер `bufr` для электронного замка. Такой же буфер резервировался и в исходной программе. В **строке 22** резервируется ячейка `melod`. Это новая ячейка, введенная в связи с доработкой алгоритма музыкальной шкатулки. В этой ячейке будет храниться текущий номер мелодии. При каждом последующем нажатии кнопки звонка этот номер будет изменяться.

В **строках 23—25** находится **блок резервирования ячеек в EEPROM**. Этот блок целиком взят из исходной программы электронного замка. В **строках 28—46** находится **таблица переопределения векторов прерываний**. Она тоже объединена из двух таблиц. Однако в связи с тем, что программа музыкальной шкатулки прерываний не использует, новая таблица полностью повторяет таблицу из программы электронного замка.

Модуль инициализации занимает **строки 47—60**. В данном случае в новый блок попали те команды, которые в исходных блоках обеих программ одинаковы. Это команды инициализации стека, инициализации портов ввода-вывода и инициализации (выключения) компаратора. Команды инициализации таймера для каждой из исходных программ отличаются друг от друга. Поэтому они исключены из общего блока инициализации. Инициализация таймера будет производиться по-разному в зависимости от выбранного режима работы (звонок или замок).

В **строке 61** начинается **основная часть программы**. За основу этой части взята программа электронного замка. Программа звонка подключается к основной части путем доработки процедуры ожидания нажатия кнопки. В остальном программа замка не претерпела никаких изменений.

Начинается основная программа с **инициализации таймера под задачи замка** (**строки 61—66**). Затем в **строках 67, 68** записывается контрольное значение в буфер `codH+codL`. В качестве контрольного

значения используется код состояния клавиатуры при полностью отпущенных кнопках. Этот код будет использоваться далее в цикле ожидания отпускания кнопок и в цикле ожидания нажатия кнопок. В **строках 69, 70** находится цикл ожидания отпускания кнопок. До сих пор программа замка не имела принципиальных изменений.

В **строках 71—75** находится **цикл ожидания нажатия кнопок**. Именно этот цикл изменен таким образом, что выполняет теперь комбинированный опрос не только кнопок ввода кода, но и кнопки звонка. И начинается цикл с опроса кнопки звонка (**строки 71—73**). В **строке 71** содержимое порта PB считывается и помещается в регистр `temp`.

В **строке 72** проверяется значение бита номер шесть полученного числа. Именно этот бит отвечает за кнопку звонка. Если кнопка звонка нажата, то значение данного бита будет равно нулю. В этом случае команда условного перехода в **строке 73** передаст управление по метке `kk1`, где находится программа воспроизведения мелодий. Если кнопка звонка не нажата, то управление передается к **строке 74**. В **строках 74** и **75** находится знакомая нам процедура проверки кнопок набора кода.

Если ни одна из кнопок набора кода не нажата, то управление передается на начало комбинированного цикла (по метке `m1`), и цикл продолжается. При обнаружении факта нажатия одной или нескольких кнопок набора кода управление переходит к **строке 76.** С этой строки программа электронного замка не имеет отличий от оригинала.

Вся **оставшаяся часть программы** занимает **строки 76—127**. **Вспомогательные процедуры** занимают **строки 128— 185**.

Программа музыкального звонка расположена в **строках 186—295**. Ее пришлось немного доработать. Прежде всего, из программы был исключен модуль опроса клавиатуры, который в программе «Музыкальная шкатулка» (**листинг 5.17**) по нажатию одной из кнопок определял номер воспроизводимой мелодии. Вместо этой процедуры в программу введена другая, которая использует в качестве номера мелодии содержимое ячейки `melod`. При каждом нажатии кнопки номер мелодии увеличивается на единицу.

Если полученный таким образом номер превысит общее количество мелодий, то он сбрасывается в ноль, а подсчет мелодий начинается сначала. В новой шкатулке используется не семь, а восемь мелодий. Других принципиальных изменений музыкальная программа не имеет.

Начинается музыкальная программа с **модуля инициализации таймера** (**строки 186—189**). В данном случае таймер настраивается на нужды программы воспроизведения звука. В **строках 190—195** расположен упомянутый выше модуль, определяющий номер мелодии. В **строке 190** текущее значение номера мелодии читается из ячейки памяти `melod` и помещается в регистр `count`.

В **строке 191** номер мелодии увеличивается на единицу. В **строке 192** полученный таким образом новый номер мелодии сравнивается с числом 8. Если номер меньше восьми, то оператор условного перехода в **строке 193** передает управление по метке `km2`, и **строка 194** не выполняется. Если номер мелодии равен или больше восьми, то выполняется **строка 194**, где номеру мелодии присваивается нулевое значение.

В **строке 195** новый номер мелодии записывается обратно в ячейку памяти `melod`. Там он хранится до следующего нажатия кнопки звонка. Кроме того, номер мелодии остается также и в регистре `count`. Именно в этот регистр помещала номер мелодии процедура сканирования клавиатуры в программе «музыкальная шкатулка».

После определения номера мелодии управление переходит к **строке 196**. В этой строке начинается уже известная нам **программа воспроизведения мелодий**. Она почти без изменений перенесена из программы (**листинг 5.17**). Но без изменений все же не обошлось.

Во-первых, пришлось переименовать некоторые метки, так как в программе электронного замка были использованы метки с теми же именами. Второе изменение связано с условием выхода из процедуры воспроизведения мелодии. В исходной программе для этого проверялось состояние сразу семи управляющих кнопок. И если хотя бы одна из них оставалась нажатой, воспроизведение мелодии продолжалось.

В нашем случае нам нужно проверить лишь одну кнопку — кнопку звонка. **Доработанная процедура проверки** расположена в **строках 204—206**. В **строке 204** читается содержимое порта PB и помещается в регистр `temp`. В **строке 205** проверяется разряд номер 6. Именно этот разряд связан с кнопкой звонка.

Если разряд равен нулю (кнопка нажата), то воспроизведение мелодии продолжается. Если разряд равен единице (кнопка отпущена), то оператор условного перехода в **строке 206** передает управление по метке `km6`, где происходит завершение процедуры формирования звука. После выключения звука команда безусловного перехода в **строке 223** передает управление по метке `main`, то есть на начало программы замка. И первыми операциями этой программы будет перенастройка таймера под требования этой основной задачи.

В **строках 289—295** находятся **таблицы всех мелодий**. В целях сокращения занимаемого места в данном примере все мелодии максимально сокращены. При повторении данной конструкции вы можете использовать мелодии целиком, взяв их из **листинга 5.17**. Еще проще скачать эту и все остальные программы в электронном виде с сайта http://book.mirmk.net. Там все восемь мелодий представлены в полном виде.

Листинг 5.21

```
;###############################################
;##                                           ##
;##              Пример 11                    ##
;##            Кодовый замок                  ##
;##         с музыкальным звонком             ##
;##                                           ##
;###############################################

;------------------------ Псевдокоманды управления

.include "tn2313def.inc"        ; Присоединение файла описаний
.list                           ; Включение листинга

.def    drebL = R1              ; Буфер антидребезга младший байт
.def    drebH = R2              ; Буфер антидребезга старший байт
.def    temp1 = R3              ; Вспомогательный регистр
.def    temp = R16              ; Второй вспомогательный регистр
.def    data = R17              ; Регистр передачи данных
.def    flz = R18               ; Фаза работы замка
.def    count = R19             ; Регистр передачи данных
.def    addre = R20             ; Указатель адреса в EEPROM
.def    codL = R21              ; Временный буфер кода младший байт
.def    codH = R22              ; Временный буфер кода старший байт
.def    loop = R23              ; Регистр счетчика
.def    fnota = R24             ; Частота текущей ноты
.def    dnota = R25             ; Длительность текущей ноты

;------------------------ Определение констант

.equ    bsize = 60              ; Размер буфера для хранения кода
.equ    kzad  = 3000            ; Задержка при сканировании кнопок
.equ    kandr = 20              ; Константа антидребезга

;------------------------ Резервирование ячеек памяти (SRAM)

        .dseg                   ; Выбираем сегмент ОЗУ
        .org    0x60            ; Устанавливаем текущий адрес сегмента

bufr:   .byte   bsize           ; Буфер для приема кода
melod:  .byte   1               ; Номер текущей мелодии

;------------------------ Резервирование ячеек памяти  (EEPROM)

        .eseg                   ; Выбираем сегмент EEPROM
        .org    0x08            ; Устанавливаем текущий адрес сегмента

klen:   .byte   1               ; Ячейка для хранения длины кода
bufe:   .byte   bsize           ; Буфер для хранения кода

;------------------------ Начало программного кода

        .cseg                   ; Выбор сегмента программного кода
        .org    0               ; Установка текущего адреса на ноль

start:  rjmp    init            ; Переход на начало программы
        reti                    ; Внешнее прерывание 0
        reti                    ; Внешнее прерывание 1
        reti                    ; Таймер/счетчик 1, захват
        rjmp    propr           ; Таймер/счетчик 1, совпадение, канал A
        rjmp    propr           ; Таймер/счетчик 1, прерывание по переполнению
        reti                    ; Таймер/счетчик 0, прерывание по переполнению
        reti                    ; Прерывание UART прием завершен
        reti                    ; Прерывание UART регистр данных пуст
        reti                    ; Прерывание UART передача завершена
        reti                    ; Прерывание по компаратору
        reti                    ; Прерывание по изменению на любом контакте
        reti                    ; Таймер/счетчик 1. Совпадение, канал B
        reti                    ; Таймер/счетчик 0. Совпадение, канал B
        reti                    ; Таймер/счетчик 0. Совпадение, канал A
        reti                    ; USI готовность к старту
        reti                    ; USI Переполнение
        reti                    ; EEPROM Готовность
```

```
46          reti                    ; Переполнение охранного таймера
    ;*****************************************************
    ;*                                                   *
    ;*               Модуль инициализации                *
    ;*                                                   *
    ;*****************************************************
    init:
    ;-------------------------- Инициализация стека

47          ldi     temp, RAMEND            ; Выбор адреса вершины стека
48          out     SPL, temp               ; Запись его в регистр стека

    ;-------------------------- Инициализация портов B/B

49          ldi     temp,0x18               ; Инициализация порта PB
50          out     DDRB, temp
51          ldi     temp,0xE7
52          out     PORTB, temp

53          ldi     temp, 0x7F              ; Инициализация порта PD
54          out     PORTD, temp
55          ldi     temp, 0
56          out     DDRD, temp

    ;-------------------------- Инициализация (выключение) компаратора

57          ldi     temp, 0x80
58          out     ACSR, temp

    ;-------------------------- Номер мелодии

59          ldi     temp,0                  ; Сбрасываем в ноль
60          sts     melod,temp
    ;*****************************************************
    ;*                                                   *
    ;*            Начало основной программы              *
    ;*                                                   *
    ;*****************************************************

    main:
    ;-------------------------- Инициализация таймера

61          ldi     temp, high(kzad )           ; Записываем коэффициент задержки
62          out     OCR1AH, temp
63          ldi     temp, low(kzad )
64          out     OCR1AL, temp
65          ldi     temp, 0x03                  ; Выбор режима работы таймера
66          out     TCCR1B, temp

67          ldi     codL,0x7F               ; Код для сравнения (младший байт)
68          ldi     codH,0x07               ; Код для сравнения (старший байт)

69  m0:     rcall   incod                   ; Ввод и проверка кода клавиш
70          brne    m0                      ; Если хоть одна не нажата, продолжаем

71  m1:     in      temp,PINB               ; Проверка кнопки звонка
72          sbrs    temp,6
73          rjmp    kk1                     ; Если нажата, переходим к звуковой части
74          rcall   incod                   ; Ввод и проверка кода клавиш
75          breq    m1                      ; Если не одна не нажата, продолжаем

76  m2:     ldi     ZH, high(bufr)            ; Установка указателя на начало буфера
77          ldi     ZL, low(bufr)             ;
78          clr     count                     ; Сброс счетчика байт

    ;---------------------- Цикл ввода кода

79  m3:     cli                             ; Запрет всех прерываний
80          ldi     data,1                  ; Вызываем задержку первого типа
81          rcall   wait                    ; К подпрограмме задержки

82  m5:     rcall   incod                   ; Ввод и проверка кода кнопок
83          st      Z+,XL                   ; Записываем его в буфер
84          st      Z+,XH
85          inc     count                   ; Увеличение счетчика байтов
```

```
 86          inc       count
 87          cpi       count, bsize          ; Проверяем, не конец ли буфера
 88          brsh      m7                    ; Если конец, завершаем ввод кода

 89          mov       codL, XL              ; Записываем код как старый
 90          mov       codH, XH

 91          ldi       data,2                ; Вызываем задержку третьего типа
 92          rcall     wait                  ; К подпрограмме задержки

 93  m6:     rcall     incod                 ; Ввод и проверка кода кнопок
 94          brne      m3                    ; Если изменилось, записываем в буфер
 95          cpi       flz,1                 ; Проверка окончания фазы ввода кода
 96          brne      m6

 97  m7:     sbic      PINB,7                ; Проверка состояния тумблера
 98          rjmp      m9                    ; К процедуре проверки кода

     ; ----------------------------- Процедура записи кода

 99          mov       data, count           ; Помещаем длину кода в data
100          ldi       addre, klen           ; Адрес в EEPROM для хранения длины кода
101          rcall     eewr                  ; Записываем в длину кода EEPROM

102          ldi       addre, bufe             ; В регистр адреса начало буфера в EEPROM
103          ldi       ZH, high(bufr)          ; В регистровую пару Z записываем
104          ldi       ZL, low(bufr)           ; адрес начала буфера в ОЗУ

105  m8:     ld        data, Z+              ; Читаем очередной байт из ОЗУ
106          rcall     eewr                  ; Записываем в длину кода EEPROM
107          dec       count                 ; Декремент счетчика байтов
108          brne      m8                    ; Если не конец, продолжаем запись

109          rjmp      m11                   ; К процедуре открывания замка

     ; ----------------------------- Процедура проверки кода

110  m9:     ldi       addre, klen           ; Адрес хранения длины кода
111          rcall     eerd                  ; Чтение длины кода из EEPROM
112          cp        count, data           ; Сравнение с новым значением
113          brne      m13                   ; Если не равны, к началу

114          ldi       addre, bufe             ; В YL начало буфера в EEPROM
115          ldi       ZH, high(bufr)          ; В регистровую пару Z записываем
116          ldi       ZL, low(bufr)           ; адрес начала буфера в ОЗУ

117  m10:    rcall     eerd                  ; Читаем очередной байт из EEPROM
118          ld        temp, Z+              ; Читаем очередной байт из ОЗУ
119          cp        data, temp            ; Сравниваем два байта разных кодов
120          brne      m13                   ; Если не равны, переходим к началу

121          dec       count                 ; Уменьшаем содержимое счетчика байтов
122          brne      m10                   ; Если не конец, продолжаем проверку

     ; ----------------------------- Процедура открывания замка

123  m11:    sbi       PORTB,4               ; Команда "Открыть замок"
124          ldi       data,3                ; Вызываем задержку третьего типа
125          rcall     wait
126          cbi       PORTB,4               ; Команда "Закрыть замок"

127  m13:    rjmp      main                  ; Перейти к началу

     ;*****************************************************
     ;*                                                   *
     ;*             Вспомогательные процедуры             *
     ;*                                                   *
     ;*****************************************************

     ; ----------------------------- Ввод и проверка 2 байтов с клавиатуры

128  incod:  push      count

129          ldi       XL,0                  ; Обнуление регистровой пары X
```

```
130          ldi      XH,0

131  ic1:    ldi      count, kandr        ; Константа антидребезга
132          mov      drebL, XL           ; Старое значение младший байт
133          mov      drebH, XH           ; Старое значение старший байт

134  ic2:    in       XL,PIND             ; Вводим код (младший байт)
135          cbr      XL,0x80             ; Накладываем маску
136          in       XH,PINB             ; Вводим код (старший байт)
137          cbr      XH,0xF8             ; Накладываем маску

138  ic3:    cp       XL, drebL           ; Сверяем младший байт
139          brne     ic1                 ; Если не равен, начинаем с начала
140          cp       XH, drebH           ; Сверяем старший байт
141          brne     ic1                 ; Если не равен, начинаем с начала

142  ic4:    dec      count               ; Уменьшаем счетчик антидребезга
143          brne     ic2                 ; Если еще не конец, продолжаем

144          cp       XL, codL            ; Сравнение с временным буфером
145          brne     ic5                 ; Если не равно, заканчиваем сравнение
146          cp       XL, codH            ; Сравниваем старшие байты

147  ic5:    pop      count
148          ret


     ; ------------------------------ Подпрограмма задержки

149  wait:   cpi      data,1              ; Проверяем код задержки
150          brne     w1
151          ldi      temp,0x40           ; Разрешаем прерывание по совпадению
152          rjmp     w2
153  w1:     ldi      temp,0x80           ; Разрешаем прерывания по переполнению
154
155  w2:     out      TIMSK,temp          ; Записываем маску
156          clr      temp
157          out      TCNT1H,temp         ; Обнуляем таймер
158          out      TCNT1L,temp

159          ldi      flz,0               ; Сбрасываем флаг задержки
160          sei                          ; Разрешаем прерывания
161          cpi      data,2              ; Если это задержка 2-го типа
162          breq     w4                  ; Переходим к концу подпрограммы

163  w3:     cpi      flz,1               ; Ожидание окончания задержки
164          brne     w3
165          cli                          ; Запрещаем прерывания

166  w4:     ret                          ; Завершаем подпрограмму


     ; ------------------------------ Запись байта в ячейку EEPROM

167  eewr:   cli                          ; Запрещаем прерывания
168          sbic     EECR, EEWE          ; Проверяем готовность EEPROM
169          rjmp     eewr                ; Если не готов ждем
170          out      EEAR, addre         ; Записываем адрес в регистр адреса
171          out      EEDR, data          ; Записываем данные в регистр данных
172          sbi      EECR, EEMWE         ; Устанавливаем бит разрешения записи
173          sbi      EECR, EEWE          ; Устанавливаем бит записи
174          inc      addre               ; Увеличиваем адрес в EEPROM
175          ret                          ; Выход из подпрограммы


     ; ------------------------------ Чтение байта из ячейки EEPROM

176  eerd:   cli                          ; Запрещаем прерывания
177          sbic     EECR, EEWE          ; Проверяем готовность EEPROM
178          rjmp     eerd                ; Если не готов ждем
```

```
179          out       EEAR, addre          ; Устанавливаем бит инициализации чтения
180          sbi       EECR, EERE           ; Устанавливаем бит инициализации чтения
181          in        data, EEDR           ; Помещаем прочитанный байт в data
182          inc       addre                ; Увеличиваем адрес в EEPROM
183          ret                            ; Выход из подпрограммы


             ;*****************************************************
             ;*                                                   *
             ;*          Процедура обработки прерываний           *
             ;*                                                   *
             ;*****************************************************

             ; --------------------------- Прерывание по совпадению

184  propr:  ldi       flz,1                ; Установка флага задержки
185          reti                           ; Завершаем обработку прерывания

             ;#####################################################
             ;#                                                   #
             ;#                  Мелодичный звонок                #
             ;#                                                   #
             ;#####################################################

             kk1:
             ;--------------------------- Инициализация таймера T1

186          ldi       temp, 0x09           ; Включаем режим CTC
187          out       TCCR1B, temp
188  km1:    ldi       temp, 0x00           ; Выключаем звук
189          out       TCCR1A, temp

             ;-------------------------- Определение номера текущей мелодии

190          lds       count,melod          ; Читаем код текущей мелодии
191          inc       count                ; Увеличение номера мелодии на 1
192          cpi       count,8              ; Проверка на последнюю мелодию
193          brne      km2                  ; Если не последняя, переход
194          clr       count                ; Обнуление счетчика
195  km2:    sts       melod,count          ; Помещаем номер в ячейку памяти


             ;-------------------------- Выбор мелодии

196  km3:    mov       YL, count               ; Вычисляем адрес, где
197          ldi       ZL, low(tabm*2)         ; хранится начало мелодии
198          ldi       ZH, high(tabm*2)
199          rcall     addw                    ; К подпрограмме 16-разрядного сложения

200          lpm       XL, Z+                  ; Извлекаем адреса из таблицы
201          lpm       XH, Z                   ; и помещаем в X

             ;-------------------------- Воспроизведение мелодии

202  km4:    mov       ZH, XH               ; Записываем в Z начало мелодии
203          mov       ZL, XL

204  km5:    in        temp, PINB           ; Читаем содержимое порт B
205          sbrc      temp, 6              ; Проверяем нажата ли еще кнопка звонка
206          rjmp      km6                  ; Если равно (кнопки отпущены) в начало

207          lpm       temp, Z              ; Извлекаем код ноты
208          cpi       temp, 0xFF           ; Проверяем, не конец ли мелодии
209          breq      m4                   ; Если конец, начинаем мелодию сначала

210          andi      temp, 0x1F           ; Выделяем код тона из кода ноты
211          mov       fnota, temp          ; Записываем в регистр кода тона
212          lpm       temp, Z+             ; Еще раз берем код ноты
213          rol       temp                 ; Производим четырехкратный сдвиг кода ноты
214          rol       temp
215          rol       temp
216          rol       temp
```

```
217          andi     temp, 0x07            ; Выделяем код длительности
218          mov      dnota, temp           ; Помещаем ее в регистр длительности

219          rcall    nota                  ; К подпрограмме воспроизведения ноты

220          rjmp     km5                   ; В начало цикла (следующая нота)

221  km6:    ldi      temp, 0x00            ; Выключаем звук
222          out      TCCR1A, temp
223          rjmp     main                  ; Переходим к началу

     ;*****************************************************
     ;*          Вспомогательные подпрограммы             *
     ;*****************************************************

     ; ---------------------- Подпрограмма 16-ти разрядного сложения
224  addw:   push     YH

225          lsl      YL              ; Умножение первого слагаемого на 2
226          ldi      YH, 0           ; Второй байт первого слагаемого = 0
227          add      ZL, YL          ; Складываем два слагаемых
228          adc      ZH, YH

229          pop      YH
230          ret

     ;----------------------- Подпрограмма исполнения одной ноты
231  nota:   push     ZH
232          push     ZL
233          push     YL
234          push     temp

235          cpi      fnota, 0x00             ; Проверка, не пауза ли
236          breq     nt1                     ; Если пауза, переходим сразу к задержке

237          mov      YL, fnota               ; Вычисляем адрес, где хранится
238          ldi      ZL, low(tabkd*2)          ; коэффициент деления для текущей ноты
239          ldi      ZH, high(tabkd*2)
240          rcall    addw                    ; К подпрограмме 16-разрядного сложения

241          lpm      temp, Z+                ; Извлекаем мл. разряд КД для текущей ноты
242          lpm      temp1, Z                ; Извлекаем ст. разряд КД для текущей ноты
243          out      OCR1AH, temp1           ; Записать в старш. часть регистра совпадения
244          out      OCR1AL, temp            ; Записать в младш. часть регистра совпадения

245          ldi      temp, 0x40              ; Включить звук
246          out      TCCR1A, temp

247  nt1:    rcall    wait1                   ; К подпрограмме задержки

248          ldi      temp, 0x00              ; Выключить звук
249          out      TCCR1A, temp

250          ldi      dnota,0                 ; Сбрасываем задержку для паузы между нотами
251          rcall    wait1                   ; Пауза между нотами

252          pop      temp                    ; Завершение подпрограммы
253          pop      YL
254          pop      ZL
255          pop      ZH
256          ret

     ;----------------------- Подпрограмма формирования задержки
257  wait1:  push     ZH
258          push     ZL
259          push     YH
260          push     YL

261          mov      YL, dnota               ; Вычисляем адрес, где хранится
262          ldi      ZL, low(tabz*2)         ; нужный коэффициент задержки
263          ldi      ZH, high(tabz*2)
264          rcall    addw                    ; К подпрограмме 16-разрядного сложения
```

```
265          lpm       YL, Z+                    ; Читаем первый байт коэффициента задержки
266          lpm       YH, Z                     ; Читаем второй байт коэффициента задержки

267          clr       ZL                        ; Обнуляем регистровую пару Z
268          clr       ZH

             ; Цикл задержки
269  ww1:    ldi       loop,255                  ; Пустой внутренний цикл
270  ww2:    dec       loop
271          brne      ww2
272          adiw      R30, 1                    ; Увеличение регистровой пары Z на единицу
273          cp        YL, ZL                    ; Проверка младшего разряда
274          brne      ww1
275          cp        YH, ZH                    ; Проверка старшего разряда
276          brne      ww1

277          pop       YL                        ; Завершение подпрограммы
278          pop       YH
279          pop       ZL
280          pop       ZH
281          ret

     ;*****************************************************
     ;*          Таблица длительности задержек            *
     ;*****************************************************

282  tabz:   .dw       128,256,512,1024,2048,4096, 8192

     ;*****************************************************
     ;*          Таблица коэффициентов деления            *
     ;*****************************************************

283  tabkd:  .dw       0
284          .dw       4748,4480,4228,3992,3768,3556,3356,3168,2990,2822,2664, 2514
285          .dw       2374,2240,2114,1996,1884,1778,1678,1584,1495,1411,1332, 1257
286          .dw       1187,1120,1057, 998, 942, 889, 839, 792

     ;*****************************************************
     ;*           Таблица начал всех мелодий              *
     ;*****************************************************
287  tabm:   .dw       mel1*2,mel2*2,mel3*2,mel4*2
288          .dw       mel5*2,mel6*2,mel7*2

     ;*****************************************************
     ;*                 Таблица мелодий                   *
     ;*****************************************************
     ;        В траве сидел кузнечик
289  mel1:   .db       109,104,109,104,109,108,108,96,108,104,108,104,108,109,109,96,255

     ;        Песенка крокодила Гены
290  mel2:   .db       109,110,141,102,104,105,102,109,110,141,104,105,107,104,109,110,255

     ;        В лесу родилась елочка
291  mel3:   .db       132,141,141,139,141,137,132,132,132,141,141,142,139,176,128,144,255

     ;        Happy births day to you
292  mel4:   .db       107,107,141,139,144,143,128,107,107,141,139,146,144,128,107,107,255

     ;        С чего начинается родина
293  mel5:   .db        99,175,109,107,106,102, 99,144,111,175,96,99,107,107,107,107,102,255

     ;        Песня из кинофильма "Веселые ребята"
294  mel6:   .db       105,109,112,149,116,64,80,148,114,64, 78,146,112,96,105,105,109,255

     ;        Улыбка
295  mel7:   .db       107,104,141,139,102,105,104,102,164,128,104,107,109,109,109,111,255
```

Программа на языке СИ

Объединение программ на языке СИ происходит точно таким же образом, как и на Ассемблере. То есть отдельно объединяются блоки описания, блоки инициализации и основные части программ. Возможный вариант комбинированной программы приведен в **листинге 5.22**. В **строках 3—13** описываются глобальные переменные, константы и массивы. Сюда вошли элементы, используемые в обеих объединяемых программах. В **строках 14—25** описаны массивы, используемые при генерации мелодий. Причем для экономии места все мелодии здесь также сокращены.

Строки 29—47 занимают вспомогательные функции для программы электронного замка. Сюда входят:

- две процедуры обработки прерываний (**строки 26—29**);
- процедура опроса клавиатуры (**строки 30—40**);
- процедура формирования задержки (**строки 41—47**).

Все эти процедуры перенесены из исходной программы без изменений.

Как и в программе на Ассемблере, за основу основной процедуры программы на СИ взята программа электронного замка. Поэтому музыкальная программа оформлена в виде отдельной функции `muz`, которая занимает **строки 48—67**. Функция представляет собой практически полную копию основной программы музыкальной шкатулки за исключением небольших изменений, о которых мы поговорим чуть дальше.

Все переменные, относящиеся именно **к задаче воспроизведения звука**, описываются внутри функции `muz` (**строки 49—51**). Лишь для хранения кода текущей мелодии используется глобальная переменная `melod` (описывается в **строке 9**).

Процедура сканирования кнопок из программы воспроизведения мелодии исключена и заменена небольшой процедуркой в **строке 67**. В этой строке происходит увеличение переменной `melod` на единицу и сравнение полученной величины с числом 8. Если `melod` больше либо равно 8, ему присваивается нулевое значение.

Второе изменение программы воспроизведения мелодии состоит в доработке процедуры проверки нажатой кнопки. Данная проверка занимает всего одну строку. В новой программе это **строка 54**. Теперь программа проверяет состояние шестого бита порта PB. Именно этот бит связан с кнопкой звонка. В остальном программа воспроизведения мелодий полностью повторяет аналогичную программу из **листинга 5.18**.

Программа кодового замка выполнена в виде главной процедуры программы и занимает **строки 68—105**. Текст этой программы почти полностью повторяет аналогичный текст в **листинге 5.20**. **Главное**

отличие состоит в новой редакции процедуры ожидания отпускания кнопки. Она доработана и превращена в **комбинированную процедуру ожидания**.

В новой программе она занимает **строки 85—86**. Это все тот же цикл `while`, но на этот раз он не пустой. В данном случае цикл содержит один оператор. Это оператор `if` в **строке 86**. О том, что оператор `if` входит в тело цикла, свидетельствует отсутствие символа «точка с запятой» в конце **строки 85**. Для сравнения, в **строке 84** точка с запятой есть. Поэтому цикл в **строке 84** не имеет в своем теле никаких команд.

Напоминаю, что в языке СИ конец строки не является признаком окончания команды. Операторы языка СИ отделяются друг от друга символом точки с запятой. Если точки с запятой нет, то все, что идет за очередным оператором, считается его продолжением.

При этом вас не должно смущать даже наличие комментария в **строке 85**. Комментарий начинается символом «//» и заканчивается в конце строки. Для компилятора любой комментарий не существует. Поэтому команда в **строке 86** считается продолжением команды в **строке 85**. Как видите, простота и эффективность языка СИ имеет и свою обратную сторону. Наличие или отсутствие всего одного символа может кардинально изменить весь алгоритм программы.

В функции `main` и в функции `muz` новой программы используются одинаковые имена меток. В программе на Ассемблере в подобной ситуации нам пришлось переименовать метки одной из программ. В языке СИ метки, проставленные внутри одной из функций, не «видны» из другой. Поэтому переименовывать метки не обязательно.

Других изменений программа электронного замка не имеет. Нам даже не пришлось переименовывать метки.

Однако в том случае, когда вы пишете программу с нуля, все же желательно не применять одинаковых имен в разных частях программы. Просто для того, чтобы самому не запутаться в этих именах.

Листинг 5.22

```
/*****************************************************
Project : Пример 11
Version : 1
Date    : 07.03.2006
Author  : Belov
Company : Home
Comments:
Кодовый замок с музыкальным звонком
Chip type           : ATtiny2313
Clock frequency     : 4,000000 MHz
Memory model        : Tiny
Data Stack size     : 32
*****************************************************/
```

```
1   #include <tiny2313.h>
2   #include <delay.h>

3   #define klfree 0x77F                        // Код состояния при полностью отпущенных кнопках
4   #define kzad 3000                           // Код задержки при сканировании
5   #define kandr 20                            // Константа антидребезга
6   #define bsize 30                            // Размер буфера для хранения кода

7   unsigned char flz;                          // Флаг задержки
8   unsigned int bufr[bsize];                   // Буфер в ОЗУ для хранения кода
9   unsigned char melod;                        // Текущий номер мелодии

10  #pragma warn-                               // Отмена предупреждающих сообщений
11  eeprom unsigned char klen;                  // Ячейка для хранения длины кода
12  eeprom unsigned int bufe[bsize];            // Буфер в EEPROM для хранения кода
13  #pragma warn+                               // Разрешение предупреждающих сообщений

    // Объявление и инициализация массивов

    // Таблица задержек
14  flash unsigned int tabz[] =  {16,32,64,128,256,512,1024};

    // Массив коэффициентов деления
15  flash unsigned int tabkd[] = {0,4748,4480,4228,3992,3768,3556,3356,3168,2990,2822,
16            2664,2514,2374,2240,2114,1996,1884,1778,1678,1584,1495,1411,1332,1257,
17            1187,1120,1057,  998,942,889,839,792};

    // Таблицы мелодий
18  flash unsigned char mel1[] = {109,104,109,104,109,108,108,96,108,104,108,104,255};
19  flash unsigned char mel2[] = {109,110,141,102,104,105,102,109,110,141,104,105,  255};
20  flash unsigned char mel3[] = {132,141,141,139,141,137,132,132,132,141,141,142,255};
21  flash unsigned char mel4[] = {107,107,141,139,144,143,128,107,107,141,139,146,255};
22  flash unsigned char mel5[] = {99,175,109,107,106,102,99,144,111,175,96,99,107,  255};
23  flash unsigned char mel6[] = {105,109,112,149,116,64,80,148,114,64,78,146,112,255};
24  flash unsigned char mel7[] = {107,104,141,139,102,105,104,102,164,128,104,107,255};

    // Таблица начал всех мелодий
25  flash unsigned char *tabm[] = {mel1, mel2, mel3, mel4, mel5, mel6, mel7};

    // Прерывание по переполнению Таймера 1
26  interrupt [TIM1_OVF] void timer1_ovf_isr(void)
    {
27      flz=1;
    }

    // Прерывание по совпадению в канале A Таймера 1
28  interrupt [TIM1_COMPA] void timer1_compa_isr(void)
    {
29     flz=1;
    }

    // Функция опроса клавиатуры и антидребезга
30  unsigned int incod (void)
    {
31  unsigned int cod0=0;                        // Создаем локальные переменные
32  unsigned int cod1;
33  unsigned char k;

34  for (k=0; k<kandr; k++)                     // Цикл антидребезга
    {
35      cod1=PINB&0x7;                          // Формируем первый байт кода
36      cod1=(cod1<<8)+(PIND&0x7F);             // Формируем полный код состояния клавиатуры
37      if (cod0!=cod1)                         // Сравниваем с первоначальным кодом
        {
38          k=0;                                // Если не равны, сбрасываем счетчик
39          cod0=cod1;                          // Новое значение первоначального кода
```

```
        }
    }
40      return cod1;
    }

    // Процедура формирования задержки
41  void wait (unsigned char kodz)
    {
42  if (kodz==1) TIMSK=0x40;                    // Выбор маски прерываний по таймеру
43  else TIMSK=0x80;
44  TCNT1=0;                                    // Обнуление таймера
45  flz=0;                                      // Сброс флага задержки
46  #asm("sei");                                // Разрешаем прерывания
47  if (kodz!=2) while(flz==0);                 // Цикл задержки
    }

    // Музыкальная программа
48  void muz (void)
    {
49  unsigned char fnota;                        // Код тона ноты
50  unsigned char dnota;                        // Код длительности ноты
51  flash unsigned char *nota;                  // Ссылка на текущую ноту

52          TCCR1A=0x00;                        // Выключение звука

            // Воспроизведение мелодии
53  m3:     nota = tabm[melod];                 // Устанавливаем указатель на первую ноту
54  m4:     if (PINB.6!=0) goto m2;             // Если кнопка звонка не нажата, закончить
55          if (*nota==0xFF) goto m3;           // Проверка на конец мелодии
56          fnota = (*nota)&0x1F;               // Определяем код тона
57          dnota = ((*nota)>>5)&0x07;          // Определяем код длительности
58          if (fnota==0) goto m5;              // Если пауза не воспроизводим звук
59          OCR1A=tabkd[fnota];                 // Программируем частоту звука
60          TCCR1A=0x40;                        // Включаем звук
61  m5:     delay_ms (tabz[dnota]);             // Формируем задержку
62          TCCR1A=0x00;                        // Выключаем звук
63          delay_ms (tabz[0]);                 // Задержка между нотами
64          nota++;                             // Перемещаем указатель на следующую ноту
65          goto m4;                            // К началу цикла
66  m2:     TCCR1A=0x00;                        // Выключаем звук
67          if (++melod>=8) melod=0;            // Увеличиваем счетчик мелодий
    }

    // Основная функция
68  void main(void)
    {
69  unsigned char ii;           // Указатель массива
70  unsigned char i;            // Вспомогательный указатель
71  unsigned int codS;          // Старый код
72  PORTB=0xE7;                 // Порт B
73  DDRB=0x18;
74  PORTD=0x7F;                 // Порт D
75  DDRD=0x00;
76  TCCR1A=0x00;                // Таймер/Счетчик 1
77  TCCR1B=0x03;
78  TCNT1=0;
79  OCR1A=kzad;
80  ACSR=0x80;                  // Аналоговый компаратор
81  melod=0;                    // Сбрасываем счетчик мелодий

82  while (1)
83  {
84  m1:       while (incod() != klfree);            // Ожидание отпускания кнопок
85            while (incod() == klfree)             // Ожидание нажатия кнопок
86              if (PINB.6==0) muz();               // К музыкальному звонку
87            TCCR1B=0x03;                          // Настройка таймера
88            ii=0;                                 // Сброс счетчика байтов
```

```
m2:         #asm("cli");                              // Запрещаем прерывания
            wait(1);                                  // Задержка 1-го типа
            codS=incod();                             // Ввод кода и запись, как старого
            bufr[ii++]=codS;                          // Запись очередного кода в буфер
            if (ii>=bsize) goto m4;                   // Проверка конца буфера

            wait(2);                                  // Задержка 2-го типа
m3:         if (incod() != codS) goto m2;             // Проверка, не изменилось ли состояние
            if (flz==0) goto m3;                      // Проверка флага окончания задержки

m4:         if (PINB.7==1) goto comp;                 // Проверка переключателя режимов
//------------------------------ Запись кода в EEPROM
            klen=ii;                                        // Запись длины кода
            for (i=0; i<ii; i++) bufe[i]=bufr[i];           // Запись всех байтов кода
            goto zamok;                                     // К процедуре открывания замка

//------------------------------ Проверка кода

comp:       if (klen!=ii) goto m1;                  // Проверка длины кода
            for (i=0; i<ii; i++)  if (bufe[i]!=bufr[i]) goto m1;          // Проверка кода

//------------------------------ Открывание замка

zamok:      PORTB.4=1;            // Открываем замок
            wait(3);              // Задержка 3-го типа
            PORTB.4=0;            // Закрываем замок
}
}
```

ГЛАВА 6

ОРГАНИЗАЦИЯ СВЯЗИ МИКРОКОНТРОЛЛЕРА И ПК ЧЕРЕЗ USB

6.1. Почему сегодня выбран именно USB интерфейс?

С развитием компьютерной техники постоянно усложняется задача самодеятельных конструкторов, которые желают, чтобы их устройства на микроконтроллерах могли связываться с компьютером, передавать туда информацию и получать команды управления.

В настоящее время практически уже полностью вышли из употребления не только LPT порты, которые были удобнее всего для организации канала компьютер-микроконтроллер, но даже уже и COM порты. Мир переходит на всевластие USB. Однако, USB-канал, это очень сложное высокотехнологичное устройство, имеющее сложный многоуровневый алгоритм работы. Кроме того, использование USB протокола в своих разработках требует лицензирования, что для самодеятельных конструкторов и мелких конструкторских фирм очень накладно. Однако, спрос рождает предложение. Производители микросхем предлагают на сегодняшний момент множество вариантов решения этой проблемы.

Например, уже разработаны модели микроконтроллеров **со встроенным каналом USB**. Правда, в продаже их найти еще затруднительно, и стоят они дорого. Гораздо более распространен второй вариант: **микросхема — адаптер USB-COM**. Многие современные микроконтроллеры содержат в себе встроенный универсальный последовательный канал, который легко можно настроить в режим, аналогичный RS232.

А, как известно, именно этот протокол использует для передачи информации COM порт. Причем не обязательно реализовывать полный COM интерфейс. Полный стандарт RS232 требует уровни сигналов на входе и выходе: высокий — плюс 12 В, низкий — минус 12 В. В случае же, если вы используете микросхему — адаптер USB-COM, то можете иметь дело с RS232, но сигналами TTL уровней (от 0 до +5 В).

6.2. Принцип действия адаптера USB-FIFO

Существует **программный метод эмуляции USB протокола**. В данном разделе я хочу остановиться на другом, самом удобном, на мой взгляд, способе подключения вашего микроконтроллерного устройства через USB порт. Этот способ предлагает фирма FTDI Chip (Future Technology Devices International Ltd.). Логотип этой фирмы вы можете видеть на **рис. 6.1**. Это способ связи через так называемый **адаптер USB-FIFO**. Вообще-то данная фирма выпускает целый спектр устройств, основанных на этом способе передачи информации. Однако мы рассмотрим в качестве примера самую простую и самую популярную микросхему этой фирмы: FT245R. Принцип работы остальных микросхем аналогичный.



Рис. 6.1. *Логотип фирмы FTDI Chip (Future Technology Devices International Ltd.)*

Как же работает адаптер USB-FIFO? Основой адаптера является буфер FIFO. Акроним FIFO является сокращением от английского «First In, First Out» (*«Первым вошел, первым вышел»*). В вычислительной технике принцип FIFO соответствует структуре данных типа «очередь».

Буфер — это несколько ячеек памяти, объединенных в единое устройство с определенной логикой работы. К нему можно применять операцию записи байта и операцию чтения байта. При записи байта он помещается в первую из ячеек памяти буфера. Если до следующего цикла записи байт не будет прочитан, то очередной байт запишется в следующую ячейку, итак далее.

Когда все ячейки заполнятся, запись данных блокируется. Когда в буфер поступает запрос на чтение, логика FIFO помещает на выход тот байт, который был записан первым. При следующем цикле чтения на выход выдается второй записанный байт. И так далее, пока очередь не исчерпается.

Структура данных типа «**очередь»** позволяет оптимизировать обмен данными между двумя устройствами, у которых чтение и запись происходят в случайные моменты времени. Если некое устройство номер 1 в процессе своей работы изъявило желание передать байт данных в устройство номер 2, а устройство номер 2 в данный момент занято и не может принять байт, то устройство номер 1 просто записывает байт в очередь и продолжает свою работу. А устройство номер 2, когда освободится, считывает байт из очереди. Это позволяет обоим устройствам уменьшить периоды ожидания готовности друг друга.

В микросхеме FT245R в качестве устройства номер 1 используется USB интерфейс, подключенный к компьютеру, а в качестве устройства

номер 2 — микроконтроллер. Микроконтроллер подключается к микросхеме посредством простой шины данных, имеющей 8 линий (D0—D7) и 4 простые линии управления. USB интерфейс, подключенный к компьютеру, реализует виртуальный COM порт, с которым ваша программа может работать как с обычным COM портом. Рассмотрим **структуру микросхемы FT245R** подробнее.

6.3. Внутренняя структура микросхемы FT245R

На **рис. 6.2** показана внутренняя структура микросхемы FT245R. Как видим, микросхема имеет не один, а два буфера FIFO. Один буфер имеет размер 128 байт и используется для передачи информации из USB к микроконтроллеру. Второй буфер имеет размер 256 ячеек и используется для передачи данных от микроконтроллера к USB интерфейсу.

Для работы всех систем микросхема имеет встроенный тактовый генератор с умножителем частоты, который может работать даже без внешнего кварцевого резонатора. Внешний кварц приходится ставить лишь при низких напряжениях питания (1,8—3,3 В).

Микросхема реализует полноскоростной аппаратный USB канал. Для питания USB интерфейса микросхема имеет встроенный стабилизатор 3,3 В.

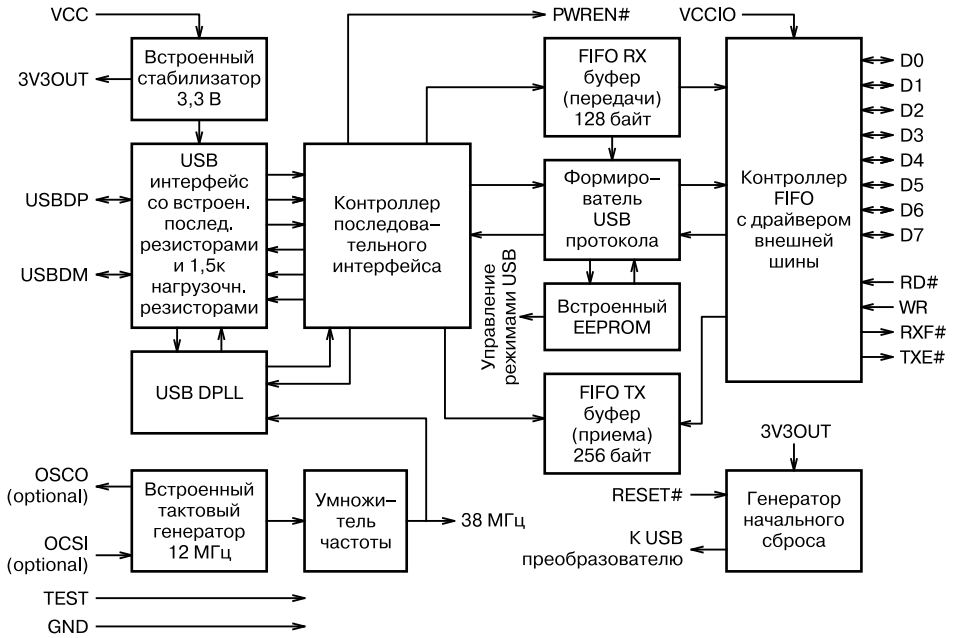


Рис. 6.2. *Блок-схема микросхемы FT245R*

Напряжение с выхода стабилизатора также подается на специальный внешний вывод микросхемы (3V3OUT) и может при необходимости быть использовано для питания маломощных устройств внешней периферии.

Микросхема также имеет встроенную энергонезависимую память EEPROM, которая используется для хранения настроек. При помощи этих настроек можно изменять некоторые режимы работы микросхемы.

Чтение и изменение содержимого EEPROM производится при помощи специальной программы FT_Prog, которую можно скачать на сайте производителя по адресу: http://www.ftdichip.com/Support/Utilities.htm

Назначение выводов микросхемы FT245R представлено в **табл. 6.1**.

Назначение выводов микросхемы FT245R *Таблица 6.1*

Контакт SSOP	Контакт QFN-32	Наименование	Тип	Описание
Группа выводов сигналов USB				
15	14	USBDP	I/O	Линия данных USB Плюс. Содержит внутренний последовательный резистор 1,5 кОм и подтягивающий резистор к линии 3,3 В
16	15	USBDM	I/O	Линия данных USB< Минус. Содержит встроенный последовательный резистор
Группа выводов питания и заземления				
4	1	VCCIO	PWR	+1,8 В…+5,25 В — питание для группы выводов FIFO Интерфейса (1-3, 5, 6, 9-14, 22, 23). В проектах, использующих напряжение USB шины, присоедините этот вывод к входу 3V3OUT, чтобы получить напряжение +3,3 В. В проектах, использующих CMOS уровни, подключите вывод к линии VCC (напряжение 5 В). На этот вывод также можно подавать напряжение от внешнего источника +1,8 В…+2,8 В для работы на более низких уровнях. Причем это внешнее напряжение должно быть сформировано из того же самого источника, что и основное питание VCC. Для этого нужно применить стабилизатор напряжения, получающий питание от линии +5 В шины USB
7, 18, 21	4, 17, 20	GND	PWR	Общий вывод всех цепей питания микросхемы
17	16	3V3OUT	Output	Выход +3,3 В встроенного стабилизатора напряжения. Этот вывод должен быть соединен с общим проводом через конденсатор 100 нФ. Главное назначение этого стабилизатора — питание напряжением 3,3 В цепей приемопередатчика USB интерфейса и питание подтягивающего резистора 1, 5 кОм для выхода USBDP. Кроме того стабилизатор может питать внешние схемы с током потребления до 50 мА. Напряжение, снимаемое с этого вывода, может также использоваться для подачи питающего напряжения на вывод VCCIO
20	19	VCC	PWR	+3,3 В…+5,25 В — питание вычислительного ядра микросхемы
25	24	AGND	PWR	Аналоговая земля — общий провод для напряжений питания умножителя тактовой частоты.
Группа выводов управления буфером FIFO				
1	30	D0	I/O	Шина данных FIFO Бит 0
2	31	D4	I/O	Шина данных FIFO Бит 4

Таблица 6.1 (продолжение)

Контакт SSOP	Контакт QFN-32	Наименование	Тип	Описание
3	32	D2	I/O	Шина данных FIFO Бит 2
5	2	D1	I/O	Шина данных FIFO Бит 1
6	3	D7	I/O	Шина данных FIFO Бит 7
9	6	D5	I/O	Шина данных FIFO Бит 5
10	7	D6	I/O	Шина данных FIFO Бит 6
11	8	D3	I/O	Шина данных FIFO Бит 3
12	9	PWREN#	Output	Низкий уровень на этом выходе свидетельствует о включении в режим USB, а высокий — о приостановке USB. Может использоваться для управления внешним ключом на основе полевого транзистора с изолированным затвором и P-каналом (MOSFET). Такой ключ позволяет управлять питанием внешних микросхем в зависимости от уровня на выходе PWREN<*. Должен быть соединен с выводом VCCIO< через резистор 10 кОм
13	10	RD#	Input	Вход записи данных в буфер FIFO через входы D0—D7. Запись байта в буфер FIFO происходит в момент перехода с высокого уровня на низкий на этом входе
14	11	WR	Input	Вход чтения байта из буфера FIFO. При переходе на этом входе с высокого уровня на низкий очередной байт данных извлекается из буфера FIFO и подается на выводы D0—D7
22	21	TXE#	Output	Выход готовности буфера FIFO для записи. При высоком уровне на этом выходе запись в буфер невозможна, а при низком — запись разрешена. (Запись производится отрицательным перепадом на входе WR). Пока идет процесс начального сброса этот выход переходит в третье состояние
23	22	RXF	Output	Вход готовности данных в буфере FIFO для чтения. Высокий уровень на этом входе возникает, если чтение данных невозможно. При низком уровне чтение разрешено. (Чтение производится импульсом низкого уровня на входе RD#). Пока идет процесс начального сброса этот выход переходит в третье состояние. Если микросхема переведена в режим разрешения автоматического пробуждения (путем записи соответствующего кода во внутренний EEPROM микросхемы), данный вывод становится входом и используется для пробуждения USB интерфейса. Если USB переходит в режим приостановки обмена (о чем сигнализирует высокий уровень на выходе PWREN#) подача импульса низкого уровня на вход RXF длительностью не менее 20 мс выводит USB из спящего режима и заставляет его возобновить обмен по USB шине
Группа выводов «разное»				
8, 24	5, 12, 13, 23, 25, 29	NC	NC	Нет внешних соединений
19	18	RESET#	Input	Вход начального сброса (низкий активный уровень). Может использоваться внешним устройством, для перезапуска микросхемы FT245R. Если внешний сброс не требуется, можно оставить этот вход свободным или соединить с линией VCC
26	26	TEST	Input	Переводит микросхему в режим проверки. В режиме нормальной работы должен быть соединен с общим проводом GND, иначе микросхема будет вести себя как дефектная
27	27	OSCI	Input	Вход модуля генератора 12 МГц. Использование не обязательно. Для нормальной работы можно оставить вход не подключенным
28	28	OSCO	Output	Выход модуля генератора 12 МГц. Использование не обязательно. Для нормальной работы можно оставить вход не подключенным

6.4. Типовая схема подключения микроконтроллера к ПК

Типовая схема подключения микроконтроллера к компьютеру через USB интерфейс при помощи микросхемы FT245R приведена на **рис. 6.3**.

После того, как вы соберете данную схему и подключите ее к USB разъему компьютера, операционная система тут же обнаружит новое устройство и предложит установить драйвера. Драйвера для данного устройства легко скачать с сайта производителя. Там вы найдете два вида драйверов.

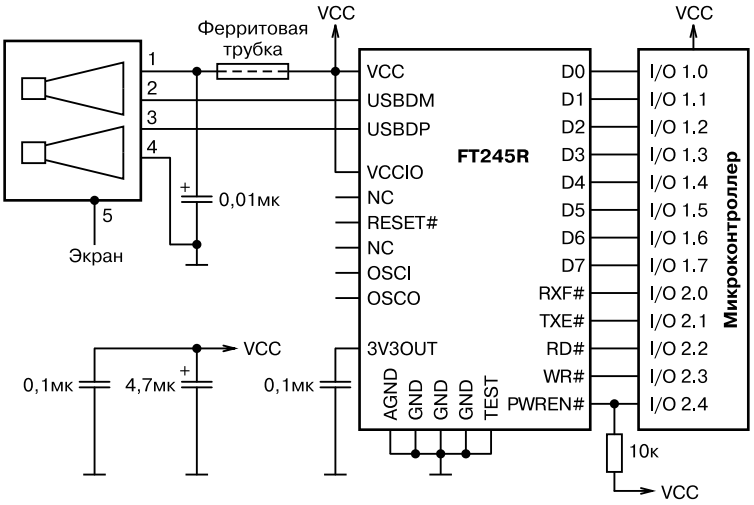


Рис. 6.3. *Типовая схема подключения микроконтроллера к ПК через USB интерфейс при помощи микросхемы FT245R*

6.5. Драйверы для адаптера и принципиальная схема

Первый тип драйвера называется **VCP** (Virtual COM port). Он реализует, как видно из названия, виртуальный COM порт в компьютере, и прикладные программы могут использовать его как стандартное COM устройство. Этот драйвер вы можете скачать по адресу:

http://www.ftdichip.com/Drivers/VCP.htm

Второй тип драйвера называется **D2XX**. Этот драйвер после установки добавляет в операционную систему специализированную библиотеку функций (специальный DLL файл), предоставляющую программисту прямой доступ к USB функциям. Подробное описание этих функций имеется в специальном руководстве, которое также можно скачать (правда, на английском языке) на сайте производителя (файл под названием D2XX Programmer's Guide).

Драйвер D2XX для этой микросхемы и файл с описанием функций можно скачать по адресу: http://www.ftdichip.com/Drivers/D2XX.htm

После установки драйверов вы можете писать свои программы, работающие с FIFO. Любой байт, переданный через виртуальный COM порт или напрямую при помощи функций D2XX, будет записан в выходной буфер FIFO микросхемы FT245R, откуда он может быть считан микроконтроллером.

Если же микроконтроллер запишет байт в микросхему FT245R, это вызовет прерывание по COM порту или просто установит флаг готовности данных (в зависимости от настройки порта). Ваша программа на компьютере может просто считать байт из COM порта или воспользоваться функцией чтения из библиотеки D2XX. При этом очередной байт будет извлечен из входного буфера FIFO микросхемы FT245R. Ну а то, как интерпретировать передаваемые байты вы должны придумать сами.

В настоящее время на радиорынках страны можно встретить в продаже не только микросхему FT245R, но также уже готовую плату адаптера как с уже установленными деталями, так и без.

На **рис. 6.4** приведен внешний вид готовой платы, а на **рис. 6.5** прилагаемая к ней принципиальная схема модуля.

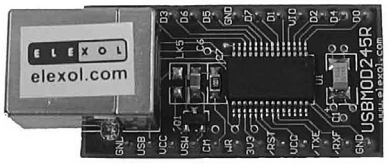


Рис. 6.4. *Внешний вид модуля готовой платы*

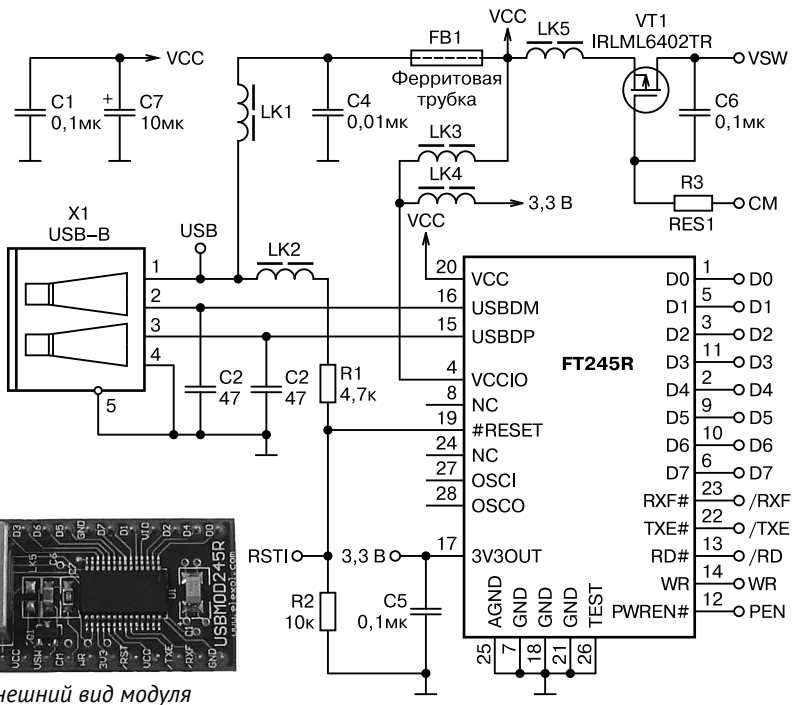


Рис. 6.5. *Принципиальная схема модуля*

ПРИЛОЖЕНИЕ 1

СВОДНАЯ ТАБЛИЦА КОМАНД АССЕМБЛЕРА МИКРОКОНТРОЛЛЕРОВ AVR

Группа команд логических операций

Мнемоника	Описание	Операция	Циклы	Флаги
AND Rd, Rr	«Логическое И» двух РОН	Rd ← Rd • Rr	1	Z, N, V
ANDI Rd, K	«Логическое И» РОН и константы	Rd ← Rd • K	1	Z, N, V
EOR Rd, Rr	«Исключающее ИЛИ» двух РОН	Rd ← Rd ⊕ Rr	1	Z, N, V
OR Rd, Rr	«Логическое ИЛИ» двух РОН	Rd ← Rd ∨ Rr	1	Z, N, V
ORI Rd, K	«Логическое ИЛИ» РОН и константы	Rd ← Rd ∨ K	1	Z, N, V
COM Rd	Перевод в обратный код	Rd ← 0FFH – Rd	1	Z, C, N, V
NEG Rd	Перевод в дополнительный код	Rd ← 00H – Rd	1	Z, C, N, V, H
CLR Rd	Сброс всех разрядов РОН	Rd ← Rd ⊕ Rd	1	Z, N, V
SER Rd	Установка всех разрядов РОН	Rd ← 0FFH	1	–
TST Rd	Проверка РОН на отрицательное (нулевое) значение	Rd ← Rd • Rd	1	Z, N, V

Группа команд арифметических операций

Мнемоника	Описание	Операция	Циклы	Флаги
ADD Rd, Rr	Сложение двух РОН	Rd ← Rd + Rr	1	Z, C, N, V, H
ADC Rd, Rr	Сложение двух РОН с переносом	Rd ← Rd + Rr + C	1	Z, C, N, V, H
ADIW Rd, K	Сложение регистровой пары с константой	Rdh:Rdl ← Rdh:Rdl + K	2	Z, C, N, V, S
SUB Rd, Rr	Вычитание двух РОН	Rd ← Rd – Rr	1	Z, C, N, V, H
SUBI Rd, K	Вычитание константы из РОН	Rd ← Rd – K	1	Z, C, N, V, H
SBC Rd, Rr	Вычитание двух РОН с заемом	Rd ← Rd – Rr – C	1	Z, C, N, V, H
SBCI Rd, K	Вычитание константы из РОН с заемом	Rd ← Rd – K – C	1	Z, C, N, V, H
SBIW Rd, K	Вычитание константы из регистровой пары	Rdh:Rdl ← Rdh:Rdl – K	2	Z, C, N, V, S
DEC Rd	Декремент РОН	Rd ← Rd – 1	1	Z, N, V
INC Rd	Инкремент РОН	Rd ← Rd + 1	1	Z, N, V

Группа команд операций с разрядами

Мнемоника	Описание	Операция	Циклы	Флаги
CBR Rd, K	Сброс разряда(ов) РОН	Rd ← Rd • (0FFH – K)	1	Z,N,V
SBR Rd, K	Установка разряда(ов) РОН	Rd ← Rd ∨ K	1	Z,N,V
CBI A, b	Сброс разряда РВВ	A.b ← 0	2	–

Мнемоника	Описание	Операция	Циклы	Флаги
SBI A, b	Установка разряда РВВ	A.b ← 1	2	–
BCLR s	Сброс флага	SREG.s ← 0	1	SREG.s
BSET s	Установка флага	SREG.s ← 1	1	SREG.s
BLD Rd, b	Загрузка разряда РОН из флага T (SREG)	Rd.b ← T	1	–
BST Rr, b	Запись разряда РОН в флаг T (SREG)	T ← Rd.b	1	T
CLC	Сброс флага переноса	C ← 0	1	C
SEC	Установка флага переноса	C ← 1	1	C
CLN	Сброс флага отрицательного числа	N ← 0	1	N
SEN	Установка флага отрицательного числа	N ← 1	1	N
CLZ	Сброс флага нуля	Z ← 0	1	Z
SEZ	Установка флага нуля	Z ← 1	1	Z
CLI	Общий запрет прерываний	I ← 0	1	I
SEI	Общее разрешение прерываний	I ← 1	1	I
CLS	Сброс флага знака	S ← 0	1	S
SES	Установка флага знака	S ← 1	1	S
CLV	Сброс флага переполнения дополнительного кода	V ← 0	1	V
SEV	Установка флага переполнения дополнительного кода	V ← 1	1	V
CLT	Сброс пользовательского флага T	T ← 0	1	T
SET	Установка пользовательского флага T	T ← 1	1	T
CLH	Сброс флага половинного переноса	H ← 0	1	H
SEH	Установка флага половинного переноса	H ← 1	1	H

Группа команд сравнения

Мнемоника	Описание	Операция	Циклы	Флаги
CP Rd, Rr	Сравнение двух РОН	? (Rd – Rr)	1	Z,N,V,C,H
CPC Rd, Rr	Сравнение РОН с учетом переноса	? (Rd – Rr – C)	1	Z,N,V,C,H
CPI Rd, K	Сравнение РОН с константой	? (Rd – K)	1	Z,N,V,C,H

Группа команд операций сдвига

Мнемоника	Описание	Операция	Циклы	Флаги
ASR Rd	Арифметический сдвиг вправо	Rd7 → Rd6 → Rd5 → Rd4 → Rd3 → Rd2 → Rd1 → Rd0	1	Z,C,N,V
LSL Rd	Логический сдвиг влево	C ← Rd7 ← Rd6 ← Rd5 ← Rd4 ← Rd3 ← Rd2 ← Rd1 ← Rd0 ← 0	1	Z,C,N,V
LSR Rd	Логический сдвиг вправо	0 → Rd7 → Rd6 → Rd5 → Rd4 → Rd3 → Rd2 → Rd1 → Rd0 → C	1	Z,C,N,V
ROL Rd	Сдвиг влево через перенос	C ← Rd7 ← Rd6 ← Rd5 ← Rd4 ← Rd3 ← Rd2 ← Rd1 ← Rd0 ← C	1	Z,C,N,V
ROR Rd	Сдвиг вправо через перенос	C → Rd7 → Rd6 → Rd5 → Rd4 → Rd3 → Rd2 → Rd1 → Rd0 → C	1	Z,C,N,V
SWAP Rd	Обмен местами тетрад	Rd(3–0) ↔ Rd(7–4)	1	–

Группа команд пересылки данных

Мнемоника	Описание	Операция	Циклы	Флаги
MOV Rd, Rr	Пересылка между РОН	Rd ← Rr	1	–
MOVW Rd, Rr	Пересылка между парами регистров	Rd +1:Rd ← Rr+1:Rr	1	–
LDI Rd, K	Загрузка константы в РОН	Rd ← K	1	–
LD Rd, X	Косвенное чтение	Rd ← [X]	2	–
LD Rd, X+	Косвенное чтение с постикрементом	Rd ← [X], X ← X+1	2	–
LD Rd, -X	Косвенное чтение с преддекрементом	X ← X-1, Rd ← [X]	2	–
LD Rd, Y	Косвенное чтение	Rd ← [Y]	2	–
LD Rd, Y+	Косвенное чтение с постикрементом	Rd ← [Y], Y ← Y+1	2	–
LD Rd, –Y	Косвенное чтение с преддекрементом	Y ← Y-1, Rd ← [Y]	2	–
LD Rd, Y+q	Косвенное относительное чтение	Rd ← [Y+q]	2	–
LD Rd, Z	Косвенное чтение	Rd ← [Z]	2	–
LD Rd, Z+	Косвенное чтение с постикрементом	Rd ← [Z], Z ← Z+1	2	–
LD Rd, -Z	Косвенное чтение с преддекрементом	Z ← Z-1, Rd ← [Z]	2	–
LD Rd, Z+q	Косвенное относительное чтение	Rd ← [Z+q]	2	–
LDS Rd, k	Непосредственное чтение из ОЗУ	Rd ← [k]	2	–
ST X, Rr	Косвенная запись	[X] ← Rr	2	–
ST X+, Rr	Косвенная запись с постикрементом	[X] ← Rr, X ← X+1	2	–
ST -X, Rr	Косвенная запись с преддекрементом	X ← X-1, [X] ← Rr	2	–
ST Y, Rr	Косвенная запись	[Y] ← Rr	2	–
ST Y+, Rr	Косвенная запись с постикрементом	[Y] ← Rr, Y ← Y+1	2	–
ST -Y, Rr	Косвенная запись с преддекрементом	Y ← Y-1, [Y] ← Rr	2	–
ST Y+q, Rr	Косвенная относительная запись	[Y+q] ← Rr	2	–
ST Z, Rr	Косвенная запись	[Z] ← Rr	2	–
ST Z+, Rr	Косвенная запись с постикрементом	[Z] ← Rr, Z ← Z+1	2	–
ST -Z, Rr	Косвенная запись с преддекрементом	Z ← Z-1, [Z] ← Rr	2	–
ST Z+q, Rr	Косвенная относительная запись	[Z+q] ← Rr	2	–
STS k, Rr	Непосредственная запись в ОЗУ	[k] ← Rr	2	–
LPM	Загрузка данных из памяти программ	R0 ← {Z}	3	–
LPM Rd, Z	Загрузка данных из памяти программ	Rd ← {Z}	3	–
LPM Rd, Z+	Загрузка данных из памяти программ и постдекремент Z	Rd ← {Z}, Z ← Z+1	3	–
SPM	Запись в программную память	{Z} ← R1:R0	–	–
IN Rd, P	Пересылка из РВВ в РОН	Rd ← P	1	–
OUT P, Rr	Пересылка из РОН в РВВ	P ← Rr	1	–
PUSH Rr	Сохранение байта в стеке	STACK ← Rr	2	–
POP Rd	Извлечение байта из стека	Rd ← STACK	2	–

Группа команд управления системой

Мнемоника	Описание	Операция	Циклы	Флаги
NOP	Нет операции	–	1	–
SLEEP	Переход в «спящий» режим	–	3	–
WDR	Сброс сторожевого таймера	–	1	–
BREAK	Приостановка программы	Используется только при отладке	–	–

Группа команд передачи управления (безусловная передача управления)

Мнемоника	Описание	Операция	Циклы	Флаги
RJMP	Относительный безусловный переход	PC ← PC + k + 1	2	–
IJMP	Косвенный безусловный переход	PC ← Z	2	–
RCALL	Относительный вызов подпрограммы	PC ← PC + k + 1	3	–
ICALL	Косвенный вызов подпрограммы	PC ← Z	3	–
RET	Возврат из подпрограммы	PC ← STACK	4	–
RETI	Возврат из подпрограммы обработки прерываний	PC ← STACK	4	I

Группа команд передачи управления (пропуск команды по условию)

Мнемоника	Описание	Условие	Циклы	Флаги
Все команды этой группы пропускают следующую за ней команду (PC ← PC + 1) при разных условиях:				
CPSE Rd, Rr	Сравнение и пропуск следующей команды при равенстве	Если Rd = Rr	1/2/3	–
SBRC Rr, b	Пропуск следующей команды если разряд РОН сброшен	Если Rr,b = 0	1/2/3	–
SBRS Rr, b	Пропуск следующей команды если разряд РОН установлен	Если Rr.b = 1	1/2/3	–
SBIC A, b	Пропуск следующей команды если разряд РВВ сброшен	Если A.b = 0	1/2/3	–
SBIS A, b	Пропуск следующей команды если разряд РВВ установлен	Если A.b = 1	1/2/3	–

Группа команд передачи управления (передача управления по условию)

Мнемоника	Описание	Условие	Циклы	Флаги
Все команды этой группы выполняют переход (PC ← PC + k + 1) при разных условиях:				
BRBC s, k	Переход, если флаг s регистра SREG сброшен	Если SREG.s = 0	1/2	–
BRBS s, k	Переход, если флаг s регистра SREG установлен	Если SREG.s = 1	1/2	–
BRCS k	Переход по переносу	Если C = 1	1/2	–
BRCC k	Переход, если нет переноса	Если C = 0	1/2	–
BREQ k	Переход по условию «равно»	Если Z = 1	1/2	–
BRNE k	Переход по условию «неравно»	Если Z = 0	1/2	–
BRSH k	Переход по условию «больше или равно»	Если C = 0	1/2	–
BRLO k	Переход по условию «меньше»	Если C = 1	1/2	–
BRMI k	Переход по условию «отрицательное значение»	Если N = 1	1/2	–
BRPL k	Переход по условию «положительное значение»	Если N = 0	1/2	–
BRGE k	Переход по условию «больше или равно» (со знаком)	Если (N и V) = 0	1/2	–
BRLT k	Переход по условию «меньше» (со знаком)	Если (N или V) = 1	1/2	–
BRHS k	Переход по половинному переносу	Если H = 1	1/2	–
BRHC k	Переход, если нет половинного переноса	Если H = 0	1/2	–
BRTS k	Переход, если флаг T установлен	Если T = 1	1/2	–
BRTC k	Переход, если флаг T сброшен	Если T = 0	1/2	–
BRVS k	Переход по переполнению дополнительного кода	Если V = 1	1/2	–
BRVC k	Переход, если нет переполнения дополнительного кода	Если V = 0	1/2	–
BRID k	Переход, если прерывания запрещены	Если I = 0	1/2	–
BRIE k	Переход, если прерывания разрешены	Если I = 1	1/2	–

ПРИЛОЖЕНИЕ 2

КАК РАБОТАТЬ С ВИРТУАЛЬНЫМ ДИСКОМ

Данная книга для уменьшения ее стоимости не комплектуется CD диском, как это сделано в других изданиях. Не смотря на это читатель, купив именно эту книгу, ничего не потеряет. Взамен CD диска автором книги разработан и размещен в Интернете его аналог — «Виртуальный диск». Виртуальный диск вы всегда можете найти по адресу http://book.mirmk.ru/wdisk

Виртуальный диск не уступает CD по объему представленной информации. Так же, как и CD, Виртуальный диск, служит для закрепления материала, изложенного в книге. Рекомендуется воспользоваться диском лишь после того, как в процессе изучения книги вы перейдете к **главе 5**.

Данный виртуальный диск используется для нескольких изданий книги. Эти издания отличаются по объему информации, имеют разные названия. Однако Виртуальный диск полностью совместим с любым из этих изданий, и может использоваться для работы с любым из них без всяких ограничений.

Так же, как и CD, Виртуальный диск содержит ряд видеоуроков, которые помогут читателю глубже изучить материал и увидеть воочию процесс отладки программы и прошивки микроконтроллера. На Виртуальном диске вы найдете полный архив всех программных примеров из книги в электронном виде. Вы всегда можете скачать программные примеры на свой жесткий диск, подключить их к среде AVR Studio или CODE Vision и самостоятельно опробовать процесс отладки на практике. Затем вы можете попробовать изменить и усовершенствовать предложенные программы и таким образом совершенствовать свои навыки программирования. Кроме видеоуроков и архива программных примеров Виртуальный диск содержит большой набор справочных материалов, а так же ссылки на сайты производителей инструментальных программ, описанных в книге.

И так, где найти и как работать с Виртуальным диском? Для доступа к диску вам потребуется компьютер, подключенный к сети Интернет. Наберите в адресной строке вашего браузера адрес виртуального диска (см. выше) и перейдите к этому адресу. Вы увидите главное меню диска (см. **рис. п1**). Из главного меню вы можете перейти к любому из разделов диска. В каждом разделе имеется кнопка (или ссылка) при помощи



Рис. п1. *Главное меню Виртуального диска*

которой вы можете вернуться назад, к основному меню. Рассмотрим все основные разделы по порядку.

Раздел «Примеры программ из книги»

При переходе по одноименной кнопке откроется окно, показанное на **рис. п2.** Это простое окно, которое содержит описание файла архива программных примеров и краткие советы по установке примеров на жесткий диск вашего компьютера. Для скачивания архива щелкните мышкой по кнопке «Загрузить». Скачанный файл – это набор файлов и директорий, упакованный программой ZIP. Распакуйте содержимое архива в корневую директорию диска «C» вашего компьютера, сохраняя всю структуру заложенных в архив директорий. В результате на вашем жестком диске должна появиться директория «C:\BookProgramm\», в которой разместится целая система подкаталогов, содержащая файлы программных примеров.

Обязательно размещайте все файлы именно в корне диска C и именно в указанной здесь папке. В некоторых случаях при переносе программ в другое место на диске проекты переставали открываться в программе AVR Studio.

Скопированные описанным выше способом примеры программ — это и примеры, описанные в книге, и дополнительные примеры, предназначенные для самостоятельного изучения. Примеры из книги вы найдете в подпапках:

«C:\BookProgramm\ATtiny2313\asm\» (программы на Ассемблере);

«C:\BookProgramm\ATtiny2313\C\» (программы на СИ).

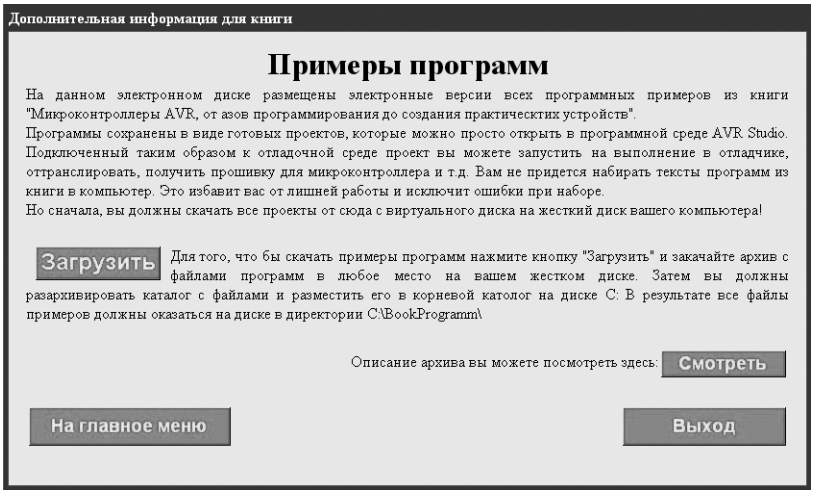


Рис. п2. *Окно раздела «Примеры программ»*

Дополнительные примеры, не описанные в книге, вы найдете в директории «C:\BookProgramm\». В их числе: программы, аналогичные примерам из книги, но адаптированные для более старого микроконтроллера AT90S2313. А также несколько программ, написанных на языке GCC.

Раздел «Видеоуроки»

При переходе к этому разделу вы увидите на экране окно, показанное на **рис. п3**. Из этого окна вы можете запустить любой из четырех видеоуроков. Сначала откроется окно по центру, которого вы увидите видеоплеер с картинкой-заставкой. Для начала воспроизведения виде-



Рис. п3. *Окно раздела «Видеоуроки»*

оурока щелкните мышкой по центру видеоплеера. После того, как видео будет запущено, разверните его на весь экран, щелкнув мышкой по специальному символу в правом нижнем углу видеоплеера. Этот символ появляется на экране только тогда, когда указатель мыши находится в пределах видео изображения.

Виртуальный диск содержит следующие видеоуроки:

- Приемы работы с программой AVR Studio (загрузка программ на Ассемблере, трансляция, отладка).
- Приемы работы с программой Code Vision (загрузка программ на СИ, трансляция, отладка).
- Работа с программной оболочкой программатора PonyProg.
- Организация рабочего места конструктора устройств на микроконтроллерах AVR.

После того, как вы изучите одну из вышеперечисленных тем в книге, можете закрепить ее при помощи соответствующего видео урока.

Раздел «Инструментальные программы»

Перейдя к этому разделу, вы увидите на экране окно, показанное на **рис. п4**. Раздел содержит набор ссылок на страницы конкретных сай-



Рис. п4. *Окно раздела «Инструментальные программы»*

тов в Интернете, на которых вы можете из первоисточника скачать последние версии основных инструментальных программ, используемых в книге. В настоящее время раздел содержит ссылки на четыре программы. Программная среда «Atmel Studio» — это последняя версия программы «AVR Studio», описанной в книге. Программная среда CODE Vision — это описанная в книге программная среда для программирования на языке СИ. Ну и два программатора: **PoniProg** и **USBasp**. Этот набор ссылок в будущем может дополняться и уточняться.

Раздел «Справочные материалы»

Главное окно этого раздела показано на рис. п5. Основа раздела — это большая таблица 8-ми разрядных микроконтроллеров производства фирмы Atmel. Таблица разбита на шесть подразделов: «AVR Tiny», «AVR Mega», «AVR XMega», «90PWM», «90CAN», «90USB». По соответствующей ссылке вы можете напрямую перейти к нужному подразделу. В самих подразделах имеются перекрестные ссылки, по которым вы можете легко переходить от одного подраздела к другому. Каждый подраздел — это большая таблица, где для каждого конкретного микроконтроллера выбранной серии приведены все его основные параметры. Такая таблица позволяет легко выбрать нужный микроконтроллер или сравнить их параметры. Щелкнув мышкой по названию любого микроконтроллера, вы переходите к фирменному его описанию, так называемому Даташиту (Datasheets). Это документ от производителя (оригинал на английском языке), содержащий полное описание микросхемы.

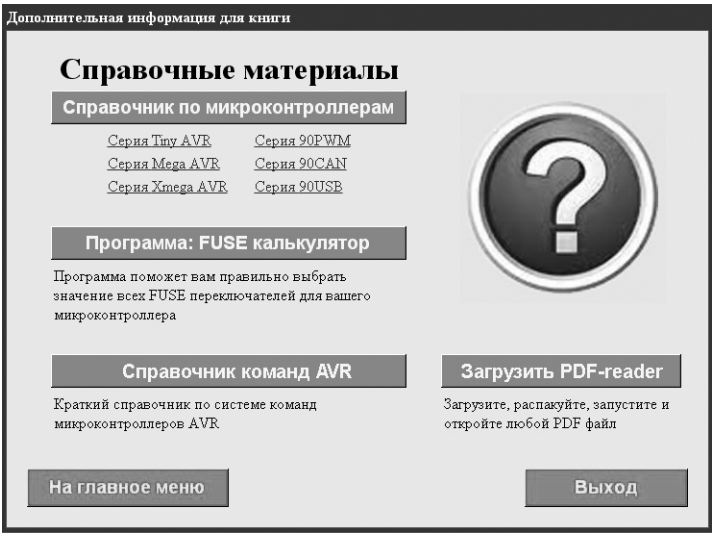


Рис. п5. *Окно раздела «Справочные материалы»*

Описание микросхемы позволяет правильно использовать все возможности микроконтроллера.

Еще один вид справочной информации, который вы найдете в разделе «Справочные материалы» — это ссылка на ON Line службу, которая называется «FUSE калькулятор». При конструировании разных устройств на микроконтроллерах часто возникает необходимость выбрать правильные значения для Фьюзов вашего микроконтроллера. Что такое Фьюзы, вы уже знаете из соответствующих разделов этой книги. Программа FUSE-калькулятор помогает быстро выбрать значение всех Фьюзов.

При нажатии кнопки «FUSE калькулятор», вы переходите на сайт калькулятора. В открывшемся окне программы вы должны выбрать из всплывающего списка тип нужного вам микроконтроллера. Первоначально в поле выбора вы увидите надпись: «Выберите микросхему». Сразу после выбора микросхемы в том же окне откроется список параметров, выбор которых в данной конкретной микросхеме контролируется при помощи уникального для данного вида микросхем набора FUSE переключателей. Наименование параметров в программе указаны на русском языке. Разобраться с параметрами вам так же помогут всплывающие подсказки. Выберите нужные значения всех параметров и в окне, озаглавленном «Результат (установите в программаторе)» сразу же появятся искомые значения Фьюзов.

Формат окна результатов точно соответствует формату окна программы PonyProg, в которое вы должны перенести значения Фьюзов, перед тем как прошить их в микроконтроллер. Кроме того, ниже окна результатов имеется маленькое окошко, в котором те же результаты представлены в формате, принятом в программаторе AVRDUDE.

Ну и последний вид информации, который вы можете получить в разделе «Справочные материалы» — это справочник команд микроконтроллеров AVR. Нажав соответствующую кнопку, вы перейдете к страничке, где в краткой форме, в виде набора таблиц описан основной набор команд и их краткое описание. Данный справочник просто необходим любому программисту, который занимается написанием программ на языке Ассемблера. Рекомендую этот справочник распечатать на бумаге.

В окне «Справочные материалы» имеется также кнопка, при помощи которой вы можете скачать с Виртуального диска простейший PDF-reader. Это программа, предназначенная для просмотра файлов, записанных в формате PDF. В этом формате, например, представлены все Даташиты, о которых мы говорили чуть выше. Если у вас не установлена другая программа для просмотра PDF файлов, скачайте этот про-

стейший просмотрщик, распакуйте архив и запишите его содержимое в любом месте на вашем жестком диске. Архив содержит папку, в которой находятся файлы программы. Запустите из этой папки файл **PDFReader.exe**. В открывшемся окне откройте любой PDF файл при помощи пункта меню **File/open**. Программа откроет файл и одновременно привяжет все файлы с расширением *.pdf к программе **PDFReader**. После этого открывать программу **PDFReader** больше не нужно. Теперь вы сможете открывать любой PDF файл в любом окне, просто щелкнув по нему мышкой.

Список литературы

1. *Белов А. В.* Конструирование устройств на микроконтроллерах. — Санкт-Петербург: Наука и Техника. — 2005.
2. *Белов А.В.* Разработка устройств на микроконтроллерах AVR: шагаем от «чайника» до профи. Книга + видеокурс на CD. — Санкт-Петербург: Наука и Техника. — 2013.
3. *Белов А.В.* «Микроконтроллеры AVR от азов программирования до создания практических устройств». Наука и Техника. — 2016.
4. *Евстифеев А. В.* Микроконтроллеры AVR семейств Tiny и Mega фирмы ATMEL / Мировая электроника. — Изд. 5. — М.: Издательский дом «Додэка-XXI». — 2008.

Список полезных ссылок на ресурсы Интернет

1. Официальный сайт данной книги и всех предыдущих изданий: http://book.mirmk.ru
2. Личный сайт автора книги: http://belov.mirmk.ru
3. Сайт фирмы Atmel, производителя микроконтроллеров AVR: http://www.atmel.com, http://www.atmel.ru
4. Сайт разработчика программы CodeVisionAVR: http://www.hpinfotech.ro
   4.1. Страница загрузки всех версий программы CodeVisionAVR: http://www.hpinfotech.ro/cvavr_download.html
5. Сайт разработчика программы Pony Prog: http://www.lancos.com
   5.1. Страница, посвященная программатору Pony Prog на сайте разработчика: http://www.lancos.com/prog.html
   5.2. Страница загрузки всех версий программы Pony Prog: http://www.lancos.com/ppwin95.html
6. Сайт программатора USBasp: http://www.fischl.de/usbasp
7. Официальные сайты издательства Наука и Техника: http://www.nit.com.ru и http://www.nit-kiev.com
8. Адрес виртуального диска книги http://book.mirmk.ru/wdisk
9. Справочник команд микроконтроллеров семейства AVR: http://fusecalc.mirmk.ru/sc/sk.htm

Уважаемые господа!
Книги издательства «Наука и Техника»

Вы можете заказать наложенным платежом
в наших интернет-магазинах
www.nit.com.ru, www.nit-kiev.com

а также приобрести

➢ **в крупнейших магазинах г. Москвы:**

ТД «БИБЛИО-ГЛОБУС»	ул. Мясницкая, д. 6/3, стр. 1, ст. М «Лубянка»	тел. (495) 781-19-00, 624-46-80
Московский Дом Книги,	ул.Новый Арбат, 8, ст. М «Арбатская», «ДК на Новом Арбате»	тел. (495) 789-35-91
Московский Дом Книги,	Ленинский пр., д. 40, ст. М «Ленинский пр.», «Дом технической книги»	тел. (499) 137-60-19
Московский Дом Книги,	Комсомольский пр., д. 25, ст. М «Фрунзенская», «Дом медицинской книги»	тел. (499) 245-39-27
Дом книги «Молодая гвардия»	ул. Б. Полянка, д. 28, стр. 1, ст. М «Полянка»	тел. (499) 238-50-01
Сеть магазинов «Новый книжный»	тел. (495) 937-85-81, (499) 177-22-11	

➢ **в крупнейших магазинах г. Санкт-Петербурга:**

Санкт-Петербургский Дом Книги	Невский пр. 28 тел. (812) 448-23-57
«Энергия»	Московский пр. 57 тел. (812) 373-01-47
«Аристотель»	ул. А. Дундича 36, корп. 1 тел. (812) 778-00-95
Сеть магазинов «Книжный Дом»	тел. (812) 559-98-28

➢ **в регионах России:**

г. Воронеж, пл. Ленина д. 4	«Амиталь»	(4732) 24-24-90
г. Екатеринбург, ул. Антона Валека д. 12	«Дом книги»	(343) 253-50-10
г. Екатеринбург	Сеть магазинов «100 000 книг на Декабристов»	(343) 353-09-40
г. Нижний Новгород, ул. Советская д. 14	«Дом книги»	(831) 277-52-07
г. Смоленск, ул. Октябрьской революции д. 13	«Кругозор»	(4812) 65-86-65
г. Челябинск, ул. Монакова, д. 31	«Техническая книга»	(904) 972 50 04
г. Хабаровск	Сеть книжно-канцелярских магазинов фирмы «Мирс»	(4212) 26-87-30

➢ **и на Украине (оптом и в розницу) через представительство издательства**

www.nit-kiev.com

e-mail: nitkiev@gmail.com

Мы рады сотрудничеству с Вами!