

В ПОМОЩЬ РАДИОЛЮБИТЕЛЮ

Майк Предко

Устройства управления роботами

схемотехника и программирование

*Архитектура микроконтроллеров PICmicro,
управление исполнительными устройствами,
подключение периферийных устройств, датчиков, детекторов*



В помощь радиолюбителю

Майк Предко

УСТРОЙСТВА
УПРАВЛЕНИЯ РОБОТАМИ:
СХЕМОТЕХНИКА
И ПРОГРАММИРОВАНИЕ

СОДЕРЖАНИЕ

Введение

8

1

Использование микроконтроллеров в автоматике

18

1.1. Управление роботом

18

1.2. Основные компоненты устройств управления

21

1.3. Память и устройства ввода-вывода

26

1.4. Прерывания

29

1.5. Периферийные устройства микроконтроллеров

32

1.6. Подключение устройства управления к роботу

34

2

Разработка программного обеспечения

36

2.1. От исходного текста программы к HEX-файлу

37

2.2. Ассемблер

40

2.3. Интерпретаторы

43

2.4. Компиляторы

46

2.5. Симуляторы и эмуляторы

51

2.6. Интегрированные средства разработки

53

3

Микроконтроллеры PICmicro

56

3.1. Основные особенности микроконтроллеров PICmicro

58

3.2. Инструментальные средства разработки программ

61

3.3. Простые схемы

79

3.4. Описание микроконтроллера PIC16F627

84

3.5. Программатор El Cheapo

120

4

Подключение к микроконтроллеру периферийных устройств

129

4.1. Аппаратные интерфейсы

130

4.2. Шаблон программы на языке C

133

4.3. Макетирование устройств на основе микроконтроллеров PICmicro

137

4.4. Межпроцессорные коммуникации

141

4.5. Реализация интерфейса RS-232

143

4.6. Программа HyperTerminal

148

4.7. Реализация интерфейса RS-232 для связи микроконтроллера с персональным компьютером	152
4.8. Двухнаправленный синхронный интерфейс	162
4.9. Устройства индикации	166
4.10. Светодиодные индикаторы	167
4.11. Управление светодиодным индикатором	170
4.12. Использование широтно-импульсной модуляции для управления аналоговыми устройствами	178
4.13. Управление яркостью светодиодного индикатора	180
4.14. Использование пьезоэлектрических излучателей и звуковых динамиков	189
4.15. Устройство звуковой сигнализации	191
4.16. Использование жидкокристаллического дисплея	195
4.17. Вывод информации на жидкокристаллический дисплей	202
4.18. Датчики	213
4.19. Механические датчики	214
4.20. Подавление дребезга контактов	218
4.21. Инфракрасный детектор столкновений	226
4.22. Инфракрасный обнаружитель объектов	229
4.23. Дистанционное управление роботом	238
4.24. Приемник сигналов дистанционного управления	241
4.25. Совмещение работы детектора объектов и приемника команд дистанционного управления	246
4.26. Ультразвуковой дальномер	252
4.27. Подключение ультразвукового дальномера к микроконтроллеру	255
4.28. Световые датчики	263
4.29. Подключение световых датчиков к микроконтроллеру	266
4.30. Звуковые датчики	279
4.31. Распознавание звуковых команд	283
4.32. Управление двигателем	289
4.33. Модели фирмы Tamiya в качестве прототипа радиолюбительских конструкций	291
4.34. Одометрия	305
4.35. Радиоуправляемый сервопривод	310
4.36. Простое радиоуправляемое устройство	312

5.1. Операционные системы реального времени	325
5.2. Пример приложения, работающего под управлением ОСРВ	330
5.3. Конечные автоматы	336
5.4. Дистанционное управление роботом, совершающим случайные блуждания	340
5.5. Поведенческое программирование	343
5.6. Нейронные сети и искусственный интеллект	348

6	Проектирование автоматических устройств	354
	6.1. Техническое задание	355
	6.2. Выбор периферийных устройств	358
	6.3. Выбор электронных компонентов и методов программирования	359
	6.4. Испытания робота	361
	6.5. Поиск ошибок	363
	6.6. Модернизация устройств	364
	Приложение	366
	Предметный указатель	399

ВВЕДЕНИЕ

Каждый год накануне Хэллоуина показывают очередной специальный выпуск известного мультсериала «Симпсоны» — нечто среднее между фильмом ужасов и научной фантастикой. В одном из моих любимых эпизодов мистер Берне, руководитель атомной станции, хочет построить робота, чтобы заменить им своих ленивых сотрудников. Единственная вещь, которой он не учитывает в своем проекте, — это управляющее устройство, контроллер. Самое простое решение заключается в том, чтобы использовать мозг Гомера Симпсона. Но тот имеет свое собственное представление о жизни и работе. Разочаровавшись в своем проекте Идеального Работника, мистер Берне вынимает мозг Гомера из машины.

Хотя описанный эпизод кажется не слишком подходящим для серьезного введения в программирование контроллеров для управления роботами, он хорошо иллюстрирует один момент, который я часто наблюдал, общаясь с настоящими разработчиками автоматических устройств. Многие из них и в самом деле уделяют не слишком много внимания разработке контроллера, а сосредотачивают свои усилия на проектировании механических частей, датчиков и двигателей. Работая над книгой, я обнаружил, что многие прекрасные проекты так и не нашли применения из-за того, что их разработчики не продумали работу управляющего устройства или не обеспечили возможность его модернизации.

Цель моей книги, как и всех остальных книг серии «Основы робототехники» (*Robot DNA*), заключается в том, чтобы снабдить проектировщика практическими методиками и надлежащим инструментом, которые смогут гарантировать работоспособность проектируемого автоматического устройства и облегчат его модификацию и модернизацию. Я считаю, что блок управления (контроллер) является важной частью любого автоматического устройства — по крайней мере, не менее важной, чем все другие его подсистемы, — и что проектировщик должен вспомнить о контроллере в самом начале своего проекта, еще до того, как тот примет свои окончательные очертания.

В данной книге я познакомлю вас с тем, как программируются контроллеры для управления подвижными механизмами — роботами. При этом будет рассмотрена совместная работа всех подсистем: входных датчиков, выходных устройств, блоков управления и периферийного оборудования связующим звеном (как говорят, *интерфейсом*) между которыми служит микроконтроллер. Познакомившись с функциями, выполняемыми всеми этими подсистемами, мы выясним необходимые подробности техники программирования микроконтроллеров, позволяющие связать все эти функции воедино и создать программу высокого уровня для управления роботом.

Хотя большое количество роботов управляется с помощью компьютеров, специальных микропроцессорных устройств, нейронных сетей или простых световых

датчиков, в данной книге упор делается на применение однокристальных микроконтроллеров. Это не означает, что приведенные здесь примеры не могут быть перенесены на другие аппаратные платформы; просто мне доставляет искреннее удовольствие использовать все преимущества именно этих устройств. К поразительной функциональности и легкости применения микроконтроллеров следует добавить удивительную устойчивость к ошибкам разработчиков, которая обеспечивается современными КМОП микросхемами. В это трудно поверить, но большинство схем, приведенных в книге, может быть собрано и проверено всего за несколько минут.

Помимо введения в искусство программирования контроллеров для управления роботами будут представлены и проиллюстрированы на нескольких примерах методы разработки устройств ввода-вывода. Это поможет вам разобраться в том, как можно обеспечить сопряжение различных устройств друг с другом. Мы будем обсуждать три различных типа устройств ввода-вывода: устройства ввода информации (датчики, сенсоры), устройства вывода и устройства управления серводвигателями. Вы узнаете, как можно программировать устройства различного типа, чтобы обеспечить наиболее гармоничное их взаимодействие.

В большинстве случаев будет использоваться микроконтроллер PIC16F627 фирмы Microchip. Он имеет 18 выводов, память команд объемом 1 Кб и еще 68 байт для хранения переменных, до шестнадцати линий ввода-вывода (некоторые из которых могут быть использованы в качестве компараторов напряжения), два 8-разрядных и один 16-разрядный таймеры, последовательный порт, простую одновекторную систему прерываний. Память этого микроконтроллера выполнена по Flash-технологии. Это означает, что он может быть многократно перепрограммирован с помощью простейшего устройства - *программатора*, и для стирания предыдущей программы не требуется источник ультрафиолетовых лучей. На сайте издательства www.dmk.ru, а также на моем персональном сайте www.myke.com можно найти описание простого программатора и программное обеспечение, необходимое для его использования.

Несмотря на явно скромные возможности однокристальных микроконтроллеров (по сравнению с большими ЭВМ), они получили чрезвычайно широкое применение. В сети Internet в свободном доступе имеется огромное количество ресурсов по их применению и программированию; о микроконтроллерах написано не так уж мало книг. Как показало мое неофициальное исследование многочисленных любительских конструкций, микроконтроллер PIC16F627 и его предшественник PIC16C84 наиболее популярны у радиолюбителей.

Примеры схем и программ, приведенные в этой книге, выполнены таким образом, чтобы в максимальной степени облегчить перенос рассмотренных решений на другие микроконтроллеры. Сейчас уже имеются сотни, а может быть, и тысячи различных микроконтроллеров, которые пригодны для управления роботами. Поэтому в программах на языке C мы будем избегать машинных команд и аппаратных особенностей, специфичных для микроконтроллеров каждого типа, акцентируясь на общих возможностях, которые присущи многим из них. Хотя в книге будет рассмотрена концепция операционной системы реального времени (RTOS — Real-Time

Operating System), большинство примеров программ использует только один вектор прерывания. Контроллер PIC16F627, как и многие другие, не может обеспечить работу RTOS в полном объеме, но существует множество других методов программирования микроконтроллеров, позволяющих облегчить труд разработчика. Эти методы будут проиллюстрированы в книге на многочисленных примерах.

Разработка программного обеспечения для микроконтроллеров PIC еще более упрощается благодаря интегрированной среде разработки MPLAB IDE (Integrated Development Environment) фирмы Microchip, а также совместимому с этой средой компилятору PICC Lite для языка C фирмы HI-TECH. Оба инструментальных средства распространяются бесплатно и доступны на сайтах www.microchip.com, www.microchip.ru и www.htsoft.com. Несмотря на это, они обеспечивают возможности, за которые при использовании других контроллеров пришлось бы заплатить не одну тысячу долларов.

При описании процесса программирования управляющих устройств роботов мы выделим три уровня задач, которые условно будут именоваться *биологический*, *механический* и *электронный*¹. Значения этих терминов проиллюстрированы на рис. 1.

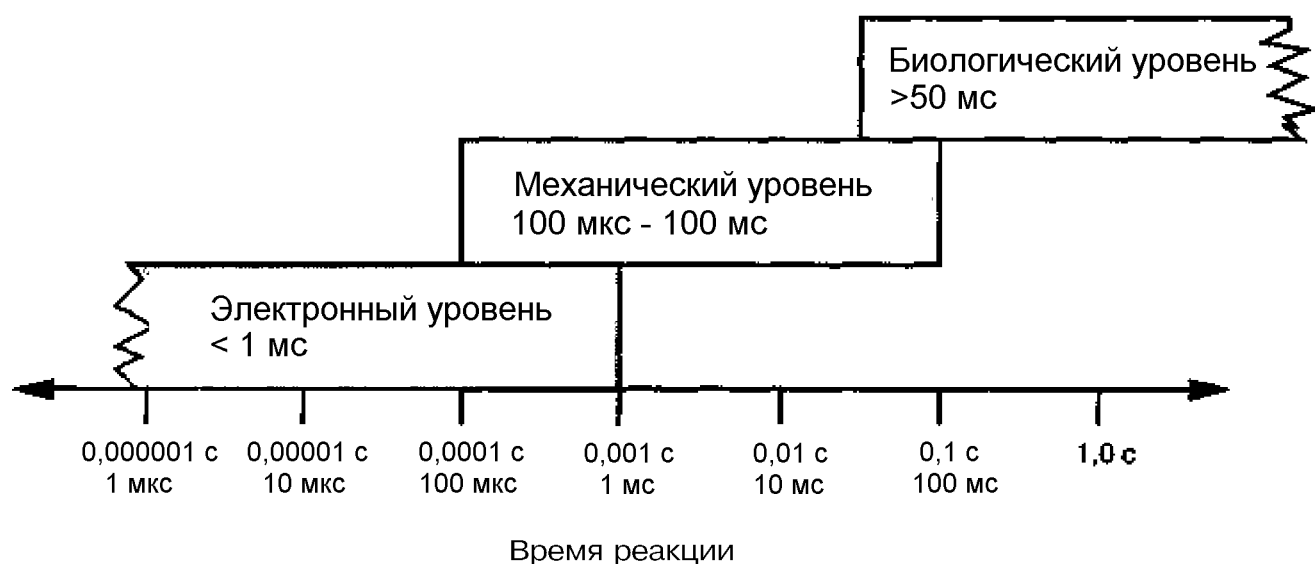


Рис. 1. Три круга задач, решаемых роботом

Биологический уровень является самым верхним - на нем контроллер должен принимать стратегические решения, поэтому программирование на этом уровне самое трудное и подразумевает обязательное использование методов искусственного интеллекта. Зато высокого быстродействия здесь не требуется: время задержки реакции на входной стимул может достигать 50 мс (1/20 секунды). Биологический уровень обеспечивает реакции, сходные с теми, за которые отвечает

¹ В оригинале автор использует придуманные им самим термины "biologic", "mechalogic" и "elelogic".
Прим. перев.

центральная нервная система человека. При этом предполагается, что специфические функции, связанные с управлением конкретных аппаратных средств и интерфейсов, берут на себя другие подсистемы. Например, в человеческом теле управление мускулами осуществляет периферическая нервная система, ее функции мы будем относить к остальным двум уровням.

Команды *механического уровня* обеспечивают управление механическими устройствами, входящими в состав автоматической системы, например электродвигателями постоянного или переменного тока. Время реакции здесь может лежать в диапазоне от 100 мкс до 100 мс. Этот уровень - самый простой с точки зрения программной и аппаратной реализации. Для управления механическими функциями мы обычно будем использовать систему прерываний.

Электронный уровень отвечает за взаимодействие друг с другом различных электронных устройств, входящих в состав автоматической системы, а также реализует интерфейсы ввода-вывода. Команды данного уровня требуют максимального быстрого действия, чтобы снизить временные затраты на передачу информации внутри системы. Обычно функции электронного уровня не слишком трудны для программирования, но необходимо увязывать их выполнение с другими процессами, происходящими в системе на иных уровнях.

В этой книге мы рассмотрим законченные примеры автоматических устройств. Хотя в фокусе нашего внимания будут контроллеры, но как уже говорилось, все подсистемы робота должны проектироваться совместно, поэтому описанные здесь подходы можно использовать при изучении других книг серии Robot DNA

Возможно, вас удивит тот факт, что простой однокристальный микроконтроллер PIC16F627, который и по своей производительности, и по объему памяти не дотягивается даже до первого компьютера Apple, может не только эффективно управлять различными периферийными устройствами робота, но еще и обеспечивать принятие решений на верхнем уровне. Тем не менее уверяю вас - с этими задачами микроконтроллер справляется намного лучше, чем мозги Гомера Симпсона.

ДЛЯ КОГО ПРЕДНАЗНАЧЕНА ЭТА КНИГА

Как и в других книгах серии Robot DNA здесь мы будем предполагать, что читатель имеет элементарные навыки программирования и знаком с основами электроники. Если вы можете написать простейшую программу, выводящую на экран приветствие «Hello World!», и когда-нибудь собирали простые электронные схемы, состоящие, может быть, всего из нескольких элементов, то эта книга не должна показаться вам слишком трудной для понимания.

Все программы в книге написаны на языке C и спроектированы так, чтобы особенности аппаратуры использовались в минимальной степени, поэтому не сложно модифицировать их в случае применения другого микроконтроллера или компилятора. При выборе другого контроллера необходимо учитывать, что он должен иметь возможность обработки прерывания от таймера и при изменении входного сигнала. Это необходимо для реализации последовательности выполнения

различных функций электронного и механического уровней. Тогда приложение верхнего (биологического) уровня сможет работать в однозадачном режиме, не «замечая», что оно периодически приостанавливается из-за программы обработки прерываний, которая обеспечивает выполнение функций остальных двух уровней.

Если вы не знакомы с языком C, советую обратиться к книге Б. Кернигана и Д. Ритчи «Язык программирования Си»¹. Она по праву считается «библией» языка C и поможет вам освоить в том числе операторы и алгоритмические конструкции, которые находят применение при разработке программ для контроллеров с помощью компилятора PICC Lite.

Некоторые склонны считать язык C трудным для изучения и использования, но вряд ли какой другой язык программирования может сравниться с ним по своим возможностям манипулирования низкоуровневыми данными. Разумеется, язык C не слишком приспособлен для написания программ управления базами данных или программирования Internet-приложений, но он просто идеален для управления аппаратными средствами. Программирование на C упрощается благодаря тому, что он является структурным и строго типизированным, а эти свойства крайне необходимы для программирования электронных схем вообще и роботов в частности.

В языке C, однако, существует возможность существенно усложнить понимание логики работы программы. Например, последовательность операторов

```
A = A + 1;
B = A » 4;
if (B == 16) {
```

может быть записана следующим образом:

```
if ((B = (++A >> 4)) == 16) {
```

Здесь значение переменной A увеличивается на единицу, переменной B присваивается значение, в 4 раза большее нового значения A, а затем следует условный оператор, выполняющийся при B = 16. В этой книге мы будем избегать таких трюков, чтобы не запутать читателя и упростить перевод программ на другие языки программирования, например Basic.

Другая особенность языка C заключается в использовании указателей. В последнее время наметилась тенденция ухода от явного использования указателей в языках программирования высокого уровня (в частности, Java). Но важно понимать, что, например, строка в языке C реализуется в виде указателя на массив символов.

Чтобы собрать конструкции, описанные в книге, требуются элементарные знания в области электроники. Они нужны хотя бы для того, чтобы при случае вы нашли замену необходимым электронным компонентам. Если вы слышали

¹ На русском языке книга опубликована в издательстве «Невский диалект» (СПб, 2000). - Прим. ред.

о законах Ома и Кирхгофа и кое-что смыслите в транзисторах и логических элементах, то для вас не составит труда реализовать описанные здесь устройства.

Умение собрать электронную схему на макетной плате еще важнее, чем знание теории. Для многих конструкций, представленных в книге, будут приведены рисунки печатных плат. Если вы захотите использовать эти устройства для управления реальными роботами, то для повышения надежности их работы предпочтительны *печатный монтаж* или *монтаж накруткой*.

Для разработки программ и их загрузки в микроконтроллер удобно использовать персональный компьютер. Я настоятельно рекомендую держать программы, относящиеся к различным устройствам, в разных папках (директориях). Для работы с нижеописанными программами требуется элементарная «компьютерная грамотность», а также умение использовать любой текстовый редактор.

Наконец, будет очень хорошо, если у вас есть доступ к сети Internet. Не только для того, чтобы заглянуть на сайт автора и скачать новые версии программ, а также поправки к тексту этой книги, но и чтобы узнать характеристики различных электронных компонентов и готовых устройств. Роботам посвящено огромное количество сайтов, серверов рассылок и других Internet-ресурсов; эта дополнительная информация поможет вам в разработке своего собственного робота.

ОСНОВНЫЕ ОБОЗНАЧЕНИЯ

В этой книге используется ряд следующих соглашений:

- *курсивом* обозначены фрагменты, на которые следует обратить особое внимание, в частности термины, вынесенные в предметный указатель;
- элементы интерфейса и клавиши обозначаются полужирным шрифтом, например: меню File (Файл), клавиша F6;
- сочетания клавиш, которые следует нажимать одновременно, записываются со знаком + (плюс), например: **Ctrl+L**;
- последовательность действий, выполняемых в меню, отмечается стрелочкой, например: Debug ⇒ Simulator Stimulus ⇒ Asynchronous Stimulus (Отладка ⇒ Входные воздействия ⇒ Асинхронные стимулы);
- фрагменты программного кода и надписи на дисплеях выделяются моноширинным шрифтом, например: `if (ButtonPress == 0)`. При этом моноширинным *курсивом* отмечены фрагменты кода, вместо которых следует подставить какое-либо значение (например, `#include "Файл"` ⇒ `<Файл>`, где *Файл* - имя файла, содержимое которого нужно включить в текст программы);
- подчеркиванием оформляются адреса Web-сайтов.

На сайте издательства «ДМК Пресс» (www.dmk.ru) вы найдете электронное приложение к данной книге, в котором содержатся все исходные тексты программ и описание простого программатора, а на персональном сайте автора (www.mukey.com) - новые версии программ и поправки к английскому изданию.

В табл. 1 расшифрованы основные обозначения, используемые в книге.

Таблица 7. Основные обозначения

Обозначение		Описание
Ω	Ом	Единица измерения электрического сопротивления, ом
k	кОм	Килоом, 1 кОм = 10^3 Ом
M Ω	МОм	МОм - мегаом, 1 МОм = 10^6 Ом
μ F	мкФ	Единица измерения емкости, микрофарада
pF	пФ	Пикофарада, 1 пФ = 10^{-12} Ф = 10^{-6} мкФ
secs	с	Единица измерения времени, секунда
msecs	мс	Миллисекунда, 1 мс = 10^{-3} с
μ secs	мкс	Микросекунда, 1 мкс = 10^{-6} с
nsecs	нс	Наносекунда, 1 нс = 10^{-9} с
psecs	пс	Пикосекунда, 1 пс = 10^{-12} с
Hz	Гц	Единица измерения частоты, герц
kHz	кГц	Килогерц, 1 кГц = 10^3 Гц
MHz	МГц	Мегагерц, 1 МГц = 10^6 Гц
GHz	ГГц	Гигагерц, 1 ГГц = 10^9 Гц
#####		Десятичное целое число
- #####		Отрицательное десятичное целое число
0x0 #####		Шестнадцатеричное целое число
0b0 #####		Двоичное целое число
n.nn x 10 ^e		Вещественное десятичное число (в экспоненциальном формате)
[]		В квадратных скобках указываются необязательные параметры
		Вертикальная черта разделяет альтернативы («или одно, или другое»)
_		Знак подчеркивания в конце строки означает перенос на следующую строку
_Label		Знак подчеркивания перед названием сигнала указывает на отрицательную логику (активным считается низкий уровень сигнала)
Register.Bit		Точка отделяет обозначение регистра от номера бита
//		Фраза, следующая за двойным слэшем, представляет собой комментарий
&		Поразрядная операция И Входы Выход A B 0 0 0 0 1 0 1 0 0 1 1 1
&&		Логическая операция И
		Поразрядная операция ИЛИ Входы Выход A B 0 0 0 0 1 1 1 0 1 1 1 1

Таблица 7. Основные обозначения (окончание)

Обозначение	Описание
, OR	Логическая операция ИЛИ
^	Поразрядная операция ИСКЛЮЧАЮЩЕЕ ИЛИ Входы Выход A B 0 0 0 0 1 1 1 0 1 1 1 0
XOR	Логическая операция ИСКЛЮЧАЮЩЕЕ ИЛИ
!	Поразрядное отрицание(побитная инверсия) Вход Выход 0 1 1 0
NOT	Логическое отрицание
+	Арифметическое сложение
-	Арифметическое вычитание
x, *	Умножение
/	Деление
%	Остаток от деления одного числа на другое
<< #	Сдвиг влево на указанное число разрядов
>> #	Сдвиг вправо на указанное число разрядов

ЗАРЕГИСТРИРОВАННЫЕ ТОРГОВЫЕ МАРКИ

Компания Microchip является владельцем следующих охраняемых торговых марок: PIC, PICmicro, ICSP, KEELOQ, MPLAB, PICSTART, PRO MATE и PIC-MASTER. Фирме microEngineering Labs, Inc. принадлежит торговая марка PicBasic. PICC и PICC-Lite - торговые марки Microchip, но лицензия на них принадлежит HI-TECH Software. HI-TECH C - торговая марка HI-TECH Software. Корпорация Microsoft владеет торговыми марками Windows 95, Windows 98, Windows NT, Windows 2000 и Visual Basic. Остальные не указанные здесь продукты и названия являются собственностью их владельцев.

БЛАГОДАРНОСТИ

Эта книга (и вся серия, в которую она входит) не появилась бы на свет без энтузиазма Скотта Грилло (Scott Grillo) - моего редактора в издательстве McGraw-Hill. Больше всего я благодарен ему за то время, что он потратил на мои книги, посвященные использованию и программированию микроконтроллеров PIC, и за неоценимую помощь, которую он оказал мне и Гордону МакКомбу (Gordon McComb), направляя наши дискуссии и помогая определить тематику каждой книги серии "Robot DNA".

Гордону МакКомбу принадлежит идея создания вышеназванной серии, и я ценю выпавшую мне возможность внести свой вклад в это дело. Мы оба провели немало часов за телефонными разговорами, обсуждая название и содержание

каждой главы. Гордон - уникальный специалист в области робототехники, и из каждого разговора с ним я извлекал для себя что-нибудь новое.

Как всегда, неоценимым источником идей для меня оказался Бен Уирз (Ben Wirz), соавтор одной из моих книг по микроконтроллерам. Я всегда наслаждаюсь нашими длительными беседами, когда мы в очередной раз пытаемся дать определение понятию «Совершенный Робот». Может быть, когда-нибудь мы наконец создадим нечто близкое к нему.

Приведенные в книге устройства могут быть разработаны на основе любого микроконтроллера, выпускаемого в настоящее время, но мой выбор пал на контроллеры PICmicro благодаря технической поддержке и инструментальным средствам проектирования/отладки фирм Microchip и HI-TECH Software. Возможности интегрированной среды разработки MPLAB IDE и номенклатура выпускаемых микроконтроллеров превосходят продукты других фирм и, без сомнения, при выборе поставщика микроконтроллеров для любого проекта Microchip должна рассматриваться в первую очередь.

Эта книга не была бы написана без помощи Грега Андерсона (Greg Anderson), Эла Ловрича (Al Lovrich) и Кима Ван Херка (Kim Van Herk) из фирмы Microchip. Их помощь и поддержку, в целом характерную для сотрудников Microchip, я воспринимаю как стандарт, по которому должны оцениваться остальные фирмы.

Я также благодарен всем помогавшим мне сотрудникам HI-TECH Software, особенно Клайду Стаббсу Смиту (Clyde Stubbs Smythe) и Марку Лукману (Mark Luckman). Качество компилятора PICC Lite, а также поддержка в его освоении достигают той высокой планки, которую установила фирма Microchip.

Как всегда, я должен поблагодарить всех подписчиков Internet-рассылки PICList (около 2000 человек), которые помогли Microchip разработать один из лучших микроконтроллеров, выпускаемых в настоящее время. В книге не хватит страниц, чтобы перечислить всех помогавших мне советом или давших ответы на многочисленные вопросы.

В течение последнего года я был приобщен ко всем радостям, выпадающим на долю владельца собаки, так что выражаю свою признательность Суси (Susi) - нашей сибирской лайке, благодаря прогулкам с которой я имел возможность отвлечься от всех проблем, связанных с книгой или программированием. Признаюсь, во время наших прогулок по Дон-Миллсу¹ я изучал траекторию блужданий Суси, пытаюсь оценить алгоритм, который заставил бы робота двигаться сходным образом.

Сейчас у нас появился щенок Лобо (Lobo), который не только укрепляет мою сердечно-сосудистую систему, но также служит еще одним неиссякаемым источником самых замысловатых траекторий для моих разработок.

Я благодарен моей дочери Марии (Marya) за ее вопросы, предложения и тот энтузиазм, с которым она проводила тестирование моих устройств (в большинстве случаев разрушительное). Я знаю не так уж много семилетних детей, которые

¹ Don Mills - город в Канаде. - Прим. *перев.*

мечтают стать генетиками и тем не менее находят время для робототехнических изысканий своих родителей.

Как всегда, безмерно благодарен своей жене, чье имя по праву звучит как *Терпение* (Patience). Спасибо ей за то, что она позволяла мне проводить бесконечные часы за компьютером и не забывала напоминать в три часа утра, что пора идти спать.

Я думаю, каждый из нас хотя бы немного изменил свои взгляды на жизнь 11 сентября 2001 года. Я выражаю благодарность всем, кто путешествует по этому миру, пытаюсь сделать его более безопасным и справедливым. Ваши попытки не напрасны, и я искренне надеюсь, что все вы благополучно вернетесь домой.

*Майк Предко
Торонто, Канада*

ГЛАВА 1

ИСПОЛЬЗОВАНИЕ

МИКРОКОНТРОЛЛЕРОВ

В АВТОМАТИКЕ

Работая над этой книгой, среди сведений, доступных в сети Internet, я собрал информацию о более чем 360 различных примерах применения микроконтроллеров при разработке автоматических устройств, чтобы выяснить, как они устроены, как управляются, какие периферийные устройства при этом используются. Оказывается, подавляющее большинство автоматических устройств управляется с помощью микроконтроллеров. Исследуя схемы и листинги программ, я обнаружил, что многие проекты могли бы быть существенно улучшены, если бы их разработчики с должным вниманием отнеслись к проектированию и программированию управляющего устройства.

Но прежде чем обсуждать методы программирования микроконтроллеров для управления роботами, хотелось бы сделать шаг назад и немного поговорить о самих микроконтроллерах и их основных характеристиках, которые имеют решающее значение при проектировании управляющих устройств.

В этой главе представлена базовая информация о микроконтроллерах и рассмотрены основные критерии, по которым следует выбирать микроконтроллер для управления автоматическими устройствами.

1.1. УПРАВЛЕНИЕ РОБОТОМ

Заглянув в любой справочник по электронным компонентам, вы обнаружите огромное количество различных устройств (и не обязательно однокристальных), которые производители называют *контроллерами* или *микроконтроллерами* (МК). Пытаясь выявить общие черты, присущие всем этим устройствам, можно заметить, что единственный объединяющий их признак - обеспечение преобразования входных сигналов в выходные согласно заданному алгоритму. Вы можете возразить, что это слишком широкое определение - так и амебу можно назвать гуманоидом только потому, что она тоже питается и размножается; но в приведенном определении есть одна тонкость, которую следует осознавать при рассмотрении задач управления подвижными автоматическими устройствами. Чтобы

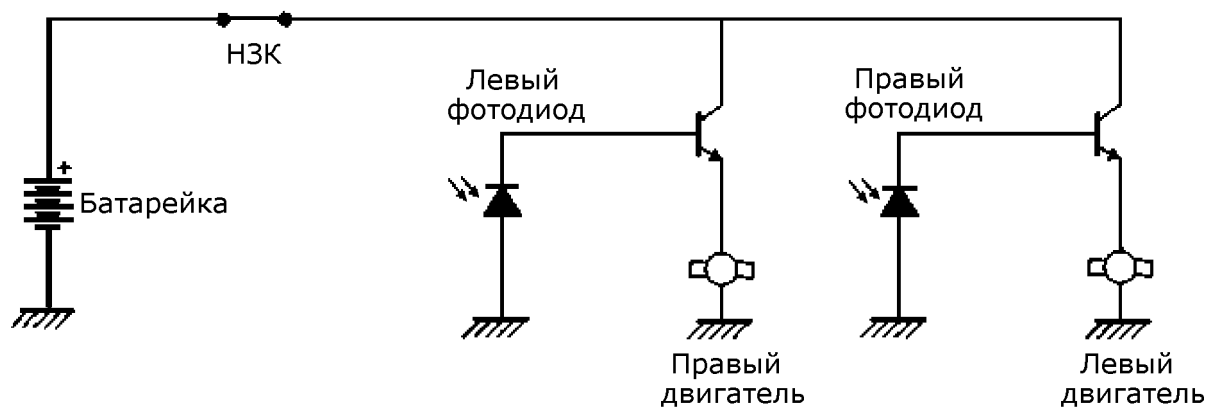


Рис. 1.1. Простейшая схема управления роботом

проиллюстрировать этот момент, рассмотрим схему робота, в задачу которого входит поиск источника света (рис. 1.1).

Здесь использован *нормально замкнутый контакт* (НЗК) - микропереключатель, который размыкает цепь, если робот наткнется на какой-нибудь предмет. В таком случае оба двигателя выключаются. Каждый из двух фотодиодов при освещении генерирует электрический ток, который подается на базу соответствующего *n-p-n* транзистора, управляя током, текущим через левый и правый двигатель. Левый фотодиод управляет правым двигателем, и наоборот. Чем больший ток генерирует левый фотодиод, тем больший ток протекает через обмотки правого двигателя. В результате робот поворачивает в более освещенном направлении. Эта простая схема очень похожа на те, что использовались для управления самыми первыми экспериментальными роботами.

Если оба фотодиода освещены одинаково, то оба двигателя работают на одном и том же токе, и робот будет двигаться по прямой в направлении источника света. Когда он наткнется на источник света, микропереключатель выключит оба двигателя. Чем ближе робот находится к источнику (или чем ярче тот светит), тем быстрее движется робот. Это происходит потому, что при увеличении яркости возрастает число фотонов, вызывающих протекание тока через фотодиоды, - как следствие, увеличивается ток базы обоих транзисторов и, соответственно, ток эмиттера, к которому подключен каждый двигатель. Скорость робота в момент столкновения с источником может оказаться слишком большой - тогда произойдет авария.

Описанная схема может считаться контроллером робота, поскольку обеспечивает заранее определенный алгоритм его движения. Она подходит под приведенное выше определение контроллера: по состоянию двух входных датчиков формируются два выходных сигнала управления двигателями. Однако это устройство не может быть запрограммировано в полном смысле этого слова; например, с его помощью нельзя обеспечить постоянную скорость движения робота. Чтобы изменить алгоритм управления, надо переделать саму схему.

Чтобы схема управления была *программируемой*, она должна уметь выполнять некоторое количество заранее определенных операций, причем подразумевается возможность с помощью входных сигналов изменять последовательность их

выполнения, подобно тому как с помощью *программы* мы задаем последовательность команд, выполняемых микропроцессором персонального компьютера.

Вот почему использование компьютера для управления нашим роботом - идеальное решение проблемы регулирования скорости движения. Например, можно измерять ток фотодиодов с помощью *аналого-цифрового преобразователя* (АЦП), который будет преобразовывать напряжение на резисторе, подключенном к фотодиоду, в цифровой код, понятный компьютеру. Последовательно с микропереключателем следует включить дополнительный резистор, чтобы сформировать еще один входной сигнал для компьютера. Когда переключатель замкнут, напряжение на нем равно нулю (сигнал низкого уровня), а при его размыкании благодаря резистору, подключенному к положительной шине источника питания, будет сформирован сигнал высокого уровня. Измененная схема показана на рис. 1.2.

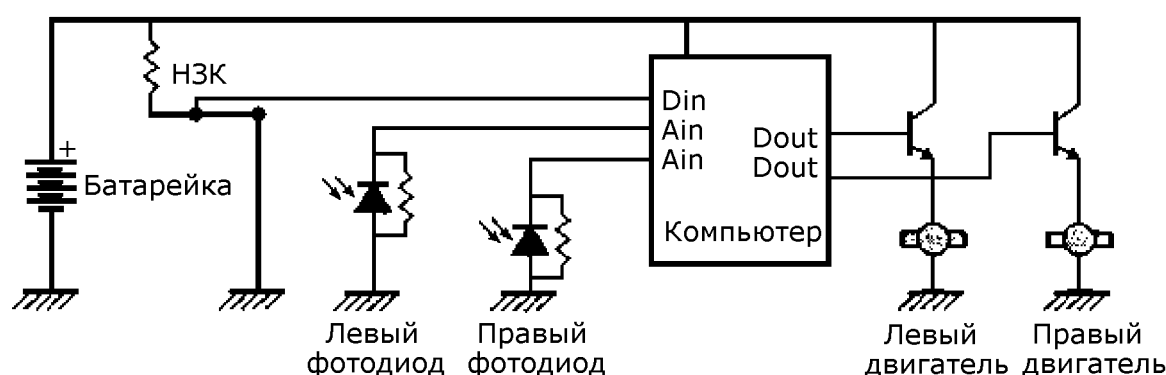


Рис. 7.2. Компьютер в качестве управляющего устройства

Приведем код программы, которая реализует тот же алгоритм управления, что и раньше, но при этом поддерживает максимальную скорость движения:

```
void main(void) {
    while (Whisker != Collision) {      // До столкновения.
        if (LeftLightSensor >= MaximumSpeed)
            RightMotor = MaximumSpeed;
        else
            // Уменьшить скорость вращения двигателя.
            RightMotor = LeftLightSensor;

        if (RightLightSensor >= MaximumSpeed)
            LeftMotor = MaximumSpeed;
        else
            // Уменьшить скорость вращения двигателя.
            LeftMotor = RightLightSensor;
    }
    // Конец цикла.
    RightMotor = 0;
    LeftMotor = 0;      // Выключить оба двигателя.
}
```

Если требуется изменить алгоритм управления, приведенный код нетрудно модифицировать.

Утверждение, что компьютер может управлять роботом, не следует понимать слишком буквально. Для этой задачи лучше подходят специальные устройства, которые будут рассмотрены ниже.

1.2. ОСНОВНЫЕ КОМПОНЕНТЫ УСТРОЙСТВ УПРАВЛЕНИЯ

Для обеспечения работы современных микропроцессоров или микроконтроллеров необходимо небольшое число дополнительных элементов. К счастью, их не слишком много, поэтому при разработке контроллера можно не беспокоиться о том, что из-за них возрастут габаритные размеры всего устройства или потребляемая от источника питания мощность. Однако следует принимать в расчет несколько важных обстоятельств.

Как уже говорилось, необходимо с самого начала процесса проектирования сформулировать и учитывать в дальнейшей работе требования к контроллеру, иначе все устройство окажется недолговечным, или микроконтроллер не сможет обеспечить заданный алгоритм управления, или устройство управления просто не поместится внутри робота.

Источником питания для всех подсистем подвижного робота обычно служит встроенная батарейка или аккумулятор. Иногда используются фотогальванические элементы, но тогда энергия должна запасаться в конденсаторах или аккумуляторах, прежде чем подаваться на электронную схему. В этом случае мощности источника питания часто хватает всего на несколько секунд, за которые робот успевает совершить несколько судорожных движений. Такое поведение от него часто и ожидается. Некоторые роботы используют для движения бензиновый двигатель и имеют небольшой генератор для вырабатывания электрической энергии, необходимой для работы электронных подсистем. И солнечные батареи, и бензиновый двигатель применяются реже, чем обычная батарейка или аккумулятор.

При выборе элемента питания необходимо учитывать, какую мощность потребляют двигатель робота и другие периферийные устройства. Часто используют два источника питания: один для двигателя, а другой для электронных схем. Это позволяет снизить помехи по цепям питания, которые возникают в момент включения/выключения двигателей. Тем не менее я предпочитаю использовать для питания один комплект батарей. Если конструкция небольшая, таким образом можно снизить стоимость и размер робота. На пути уменьшения стоимости и веса конструкции разработчика ждет приятный сюрприз. Чем меньше масса робота, тем меньшей может быть мощность двигателя, а вместе с тем его стоимость, размер и мощность, потребляемая от источника питания. Следовательно, для него может использоваться менее мощная и более легкая батарейка. А это еще более удешевит всю конструкцию и позволит применить еще менее мощный двигатель. И так далее.

Описанный эффект позволяет создать недорогого робота с минимально возможными размерами и массой. Только не слишком увлекайтесь, иначе робот не сможет выполнять заданные функции из-за того, что окажется слишком легким и слабосильным.

Разработчик может использовать щелочные, никель-кадмиевые или кислотные аккумуляторы. При выборе элемента питания следует учитывать:

- размеры и вес робота;
- требуемое напряжение питания;

- время «жизни» робота;
- скорость его движения;
- стоимость;
- время зарядки аккумулятора.

Элементы различного типа вырабатывают разные напряжения: угольные элементы - около 1,5 В, щелочные - 1,7 В и более, никель-кадмиевые аккумуляторы - 1,2 В. Как показано на рис. 1.3, напряжение аккумуляторов по мере их разряда почти не уменьшается, в то время как у одноразовых батареек оно линейно убывает со временем.

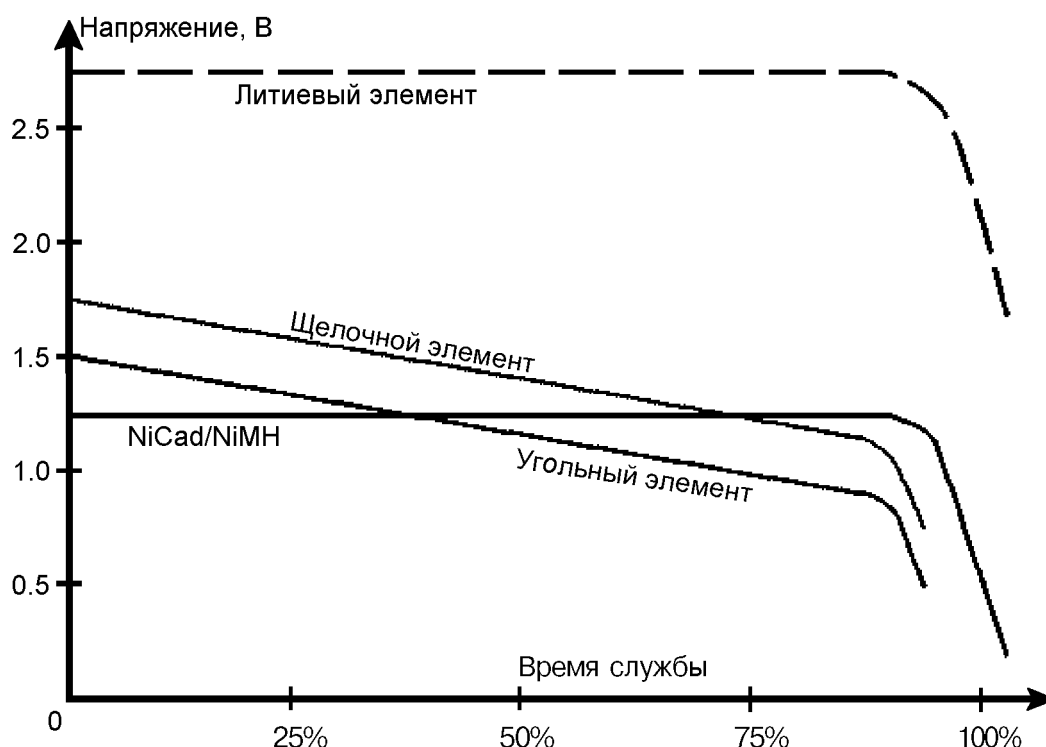


Рис. 1.3. Изменение напряжения по мере разряда батареи

Необходимо помнить, что важный параметр любого источника питания - его внутреннее сопротивление. Чем оно больше, тем меньше энергии достается двигателю, потому что значительная ее часть расходуется внутри источника. При этом из-за нагрева ухудшаются его характеристики (напряжение и емкость и, соответственно срок службы). Кроме того, при большом внутреннем сопротивлении источника питания увеличиваются броски напряжения при включении/выключении двигателей, поэтому необходимо применять более сложные схемы фильтрации напряжения.

Бытует мнение, что выгоднее покупать дешевые угольные батарейки, чем более дорогие щелочные или никель-кадмиевые, потому что и первые, и последние имеют практически одинаковую емкость (параметр, измеряемый в ампер-часах). Это так, но не стоит злоупотреблять использованием в устройствах дешевых элементов: они обычно имеют повышенное внутреннее сопротивление. Замечу, что этот параметр всегда указывается в справочных данных на любые элементы питания.

Проблема, связанная с внутренним сопротивлением, проиллюстрирована на рис. 1.4. Комплект батарей является источником питания для контроллера и двигателя. Внутреннее сопротивление каждого элемента можно учесть с помощью резистора, включенного последовательно с источником. Его сопротивление равно сумме внутренних сопротивлений всех элементов питания, включенных последовательно друг с другом. Согласно закону Кирхгофа, сумма напряжений, падающих на элементы замкнутой цепи, равна электродвижущей силе (проще говоря, напряжению) источника питания. Поэтому напряжение на нагрузке (контроллере и двигателе) тем меньше, чем больше напряжение, падающее на резисторе. Но последнее при фиксированном токе нагрузки возрастает с увеличением сопротивления резистора.

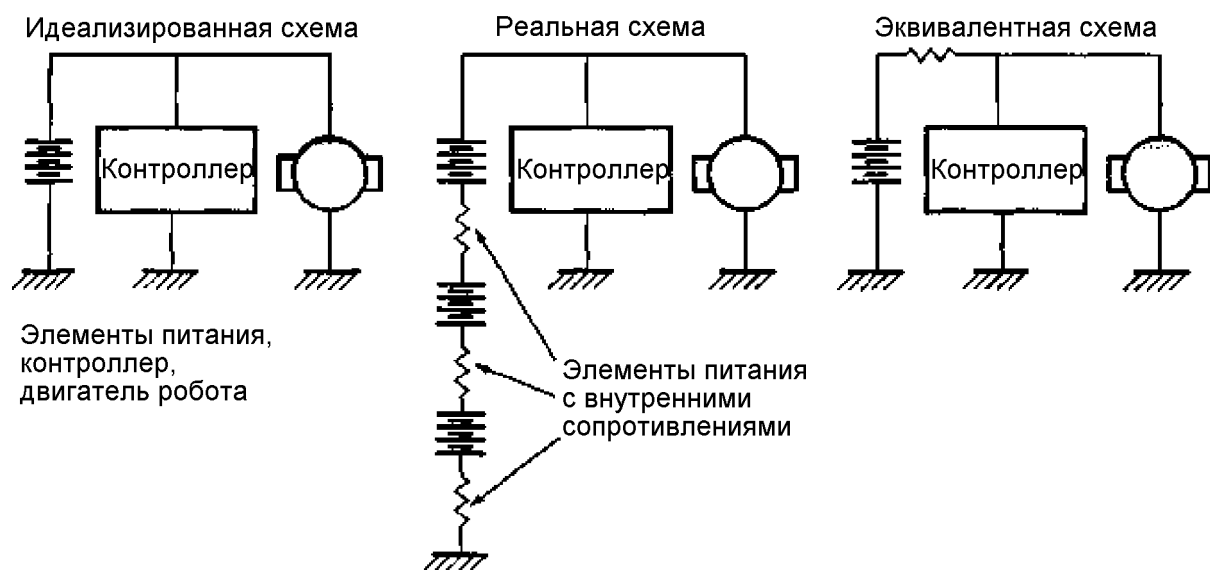


Рис. 1.4. Подключение источника питания

Самое низкое внутреннее сопротивление у щелочных батареек и никель-кадмиевых аккумуляторов, специально предназначенных для работы при высоких токах нагрузки. Например, 9,6-вольтовые никель-кадмиевые аккумуляторы, выпускаемые для дистанционно управляемых моделей, имеют очень низкое внутреннее сопротивление и могут быть приобретены вместе с зарядным устройством по умеренной цене.

После выбора элементов питания можно приступить к проектированию регулятора и стабилизатора напряжения. Обычно для этого используют микросхему 78(L)05. Если имеется только один комплект батарей для питания двигателя и контроллера, то придется использовать преобразователь напряжения, поскольку эти два устройства требуют существенно разных напряжений питания. В некоторых случаях вообще можно отказаться от регулятора напряжения, взяв несколько элементов питания, включенных последовательно, чтобы обеспечить требуемое напряжение от 2,5 до 6,0 В.

В любом случае необходимо позаботиться о фильтрации напряжения питания и подключить между положительной и отрицательной шинами источника два конденсатора: один емкостью по крайней мере 10 мкФ (он будет обеспечивать

фильтрацию низкочастотных помех) и второй емкостью около 1 нФ или меньше (для фильтрации высокочастотных помех). В некоторых случаях не мешает катушка индуктивности, включенная последовательно с источником питания.

Для сброса микроконтроллера (выработки сигнала RESET) обычно используются простейшие цепи. Часто достаточно подключить внешний резистор между входом сброса и положительной шиной источника питания (V_{CC}) или массой (GND). Некоторые микроконтроллеры сами сбрасываются в исходное состояние, если напряжение питания опускается ниже установленного предела. Другие чипы имеют встроенную схему формирования сигнала сброса, срабатывающую, когда напряжение питания достигает заданного уровня. Даже если используемый микроконтроллер и не поддерживает таких возможностей, в любом случае несложно построить схему на нескольких транзисторах, которая будет вырабатывать сигнал сброса.

У многих микроконтроллеров вход сброса используется также и для **программирования**. Для включения режима программирования обычно необходимо подать на этот вход повышенное напряжение. Поэтому схема формирования сигнала сброса должна проектироваться так, чтобы формируемое ею напряжение никогда не превысило уровней питания V_{CC} или V_{DD} .

Следует учитывать и повышенные механические нагрузки, которые должна выдерживать схема управления роботом по сравнению со схемами, находящими применение в обычных компьютерах. Поэтому следует обратить внимание на особенности построения генератора тактовых импульсов.

Обычно для тактирования используется схема, показанная на рис. 1.5 (**релаксационный генератор**). Здесь для задержки сигнала между инвертором и схемой управления предназначена RC-цепочка. Похожая схема часто встроена в микроконтроллер - тогда не требуется никаких дополнительных внешних элементов (в некоторых случаях используется внешний резистор, являющийся частью RC-цепи и задающий частоту генерации). Преимущества описанного тактового генератора - дешевизна и работоспособность при колебаниях напряжения питания.

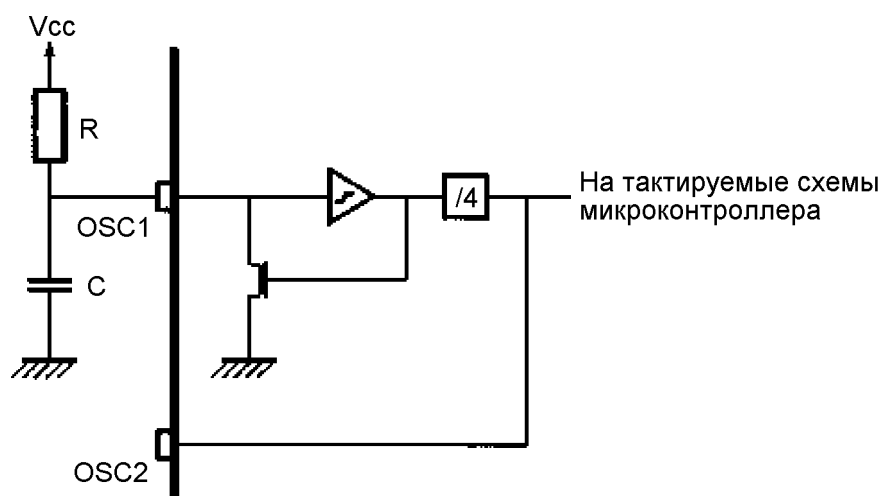


Рис. 1.5. RC-генератор тактовых импульсов

К недостаткам следует отнести частотную нестабильность. К тому же для релаксационной схемы частоту генерации нельзя прогнозировать точнее, чем $\pm 10\%$. Обычно большой точности и не требуется, но если необходимо обеспечить асинхронный обмен информацией между двумя электронными схемами, то именно точность тактового генератора может стать решающим фактором.

Иногда в схеме тактовых генераторов используют кварцевые резонаторы, но они не так устойчивы к механическим воздействиям и дороже, чем керамические резонаторы. Те и другие подключаются к контроллеру сходным образом (рис. 1.6).

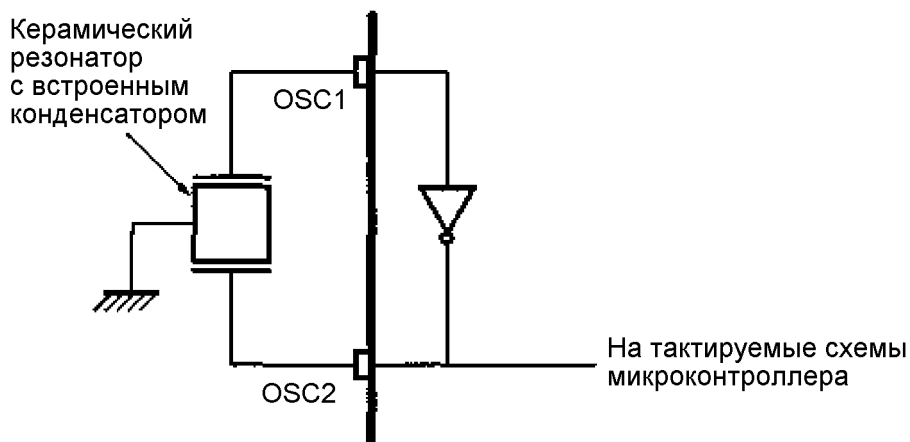


Рис. 1.6. Подключение керамического резонатора

Многие кварцевые резонаторы имеют встроенную емкость, которая упрощает схему.

При подключении керамического резонатора к микроконтроллеру следует использовать соединения минимально возможной длины, чтобы избежать искажений сигнала при передаче, а также наводок от соседних цепей.

Следует учитывать, что кварцевые резонаторы обеспечивают наибольшую точность по частоте генерации - до 0,01%, в то время как керамические резонаторы - от 0,5 до 2,0%. Обычно точности последних более чем достаточно для реализации межпроцессорных коммуникаций, но в некоторых электронных схемах требуется использование кварца.

Наконец, последний компонент, который предстоит рассмотреть, - выключатель питания. О нем часто (и незаслуженно!) забывают: то ли разработчик думает, что пользователь будет вынимать батарейки из робота каждый раз, когда захочет его выключить, то ли считает, что управляющая программа настолько хороша, что сама выключает питание двигателей, пока робот не используется. Существует хороший способ выключения питания робота - я называю его «большая красная кнопка» (БКК). Без БКК пользователь может забыть, что робот остался включенным после того, как он натолкнется на какое-нибудь препятствие и из-за этого перестанет двигаться. Не забывайте о выключателе. Иначе ваш робот может вдруг начать двигаться в самый неподходящий момент, из-за воздействия какой-нибудь случайной помехи — и слетит с полки.

1.3. ПАМЯТЬ И УСТРОЙСТВА ВВОДА-ВЫВОДА

Разрабатывая контроллер для робота, вы должны четко осознавать все требования, которые предъявляет контроллер к системе памяти, а также понимать, каким способом программа загружается в эту память. Кто-то может подумать, что контроллер программируется раз и навсегда, но это не так. Во время разработки и отладки управляющего алгоритма перепрограммирование приходится производить десятки раз; кроме того, оно необходимо для модернизации устройства и/или алгоритма управления.

Можно выделить два типа запоминающих устройств. Память первого типа - *энергонезависимая*, называемая также *постоянной* (ПЗУ - постоянное запоминающее устройство, ROM - Read Only Memory). В ней хранится программа, управляющая роботом, она не стирается при выключении питания. Если вы знакомы с персональными компьютерами, вас может удивить тот факт, что программа хранится в ПЗУ. Однако не следует забывать, что у робота нет дисководов или жесткого диска для хранения программ и данных. Как только робот включается, он должен немедленно начать выполнять свою программу.

В отсутствие дисков есть своя положительная черта: нет необходимости в дисковой операционной системе и интерфейсе прикладных программ (API - Application Programming Interface). Впрочем, тут есть и подвох: многие низкоуровневые функции, выполнение которых обычно обеспечивает операционная система, приходится программировать самостоятельно. Для универсальных ЭВМ это большой недостаток, но автономные роботы с нестандартным аппаратным интерфейсом - другое дело: ведь широкий набор функций API в большинстве случаев не используется в полной мере.

Память для хранения программы может быть встроенной в микроконтроллер или внешней. В устройствах, описанных в этой книге, будет использоваться первый вариант.

Перечислим основные виды ПЗУ:

- *масочное ПЗУ*. Память этого вида не может быть перепрограммирована, поэтому используется только при серийном производстве;
- *программируемое ПЗУ (PROM — Programmable ROM)* может быть запрограммировано лишь один раз путем выжигания перемычек внутри кристалла, поэтому такая память в настоящее время уже почти не используется;
- *память с произвольной выборкой (RAM - Random Access Memory, она описана ниже)* является энергозависимой, так как теряет записанную в нее информацию при выключении питания, однако современные КМОП микросхемы потребляют настолько малый ток (микроамперы), что в комплекте с батареей микросхема RAM может в течение долгого времени выполнять роль ПЗУ;
- в большинстве контроллеров используются перепрограммируемые микросхемы памяти (EPROM - Erasable PROM). Информация, записанная в них, может быть удалена с помощью ультрафиолетовых лучей (такие микросхемы

имеют на корпусе кварцевое окошко¹ - см. рис. 1.7). Некоторые микросхемы этого типа не имеют окна для стирания и называются *однократно программируемыми* (OTP - One-Time Programmable);

- современные контроллеры используют так называемую Flash-память (EEPROM - Electrically Erasable ROM), в которой старая информация стирается с помощью электрического напряжения без использования ультрафиолетовых лучей. На рис. 1.8 показано, как устроена ячейка Flash-памяти. Плавающий затвор полевого транзистора может заряжаться или разряжаться с помощью управляющего электрода. Когда он заряжен, в кремниевой подложке образуется зона проводимости, которая позволяет электрическому току протекать между выводами стока и истока.

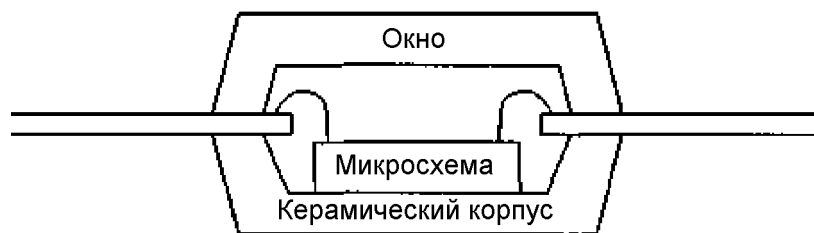


Рис. 1.7. Керамический корпус EPROM с окном для стирания информации

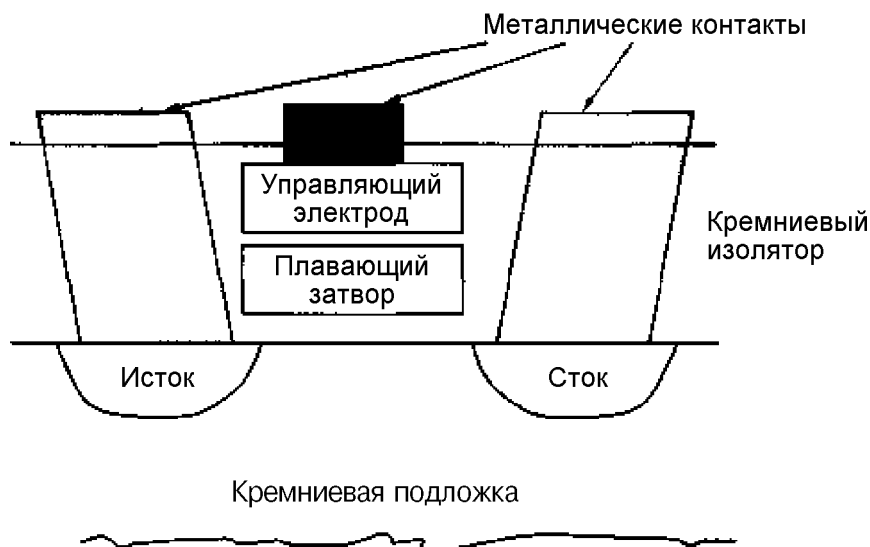


Рис. 1.8. Ячейка Flash-памяти

Существует небольшая разница между микросхемами EEPROM и Flash-памятью. В первом случае можно адресовать любую ячейку, а во втором - стирание происходит сразу для всего блока, образованного группой соседних ячеек памяти. Благодаря тому что цепи стирания не дублируются для каждой ячейки, а являются общими для всего блока, Flash-память оказывается несколько дешевле, чем EEPROM.

¹ Обычное стекло задерживает ультрафиолетовое излучение. - Прим. перев.

Для того чтобы запрограммировать микросхему EPROM, Flash или EEPROM, необходимо на ее входы подать программируемые данные, а также (в определенной последовательности) специальные управляющие сигналы согласно *алгоритму программирования*. Для различных микросхем этот алгоритм обычно разный. Данные могут записываться в микросхему памяти либо *последовательно* (бит за битом), либо *параллельно* (несколько разрядов одновременно). Для программирования каждой ячейки может требоваться время от сотен наносекунд до десятков миллисекунд. Вся микросхема программируется, таким образом, за несколько минут или даже несколько секунд.

Для записи данных в микросхемы EPROM, Flash или EEPROM выпускается множество программаторов; кроме них существует немало радилюбительских конструкций. На сайте издательства www.dmk.ru, а также на моем сайте www.myke.com можно найти драйвер и схему разработанного мною простого программатора Flash-памяти, которая используется в микроконтроллерах PIC. Этот программатор похож на десятки других устройств, которые обычно описываются на радилюбительских страничках в сети Internet. Коммерческие разработки отличаются от любительских более широким набором программируемых микросхем, а также тем, что могут работать в автономном режиме (без использования компьютера или других устройств). При выборе программатора необходимо убедиться, что он распознает формат файлов, которые генерирует используемая вами среда разработки программ (IDE).

Наконец, память второго типа - это уже упоминавшаяся *память с произвольной выборкой* (RAM, называемая также *оперативной*, ОЗУ). Она используется для хранения данных, которыми манипулирует программа. И чтение, и запись данных в этом случае происходят *одинаково* - никакого программатора не требуется. Однако память RAM является *энергозависимой*, поэтому при выключении питания даже на короткое время вся записанная в нее информация теряется.

Обычно выделяют два вида микросхем RAM - *динамическую* (DRAM) и *статическую* (SRAM). Для хранения данных в большинстве персональных компьютеров используется DRAM; в последнее время это обычно так называемая *синхронная* DRAM (SDRAM), позволяющая увеличить быстродействие компьютера.

Из-за того что ячейки (S)DRAM быстро разряжаются, их приходится периодически *регенерировать* (обновлять записанную в них информацию). Зачастую на это уходит до 5% всего времени работы компьютера. Тем не менее память (S)DRAM очень широко применяется благодаря своей низкой удельной стоимости (в расчете на каждый бит хранимой информации).

Что касается микроконтроллеров, в них чаще применяют *статическую память* (SRAM). Каждая ячейка SRAM состоит из четырех-шести транзисторов, как показано на рис. 1.9. Преимущество SRAM состоит в том, что в режиме хранения данных она потребляет очень маленький ток и не требует никакой регенерации. Но по своим размерам каждая ячейка статической памяти в несколько раз больше динамической.

Память SRAM часто содержится внутри микроконтроллера, но обычно ее емкость невелика - иногда всего несколько десятков байт. Кроме того, вы не должны забывать, что сама программа хранится в другом месте - в ПЗУ. Это необычно

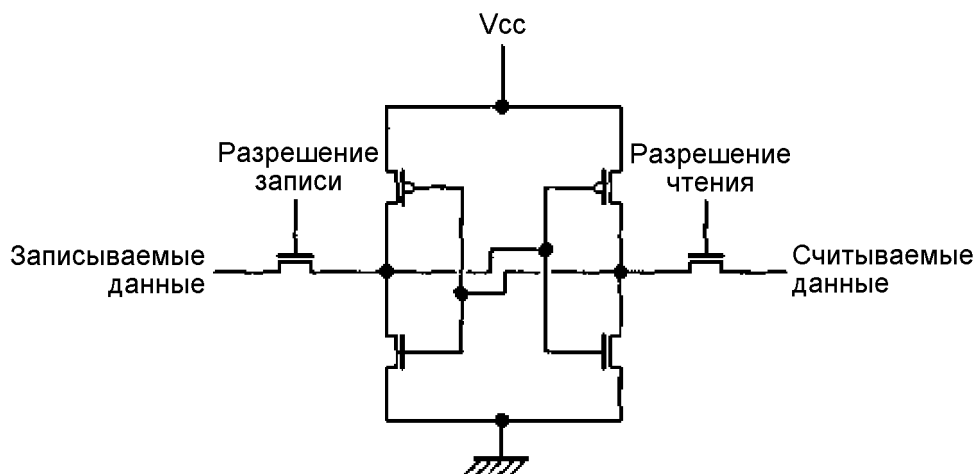


Рис. 1.9. Ячейка статической памяти

для программиста, привыкшего работать с персональными компьютерами, в которых память программ и память данных обычно не разделяются. Чтобы не использовать слишком много памяти RAM, советую, во-первых, снизить до минимума количество и длину текстовых строк в вашей программе, а во-вторых, стараться хранить их в ПЗУ вместе с программой. Как бы то ни было, несмотря на небольшой объем RAM в микроконтроллерах, вы без труда сможете разрабатывать довольно сложные программы управления роботом.

1.4. ПРЕРЫВАНИЯ

Я нередко поражаюсь тому, как много дипломированных инженеров и компьютерных специалистов обычно боятся использовать прерывания в своих разработках. Причину, вероятно, следует искать в их первых, не всегда удачных попытках программирования прерываний, а может, и в невежестве преподавателей. По моему мнению, прерывание - наиболее естественный способ реализации поведения робота на верхнем (биологическом) уровне.

Использование прерываний избавляет разработчика от программирования многочисленных циклов опроса датчиков и уменьшает вероятность того, что входные данные окажутся потерянными. Кроме того, благодаря прерываниям можно проще и точнее обеспечить точную синхронизацию событий, происходящих в системе. Поэтому во всех примерах, приведенных в данной книге, я использовал по крайней мере одно прерывание - по таймеру.

На рис. 1.10 показано, как происходит обработка прерывания. Можно выделить шесть этапов этого процесса:

1. Выполняется основная программа, прерывания разрешены.
2. Приходит запрос на прерывание (interrupt request) от аппаратуры.
3. Выполнение основной программы приостанавливается, содержимое регистров процессора сохраняется в специальном месте. Выбирается так называемый *вектор прерывания* - адрес программы, которая должна его обрабатывать.
4. Запускается программа обработки прерывания (interrupt handler). В самом начале ее выполнения обычно запрещаются все другие прерывания.

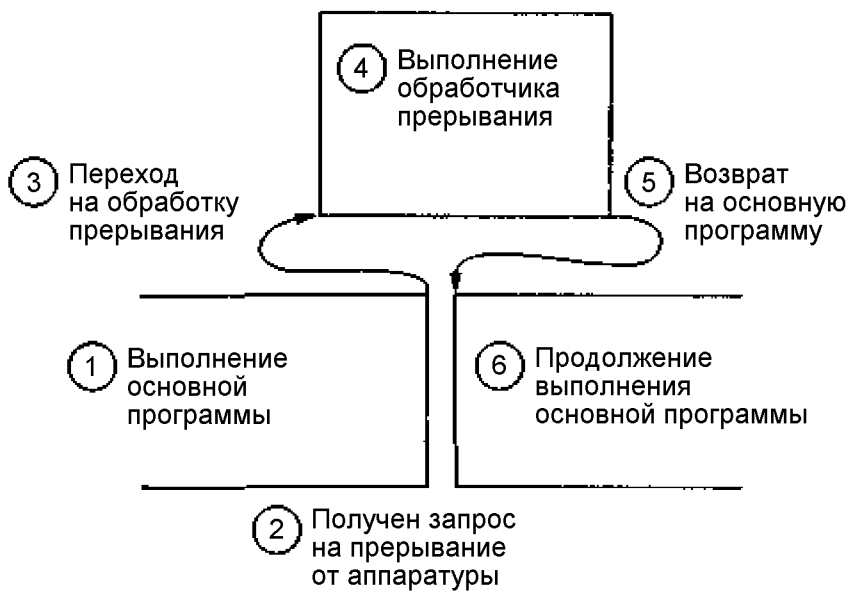


Рис. 1.10. Обработка прерывания

- 5. Обработка прерывания завершается, значения регистров восстанавливаются, разрешается обработка других прерываний.
- 6. Выполнение передается основной программе.

Обработкой прерываний руководит специальное устройство внутри микроконтроллера - *контроллер прерываний* (interrupt controller). На рис. 1.11 показаны временные диаграммы работы контроллера прерываний в микросхемах PICmicro. Здесь приведен пример передачи управления на вектор 0x0004.

Следует помнить, что при обработке прерывания необходимо сбросить сигнал запроса, иначе после его обработки запрос будет продолжать действовать и вместо

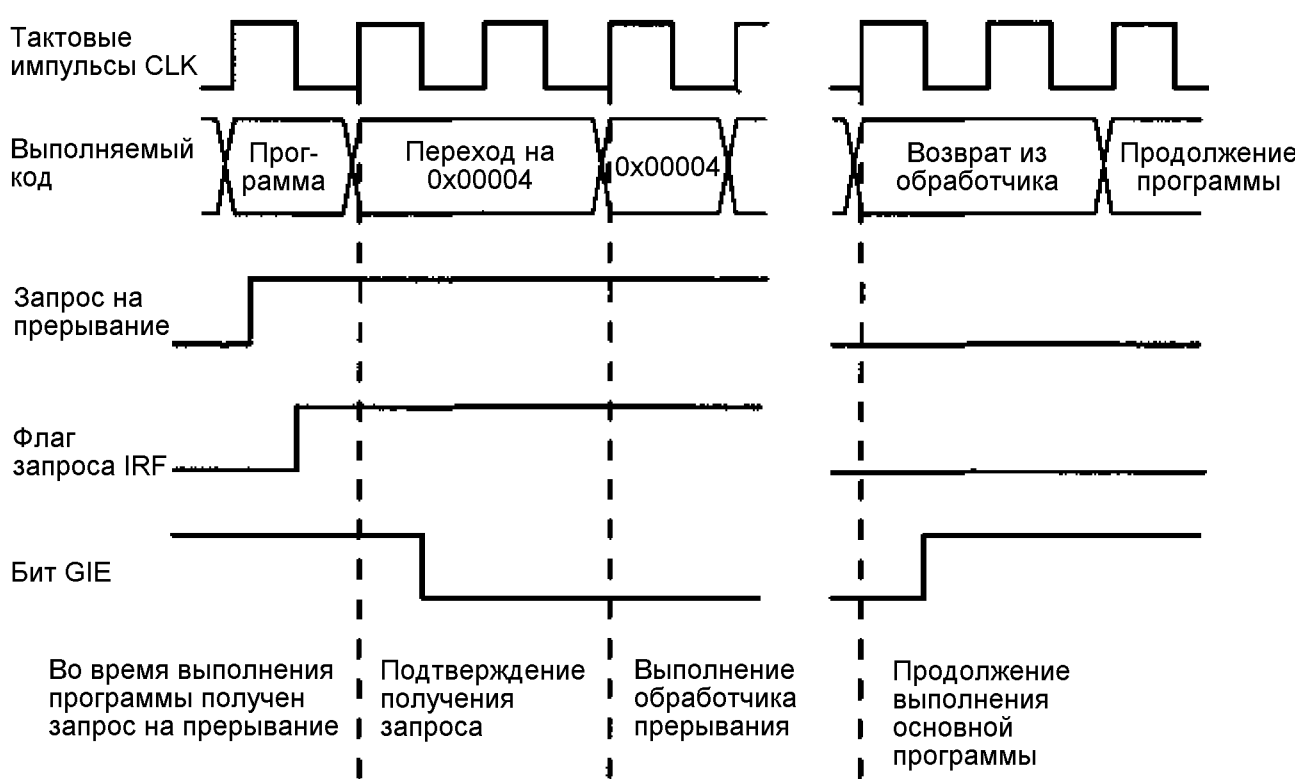


Рис. 1.11. Обработка прерывания в микроконтроллерах PICmicro

возвращения к выполнению основной программы мы рискуем вернуться обратно к обработке этого запроса (см. рис. 1.11). Если сигнал запроса IRF (Interrupt Request Flag) вовремя не сбросить в 0 до возвращения к основной программе, то контролер прерываний «увидит», что флаг запроса IRF все еще продолжает действовать, и опять передаст управление вектору 0x0004.

Теперь обсудим, как контроллер различает, откуда пришел запрос и какая программа должна его обрабатывать. В микроконтроллерах PICmicro используется только один вектор прерываний, поэтому достаточно просто определить источник запроса. В больших микропроцессорах (например, в процессоре i80x86, применяемом в IBM PC), существует множество векторов прерываний, запросы на которые могут поступить одновременно.

В микроконтроллерах PICmicro нельзя начать обработку какого-либо запроса на прерывание, если еще не завершилась обработка предыдущего. В персональных компьютерах такое возможно.

Программа обработки прерывания должна быть максимально простой и выполняться как можно быстрее, ведь при обработке запроса мы не знаем, как скоро появится следующий. Чтобы уменьшить вероятность того, что какому-либо запросу придется слишком долго ждать своей обработки, намного проще обеспечить быструю обработку запросов, чем усложнять контроллер прерываний.

В начале этого раздела упоминалось о том, с какой опаской многие разработчики используют прерывания в своих программах, а также подчеркивалось, как важно понимать организацию обработки прерываний при разработке автоматических устройств. Надеюсь, все, что здесь сказано, не отпугнет вас от использования прерываний в ваших конструкциях; в конце концов, программирование прерываний на языке высокого уровня - не такая уж сложная задача.

Чтобы не быть голословным, приведу простой пример программы обработки прерывания, которая будет использоваться во многих проектах, описанных в этой книге:

```

// Процедура обработки прерываний от таймера TMRO.
void interrupt tmr0_int(void) {
    if (TOIF) {
        TOIF = 0;
        RTC++;
    }
}
```

Модификатор `interrupt` в заголовке показывает, что далее следует процедура обработки прерывания. Она будет запускаться на выполнение каждый раз, когда контроллер получит запрос на обработку прерывания. Затем проверяется источник запроса (условный оператор `if`). Если прерывание запрошено таймером TMRO, то необходимо сбросить сигнал запроса (`TOIF = 0`) и увеличить на единицу счетчик «тиков» таймера.

Разумеется, приведенный пример слегка упрощен, но при добавлении к написанному коду новых функций общая структура обработчика прерываний не изменяется.

В главе 4 будет показано, как используются прерывания для реализации функций механического и электронного уровней, как при этом можно избежать применения

низкоуровневого ассемблера, а также - и это самое важное - как использование прерываний позволяет обеспечить простое и эффективное управление роботом.

1.5. ПЕРИФЕРИЙНЫЕ УСТРОЙСТВА МИКРОКОНТРОЛЛЕРОВ

Во всех микроконтроллерах есть некоторое количество встроенных устройств ввода-вывода (УВВ), к которым можно обращаться при разработке программ. У простейших микроконтроллеров имеется только порт цифрового ввода-вывода и таймер; более сложные чипы поддерживают возможность ввода и/или вывода аналоговой информации.

Цифровые порты используются чаще всего. На рис. 1.12 приведен пример цифрового УВВ, имеющегося в микроконтроллерах 8051 фирмы Intel. Здесь вывод микроконтроллера работает как вход, если нижний транзистор закрыт. Верхний транзистор приоткрыт и используется для «подтягивания» потенциала входной линии до положительного напряжения источника питания, поэтому на отключенном от внешних устройств входе действует сигнал высокого уровня. Если внешнее устройство подает на вход микроконтроллера сигнал низкого уровня, то при этом вход соединяется с массой при помощи низкого внутреннего сопротивления источника входного сигнала, который поэтому легко «побеждает» не до конца открытый верхний транзистор и не мешает установлению на входе напряжения низкого уровня.

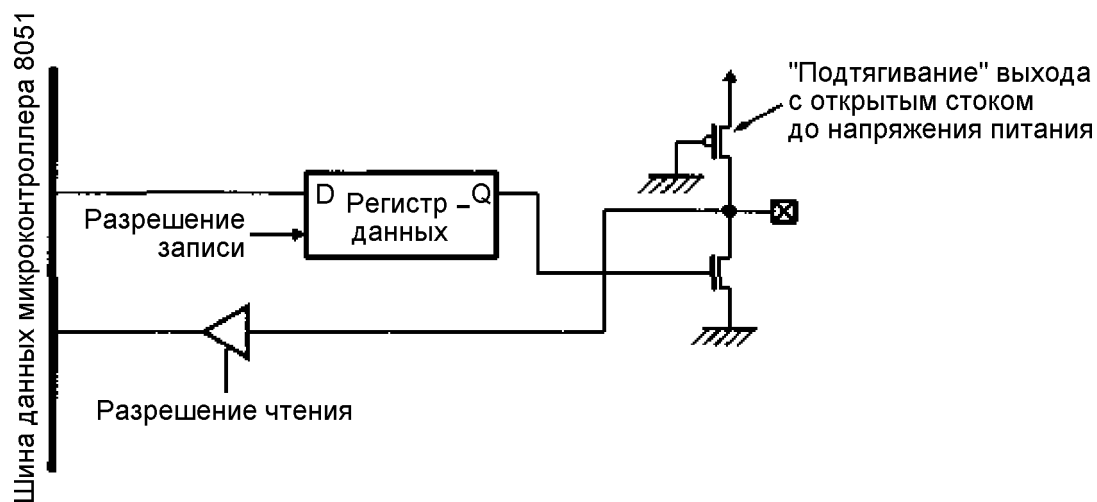


Рис. 1.12. Параллельный порт ввода-вывода в микроконтроллерах 8051

Когда вывод должен работать в качестве выходного и необходимо установить сигнал низкого уровня, открывается нижний транзистор. Он включен по схеме с *открытым стоком*, поэтому не может самостоятельно обеспечить на выходе напряжение высокого уровня. Для этого используется верхний транзистор.

Во многих микроконтроллерах, в том числе и в PICmicro, порты ввода-вывода снабжены специальным устройством, имеющим три логических состояния. Его схема приведена на рис. 1.13. Благодаря тому что выходной сигнал снимается с буферного элемента (изображен в виде треугольника), а не с выхода регистра,

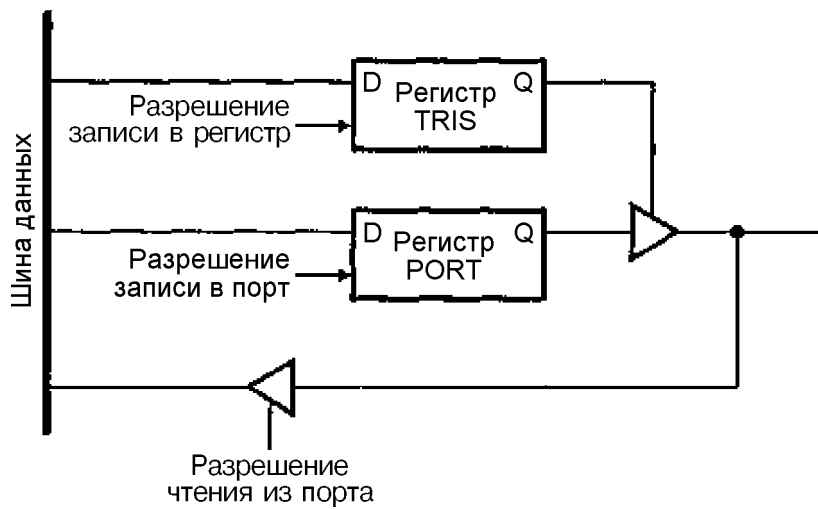


Рис. 1.13. Структурная схема порта ввода-вывода в микроконтроллерах PICmicro

микроконтроллер при чтении из порта, в который только что записал данные, всегда может «узнать», не изменило ли внешнее устройство уровень напряжения на данном выводе.

Как уже говорилось, некоторые микроконтроллеры могут вводить или выводить аналоговые сигналы. Эта возможность чрезвычайно полезна, например, если робот должен контролировать уровень громкости звука или яркость источника света.

Однако прежде, чем вы займетесь проектированием робота, управляемого голосом, позвольте заметить, что задача эта не столь проста, как может показаться на первый взгляд, и многочисленные команды специалистов потратили не один миллион долларов, пытаясь ее решить. Впрочем, построить робота, реагирующего на простые звуки (вроде хлопка в ладоши), не так уж сложно.

Иногда порты ввода-вывода используют для того, чтобы расширить возможности микроконтроллера, подключив к его внутренней шине внешние блоки ROM и/или RAM.

Кроме того, один микроконтроллер может служить внешним устройством для другого. В этом случае, например, один контроллер выполняет функции механического и электронного уровней, а второй обеспечивает управление на биологическом уровне. Сопряжение контроллеров облегчается тем, что многие модели имеют встроенные интерфейсы ввода-вывода, например интерфейс PC (Inter-Computer Communications).

Большое значение для автоматических устройств имеет наличие внутренних таймеров. С их помощью обеспечивается программирование сложных протоколов обмена информацией между различными устройствами, а также выполнение заданной последовательности действий с необходимыми задержками во времени. Обычно счетчик таймера постоянно работает, периодически увеличивая свое значение, пока не достигнет максимально возможного; затем он снова сбрасывается в нулевое состояние. В этот момент может быть выработано прерывание или сгенерирован специальный выходной сигнал. Для расширения функциональных возможностей многих таймеров предусмотрены дополнительные делители частоты — так называемые *прескалеры* и *постскалеры* (prescaler и postscaler). Прескалер

(предварительный делитель) выполняет деление входной частоты сигнала, поступающего на таймер, а постскалер делает то же самое для выходного сигнала таймера, вырабатываемого в момент его переполнения. Оба делителя, использованные вместе или по отдельности, позволяют расширить рабочий интервал таймера.

В большинстве микроконтроллеров имеются встроенные средства, обеспечивающие их сопряжение с компьютером. Чаще всего в этих целях используются NRZ-кодирование сигнала (NonReturn to Zero - кодирование без возврата к нулю) и интерфейс, основанный на стандартных протоколах RS-232 или RS-485.

Кроме этого, некоторые микроконтроллеры поддерживают интерфейсы:

- SPI (Serial Peripheral Interface) - последовательный периферийный интерфейс;
- Microwire;
- PC;
- CAN (Common Automotive Network);
- USB (Universal Serial Bus) — универсальная последовательная шина.

Приложив немного старания, вы всегда найдете микроконтроллер, который имеет интерфейс, необходимый для вашего устройства. Например, порывшись в сети Internet, я без труда отыскал микроконтроллеры, имеющие:

- контроллер управления шаговым двигателем;
- контроллер управления двигателем постоянного тока;
- контроллер управления однофазным двигателем переменного тока;
- контроллер управления трехфазным двигателем переменного тока;
- выходной видеосигнал в стандарте NTSC (National Television Standards Committee);
- аудиовход и выход;
- контроллер LCD (Liquid Crystal Display - жидкокристаллический дисплей);
- интерфейс с клавиатурой и мышью персонального компьютера.

Хотя некоторые из этих возможностей и не так уж важны для мобильного автоматического устройства, однако бывают ситуации, в которых без них не обойтись.

1.6. ПОДКЛЮЧЕНИЕ УСТРОЙСТВА УПРАВЛЕНИЯ К РОБОТУ

После того как разработчик выбрал микроконтроллер и решил, какие операции он будет выполнять, можно, наконец, заняться подключением микроконтроллера к остальным подсистемам робота. Это та граница, за которой кончается идеальный мир программирования и начинается реальная действительность с ее бросками питания, разрядившимися батареями, вибрациями и электромагнитными наводками от других электрических устройств. Разработчик должен учитывать все эти помехи.

К счастью, современные микроконтроллеры весьма устойчивы ко всем мешающим факторам, что облегчает их использование в автоматических устройствах. Пожалуй, самая трудная задача - сопряжение микроконтроллера со всеми периферийными устройствами, входящими в состав робота.

Несколько советов, приведенных ниже, помогут вам обеспечить надежное функционирование микроконтроллера и упростят разработку электрического интерфейса между контроллером и периферийными устройствами. Практически во всех случаях, когда для разрешения проблем возникает необходимость в пошаговой отладке управляющей программы, разработчик должен четко оделить функционирование самого контроллера от интерфейсной части, обеспечивающей его сопряжение с периферийными устройствами.

В первую очередь необходимо удостовериться в надежном функционировании устройств, обеспечивающих работу микроконтроллера (источника питания, тактового генератора, схемы формирования сигнала сброса). Если в вашем распоряжении имеется осциллограф, следует удостовериться, что все сигналы имеют правильную форму и не слишком искажены из-за шумов и переходных процессов в цепях. Нелишне соблюдать следующие правила:

- при разработке схемы контроллера старайтесь, чтобы длина проводников, по которым подаются сигналы тактирования, была минимально возможной (кроме всего прочего, это позволит снизить наводки на другие части схемы);
- убедитесь, что общий провод источника питания надежно подключен к схеме и обе шины питания не образуют замкнутых контуров, индуктивность которых может стать источником помех из-за электромагнитных наводок;
- в идеале источник питания контроллера должен быть изолирован от двигателей робота и располагаться как можно дальше от них;
- если различные блоки работают от разных источников питания, то следует убедиться, что общий провод («земля») каждого источника надежно соединен с другими; замечу, что обычно выгоднее использовать несколько источников, расположенных поблизости к тем блокам, для питания которых они предназначены, чем предусматривать в своей разработке один большой блок питания (кроме всего прочего, такой подход позволяет решить многие проблемы с прокладкой шин, особенно между частями конструкции, которые должны двигаться относительно друг друга);
- для снижения потребляемой мощности и уменьшения переходных процессов полезно снижать тактовую частоту контроллера до приемлемых значений. Как правило, такая экономия не слишком существенна, но можно назвать случаи, когда контроллер потребляет значительную мощность, сравнимую с мощностью остальных его подсистем;
- желательно, чтобы все электронные блоки работали от 5-вольтового источника питания - фактически это значение является стандартным для логических микросхем (кроме того, 5-вольтовые элементы более устойчивы к помехам, чем, например, 2,5- или 3,3-вольтовые);
- используйте только многократно программируемый контроллер и располагайте его на специальном разъеме (socket), из которого он при необходимости может быть легко извлечен. Следует помнить, что в результате отладки своей конструкции вы не раз захотите изменить код программы, и лучше заранее позаботиться о том, чтобы каждое такое изменение не требовало лишнего времени и усилий.

ГЛАВА 2

РАЗРАБОТКА

ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Во введении я уже объяснял, что удобно выделить три уровня задач, решаемых программой управления роботом. Разбиение на биологический, механический и электронный уровни обычно упрощает процесс разработки и позволяет добиться наилучших результатов. Во-первых, такое деление упрощает код программы, которая должна решать только свой круг задач. Во-вторых, при многоуровневом проектировании возможно обеспечить лучшее согласование различных функций, выполняемых роботом, чем в случае, когда все эти функции реализуются в одной общей программе. Ниже мы рассмотрим образцы программ, написанных для решения задач разных уровней, а также обсудим, как они могут быть связаны друг с другом. Как вы увидите, все примеры спроектированы таким образом, чтобы в максимальной степени облегчить стыковку различных программных модулей.

В этой главе речь пойдет об использовании различных инструментальных средств, которые необходимы при разработке программного обеспечения (ПО) для микроконтроллеров. Мы начнем с самых простых инструментов программиста и постепенно дойдем до наиболее сложных. Будет рассмотрен самый общий инструментарий, применяемый во всех случаях, а не только для программирования роботов. Позже в книге мы обсудим основные шаблоны проектирования программ, которые полезны именно для применения в области робототехники.

Надо заметить, что, хотя существуют специализированные средства проектирования автоматических устройств, в большинстве случаев достаточно использования универсальных инструментов разработки, описанных в этой главе.

Обсуждение начнем с инструментальных средств наиболее высокого уровня, чтобы дать ясное понимание того, на что способны эти средства и зачем они применяются, какие у них преимущества и недостатки. Полагаю, что у вас не возникнет затруднений при установке компилятора PICC Lite и интегрированной среды разработки MPLAB IDE. Обе программы можно найти в свободном доступе на сайтах www.microchip.com и www.htsoft.com. Если у вас все-таки возникнут трудности с установкой, желательно обратиться к ресурсам, посвященным элементарным навыкам работы на персональном компьютере, а также потратить некоторое время на изучение основ программирования на языке C.

Интегрированная среда разработки MPLAB фирмы Microchip, подробно обсуждаемая в главе 3, предназначена для редактирования текста программы, ее компиляции и осуществления моделирования (simulating) процесса выполнения

программы (даже в отсутствие самого микроконтроллера), а также для загрузки программы в микроконтроллер.

Компилятор PICC Lite фирмы HI-TECH Software - свободно распространяемый компилятор языка C для микроконтроллера PIC16F84. Этот компилятор хорошо интегрируется со средой разработки MPLAB. Оба программных продукта в совокупности представляют мощный инструмент разработки программного обеспечения для микроконтроллеров, обеспечивающий возможности программирования и отладки, за которые при использовании других контроллеров пришлось бы заплатить тысячи долларов.

К недовольствию пользователей рабочих станций, компьютеров Apple и приверженцев Linux, большинство инструментальных средств разработки ПО работает только на персональных компьютерах PC под управлением операционной системы Windows (так называемые Wintel PC). Инструментальные средства, разработанные для рабочих станций, как правило, очень дороги и к тому же обычно входят в еще более дорогие программные пакеты или прилагаются к некоторым устройствам. С другой стороны, все еще доступны бесплатные средства разработки, ориентированные на операционные системы Windows 3.x или даже MS DOS. К сожалению, они, как правило, не поддерживают многие полезные возможности и не могут работать с новыми микроконтроллерами.

В последнее время все доступнее становится специализированное программное обеспечение для операционной системы Linux. Для многих микроконтроллеров уже существуют версии компиляторов GCC языка C, разработанные в рамках проекта GNU; они поддерживают многие возможности из тех, что описаны в этой книге применительно к компилятору PICC Lite. Кроме того, существуют версии инструментальных средств для системы UNIX, компьютеров Macintosh и др.

Чтобы избежать проблем, связанных с поиском средств разработки ПО для определенного микроконтроллера, лучше использовать инструменты, рассчитанные на операционную систему Windows. Они не только дешевле, но и обеспечивают максимальную переносимость ваших проектов на другие платформы.

2.1. ОТ ИСХОДНОГО ТЕКСТА ПРОГРАММЫ К HEX-ФАЙЛУ

Прежде всего обсудим, с какими файлами приходится иметь дело программисту, когда он разрабатывает ПО для микроконтроллеров.

Исходный текст программы ("исходник"), как известно, хранится в текстовом файле в коде ASCII (American Standard Code for Information Interchange; произносится "аски"). Этот текст содержит операторы используемого языка программирования, а также комментарии, в которых автор программы может пояснить назначение каждого оператора. Текст программы может быть написан с использованием любого текстового редактора (Блокнот, WordPad). Простой текстовый редактор также имеется в составе интегрированной среды разработки.

Написанный программистом исходный текст обрабатывается специальной программой - *интерпретатором* или *компилятором*. В результате работы компилятора

получается так называемый *объектный файл* или *HEX-файл* (шестнадцатеричный файл), который с помощью программатора может быть загружен в память микроконтроллера. В случае использования интерпретатора дело обстоит несколько иначе: программа-интерпретатор встроена в микроконтроллер и может непосредственно выполнять команды, записанные в исходном тексте программы.

Текст программы может содержать ссылки на другие (обычно тоже текстовые) файлы, в которых описаны используемые в программе функции или определены необходимые данные. Такие включаемые файлы обычно имеют расширение .inc. В случае использования языка С файлам, включаемым в исходный текст программы, присвоено расширение .h. Использование отдельных файлов упрощает процесс программирования, и после приобретения необходимого опыта вы сможете самостоятельно разработать библиотеку включаемых файлов.

В результате компиляции создается файл с расширением .obj или .hex. HEX-файл уже может загружаться в память микроконтроллера, а OBJ-файл предназначен для последующего связывания с другими OBJ-файлами для формирования окончательного HEX-файла. Эту работу выполняет так называемый *редактор связей* (компоновщик, linker).

Наиболее популярный формат HEX-файлов - это 8-разрядный формат INXM8. Именно в этом формате должен быть записан HEX-файл для программатора EI Cheapo. Заметим, что для микроконтроллеров фирмы Motorola обычно используется другой формат, так называемый S9.

Приведем пример HEX-файла, подготовленного для микроконтроллера PIC в формате INXM8:

```
:10000000FF308600831686018312A001A101A00B98
:0A0010000728A10B07288603072824
:02400E00F13F80
:00000001FF
```

Каждая шестнадцатеричная цифра представлена в текстовом файле в коде ASCII, то есть занимает один байт. В каждой строке указаны стартовый адрес и данные, которые должны быть загружены в память начиная с этого адреса. Смысл каждой позиции в текстовом файле поясняется в табл. 2.1.

Таблица 2.1. Формат 1NXM8

Порядковый номер позиции	Назначение
1	Всегда содержит символ : (двоеточие) - указывает начало строки
2-3	Число команд, содержащихся в данной строке (две шестнадцатеричные цифры на каждую команду)
4-7	Стартовый адрес, начиная с которого будут размещаться команды, в формате Big Endian (начиная со старшего байта, как в процессорах Motorola)
8-9	Тип строки: 00 - программа или данные, 01 — конец
10-13	Первая команда, которая должна загружаться в память начиная с указанного адреса, в формате Little Endian (начиная с младшего байта, как в процессорах Intel)
...	Другие команды, 4 символа на каждую

Таблица 2.1. Формат INXM8 (окончание)

Порядковый номер позиции	Назначение
Предпоследние два байта	Контрольная сумма строки
Последние два байта (не отображаются в текстовом редакторе)	ASCII-коды символа возврата каретки (0x0D) и перевода строки (0x0A)

Для вычисления контрольной суммы необходимо сложить все байты строки и вычесть младший байт полученной суммы из шестнадцатеричного числа 0x0100. Для примера найдем контрольную сумму второй строки приведенного выше фрагмента HEX-файла:

0A
00
10
00
07
28
A1
0B
07
28
86
03
07
+ 28

ЮС

Вычтем младший байт результата из 0x0100:

0x0100
- 0x00DC

0x0024

Полученное **шестнадцатеричное** значение 24 записано в конце второй строки рассматриваемого HEX-файла.

К созданному компилятором объектному файлу компоновщик может добавить нужные библиотечные файлы, содержащие код функций, необходимых для выполнения основной программы. Например, в случае использования языка высокого уровня в этих библиотечных файлах могут быть описаны команды умножения и деления, операции над комплексными числами или команды ввода-вывода. Подобные библиотеки обычно поставляются вместе с инструментальным средством, но программист может использовать и библиотечные объектные файлы из любых других источников, в том числе свои собственные.

Компоновщики обычно имеют многочисленные настройки, определяющие различные режимы работы по созданию конечного HEX-файла из одного или нескольких OBJ-файлов. К счастью, значения большинства параметров, установленные *по умолчанию*, обеспечивают наиболее часто используемый режим компоновки. При этом программист может быть уверен, что функции, которые хотя

и содержатся в подключаемой библиотеке, но ни разу не используются в программе, будут исключены при компоновке из конечного HEX-файла. В результате будет создан загрузочный файл минимального размера. Обычно в объектных библиотеках содержатся десятки или сотни различных функций, и глупо было бы зря расходовать память микроконтроллера, загружая в нее те из них, которые никогда не используются.

Компиляция приложения, состоящего всего из одного исходного файла и не обращающегося к каким-либо другим программным модулям, - довольно простое дело. В таких программах очень легко разобраться; притом их не слишком сложно отлаживать. В противном случае всегда существует некоторая опасность неправильного обращения к ресурсам других модулей. Но иногда затруднительно разработать приложение, состоящее всего из одного файла, например, если над проектом работает не один программист. В этом случае удобнее разбивать большую программу на отдельные модули.

Сложные программные проекты, включающие в себя множество исходных файлов и библиотек объектных модулей, хорошо подходят для тех случаев, когда в группе несколько программистов. Но такой подход требует строгой проверки работоспособности каждого программного модуля по отдельности. В некоторых случаях, когда для объектных модулей недоступны исходные тексты, процесс компоновки всего приложения может вызвать некоторые затруднения, а отладка становится еще более сложной.

2.2. АССЕМБЛЕР

Считается, что *язык ассемблера* (или просто *ассемблер*) позволяет разрабатывать наиболее эффективный машинный код. Однако этот язык (как говорят, *язык низкого уровня*) труден для изучения и не слишком легок в использовании, поэтому запрограммированные с его помощью приложения не всегда оказываются **самыми** эффективными. Несмотря на это обстоятельство, ассемблирование является любимым занятием многих разработчиков ПО для управления роботами, потому что компиляторы для этого языка, предназначенные для работы с самыми различными микроконтроллерами, обычно бесплатно распространяются через Internet. Кроме того, программирование на ассемблере - само по себе увлекательное занятие.

Прежде чем начать разработку программ на языке низкого уровня, вы должны хорошо разобраться в том, как работает микроконтроллер. Причем недостаточно только познакомиться с системой его команд. Программиста подстерегает множество ловушек, которые не видны тем, кто просто смотрит на список команд процессора, не пытаясь написать действительно работающую программу.

Говоря, что необходимо разобраться в том, как работает микроконтроллер, я не имею в виду, что надо досконально знать принципы его функционирования на аппаратном уровне. Нет, достаточно просто понимать, как он выполняет свои

команды. Многие команды, которые при первом знакомстве кажутся одинаковыми у разных процессоров, на самом деле работают несколько по-разному, причем это трудно заметить, изучая систему команд процессора.

Рассмотрим для примера следующий фрагмент программы на языке C:

```
if (A == 0)      // Если значение переменной A равно нулю,
    B++;        // то увеличить значение переменной B на 1.
```

Для обычного 8-разрядного процессора ассемблерный код, соответствующий этому фрагменту, будет выглядеть следующим образом:

```
move Acc, A      ; Загрузить в аккумулятор значение переменной A.
add Acc, #0       ; Прибавить к нему 0, чтобы затем сравнить результат с 0.
jnz Skip         ; Если результат не нулевой, то перейти на метку Skip,
move Acc, B       ; в противном случае выполнить инкремент B.
add Acc, «1
move B, Acc
Skip:
```

Для многих микроконтроллеров предусмотрена специальная команда для инкремента и декремента (увеличения и уменьшения на 1); в этом случае соответствующие строки программы могут быть изменены. Например, три последние команды в нашем примере заменятся одной-единственной командой инкремента переменной B.

У некоторых процессоров (в том числе и у микроконтроллеров PIC) команда `move` изменяет состояние флажков; это позволяет исключить из текста программы строку

```
add Acc, #0
```

Кроме того, обычно микроконтроллеры имеют специальную инструкцию, которая в зависимости от состояния каких-либо флагов позволяет выполнить или пропустить следующую команду. Например, если использовать инструкцию `skpnz`¹ (SKip if Not Zero - пропустить, если не 0), то наша программа еще более упростится:

```
movf A, f ; Установить флаг нуля, если A = 0.
skpnz     ; Пропустить следующую команду, если не установлен флаг нуля.
incf B, f ; Выполнить инкремент B.
```

Кроме простоты измененный фрагмент программы имеет много достоинств, которые, возможно, незаметны на первый взгляд:

- состояние аккумулятора не изменяется - мы просто устанавливаем флаги в соответствии со значением переменной A;
- при использовании команды `skpnz` отпадает необходимость расставлять метки и применять условные команды перехода;

¹ Инструкция `skpnz` - это **макрокоманда**, которой на самом деле соответствует машинная команда `btfsc STATUS, Z`. - *Прим. перев.*

- количество машинных циклов («тиков» тактового генератора), в течение которых будет выполняться программа, не зависит от того, выполнилась команда инкремента или нет.

Однако в последнем примере есть один недостаток: сравнивая ассемблерный код с исходным текстом программы на языке C, сразу и не поймешь, что оба фрагмента делают одно и то же.

Приведенный пример показывает, как важно понимать назначение и побочный эффект каждой команды процессора, чтобы разрабатывать действительно эффективный код. Поверьте, усилия, затраченные на изучение ассемблера, с лихвой окупятся при программировании!

Когда вы достаточно хорошо освоите программирование на ассемблере, то сможете представить свою программу как «кинофильм», каждый кадр которого соответствует отдельной команде процессора. Тогда даже без использования отладчика вам будет понятно назначение каждой строчки программы, и вы без труда найдете ошибки, которые, возможно, в ней содержатся.

Но даже опытным программистам на ассемблере стоит учитывать несколько важных моментов. Первое, о чем следует предупредить: программа, разработанная для одного микроконтроллера, не сможет работать на другом. Ее перенос на другую аппаратную платформу может потребовать значительных усилий. Это так же трудно, как описать другому программисту во всех подробностях кадры того «кинофильма», который прокручивался у вас в голове. Во-вторых, иногда довольно сложно использовать код, написанный на ассемблере, в программе на языке высокого уровня. Дело в том, что процедуры и функции в высокоуровневых языках имеют входные и выходные параметры. И при разработке ассемблерной процедуры, которая будет вызываться из программы, написанной, например, на C, приходится принимать во внимание все тонкости работы с этими параметрами. Корректность передачи данных в ассемблерную процедуру и правильность возвращаемых значений могут потребовать кропотливого тестирования.

Когда пытаешься разобраться в программе, написанной на языке ассемблера, обычно не так уж просто осознать ее структуру и назначение отдельных частей. И это самый большой недостаток ассемблерных программ. Они обычно плохо структурированы и отвлекают внимание программиста на особенности аппаратуры, скрывая главную идею, заложенную в тот или иной фрагмент кода.

Языки высокого уровня, напротив, весьма близки к естественному английскому языку. Например, английское предложение

if the contents of variable A are equal to zero, then increment the contents of variable B
(если значение переменной A равно нулю, то увеличить на единицу значение переменной B)

на языке Basic запишется следующим образом:

if (A = 0), then B = B + 1

а на языке C:

if (A == 0)
B++;

Записи на языках Basic или C по структуре близки к вышеуказанному предложению, а вот ассемблерная программа для контроллера PIC будет не настолько очевидной.

Материальные соображения оказываются не последними, когда разработчику приходится выбирать язык программирования для своего будущего проекта. Инструментальные средства разработки приложений на языке ассемблера бесплатны для большинства микроконтроллеров и обычно распространяются по сети Internet; порой это оказывается решающим фактором, заставляющим программистов писать свои программы на языке низкого уровня.

Несмотря на отмеченные недостатки, ассемблер был и остается весьма полезным инструментом. Только на этом языке часто оказывается возможным реализовать критичные участки программы, работающие с аппаратурой и обеспечивающие точно заданные временные задержки между отдельными операциями.

Тем не менее в этой книге, как уже отмечалось, я постараюсь не использовать язык низкого уровня - во-первых, чтобы не затруднять понимание программного кода, а во-вторых, чтобы облегчить перенос программ на другие микроконтроллеры.

2.3. ИНТЕРПРЕТАТОРЫ

Когда персональные компьютеры только появились, интерпретаторы (такие как Basic) были довольно распространены. Большая часть программного обеспечения и даже обучение программированию было ориентировано на их использование. Именно интерпретатор языка Basic загружался в память первых компьютеров IBM PC при их включении и брал на себя выполнение тех функций, которые сейчас считаются прерогативой операционной системы.

Интерпретатором называется специальная программа, которая запускается на компьютере и выполняет исходный текст программы. Поэтому не требуется какое-либо преобразование исходного файла, как в случае использования компилятора. Кроме того, интерпретатор обычно обеспечивает простейший интерфейс пользователя, позволяя ему вводить данные, редактировать программу и сохранять ее на диске или магнитной ленте.

Приведем пример простой программы на языке Basic:

```
for i = 1 to 4
  print i
next i
end
```

Как только пользователь вводит команду, интерпретатор сразу выполняет ее. В нашем примере имеется оператор цикла, который выполнится четыре раза, для значений параметра *i* от 1 до 4. Поэтому будет распечатано четыре числа:

```
1
2
3
4
```

После этого интерпретатор выведет приглашение (prompt) для ввода следующей команды.

Программа-интерпретатор, встроенная в память компьютера, позволяет фирмам-изготовителям предоставить пользователям машину, готовую к использованию. Вместо того чтобы вводить свою программу в машинных кодах с помощью, например, кнопок, расположенных на лицевой панели компьютера, пользователь получает в свое распоряжение удобный интерфейс для общения с машиной.

Интерпретатор, встраиваемый в некоторые микроконтроллеры, чаще всего работает с помощью какого-либо последовательного интерфейса. Например RS-232. Как правило, такой интерпретатор дает пользователю возможность загружать свои программы в микроконтроллер с помощью персонального компьютера. Как и для первых PC, интерпретаторы для микроконтроллеров обычно основаны на языке Basic.

Можно привести несколько доводов в пользу использования интерпретатора при разработке автоматического устройства.

Во-первых, интерпретатор - это инструмент, дружелюбный пользователю. Он позволяет быстро проверить ту или иную вашу идею, не прибегая к кропотливому процессу перекомпиляции всего приложения. Вы просто подключаете своего робота к разьему последовательного порта компьютера, изменяете нужную строку программы и проверяете, что из этого получилось. Конечно, есть и обратная сторона такой легкости изменения кода - часто программист забывает о необходимости документировать все изменения, вносимые в свой проект, а иногда по оплошности и вовсе не сохраняет на диске компьютера измененную программу.

Во-вторых, программа, рассчитанная на работу с интерпретатором, обычно не использует особенности аппаратуры и поэтому легко может быть перенесена на другой микроконтроллер.

В-третьих, у программ-интерпретаторов обычно много встроенных функций, к которым можно обратиться, не прибегая к созданию низкоуровневых процедур для работы с различными интерфейсами аппаратных средств (электронный уровень) или с электромеханическими устройствами робота (механический уровень). Например, разработчик сможет воспользоваться готовой процедурой для подавления дребезга механических контактов или вызвать заложенную в интерпретатор функцию широтно-импульсной модуляции (ШИМ) для регулирования скорости вращения двигателя.

В-четвертых, язык Basic и другие языки, используемые в интерпретаторах, очень просто изучить и применять; к тому же они уже получили распространение в мире программирования, и разработчику программ для роботов не придется осваивать какой-то специальный язык. О языке Basic написано немало книг, и имеется большое количество готовых программ, которые можно будет использовать в проекте.

Наконец, большинство интерпретаторов для микроконтроллеров бесплатно распространяется по сети Internet. (Например, интерпретатор, разработанный мной для микроконтроллера PIC16F87х, можно скачать с сайта www.myke.com.) В большинстве случаев доступна не только программа-интерпретатор, но и ее исходный текст на каком-либо языке, поэтому при необходимости пользователь может изменить заложенные в нее функции.

Разумеется, используя интерпретатор, программист должен мириться и с некоторыми его недостатками. Самый важный из них - низкая скорость выполнения программ. Программа, написанная на ассемблере, обычно выполняется процессором со скоростью несколько миллионов машинных команд в секунду, а написанная на компилирующем языке высокого уровня - несколько десятков тысяч операторов в секунду. Интерпретатор же способен выполнить всего сотню, в лучшем случае тысячу операторов в секунду. Для функций биологического уровня такая медлительность еще допустима, но выполнение функций нижних уровней часто требует большего быстрогодействия.

Программист должен знать некоторые особенности использования интерпретатора при разработке автоматического устройства.

Во-первых, многочисленные комментарии к тексту программы, которые можно только приветствовать в случае применения компилятора, могут существенно замедлить работу интерпретатора. Ведь последнему приходится читать программу строка за строкой и тут же ее выполнять; при этом требуется дополнительное время на то, чтобы распознать и исключить из рассмотрения символы, входящие в комментарий.

Во-вторых, при использовании команд перехода (таких как `goto`) интерпретатору приходится просматривать всю программу в поисках метки, на которую должен быть сделан переход. Это требует много времени, поэтому во многих ранних версиях языка Basic все строки программы нумеровались - ведь найти нужную строку по номеру гораздо проще, чем просматривать весь текст в поисках символической метки.

В-третьих, при использовании интерпретатора сложно реализовать заданные временные промежутки между отдельными командами, потому что затруднительно предсказать, как долго будет выполняться та или иная команда.

В-четвертых, программа-интерпретатор занимает довольно много памяти, так что на саму программу управления роботом остается меньше места. Немногие микроконтроллеры способны разместить интерпретатор в своем внутреннем ПЗУ, поэтому обычно приходится использовать внешние микросхемы памяти. Здесь опять можно вспомнить о комментариях — в скомпилированной программе они не занимают ни одного дополнительного байта, но интерпретатору приходится держать в памяти весь текст программы, в том числе и ненужные ему комментарии.

Следует заметить, что возможности отладки программ, встроенные в большинство интерпретаторов, достаточно скромны.

Работая с аппаратными средствами микроконтроллера, интерпретатор должен обращаться к его регистрам. Но из-за невысокой скорости работы процесс чтения и записи происходит слишком медленно, и система обработки прерываний может не успеть обработать запрос какого-нибудь внешнего устройства.

Хотя интерпретатор позволяет вносить изменения в текст программы без использования программатора, загрузка интерпретатора в память микроконтроллера требует того же самого программатора. Правда, в некоторых случаях микроконтроллер может быть приобретен вместе со встроенным в него интерпретатором,

но это значительно увеличивает его стоимость. К тому же редко приходится рассчитывать на то, что встроенный интерпретатор будет поддерживать те функции, которые потребуются разработчику для его проекта.

Наконец, стоит упомянуть о модулях Parallax Basic Stamp, в которых используется подход, совмещающий преимущества интерпретатора и компилятора. Такие модули программируются на языке PBasic, который каждый оператор исходного текста программы компилирует в специальный код (последовательность *токенов*). Этот код, а не сам текст программы, загружается в память модуля Basic Stamp. Каждый токен обычно занимает в памяти один байт и кодирует некоторую встроенную функцию. Но последовательность токенов - не совсем то же самое, что исходный текст программы, поэтому Basic Stamp не может считаться «истинным» интерпретатором. Basic Stamp часто используется разработчиками, так как недорог, легко программируется, поддерживает много полезных встроенных функций и в сети Internet для него доступно большое число примеров программ.

Выбирая подходящий интерпретатор, разработчик должен обратить внимание на наличие следующих встроенных функций:

- возможность сохранения программы в ПЗУ;
 - поддержка простого последовательного интерфейса;
 - наличие широтно-импульсного модулятора;
 - поддержка интерфейсов PC, SPI, CAN и др.;
 - наличие функции подавления дребезга контактов;
 - поддержка аналого-цифрового преобразователя (АЦП).
-

2.4. КОМПИЛЯТОРЫ

Как уже говорилось, компилятор преобразует исходный текст программы в машинный код, предназначенный для исполнения процессором. При этом большинство современных компиляторов для языков высокого уровня выполняет еще некоторые действия, пытаясь упростить код программы и/или увеличить скорость ее выполнения. Если вы решили освоить язык ассемблера, следует иметь в виду, что машинный код, произведенный компилятором, - прекрасный пример для изучения.

Все примеры, приведенные в этой книге, написаны на языке C. Для компиляции программ использовался компилятор PICC Lite фирмы HI-TECH Software. Язык C для микроконтроллеров основан на стандарте ANSI C (он описывается в приложении). Код, производимый PICC Lite, весьма эффективен. Кроме того, этот компилятор прекрасно интегрируется со средой разработки MPLAB.

Рассмотрим работу компилятора на примере. Пусть исходный текст программы на C содержит строку

```
.     A = B + (C * D);
```

Для перевода этого фрагмента текста в машинный код компилятору требуется выполнить ряд действий. Большинство компиляторов при этом использует

специальную структуру микропроцессора - так называемый *стек* (stack). Во время синтаксического разбора исходного текста все значения, встречаемые в выражении, помещаются в этот стек в порядке, обратном тому, который имел место в программе. По мере вычислений полученные значения «выталкиваются» из стека.

Таблица 2.2. Основные типы операторов языка высокого уровня

Оператор	Тип
имя = ...	Оператор присваивания
if (. . .)	Условный оператор
имя { . . . }	Вызов подпрограммы (функции)
тип имя [= константа]	Объявление переменной
[тип] имя {параметр, . . . }	Объявление подпрограммы (функции)

В первую очередь компилятору требуется определить тип оператора. Обычно выделяют пять типов, указанные в табл. 2.2.

В первых трех случаях часть оператора, обозначенная многоточием, называется выражением. *Выражение* - это константа или переменная либо несколько констант и/или переменных, соединенных знаками арифметических, логических и других операций. В приложении приведен список операций, допустимых в PICC Lite. Аналогичные операции имеются в стандартном C и во многих других языках программирования.

Значения констант и переменных, а также промежуточные результаты во время вычислений хранятся в стеке. Само выражение обычно переводится компилятором в так называемую *постфиксную запись*, в которой знаки операций располагаются после операндов в $c \ D \ * \ +$ или $c \ D \ * \ в \ +$.

В памяти компьютера постфиксное представление выражения часто хранится в виде двоичного дерева. Пример постфиксной записи выражения

$A = B + (C * D);$

приведен на рис. 2.1.

Приоритет операций в постфиксной записи определяется тем порядком, в котором данные помещались в дерево (снизу вверх, справа налево), поэтому нет необходимости расставлять скобки. Для нашего примера компилятор выполнит такую последовательность операций со стеком:

$A = B + (C * D)$

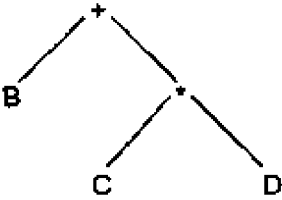


Рис. 2.1. Дерево разбора

- поместить в стек D;
- поместить в стек C;
- выполнить умножение двух верхних чисел в стеке, удалить их из стека и поместить в его вершину результат;
- поместить в стек B;

- выполнить сложение двух верхних чисел в стеке, удалить их из стека и поместить в его вершину результат;
- взять из вершины стека результат и поместить его в переменную A

Читатель, возможно, помнит карманные калькуляторы, которые программировались сходным образом. Такая запись вычислений называется *обратной польской нотацией* (Reverse Polish Notation - RPN).

То, что результаты вычислений сохраняются в стеке, важно для операторов вызова подпрограмм, а также для условных операторов. Первые берут входные данные из стека и результат своей работы помещают туда же, а вторые в зависимости от значения, находящегося в вершине стека, либо выполняют, либо пропускают указанный в их теле оператор или последовательность операторов.

Другой способ вычислений заключается в том, что промежуточные результаты хранятся в специальных ячейках памяти, которые задаются непосредственно. Проиллюстрируем это на нашем примере:

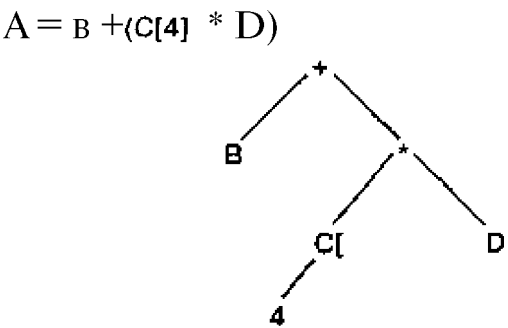
```
Temp1 = C;  
Temp2 = D;  
Temp3 = B;  
  
Temp1 = Temp1 * Temp2;  
Temp1 = Temp1 + Temp3;  
  
A = Temp1
```

Но это более сложный способ, чем использование стека.
Рассмотрим теперь пример, когда в вычислениях участвует элемент массива:

```
A = B + ( C[4] * D );
```

Дерево выражения представлено на рис. 2.2. Последовательность операций такова:

- поместить в стек D;
- поместить в стек 4;
- поместить в стек элемент массива C, номер которого указан в вершине стека (удалив этот номер из стека);
- выполнить умножение двух верхних чисел в стеке, удалить их из стека и поместить в его вершину результат;
- поместить в стек B;
- выполнить сложение двух верхних чисел в стеке, удалить их из стека и поместить в его вершину результат;
- взять из вершины стека результат и поместить его в переменную A



Это можно записать короче:

```
Push      D  
Push      4  
Push      C[
```

Рис. 2.2. **Дерево** разбора в случае использования элемента массива

Execute	*
Push	B
Execute	+
Pop	A

Здесь использованы следующие сокращения: `push` - поместить указанное значение в вершину стека; `pop` - извлечь значение из вершины стека и записать его в указанную переменную; `execute` - выполнить операцию; квадратная скобка обозначает элемент указанного массива, номер которого хранится в вершине стека.

Теперь рассмотрим, как работают вызовы функций. Допустим, мы описали функцию двух переменных

```
int Func(int varA, int varB)
```

и используем ее при вычислении выражения

```
A = B + Func(C[4], D )
```

Дерево разбора для этого случая приведено на рис. 2.3. Последовательность операций со стеком такова:

Push	D
Push	4
Push	C[
Call	Func
Push	B
Execute	+
Pop	A

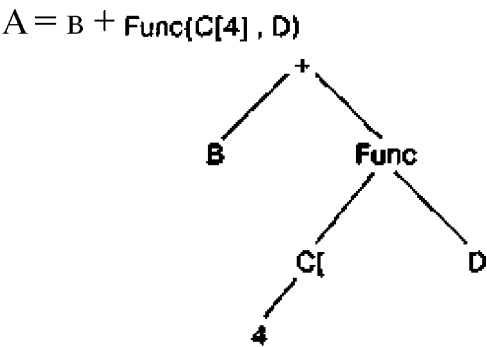


Рис. 2.3. Дерево разбора в случае использования функции

К тому моменту, когда производится вызов функции, в вершине стека уже находятся оба параметра, необходимых для ее выполнения. Параметр, указанный в тексте программы последним, находится в самой вершине стека, а тот, что указан первым, - в стеке под ним. Это происходит потому, что параметры помещаются в стек в том же порядке, в каком они указаны в тексте программы.

Теперь рассмотрим следующую функцию на языке C:

```
int Func(int varA, int varB)
{
    varA = varA + 1;
    return varA * varB;
}
```

Соответствующий машинный код, получившийся после компиляции, будет выглядеть следующим образом:

Func:	
; Вычислить varA = varA + 1.	
Push	StackTop - 1 ; Поместить varA в вершину стека.
Push	1 ; Поместить 1 в вершину стека.
Execute	+ ; Сложить два числа в стеке.
Pop	StackTop - 1 ; Взять число из вершины стека и по-

местить его в `varA`.

```
; Вычислить и вернуть результат varA * varB.
Push      StackTop - 1 ; Поместить в вершину стека varA.
Push      StackTop      ; Поместить в вершину стека varB.
Execute    *              ; Выполнить умножение двух верхних чисел в стеке.

return
```

Здесь в переменной `StackTop` хранится указатель на вершину стека, какой она была в момент вызова функции. Благодаря этому указателю функция получает доступ к своим параметрам.

Фрагмент программы, вызывающий нашу функцию, выглядит примерно так:

```
Push      varA
Push      varB
Call      Func
Pop       StackTop - 1 ; Поместить результат в первый параметр функции,
                      ; то есть в varA.
Pop       BitBucket   ; Удалить из стека второй параметр - varB.
; Результат выполнения функции - в вершине стека.
```

Результат работы реального компилятора может несколько отличаться от вышеприведенного. Например, многие компиляторы выполняют оптимизацию кода. Тогда фрагмент программы

$$A = B + (C * (4 \cdot 2));$$

будет заменен на

$$A = B + (C * 8);$$

В результате получим последовательность стековых операций:

```
Push      8
Push      C
Execute    *
Push      B
Execute    +
Pop       A
```

В случае, когда при разработке приложения используются несколько разных языков программирования (например, если в проекте участвуют несколько программистов), каждый исходный файл, написанный одним из программистов на известном ему языке, переводится компилятором в отдельный объектный файл. Окончательную сборку нескольких объектных файлов выполняет *компоновщик* (linker). Если исходный текст программы очень велик (более 10000 строк кода), целесообразно также разбить его на несколько программных модулей. В этом случае каждый модуль компилируется и отлаживается отдельно.

Программист должен осознавать несколько важных моментов, касающихся использования компиляторов. Результатом работы дешевого компилятора может оказаться машинный код, выполняемый не быстрее, чем при использовании интерпретатора, и к тому же занимающий намного больше памяти. Следует заметить, что многие компиляторы языков высокого уровня весьма недешевы, в то время как ассемблеры для большинства микроконтроллеров совершенно бесплатны.

Сообщения об ошибках (errors) и предупреждения (warnings), выдаваемые компиляторами, порой не слишком ясны начинающему программисту (иногда они вообще не соответствуют действительным ошибкам в программе).

Никогда не загружайте программу в микроконтроллер, пока не добьетесь того, чтобы процесс компиляции прошел без всяких предупреждающих сообщений: ведь даже самые на первый взгляд безобидные сообщения во время компиляции могут привести к тому, что программа не будет работать как надо (или даже совсем не будет работать).

2.5. СИМУЛЯТОРЫ И ЭМУЛЯТОРЫ

Иногда отладка программы оказывается очень трудным делом, особенно если это программа для управления роботом. Позже в этой книге мы обсудим, какими индикаторами следует снабдить автоматическое устройство, чтобы всегда было ясно, в каком состоянии находится его управляющая программа. Но, так или иначе, прежде чем загрузить программу в память микроконтроллера, следует сначала позаботиться о том, чтобы отладить ее на персональном компьютере. Несколько минут, потраченных на это, с лихвой компенсируют многие часы мучений, которых может потребовать отладка программы на «живом» роботе.

Для отладки программы без использования самого микроконтроллера, для которого она разрабатывалась, предназначено специальное инструментальное средство, называемое *симулятором* (simulator).

Не имея доступа к реальной аппаратуре, симулятор работает с *программной моделью* микроконтроллера, имитируя все его действия при выполнении не только арифметических или логических операций, но и многих команд ввода-вывода. К сожалению, симулятор не способен точно моделировать работу микроконтроллера с периферийными устройствами. Например, симулятор, входящий в состав MPLAB IDE, являясь превосходным инструментом разработки программ, все же оказывается не в состоянии поддерживать отладку некоторых сложных операций ввода-вывода.

Перечислим основные особенности, которые следует иметь в виду при выборе симулятора:

- возможность отладки программ на уровне исходного текста (source-code level debugging);
- точное моделирование времени выполнения каждой команды (это не слишком важно для высокоуровневых программ, приведенных в данной книге, но может оказаться необходимым при программировании на ассемблере процедур, работающих с периферийными устройствами);
- возможность задавать набор и последовательность сигналов (stimulus), поступающих на входы виртуального микроконтроллера во время отладки программы.

Симулятор можно представить в виде набора нескольких блоков (рис. 2.4), соединенных между собой и управляемых с помощью пользовательского

интерфейса симулятора. Реальные программы-симуляторы, возможно, используют большее количество блоков, но для нашего обсуждения хватит и тех пяти, что изображены на рис. 2.4.



Рис. 2.4. Структура программного симулятора

Память программ моделирует работу реальной памяти микроконтроллера, в которую с помощью программатора загружается управляющая программа. *Модель процессора* имитирует все действия настоящего процессора, который выбирает из памяти программ очередную команду и необходимые данные, вычисляет результат и помещает его в указанное место, после чего выбирает следующую команду и т.д. *Регистровый файл* моделирует работу регистров микроконтроллера.

Модель устройств ввода-вывода (УВВ) предназначена для чтения из заранее подготовленного файла значений сигналов, которые необходимо подать на входные порты виртуального микроконтроллера, и позволяет прочитать состояние его выходных портов на каждом этапе процесса отладки.

Обратите внимание на важный момент во время отладки с помощью симулятора: при каждом прогоне программы можно обеспечить одну и ту же последовательность подаваемых на микроконтроллер входных сигналов. В противном случае было бы сложно выявить ошибки в программе.

Модель процессора по праву считается сердцем симулятора. Она позволяет не только имитировать выборку команд из памяти, их выполнение и обращение к регистрам микроконтроллера, но также управляет многими его аппаратными средствами, например системой прерываний.

Важно подчеркнуть, что за работой симулятора программист следит в том же окне, где расположен исходный текст программы, то есть моделирование работы микроконтроллера осуществляется на уровне операторов языка высокого уровня, а не на уровне ассемблерных команд (хотя последнее тоже возможно).

Другой способ отладки программ, более приближенный к реальности, предполагает использование так называемого внутрисхемного эмулятора. *Эмулятор*

(emulator) - это специальное устройство, которое подключается вместо микроконтроллера к той схеме, в которой он должен работать. Работой эмулятора с помощью специальной программы управляет персональный компьютер. В результате удастся максимально точно имитировать поведение микроконтроллера в целевом устройстве - ведь теперь виртуальным оказывается только сам микроконтроллер, а все порты ввода-вывода и аппаратура, работающая с ними, выполнены «в железе» (рис. 2.5). При использовании эмулятора можно наблюдать реальные сигналы на выводах отлаживаемой схемы.



Рис. 2.5. Подключение внутрисхемного эмулятора

Скажу больше: эмулятор часто выполняется на основе того же самого микроконтроллера, который он должен эмулировать. К обычному микроконтроллеру добавляются специальные устройства для обеспечения связи процессора с компьютером и управления пошаговым режимом работы. Именно так выполнены эмуляторы фирмы Microchip.

Можно отметить два недостатка, присущих эмуляторам. Во-первых, настоящие эмуляторы весьма дороги. Поэтому выпускаются более простые, такие как MPLAB-ICD Debugger фирмы Microchip. Этот инструмент предназначен для отладки устройств на базе микроконтроллеров PIC16F87х и поддерживает большинство функций, присущих дорогим эмуляторам, но его возможности по отладке некоторых команд, работающих с аппаратурой, ограничены.

Во-вторых, эмулятор, вставляемый в разъем целевого устройства вместо микроконтроллера, должен иметь связь с компьютером. Часто это не столь уж страшное ограничение, но в случае отладки управляющей программы для мобильного устройства обеспечить такую связь иногда затруднительно.

При выборе эмулятора в первую очередь необходимо выяснить, поддерживает ли он возможность отладки на уровне исходного текста программы. Если это не так, вам придется потратить уйму времени, разбираясь в машинном коде и путаясь в абсолютных адресах команд перехода.

2.6. ИНТЕГРИРОВАННЫЕ СРЕДСТВА РАЗРАБОТКИ

Многие программисты уже привыкли ко всем удобствам, которые предоставляют интегрированные средства разработки (IDE), например Borland Turbo Pascal или Microsoft Visual Basic и Visual C++, входящие в состав Microsoft Visual

Development Studio. Такие специалисты смогут почувствовать себя в знакомой среде, работая с программой MPLAB (рис. 2.6).

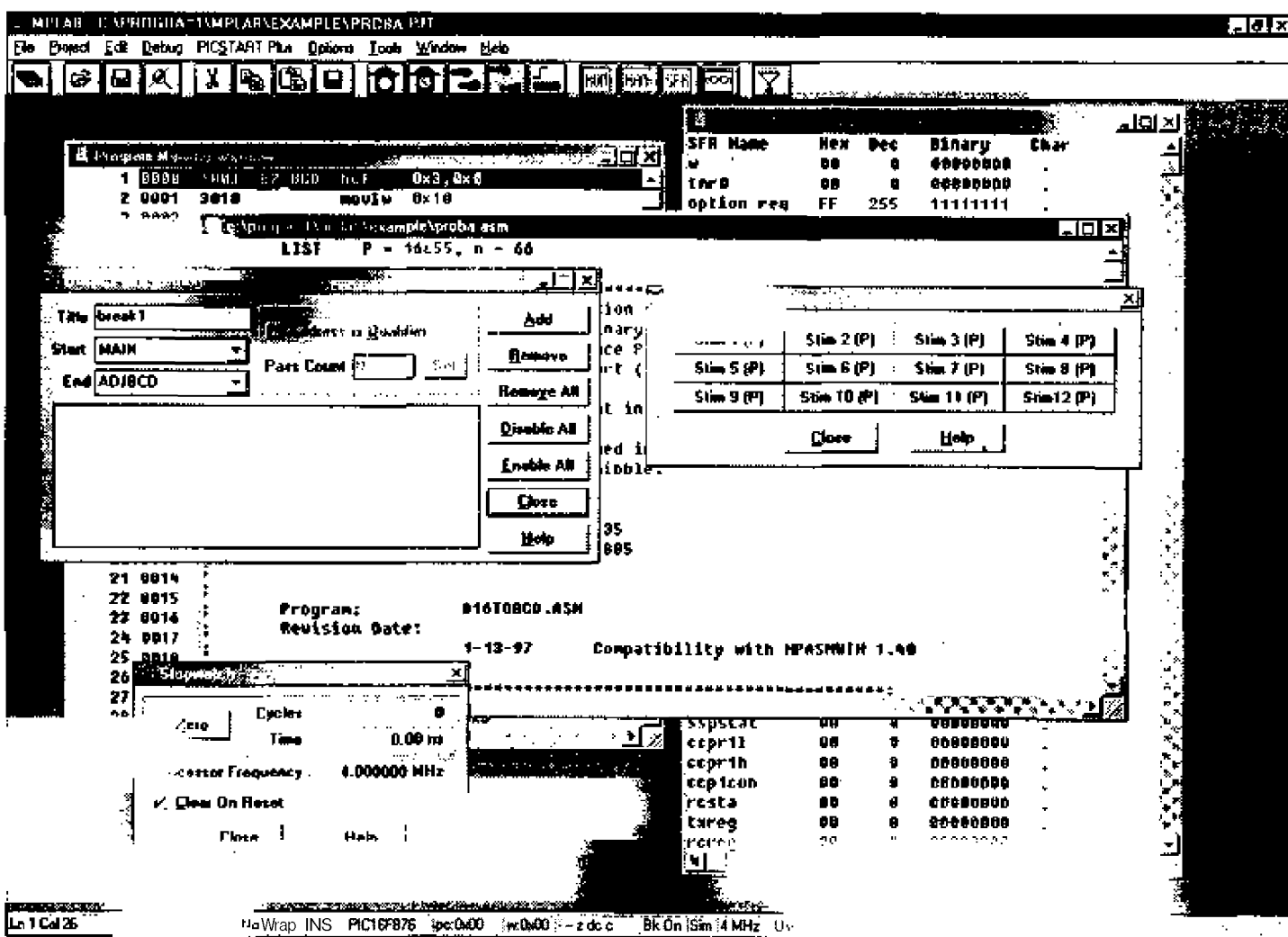


Рис. 2.6. Интерфейс MPLAB

Это средство разработки программ для микроконтроллеров фирмы Microchip включает в себя следующие инструменты:

- текстовый редактор (editor);
- ассемблер (assembler);
- компилятор (compiler);
- компоновщик (linker);
- симулятор (simulator);
- эмулятор (emulator);
- программатор (programmer).

Назначение любой интегрированной среды заключается в том, чтобы предоставить программисту весь спектр услуг, которые требуются на этапах разработки и отладки программы, избавив разработчика от необходимости обращаться к каким-либо другим инструментам и иметь дело с многочисленными файлами, образующимися при работе каждого из используемых инструментальных средств.

Кроме того, интегрированная среда разработки программ должна поддерживать возможности:

- добавления новых моделей микроконтроллеров;
- подключения других компиляторов языков высокого уровня;
- работы с различными программаторами и эмуляторами.

Некоторые (очень дорогие) средства разработки интегрируются с программами автоматизированного проектирования электронных схем, что позволяет разработчику в одной среде проектировать и отлаживать не только программную, но и аппаратную часть своего устройства.

В заключение отметим, что при выборе интегрированного средства разработки программ для микроконтроллеров следует обращать внимание на то, поддерживает ли встроенный редактор те же функции, что и обычные текстовые редакторы, такие как Блокнот или WordPad. Если это не так, то могут возникнуть затруднения при попытке вставить фрагмент исходного текста программы из другого файла (например, из HTML-страницы). Также необходимо убедиться, что формат выходного HEX-файла является стандартным и совместим с имеющимся программатором.

ГЛАВА 3

МИКРОКОНТРОЛЛЕРЫ PICMICRO

Уже более десяти лет я использую в своей работе микроконтроллеры PIC, изготовленные по технологии КМОП (комплементарные металло-оксидные-полупроводниковые схемы, CMOS). Мне знакомы и микроконтроллеры предыдущего поколения, изготовленные по технологии n-МОП (на МОП транзисторах с n-каналом, NMOS). В своей приверженности микроконтроллерам PIC я не одинок: эти микросхемы имеют достаточную производительность, легко программируются и весьма популярны среди как профессионалов, так и радиолюбителей.

Кроме того, широкий спектр различных микроконтроллеров PIC способен удовлетворить самые разнообразные запросы разработчиков, которым не составит большого труда найти для своего проекта самую подходящую модель, обладающую набором необходимых функций. Каждый год фирма Microchip объявляет о выпуске около сотни новых модификаций — с новыми функциями, заключенных в различные корпуса. Многие старые модели получают новую жизнь, подвергаются модернизации, улучшаются их характеристики и снижается стоимость.

Перечислим только самые основные особенности большинства микроконтроллеров PICmicro:

- возможность выбора различных конфигураций во время программирования;
- возможность работы с различными тактовыми генераторами (в том числе использование контура фазовой подстройки частоты, ФАПЧ);
- сопряжение с различными схемами формирования сигнала сброса (в том числе формирование сигнала сброса внутри микроконтроллера);
- широкий спектр возможностей ввода-вывода, в том числе:
 - ввод-вывод сигналов большой мощности;
 - выходы с открытым коллектором (ОК);
 - последовательный ввод-вывод с использованием кода NRZ (кодирование без возвращения к нулю);
 - синхронный последовательный порт ввода-вывода;
 - аналоговый вход с компараторами или АЦП;
- многократно программируемая встроенная память (Flash) для хранения программ и данных;
- встроенные таймеры-счетчики с возможностью настройки режимов их работы;
- одновекторная система прерываний, настраиваемая на работу с различными источниками запросов (некоторые микроконтроллеры PIC поддерживают несколько векторов прерываний).

Microchip была одной из первых фирм, которые начали производить недорогие микроконтроллеры и обеспечили масштабную поддержку своих клиентов. Многие микроконтроллеры PIC имеют возможность *внутрисхемного программирования* (In-Circuit Serial Programming, ICSP). Это означает, что нет необходимости вынимать программируемый микроконтроллер из целевой схемы и вставлять его в программатор.

Кроме того, фирма Microchip одной из первых начала использовать в своих микросхемах перепрограммируемые ПЗУ с электрическим стиранием (EEPROM). С тех пор окошко для стирания программ с помощью ультрафиолетовых лучей навсегда исчезло с корпуса микроконтроллеров.

На основе чипов PICmicro разрабатывается множество коммерческих устройств. На рынке микроконтроллеров Microchip находится на втором месте (после Motorola). Большое количество автоматических устройств разрабатывают с использованием плат Parallax Basic Stamp, которые основаны на микроконтроллерах PIC.

Для PICmicro доступна подробная документация (включающая многочисленные примеры использования этих микроконтроллеров при разработке различных автоматических устройств); имеется программное обеспечение для разработки и отладки программ; множество различных программаторов, симуляторов и эмуляторов. В Internet доступны тысячи ресурсов, посвященные различным аспектам проектирования устройств на основе микроконтроллеров PIC. Можно сказать, что эти микроконтроллеры практически так же распространены, как персональные компьютеры PC.

Есть, конечно, и недостатки, но их не так уж много. Некоторым разработчикам, привыкшим к процессорам принстонской архитектуры, не нравится, что для программирования микроконтроллеров PIC на языке низкого уровня они должны изучать особенности гарвардской архитектуры¹, на которой основаны микроконтроллеры Microchip. Кроме того, говорят, что эти микроконтроллеры несколько проигрывают другим по соотношению «цена/производительность». Но лично я считаю, что гарвардская архитектура лучше всего подходит для программирования автоматических устройств, а если говорить о цене и производительности, фирма Microchip не прекращает работу по улучшению своей продукции.

Начиная работу над этой книгой, я разрабатывал все примеры программ (в том числе приведенные в следующей главе) для чаще всего используемого на тот момент микроконтроллера PIC16F84. Главным доводом в пользу этого контроллера было то, что именно для него предназначался бесплатно распространяемый компилятор C фирмы HI-TECH Software; это позволяло разрабатывать программы на языке высокого уровня. Кроме того, PIC16F84 имел Flash-память и поддерживал возможность внутрисхемного программирования.

Но пока продолжалась работа над книгой, вышла новая версия компилятора PICC Lite, умеющая генерировать код для контроллера PIC 16F627. Этот контроллер наделен всеми чертами своего предшественника PIC16F84 и вдобавок к этому

¹ В процессорах гарвардской архитектуры разделены память команд и память данных. - Прим. перев.

имеет расширенные функции ввода-вывода, позволяющие существенно упростить разработку автоматических устройств. Поэтому все приведенные здесь программы и схемы устройств были без особого труда переориентированы на новый микроконтроллер.

В этой главе будет подробно описано функционирование микроконтроллера PIC16F627. Мне хотелось бы быть уверенным, что вы досконально изучите его устройство и особенности программирования, перед тем как приступить к разработке автоматических устройств на его основе.

Необходимо отметить также, что вместо зарегистрированного названия PICmicro для микроконтроллеров фирмы Microchip часто используют короткую аббревиатуру PIC (Peripheral Interface Controller - периферийный интерфейсный контроллер). Именно так назвала свое детище фирма General Instruments, от которой впоследствии отпочковалась Microchip; вот почему это название до сих пор не вполне корректно закрепляют за микроконтроллерами PICmicro.

3.1. ОСНОВНЫЕ ОСОБЕННОСТИ МИКРОКОНТРОЛЛЕРОВ PICMICRO

Имеется несколько различных семейств микроконтроллеров PIC, у каждого из которых свои характерные особенности. Микроконтроллеры (МК) разных семейств отличаются друг от друга программно доступными регистрами для работы с периферийными устройствами, организацией системы прерываний, наличием некоторых команд (например, операции умножения, выполняемой за один цикл работы процессора). Основные отличия перечислены в табл. 3.1.

Таблица 3. 1. Семейства микроконтроллеров PICmicro

Семейство	Разрядность команд, бит	Количество регистров*	Примечания
Младшее	12	32х4	1. Хорошо подходят для управления несложными периферийными устройствами 2. Не имеют системы прерываний 3. Имеют один таймер 4. Работают на частоте до 20 МГц 5. Имеют ограниченные возможности ввода-вывода 6. Не имеют Flash-памяти программ 7. Программируются только в параллельном режиме (исключение составляют PIC12C5xx и PIC16C505, поддерживающие возможность последовательного внутрисхемного программирования)
Среднее	14	128х4	1. Хороший выбор для устройств общего применения 2. Наиболее популярны среди остальных МК P/C 3. Имеют наибольшее количество модификаций 4. Одновекторная система прерываний 5. До трех таймеров

Таблица 3.1. Семейства микроконтроллеров PICmicro (окончание)

Семейство	Разрядность команд, бит	Количество регистров*	Примечания
			6. Работают на частоте до 20 МГц 7. Поддерживают расширенные возможности ввода-вывода (аналоговый ввод-вывод, драйвер жидкокристаллического дисплея, последовательный интерфейс и пр.) 8. В большинстве своем имеют Flash-память программ 9. Поддерживают внутрисхемное программирование в последовательном коде 10. Некоторые МК имеют встроенные средства пошаговой отладки программ
PIC17Cxx(x)	16	224x8, 48 регистров специальных функций (SFR)	1. Внешняя шина/параллельный порт 2. Архитектура, отличающаяся от остальных МК PICmicro 3. Расширенная система команд, индексная адресация 4. Небольшое количество модификаций 5. Несколько векторов прерываний 6. Три таймера 7. Работают на частоте до 33 МГц 8. Поддерживают расширенные возможности ввода-вывода 9. Не имеют Flash-памяти программ 10. Программируются только в параллельном режиме; поддерживают режим самопрограммирования (self-programming)
PIC18C/Fxx2	16	256x16	1. Улучшенная архитектура, основанная на архитектуре среднего семейства 2. Могут адресовать до 2 Мб памяти программ, до 4 Кб памяти данных 3. В скором времени полностью заменят МК среднего семейства 4. Расширенная система команд, индексная адресация 5. Один вектор прерывания, программирование приоритетов источников запроса 6. Расширенные возможности ввода-вывода, аналогичные МК среднего семейства 7. Работают на частоте до 40 МГц (тактовый генератор на 10 МГц и ФАПЧ для умножения частоты) 8. Имеют Flash-память программ 9. Поддерживают внутрисхемное программирование в последовательном коде и самопрограммирование (self-programming)

* Указано количество регистров в одном банке памяти и количество банков.

МК всех семейств выполняют каждую машинную команду за четыре такта. В большинстве случаев поддерживается тактовая частота до 20–33 МГц. Некоторые модели используют кольцо фазовой автоподстройки частоты (ФАПЧ) для умножения частоты тактового генератора в четыре раза.

В табл. 3.2 перечислены некоторые часто используемые микроконтроллеры PICmicro различных семейств.

Таблица 3.2. Микроконтроллеры PICmicro

Модель	Семейство	Основные особенности*	Области применения
12C5xx	Младшее	IOSCR	Простые интерфейсные устройства
12C6xx	Среднее	ADC, IOSCR, Data EEPROM	Простые интерфейсные устройства
14C000	Среднее	ADC	Управление питанием
16C5x	Младшее		Простые устройства
16C505	Младшее		Простые устройства
16HV540	Младшее	Регулятор напряжения	Простые устройства
16C55x	Среднее		Простые устройства
16C6x	Среднее		Простые цифровые устройства
16C62x	Среднее	VC	Аналоговые устройства
16F62x	Среднее	VC, FP	Аналоговые устройства
16C7x	Среднее	ADC	Аналоговые интерфейсы
16F8x	Среднее	FP	Разработка устройств на основе микроконтроллеров
16F87x	Среднее	ADC, FP	Аналоговые интерфейсы Разработка устройств
16C9xx	Среднее	ADC, PC	Аналоговые интерфейсы
17Cxx	Старшее	Внешняя память	Сложные устройства
18Cxxx	18Cxxx	ADC, PC	Сложные аналоговые/цифровые интерфейсные устройства

* Обозначения: IOSC - Internal OSC/Reset (встроенный тактовый генератор и схема формирования сигнала сброса); ADC - наличие АЦП; Data EEPROM - многократно программируемая память данных с электрическим стиранием; VC - Voltage Comparator (аналоговый вход с компаратором напряжения); FP - Flash Program Memory (Flash-память программ).

Заметим, что память программ в микроконтроллерах PICmicro измеряется не байтами, а количеством машинных команд, так как разрядность каждой команды составляет в МК различных семейств от 12 до 16 бит.

Система команд МК PICmicro разработана таким образом, что число различных команд, необходимых для программирования приложений, раза в три меньше, чем обычно требуется при использовании других микроконтроллеров.

3.2. ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРОГРАММ

Как уже говорилось, для МК PICmicro бесплатно доступно большое число различных инструментальных средств, в том числе интегрированная среда разработки программ MPLAB фирмы Microchip, а также компилятор языка высокого уровня PICC Lite фирмы HI-TECH Software. Используемое в книге ПО работает на платформе Win32 под управлением операционных систем Windows 95, Windows NT, Windows 2000, Windows ME, Windows XP фирмы Microsoft.

PICC Lite - это бесплатная версия компилятора языка C, разработанного фирмой HI-TECH Software и предназначенного специально для МК PICmicro. Хотя этот компилятор использует командную строку MS DOS, он легко интегрируется со средой разработки MPLAB, позволяя проводить отладку исходного кода в диалоговом режиме. Как будет показано в следующей главе, с помощью PICC Lite можно разрабатывать довольно сложные программы, не прибегая к использованию ассемблера. Краткое описание языка PICC Lite приводится в приложении.

Полная (не бесплатная) версия компилятора PICC может генерировать машинный код для любого микроконтроллера PICmicro и имеет для каждого из них специальные включаемые файлы. Что касается бесплатной версии PICC Lite, то она поддерживает пока только МК PIC16C84, PIC16F84, PIC16F84A и PIC16F627. Этот набор может показаться весьма скудным, но МК PIC16F84 благодаря наличию Flash-памяти программ уже давно традиционно используется в радиолюбительских конструкциях, а пришедший ему на смену PIC16F627 (равно как и PIC16F628, имеющий больший объем памяти программ) постепенно становится новым стандартным МК для разработки устройств на микроконтроллерах PICmicro.

PICC Lite совместим со стандартом ANSI/ISO C, но следует иметь в виду некоторые его особенности, важные при проектировании программ управления автоматическими устройствами на основе микроконтроллеров. Этот компилятор специально разрабатывался для совместного использования с интегрированной средой MPLAB и компоновщиком объектных модулей фирмы Microchip, чтобы сделать возможными компиляцию и отладку программ на уровне исходного текста, а также вставку низкоуровневых команд микроконтроллеров PIC в текст программ на языке C. Компилятор PICC Lite отводит для целых переменных типа `int` два байта памяти (это может показаться необычным тому, кто привык программировать 8-разрядные микроконтроллеры на языке низкого уровня, и не совсем согласуется с требованиями стандарта ANSI C, который предписывает в таком случае использовать разрядность, совпадающую с размером машинного слова). Однако большинство программистов уже привыкли, что данные типа `int` являются 16-разрядными. Поэтому для 8-разрядных переменных следует использовать привычный в таких случаях тип `char`. Все регистры микроконтроллеров PIC описаны в файле `pic.h` и доступны в программах на языке C.

Следует упомянуть о нескольких важных ограничениях компилятора PICC Lite. Из-за небольшого аппаратного стека микроконтроллеров PIC (глубиной не более 8 слов) в программах на языке C нельзя использовать *рекурсию*. Кроме того, недопустимы *повторно входимые функции*, которые могут вызываться из процедуры обработки прерывания, возникшего в момент выполнения этой же функции, но вызванной из главной программы. В большинстве случаев это не слишком сильное ограничение.

В данной книге я постараюсь не использовать возможность компоновщика генерировать HEX-файл из нескольких объектных модулей. Для простоты понимания логики работы приведенных примеров, а также для упрощения процесса отладки мы везде будем использовать для компиляции только один файл с исходным текстом программы. Но с помощью компилятора PICC Lite можно разрабатывать и сложные приложения, исходные тексты которых могут быть разбросаны по разным файлам.

Следует сказать, что возможность вставки ассемблерного кода, поддерживаемая компилятором PICC Lite, снижает необходимость применения нескольких объектных файлов при сборке будущего приложения. В нескольких примерах я буду использовать такие машинные команды, как `sleeri` или `clrwdt`. Выше уже говорилось о том, что машинный язык микроконтроллеров PICmicro несколько отличается от машинного языка процессоров принстонской архитектуры (в том числе и микроконтроллеров Motorola 68HC11).

Первоначальные версии представленных в книге программ разрабатывались для МК PIC16F84. И хотя он не имеет некоторых возможностей, обеспечиваемых более новым МК PIC16F627, но и для него с помощью компилятора PICC Lite удастся создавать достаточно простые и эффективные программы. После выпуска новой версии PICC Lite, способной работать с PIC16F627, все приведенные в книге программы были модифицированы для этого микроконтроллера.

Вот уже более пяти лет я использую в своей работе интегрированное инструментальное средство MPLAB, разработанное фирмой Microchip специально для своих микроконтроллеров. Новые версии этой программы, в которых исправлены замеченные ошибки, а также добавлена поддержка новых микроконтроллеров, можно бесплатно загрузить с сайта www.microchip.com.

Среда разработки MPLAB имеет следующие особенности:

- совместимый с Microsoft текстовый редактор;
- включаемые файлы с описаниями регистров микроконтроллеров PICmicro всех семейств и модификаций;
- встроенный ассемблер для всех микроконтроллеров PICmicro;
- компоновщик OBJ-файлов;
- возможность настройки интерфейса оболочки;
- встроенный симулятор (с ограниченной поддержкой функций ввода-вывода);
- интерфейс для внутрисхемного эмулятора MPLAB-ICE и PICMASTER;
- интерфейс отладчика для MPLAB-ICD (In-Circuit Debugger - внутрисхемный отладчик);
- программный интерфейс с программаторами PICSTART и PROMATE II.

Интегрированная среда MPLAB имеет интуитивно понятный интерфейс пользователя, привычный тем, кто знаком с аналогичным программным обеспечением фирмы Microsoft. Основные команды текстового редактора MPLAB приведены в табл. 3.3.

Таблица 3.3. Основные команды текстового редактора MPLAB

Комбинация клавиш	Команда
<i>t</i>	Переместить курсор на одну строку вверх
<i>↓</i>	Переместить курсор на одну строку вниз
<i>←</i>	Переместить курсор на один символ влево
<i>→</i>	Переместить курсор на один символ вправо
<i>Page Up</i>	Сдвинуть текст на одну экранную страницу вверх
<i>Page Down</i>	Сдвинуть текст на одну экранную страницу вниз
<i>Ctrl+←</i>	Поставить курсор перед первым символом текущего слова
<i>Ctrl+→</i>	Поставить курсор перед первым символом следующего слова
<i>Ctrl+↑</i>	Поставить курсор в начало просматриваемого текста
<i>Ctrl+↓</i>	Поставить курсор в конец просматриваемого текста
<i>Home</i>	Переместить курсор в начало текущей строки
<i>End</i>	Переместить курсор в конец текущей строки
<i>Ctrl+Home</i>	Поставить курсор перед первым символом файла
<i>Ctrl+End</i>	Переместить курсор в конец файла
<i>Shift+←</i>	Расширить выделение на один символ влево
<i>Shift+→</i>	Расширить выделение на один символ вправо
<i>Shift+↑</i>	Расширить выделение на одну строку вверх
<i>Shift+↓</i>	Расширить выделение на одну строку вниз
<i>Ctrl+Shift+←</i>	Расширить выделение на одно слово влево
<i>Ctrl+Shift+→</i>	Расширить выделение на одно слово вправо
<i>Ctrl+Z</i>	Отменить последнее изменение
<i>Ctrl+X</i>	Вырезать выделенный текст в буфер обмена
<i>Ctrl+C</i>	Скопировать выделенный текст в буфер обмена
<i>Ctrl+V</i>	Вставить текст из буфера обмена там, где стоит курсор
<i>F3</i>	Искать текст по образцу
<i>Ctrl+S</i>	Сохранить текущий файл
<i>Ctrl+O</i>	Открыть файл
<i>Ctrl+N</i>	Создать новый файл
<i>Ctrl+P</i>	Распечатать текущий файл или выделенный фрагмент

С программой MPLAB интегрируется большое число выпускаемых Microchip устройств, которые подключаются к персональному компьютеру. В течение нескольких лет я использовал одно из таких устройств - PISTART Plus. Другое устройство - внутрисхемный отладчик MPLAB ICD для микроконтроллеров PIC16F876 и PIC16F877 - обладает многими чертами внутрисхемного эмулятора, полностью интегрируется с MPLAB и существенно облегчает отладку программ.

Профессиональным разработчикам рекомендую программатор PROMATE II и внутрисхемный эмулятор MPLAB ICE. В отличие от многих других программаторов, PROMATE II может работать со всеми микроконтроллерами PICmicro, а эмулятор MPLAB ICE - наилучшее из всех аналогичных устройств, с которыми мне приходилось встречаться.

Интегрированная среда разработки MPLAB позволяет отображать состояние всех регистров микроконтроллеров PICmicro во время работы с симулятором. Все

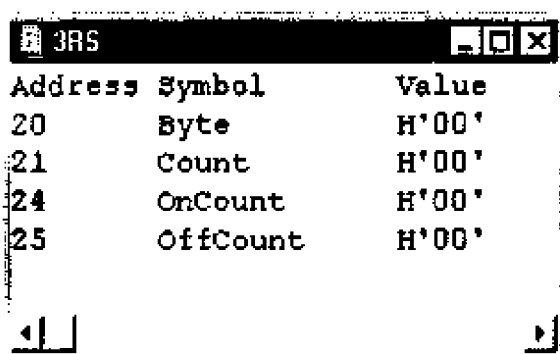


Рис. 3.1. Просмотр содержимого регистров и значений переменных в окне **Watch**

программы, приведенные в этой книге, перед загрузкой в микроконтроллер прошли предварительную проверку с использованием симулятора, встроенного в MPLAB. Для отображения содержимого регистров и переменных в MPLAB, как и в большинстве других интегрированных средств, предназначено специальное окно просмотра **Watch** (рис. 3.1).

Для отображения окна **Watch** необходимо воспользоваться главным меню MPLAB, выполнив команды **Window** $\Rightarrow$ **Watch Windows** $\Rightarrow$ **New Watch Window** (Окно $\Rightarrow$ Окна просмотра $\Rightarrow$ Новое окно просмотра).

После этого появится диалоговое окно выбора регистров. Нажав на кнопку **Properties** (Свойства), можно изменить формат отображающихся в окне **Watch** значений (см. рис. 3.2).

Окно **Watch** должно создаваться только после того, как приложение будет скомпилировано без ошибок. До этого список регистров и имен, определенных в программе, может быть недоступным для внесения добавлений.

Содержимое регистров может отображаться в шестнадцатеричном, десятичном и двоичном форматах, а также в виде символов ASCII. Размер данных может быть равным 1, 2, 3 или 4 байтам. Для значений, которые занимают в памяти несколько байтов, можно определить формат High:Low (сначала старший байт - как в процессорах и микроконтроллерах Motorola) или Low:High (сначала младший байт - как в процессорах Intel). Кроме того, в окне **Watch** можно отображать состояние отдельных двоичных разрядов заданных регистров.

В электронном приложении к этой книге (его можно скачать с сайта www.dmk.ru) в папке Code вместе с примерами программ имеются образцы настроек для окна **Watch**. Эти файлы имеют расширение .wat, и их можно загрузить в MPLAB, выполнив команды меню **Window** $\Rightarrow$ **Watch Windows** $\Rightarrow$ **Load Watch Window** (Окно $\Rightarrow$ Окна просмотра $\Rightarrow$ Загрузить окно просмотра). Файлы настроек представлены в текстовом виде, их формат достаточно прост, поэтому вы без труда изучите их содержимое с помощью любого тестового редактора. Однако я не рекомендую изменять текст конфигурационных файлов с помощью того же редактора, иначе в момент их загрузки может произойти ошибка, и программа MPLAB зависнет.

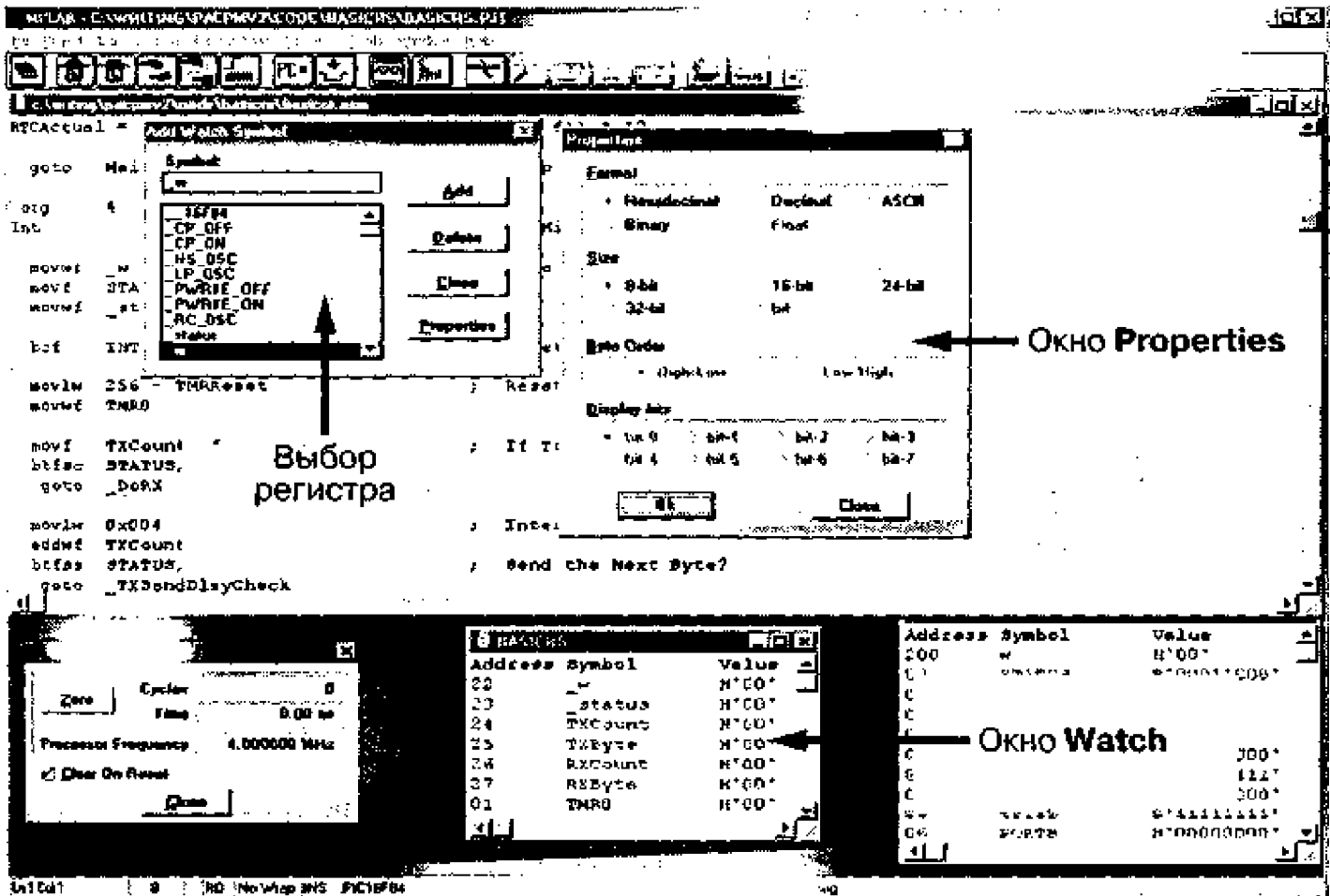


Рис. 3.2. Настройка окна Watch

Симулятор, встроенный в MPLAB, позволяет моделировать большинство команд микроконтроллеров PIC16F84 и PIC16F627. К сожалению, это не относится к некоторым операциям ввода-вывода. Например, в симуляторе недоступна работа с аналоговыми компараторами и блоком US ART (Universal Synchronous/Asynchronous Receiver/Transmitter-универсальный синхронный/асинхронный приемопередатчик). Позже мы обсудим способ, позволяющий в некоторых случаях обойти это ограничение.

Другой недостаток встроенного симулятора - его невысокое быстродействие. Он может выполнять от 1000 до 3000 машинных команд в секунду¹. Это значит, что моделирование каждой секунды работы реальной программы может занять до 5 мин. Если учесть, что выполнение кода верхнего (биологического) уровня может требовать нескольких секунд реального времени, такая медлительность симулятора причиняет заметные неудобства при отладке сложных программ.

Задать входные воздействия (stimulus - стимулы) при моделировании работы программы с помощью симулятора MPLAB можно пятью способами. Такое обилие возможностей, вероятно, обескуражит начинающего пользователя, особенно если учесть, что большинство других симуляторов предоставляет не более двух

¹ Скорость работы симулятора зависит от тактовой частоты процессора, а также от активности мыши во время его работы. — Прим. перев.

способов для выполнения этого действия. Но благодаря разнообразию средств программист может наиболее эффективно решать задачи по отладке своих приложений. В следующих главах мы подробнее обсудим различные аспекты отладки, а здесь изучим самые общие принципы работы со стимулами.

Наиболее универсальный метод определения входных воздействий основан на использовании диалогового окна **Asynchronous Stimulus** (асинхронные входные воздействия) - рис. 3.3.

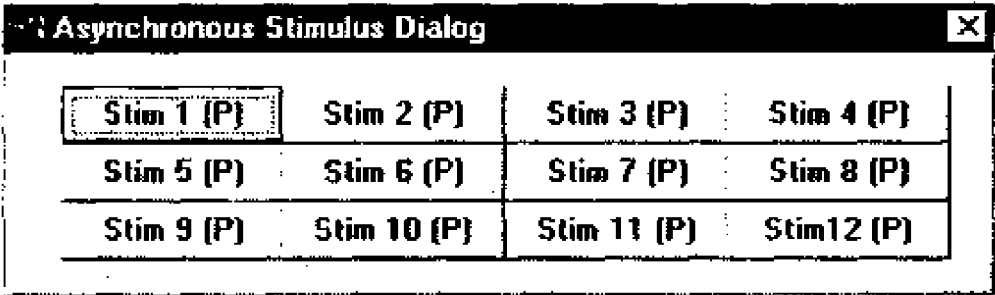


Рис. 3.3. Окно задания асинхронных входных воздействий

Это окно вызывается на экран в результате выполнения команд меню **Debug ⇒ Simulator Stimulus ⇒ Asynchronous Stimulus** (Отладка ⇒ Входные воздействия ⇒ Асинхронные стимулы). С помощью контекстного меню, вызываемого щелчком правой кнопки мыши, можно сопоставить любую из двенадцати кнопок этого диалогового окна конкретному выводу микроконтроллера, а также указать, как будет изменяться входной сигнал при нажатии на эту кнопку во время отладки приложения. Поддерживаются четыре режима: **Pulse** (Импульс), **Low** (Низкий), **High** (Высокий) и **Toggle** (Переключатель).

В режиме **Pulse** на соответствующем входе микроконтроллера будет сформирован одиночный импульс, то есть входной сигнал изменит свое значение на противоположное и снова вернется в исходное состояние в течение одного командного цикла работы процессора. Это удобно для тактирования таймера или генерации запроса на внешнее прерывание.

Режимы **Low** и **High** позволяют подать на вход сигнал соответственно низкого и высокого уровня. Таким образом, запрограммировав две кнопки диалогового окна **Asynchronous Stimulus** для работы с одним и тем же выводом микроконтроллера (одну - для высокого, а вторую - для низкого уровня), можно в любой момент задать значение входного сигнала на этом выводе.

Режим **Toggle** позволяет переключить состояние входного сигнала на противоположное.

Должен признаться, что мне не слишком нравится только что описанный способ определения входных воздействий. Наверное, потому, что при отладке своих программ я слишком часто забывал в нужный момент нажать нужную кнопку, а нередко нажимал кнопки в самые неподходящие моменты. Кроме того, приведенный способ задания стимулов раз в двадцать замедляет процесс отладки.

Более удобный метод указания входных воздействий основан на использовании специального текстового файла (*файла стимулов*, stimulus file). Каждая его

строка определяет состояние сигнала на любом выводе микроконтроллера в каждый момент времени.

Приведем простой пример файла стимулов:

! Пример файла стимулов

Step	RB0	RB1
1	0	0
100	0	1
200	1	1
300	1	0
400	0	0
500	0	1
600	1	1
700	1	0
800	1	0

Восклицательный знак ставится в начале комментария (весь текст справа от него до конца строки игнорируется). Следующая строка начинается директивой Step; за ней следуют имена выводов микроконтроллера, для которых мы хотим задать входные сигналы. В нашем примере это выводы RB0 и RB1. В общем случае могут быть заданы любые выводы, их имена всегда начинаются с буквы R, за которой следует имя порта (A, B, C и т.д.) и номер бита (от 0 до 7). Имя сигнала сброса - MCLR (Master Clear).

Далее для каждого значения счетчика машинных команд (1, 100, 200 и т.д.) задается соответствующее значение входного сигнала. В приведенном примере сигнал сброса не используется, поэтому после того как процесс моделирования дойдет до 800, оба вывода останутся в состоянии, определяемом последней строкой файла, и будут находиться в этом состоянии, пока не будет подан сигнал MCLR.

Для отображения числа выполненных машинных циклов можно вызвать на экран окно **Stopwatch**, выполнив с помощью меню MPLAB команды **Window** $\Rightarrow$ **Stopwatch** (Окно $\Rightarrow$ Окно останова) - рис. 3.4.

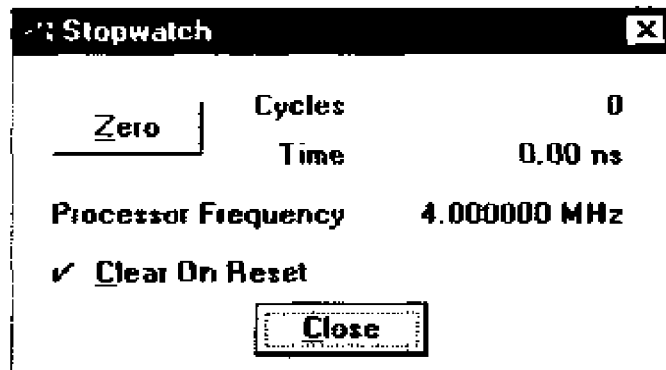


Рис. 3.4. Окно **Stopwatch**

Обычно я использую параметр **Clear on Reset** (Очистить при сбросе), предназначенный для автоматического сброса счетчика выполненных машинных команд при подаче сигнала сброса микроконтроллера. В любой момент счетчик может быть обнулен нажатием на кнопку **Zero** (Обнулить) в окне **Stopwatch**. Однако пользоваться этой возможностью следует осторожно. Не следует забывать, что после обнуления счетчика файл стимулов опять будет просматриваться с самого начала. Иногда это полезно для того, чтобы организовать цикл, заново повторяя последовательность всех воздействий, описанных в файле стимулов.

В окне **Stopwatch** отображается частота процессора; ее можно изменить, выполнив команды **Option ⇒ Development Mode ⇒ Clock** (Настройки ⇒ Режим разработки ⇒ Тактовая частота).

Для вычисления номера шага, который надо указать в файле стимулов, можно использовать простую формулу¹:

$$\text{Step} = t \times F / 4,$$

где F - заданная частота процессора; t - задержка во времени относительно момента начала выполнения программы (в секундах).

Например, при частоте 3,58 МГц задержка в 250 мкс соответствует значению счетчика машинных циклов:

$$\text{Step} = 250 \text{ мкс} \times 3,58 \text{ МГц} / 4 = 223,75,$$

что после округления дает 224. Именно это значение и надо указать в первой колонке файла стимулов.

Для подачи на входы микроконтроллера периодической последовательности прямоугольных (тактовых) импульсов надо с помощью главного меню MPLAB выполнить команды **Debug ⇒ Simulator Stimulus ⇒ Clock Stimulus** (Отладка ⇒ Входные воздействия ⇒ Тактовые импульсы). В результате на экране появится диалоговое окно (рис. 3.5), в котором можно будет выбрать нужный вывод (**Stimulus Pin**), длительность импульса высокого уровня (**High**) и длительность паузы между импульсами (**Low**). Период генерируемой последовательности прямоугольных импульсов будет равен сумме этих двух величин. При выборе параметра **Invert** (Инверсия) последовательность будет инвертирована, то есть сразу после подачи сигнала сброса генерируемый стимул примет низкий уровень. Как

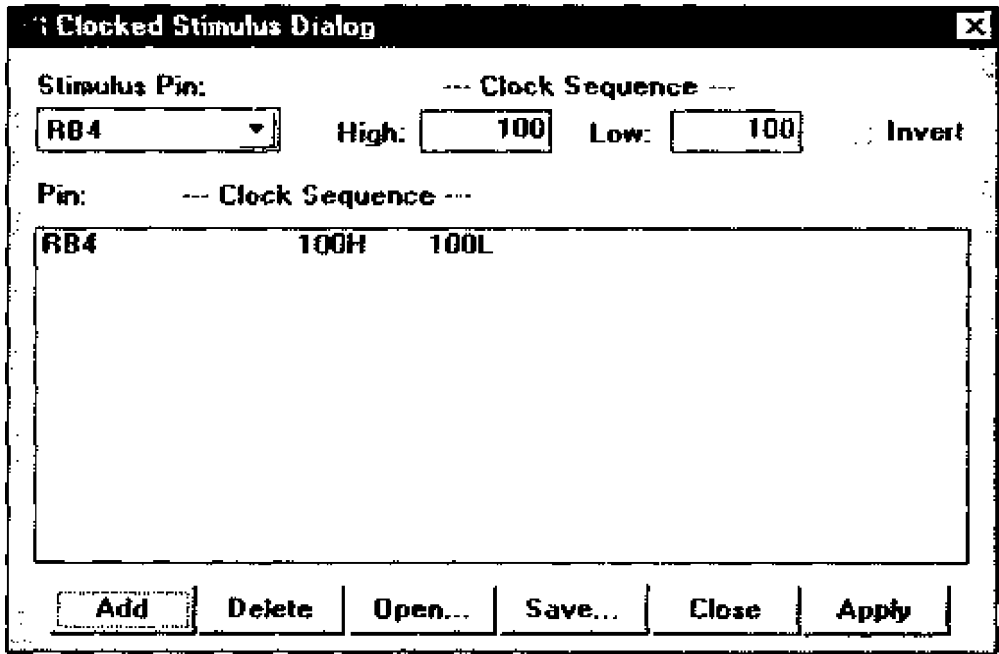


Рис. 3.5. Диалоговое окно для настройки подачи тактовых импульсов

¹ Деление на четыре необходимо, потому что микроконтроллеры PICmicro выполняют одну машинную команду за четыре периода работы тактового генератора. - Прим. перев.

и раньше, все промежутки времени указываются в циклах работы счетчика выполняемых команд.

Тактовые входные импульсы обычно используются для отладки функций по вводу-выводу, для подачи регулярно повторяющихся импульсов на входы микроконтроллера, а также для отладки реакций на внешние прерывания.

Как уже говорилось, с помощью симулятора невозможно провести моделирование некоторых сложных операций ввода-вывода. Этот недостаток может быть частично компенсирован с помощью так называемых *регистровых стимулов* (Register Stimulus). Программист может во время отладки задать в шестнадцатеричном коде значение любого регистра, выполнив команды меню Debug ⇒ Simulator Stimulus ⇒ Register Stimulus ⇒ Enable (Отладка ⇒ Входные воздействия ⇒ Регистровые стимулы ⇒ Включить) - рис. 3.6.

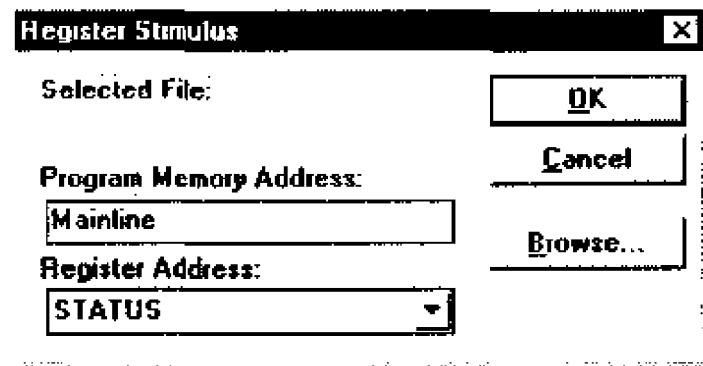


Рис. 3.6. Выбор значений регистров во время отладки

В появившемся диалоговом окне необходимо выбрать адрес команды или метку программы, которая будет определять момент подачи стимула, а также указать имя или адрес регистра, которому будет принудительно присвоено заданное значение. После этого, нажав на кнопку **Browse...**, надо выбрать заранее подготовленный файл, в котором определены нужные значения в шестнадцатеричном коде. Это простой текстовый файл; каждая его строка должна содержать две шестнадцатеричные цифры, например:

```
00
03
05
09
12
1C
1A
06
2E
```

Никакие комментарии или многобайтные значения здесь не допускаются. Теперь при отладке программы каждый раз, когда выполнение будет доходить до указанного адреса (или символьной метки), в заданный регистр будет записываться очередное значение, взятое из этого файла. Таким способом можно задать значение только одного регистра.

Регистровые стимулы удобно использовать для моделирования сложных операций ввода-вывода, которые не поддерживает встроенный симулятор MPLAB, в частности для отладки программы, использующей USART (этот пример будет рассмотрен позже).

Для изменения значений сразу нескольких регистров можно воспользоваться окном Register Modify (Модификация содержимого регистров), показанным на

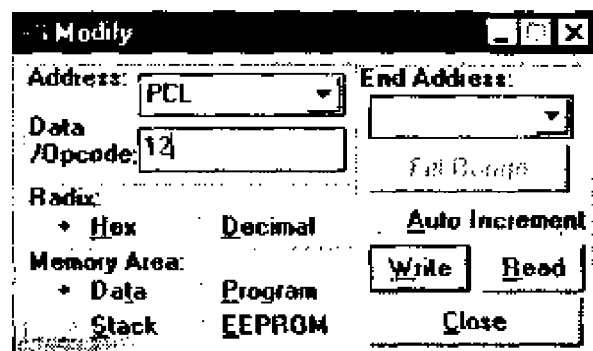


Рис. 3.7. Окно Register Modify
для изменения содержимого регистров

рис. 3.7, которое можно вызвать на экран с помощью команд меню **Window ⇒ Modify** (Окно ⇒ Модификация).

Таким способом можно изменить значения любых регистров, в том числе и рабочего регистра.

Все пять описанных способов моделирования входных воздействий могут использоваться совместно.

Дополнительное удобство, предоставляемое интегрированной средой MPLAB, заключается в том, что при определении вход-

ных воздействий все регистры и выходы микроконтроллера доступны по именам, а вместо абсолютных адресов программы можно указывать символьные метки.

Теперь пора обсудить использование компилятора PICC Lite. Этот инструмент разработчика хорошо интегрируется со средой MPLAB, и вы удивитесь, как быстро мы сможем создавать с его помощью работающие программы. Есть вы знакомы с языком C, то, безусловно, оцените и такую приятную особенность PICC Lite, как подробные сообщения, выдаваемые во время работы компилятора. Например, предупреждение (warning) выдается при попытке использовать конструкцию

```
if (variable = constant)
```

вместо правильного оператора

```
if (variable == constant)
```

Рассмотрим пример работы с компилятором в среде MPLAB. Для того чтобы проиллюстрировать весь цикл разработки приложения, в том числе и процесс отладки с помощью встроенного симулятора, сделаем в нашей программе на языке C несколько ошибок, как синтаксических, так и смысловых. Ошибки первого типа мы устраним после того, как увидим сообщение компилятора, а для исправления вторых потребуется провести моделирование работы программы в симуляторе.

Рассматриваемый ниже текст программы содержится в электронном приложении к этой книге, которое вы найдете на сайте www.dmk.ru. в файле example.c (папка Code\Example):

```
#include <pic.h>

// 31 марта 2002 - Myke Predko.
//
// Пример программы на языке C.
//
// Программа содержит синтаксические и смысловые ошибки.
//
// Используется микроконтроллер PIC16F84, частота тактирования 4 МГц.
// Младший бит порта В (обозначается RBO) переключается
// с максимально возможной частотой.
//
// Глобальные переменные:
//
// Слово конфигурации:
__CONFIG(0x03FF1);
```



```

// XT генератор тактовых импульсов.
// Включить PWRT.
// Выключить сторожевой таймер.
// Выключить защиту кода программы.

// Главная программа:
void main(void)
{
    while (1 == 1)
        RB0 = RB1 " 1; // Инверсия бита RB0.
} // Конец главной программы.

```

Запустите **MPLAB**. В нижней строке окна программы отображается название микроконтроллера, для которого в данный момент разрабатывается программа. Для нашего примера необходим **PIC16F84**. Если вы видите другое имя, выполните команду меню **Options** $\Rightarrow$ **Development Mode** (Настройки $\Rightarrow$ Режим разработки) и укажите нужную модель МК, открыв вкладку **Tools** (Инструменты).

После этого можно создать новый проект, выполнив команды **Project** $\Rightarrow$ **New Project** (Проект $\Rightarrow$ Новый проект) - рис. 3.8.

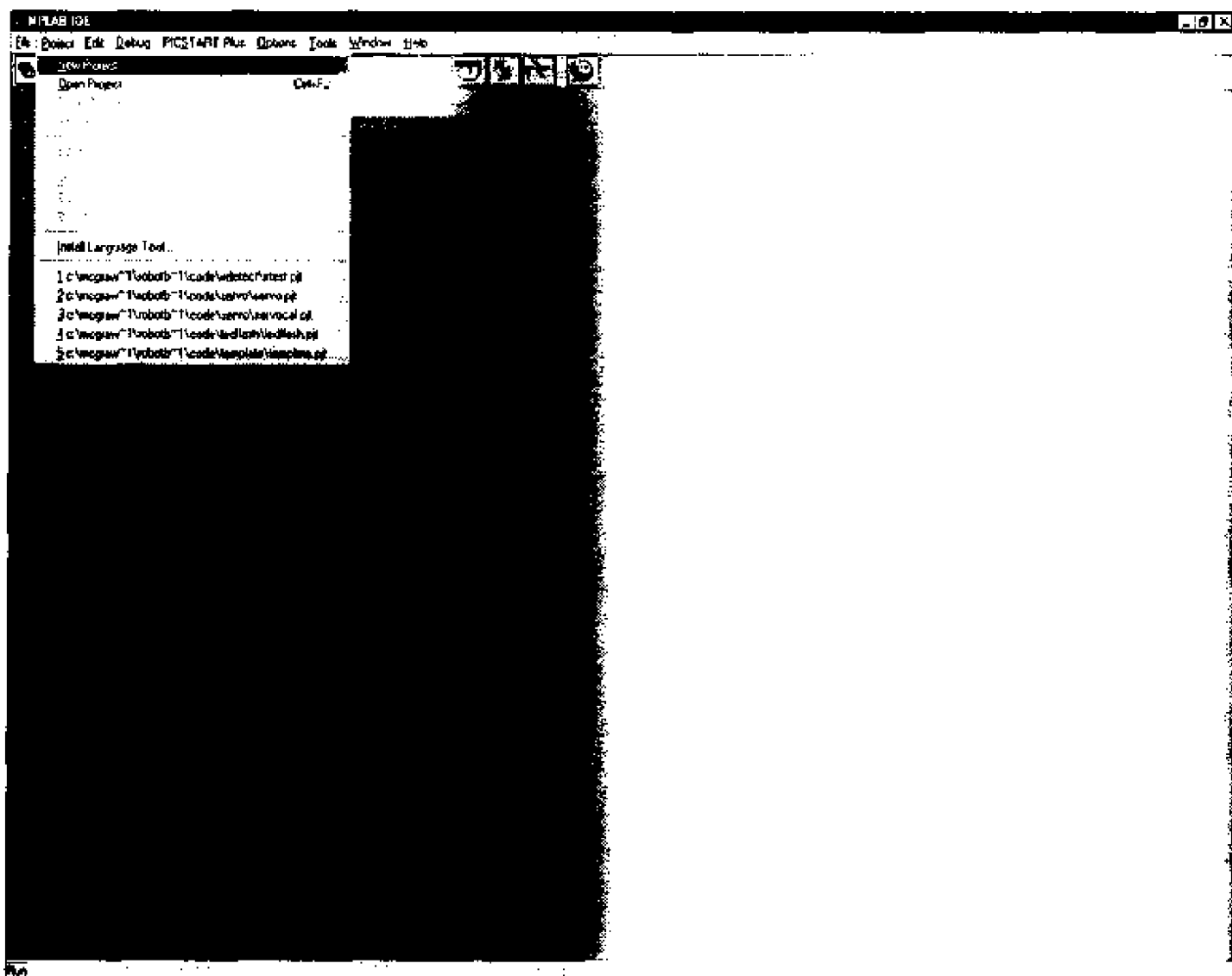


Рис. 3.8. Начало создания нового проекта в **MPLAB**

В появившемся окне (рис. 3.9) необходимо задать имя будущего проекта, а также указать диск и каталог для размещения файлов. В примере на рис. 3.10 выбраны имя `example.pjt` и каталог `Code\Example` на диске `C:`.

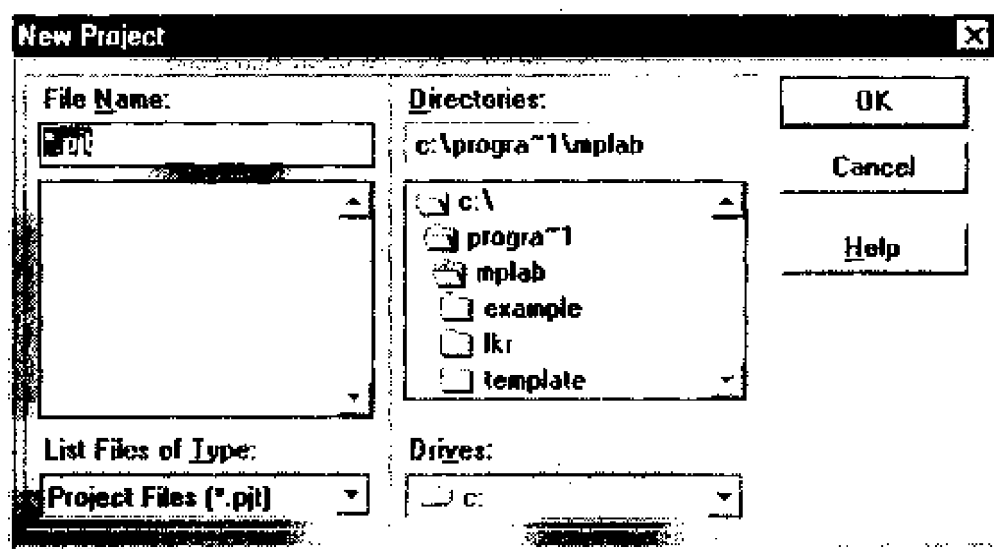


Рис. 3.9. Диалоговое окно создания нового проекта

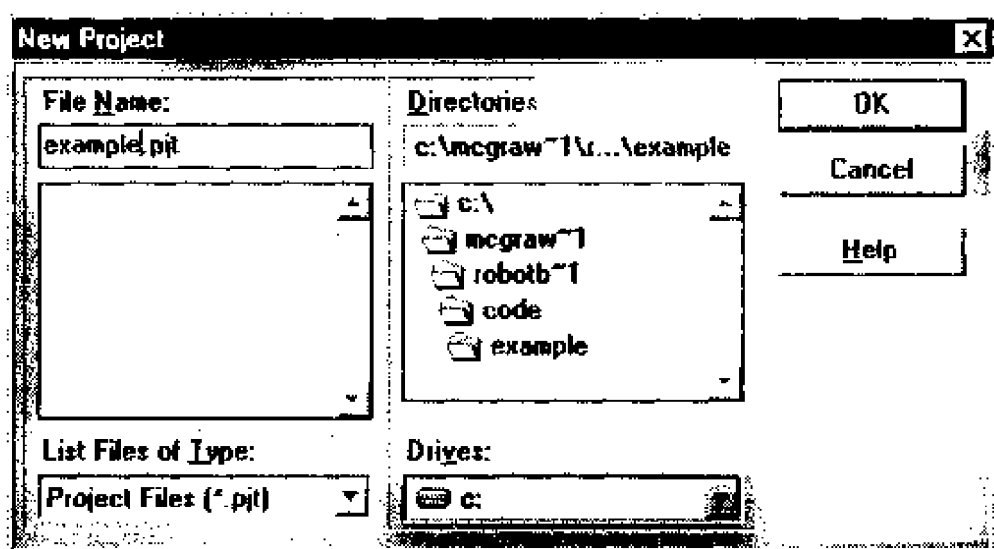


Рис. 3.10. Выбор имени и места размещения проекта

После нажатия кнопки **ОК** появится диалоговое окно **Edit Project** (Редактировать свойства проекта) для настройки параметров проекта. По умолчанию MPLAB считает, что приложение будет написано на ассемблере, поэтому в поле **Language Tool Suite** (Инструментальное средство) значится Microchip. Необходимо выбрать HI-TECH PICC Lite (рис. 3.11).

Затем необходимо выделить с помощью мыши целевой файл `example.hex` и нажать кнопку **Node Properties...** (Свойства) для установки необходимых режимов компиляции (рис. 3.12).

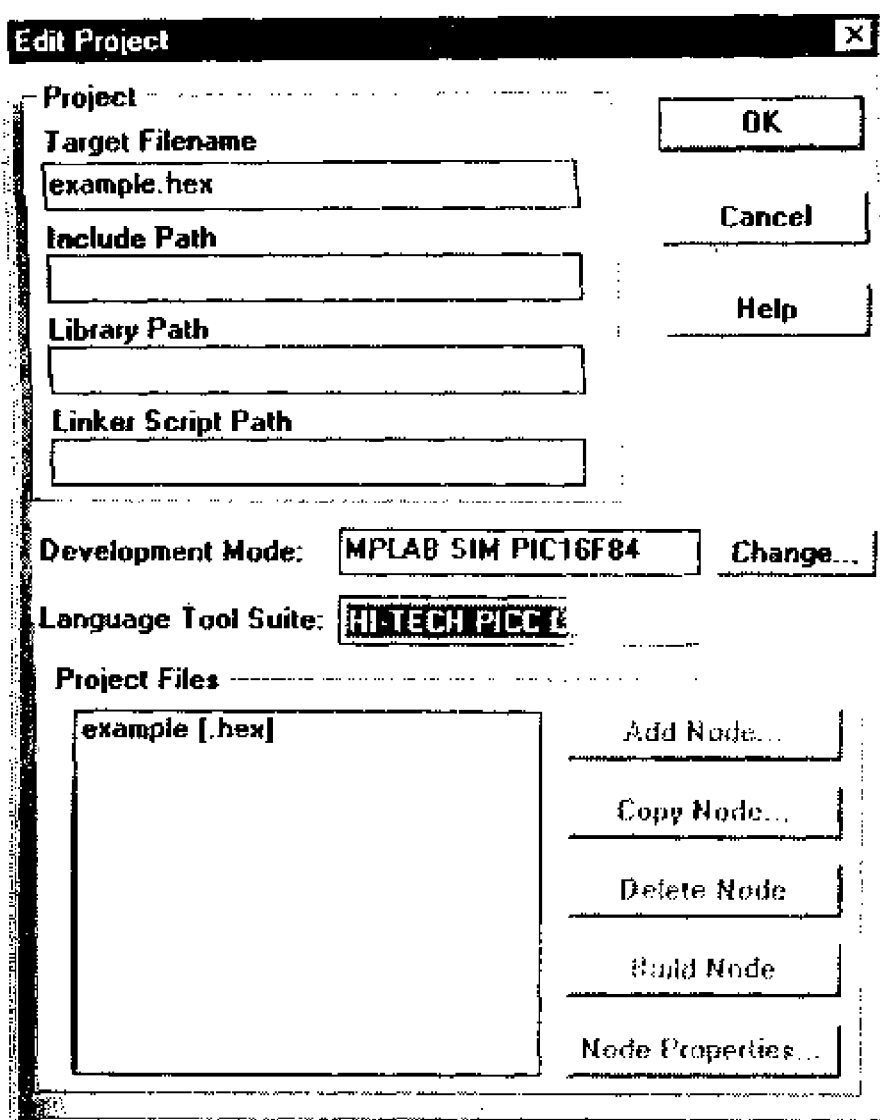


Рис. 3.11. Настройка параметров проекта

Выберем следующие параметры:

- **Information Message** - Quiet (Выключить режим расширенных сообщений);
- **Generate debug info** - On (Генерировать информацию для отладчика);
- **Assembler Optimization** - On (Включить оптимизацию ассемблерного кода);
- **Global Optimization** - On (Включить общую оптимизацию);
- **Floating point for double** - 24-bit (24-разрядные вещественные значения);
- **Error file** - On (Генерировать файл ошибок);
- **Append Errors to file** - On (Ошибки записывать в конец файла).

По окончании настройки нажмите кнопку **OK**. В результате вы вернетесь в окно **Edit Project**, где теперь надо нажать кнопку **Add Node** (Добавить файл к проекту), а затем указать файл с исходным текстом программы. В примере на рис. 3.13 выбран файл `example.c`.

Для правильной работы MPLAB необходимо, чтобы все файлы, подключаемые к проекту, находились в той папке, которая была задана при создании данного

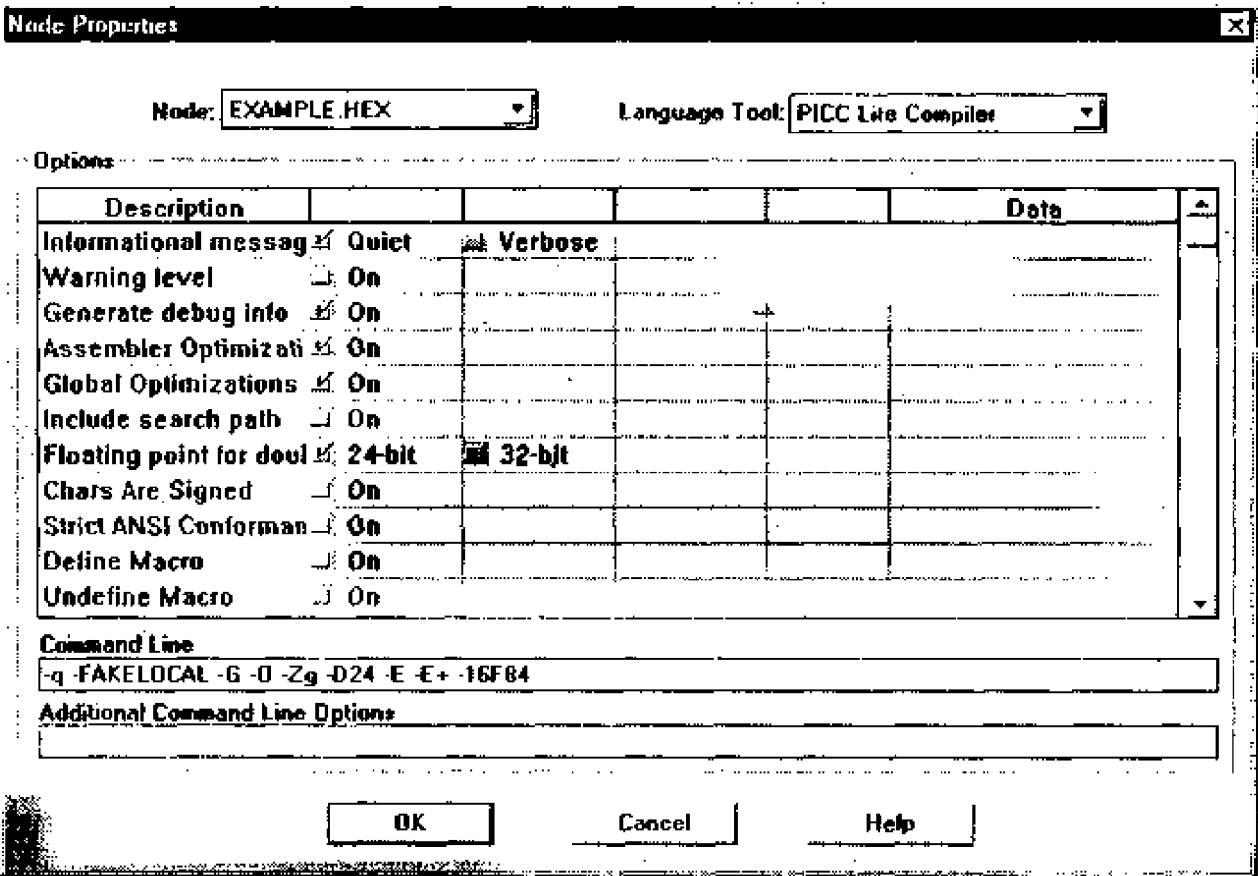


Рис. 3.12. Выбор режимов компиляции в окне *Node Properties*

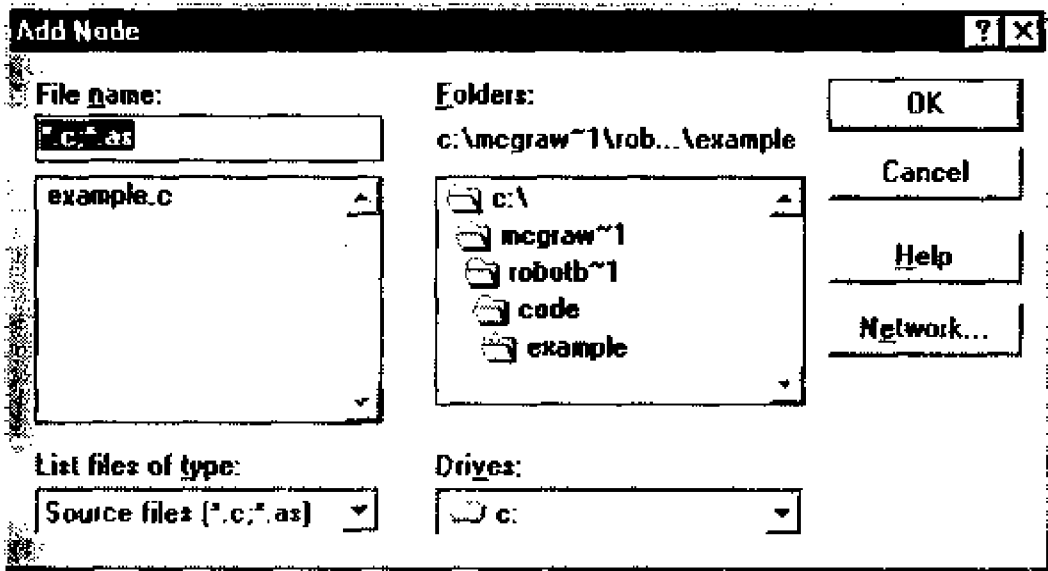


Рис. 3.13. Выбор файла с исходным текстом программы на языке C

проекта (см. рис. 3.10). В противном случае будет выдано предупреждающее сообщение.

После выбора файла с исходным текстом программы и нажатия кнопки **ОК** требуется подтвердить заданные параметры, еще раз нажав **ОК** в диалоговом окне **Edit Project**. Окно программы MPLAB примет первоначальный вид (см. рис. 3.8).

Теперь можно просмотреть или отредактировать исходный текст программы, выполнив команду меню **File ⇒ Open** (Файл ⇒ Открыть). После выбора файла `example.c` окно MPLAB примет вид, показанный на рис. 3.14.

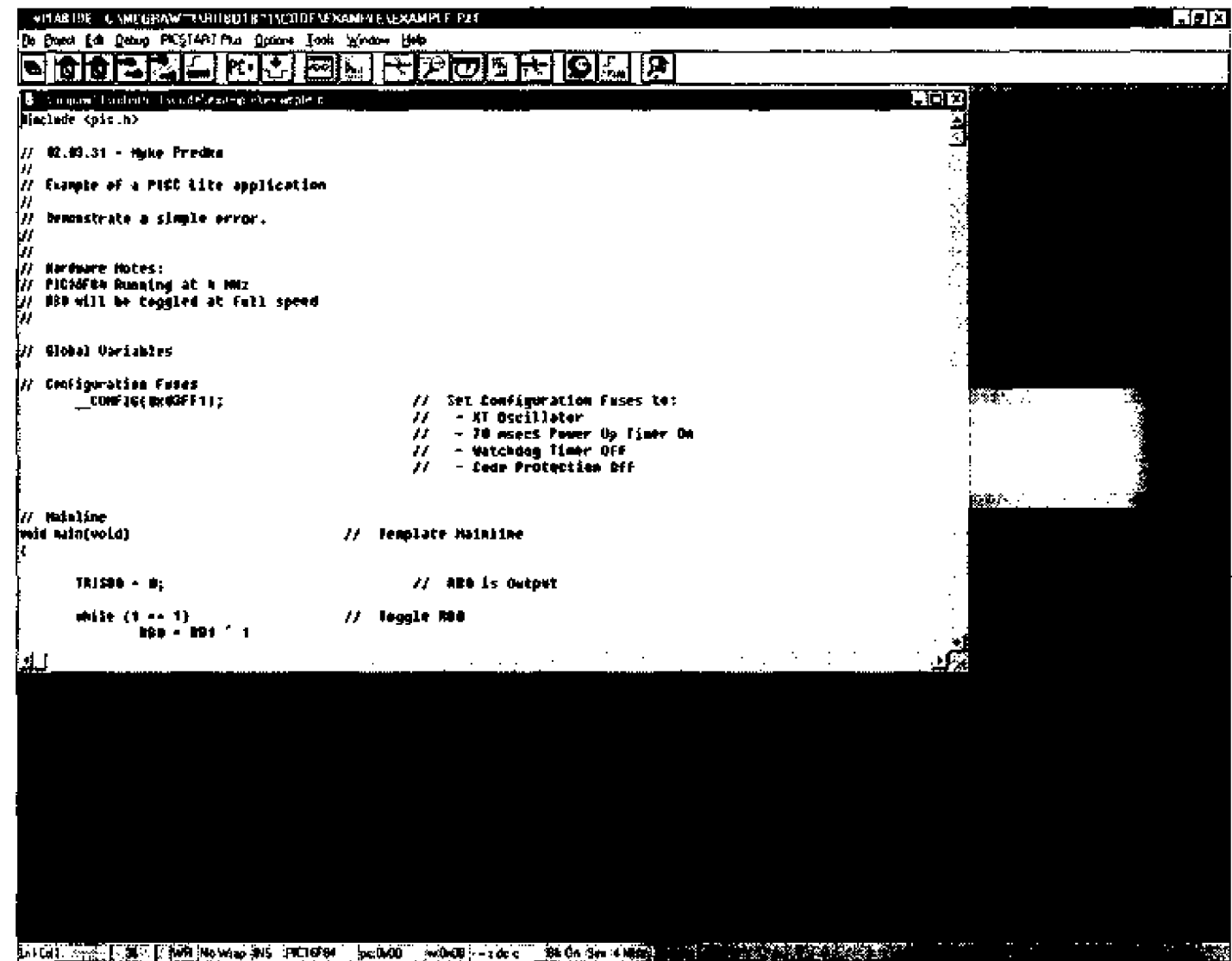


Рис. 3.14. Редактирование исходного текста программы на языке C

Для того чтобы выполнить компиляцию приложения, нажмите клавиши **Ctrl+F10** или выберите команды меню **Project ⇒ Build All** (Проект ⇒ Компилировать все). В результате будет вызван компилятор **PICC Lite¹**. Наша программа содержит синтаксическую ошибку - отсутствует точка с запятой после оператора

```
RB0 = RB0 ^ 1
```

¹ Если будет выдано сообщение «Unable to find build tool» (Не могу найти компилятор), необходимо выполнить команду меню **Project ⇒ Install Language Tool** (Проект ⇒ Инсталлировать инструментальное средство), после чего указать в поле **Language Suit** (Инструментальное средство) компилятор **HI-TECH PICC Lite**, в поле **Tool Name** (Имя инструментального средства) - **PICC Lite Compiler**, а в поле **Executable** (Имя исполняемого файла) - путь к исполняемому файлу **PICL.EXE**. Аналогично следует задать путь к компоновщику **PICC Lite Linker** и имя соответствующего исполняемого файла **HLINK.EXE**. — Прим. перев.

Поэтому компилятор выдаст сообщение

```
Error[000] C:\ROBOT\CODE\EXAMPLE\EXAMPLE.C 32 : ; expected
```

Здесь 32 - это номер строки, в которой обнаружена ошибка. Часто бывает (как и в нашем случае), что ошибка на самом деле содержится в предыдущей строке. Двойной щелчок мыши по строке сообщения об ошибке позволяет переключиться в окно исходного текста программы, причем курсор будет указывать то место, где она замечена.

Добавим точку с запятой. Заодно исправим еще одну ошибку: укажем нулевой бит регистра В вместо первого, чтобы получилось

```
RBO = RBO " 1;
```

Сохраним изменения и снова выполним компиляцию — теперь она должна пройти без ошибок. Будут выведены на экран следующие обычные в таких случаях сообщения:

```
Memory Usage Map:
```

```
Program ROM   $0000 - $0005  $0006 (      6) words
Program ROM   $03FC - $03FF  $0004 (      4) words
               $000A (     10) words total Program ROM
```

```
Config Data   $2007 - $2007  $0001 (      1) words total Config Data
```

```
Program statistics:
```

```
Total ROM used      10 words (1.0%)
```

```
Total RAM used       0 bytes (0.0%)
```

```
Build completed successfully.
```

После успешной компиляции можно закрыть окно **Build Result** (Результаты компиляции).

Теперь все готово для запуска симулятора. Выполните команды меню **Window** ⇒ **Watch Windows** ⇒ **Load Watch Window** (Окно ⇒ Окна просмотра ⇒ Загрузить окно просмотра) и укажите файл Code\Procwat\PIC16F84.WAT. Появившееся в результате окно просмотра можно растянуть, чтобы в нем отображались одновременно все указанные регистры, и перетащить на любое удобное место, чтобы окно не мешало отладке программы (рис 3.15). Нажатием на самую левую кнопку панели инструментов MPLAB можно быстро переключать эту панель в режим отладки и обратно. Название текущего режима отображается в строке состояния в правом нижнем углу.

Переключитесь в режим **Debug**.

Нажмите кнопку **Reset Processor** (Сброс процессора), чтобы осуществить сброс микроконтроллера (названия кнопок отображаются в строке состояния). Для сброса можно также использовать клавишу F6. В строке состояния можно узнать адрес текущей команды, то есть содержимое счетчика команд (PC - Program Counter). После сброса он хранит значение 0x03FC. Это адрес начала

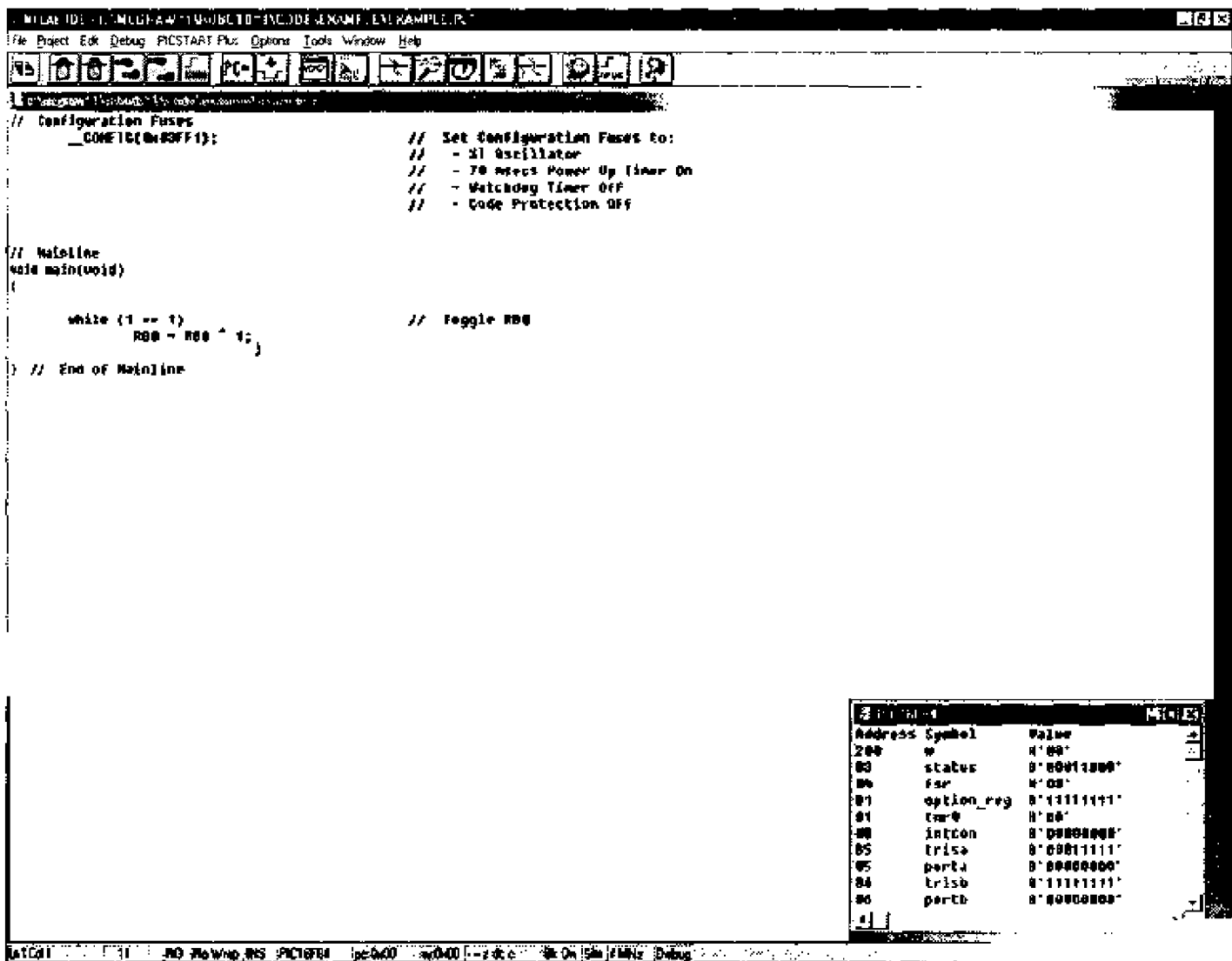


Рис. 3.15. Проект готов к отладке с помощью встроенного симулятора MPLAB

программы. Первой исполняемой строкой нашей программы является та, в которую мы не так давно внесли небольшие исправления. Именно эта строка будет сейчас выделена черным цветом (рис. 3.16).

Теперь можно использовать команды **Step** или **Step Over** (соответствующие клавиши - F7 и F8), чтобы выполнять программу по шагам. Однако может показаться, что при этом ничего не происходит. Подсвеченной остается та же строка исходного текста программы, содержимое окна просмотра не изменяется.

Как мы узнаем чуть позже в данной главе, для управления выполнением операций ввода-вывода используется специальный регистр TRIS (tri-state buffer enable). Такой регистр существует для каждого порта ввода-вывода, имеющегося в контроллере. Каждый его бит после сброса по умолчанию устанавливается в 1. Это означает, что все линии соответствующего порта используются для *ввода* информации. Нам необходимо сконфигурировать младший бит порта В для работы в режиме *вывода*, поэтому придется сбросить младший бит регистра TRISB

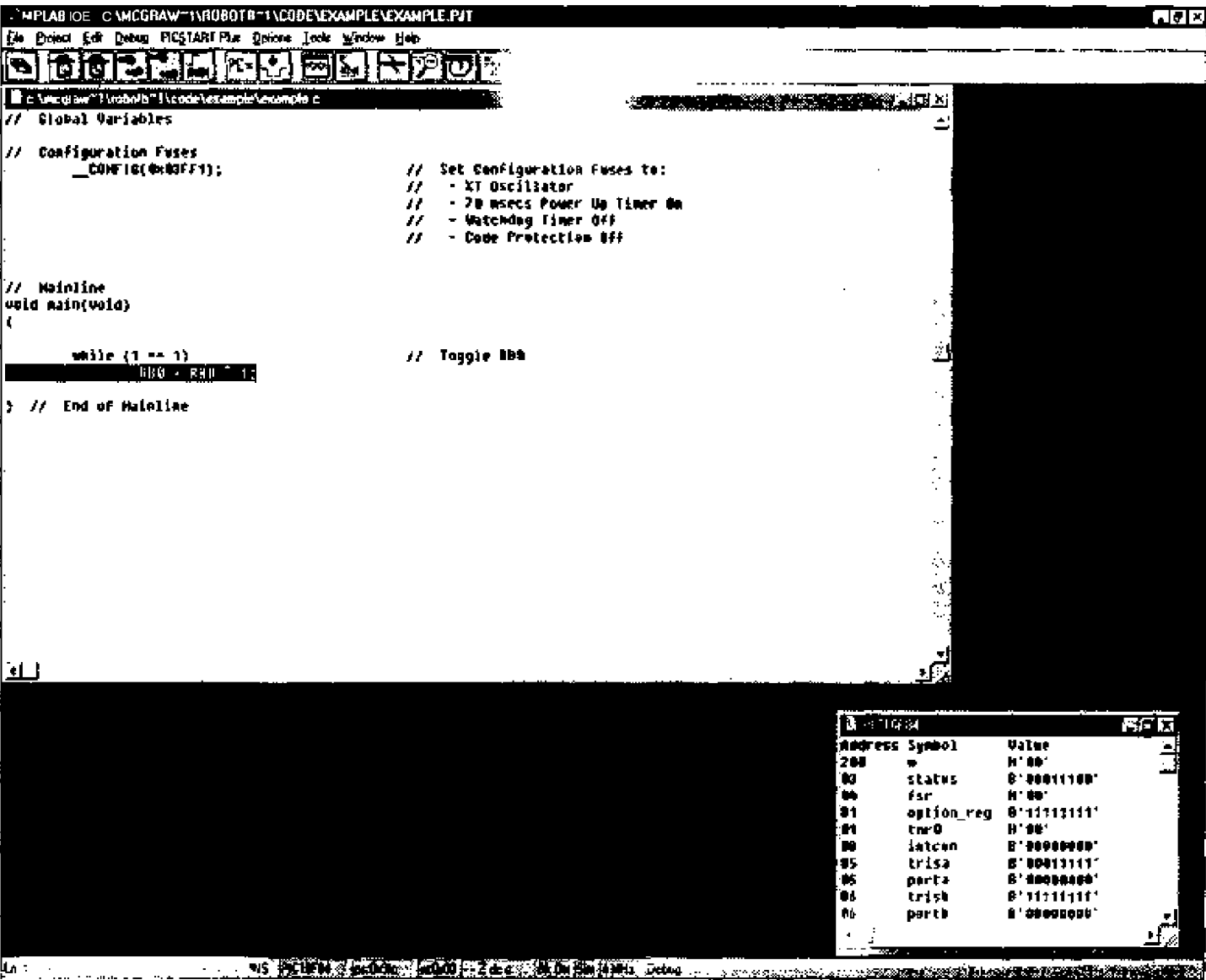


Рис. 3.16. Окно симулятора после подачи сигнала сброса

в нулевое состояние. Для этого добавим к нашей программе (перед оператором бесконечного цикла) строку

```
TRISBO = 0;
```

Снова выполним компиляцию (**Ctrl+F10**), подадим сигнал сброса (F6) и выполним несколько шагов отладки (F7 или F8). Теперь все в порядке: содержимое регистров в окне просмотра с каждым шагом изменяется, для привлечения внимания изменившиеся значения выделяются красным цветом (рис. 3.17).

Убедитесь, что состояние младшего бита регистра PORTB меняется на каждом шаге отладки. При этом также будет «мигать» второй (считая с нуля)¹ бит регистра STATUS.

Надеюсь, что после такого элементарного упражнения вы без труда сможете использовать компилятор и отладчик для создания приложений на языке C.

¹ Напомним, что биты нумеруются справа налево. Во втором разряде регистра STATUS находится признак нулевого результата. - Прим. перев.

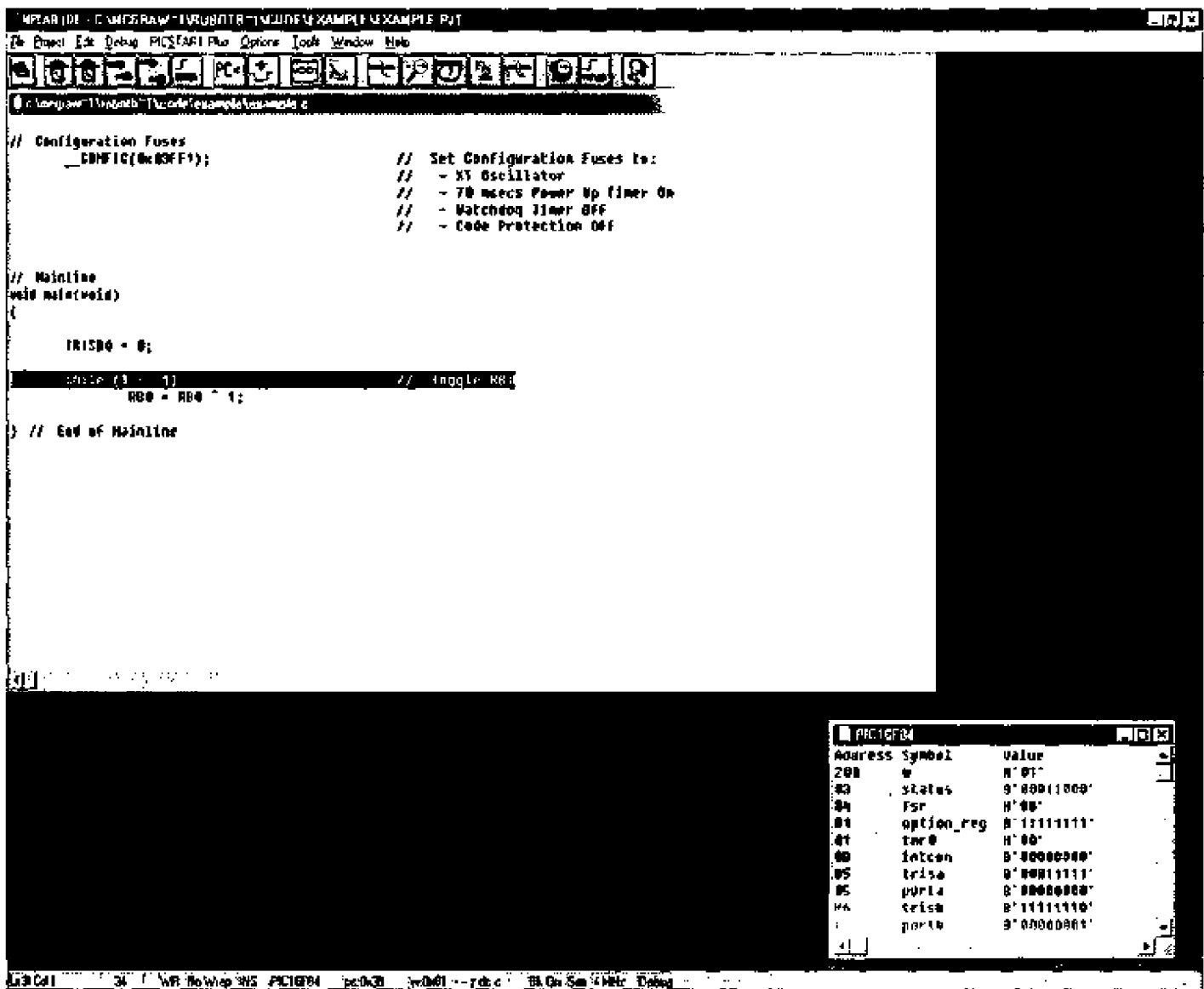


Рис. 3.17. Процесс отладки приложения

3.3. ПРОСТЫЕ СХЕМЫ

В своей книге «Handbook of Microcontrollers»¹ я утверждал, что если какой-либо микроконтроллер требует напряжения питания более +5 В, а также применения сложных внешних схем сброса и тактирования, лучше отказаться от использования такого МК. И сейчас готов повторить то же самое - наша жизнь слишком коротка, чтобы растрачивать ее на проектирование специальных схем, которые должны обеспечивать работу микроконтроллеров.

Это тем более верно, если учесть, что в распоряжении разработчика имеется МК PIC16F627 с большим числом встроенных функций, при использовании которого можно полностью сосредоточиться на процессе разработки программы и не тратить усилия на подключение дополнительных схем.

¹ Русский перевод: Предко М. Руководство по микроконтроллерам. В 2-х тт. - М.: Постмаркет, 2001. - Прим. перев.

Микроконтроллер PIC16F627 весьма неприхотлив, его работоспособность сохраняется при изменении напряжения питания от 2,5 до 6 В. Некоторые модели работают даже при 2,5 В. Но, как правило, разумно использовать в проектах источник +5 В, чтобы избежать возможных проблем с питанием других микросхем.

Для обеспечения такого питания я обычно применяю микросхему 78(L)05¹ или стабилитрон на 5,1 В. Пример включения микросхемы 7805 показан на рис. 3.18.

Простой стабилизатор напряжения на основе стабилитрона представлен на рис. 3.19.

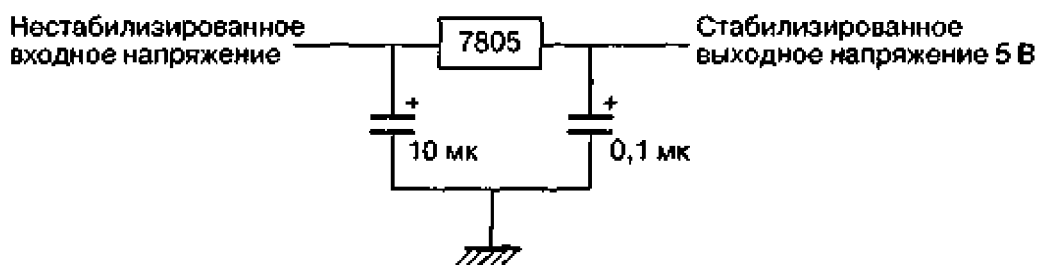


Рис. 3.18. Использование микросхемы 7805 в качестве источника питания микроконтроллера



Рис. 3.19. Источник питания на стабилитроне

Для выбора номинала дополнительного резистора можно использовать формулу

$$R = (U_{\text{вх}} - U_{\text{ст}}) / I_{\text{вых}}.$$

Стабилитрон должен иметь максимальную рассеиваемую мощность более $U_{\text{ст}} I_{\text{вых}}$.

Кроме источника питания для работы микроконтроллера необходимо сформировать сигнал начального сброса (RESET). Он должен генерироваться каждый раз при включении источника питания, а также в случае, если надо заново начать выполнение программы. Микроконтроллеры PICmicro поддерживают несколько способов формирования сигнала сброса. Сигнал RESET всегда подается низким уровнем на вход MCLR . При этом сбрасываются все таймеры, регистры ввода-вывода и программный счетчик PC.

¹ Отечественный аналог КР142ЕН5А. - Прим. перев.

Сброс может быть сформирован источниками трех типов: внешними, внутренними и детектором падения напряжения питания (brown-out detection - BOD).

Внутренний источник сигнала сброса, имеющийся у многих микроконтроллеров PICmicro, срабатывает во время включения источника питания, когда напряжение питания достигает нужного уровня. Если такая возможность отсутствует, можно использовать простую внешнюю схему, показанную на рис. 3.20. Она не только подает сигнал низкого уровня на вход `_MCLR`, но также может обеспечить ручной сброс при нажатии кнопки **Reset**.

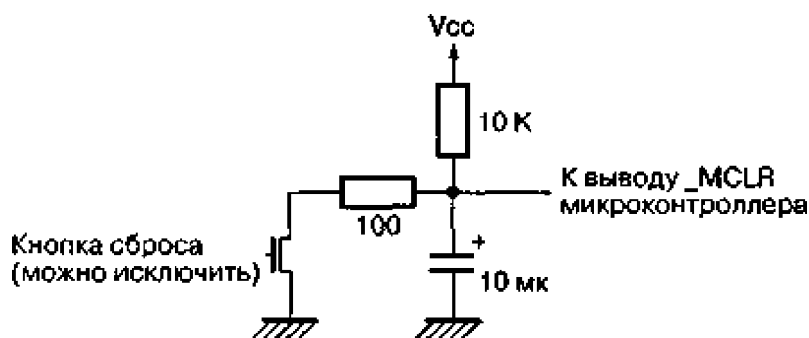


Рис. 3.20. Внешняя схема формирования сигнала *RESET*

Следует помнить, что вход `_MCLR` также используется для подачи напряжения программирования V_{pp} . Поэтому необходимо позаботиться о том, чтобы выходной сигнал, формируемый схемой сброса, не достиг 12 В. У некоторых микроконтроллеров (например, PIC12Cxx и PIC16C505) вывод `_MCLR`, запрограммированный для работы в качестве входного, по умолчанию используется для подачи цифровых импульсов, и необходимо записать специальное значение в регистр `OPTION`, чтобы иметь возможность использовать этот вход для сигнала сброса.

В некоторых микроконтроллерах имеется специальный детектор падения напряжения питания (BOD), который формирует сигнал в случае, если напряжение на выводах V_{dd} или `_MCLR` падает ниже заданного уровня (обычно 4,5 В). Хотя сами микроконтроллеры нормально работают при напряжении питания 4,5 В, многие другие микросхемы, которые могут использоваться совместно с МК, при таком напряжении часто оказываются неработоспособными.

Если микроконтроллер не имеет встроенного детектора падения напряжения питания, то его несложно реализовать с помощью простой схемы, показанной на рис. 3.21.

Некоторые микроконтроллеры имеют программируемый детектор падения напряжения питания, а другие требуют для этого использования внешних элементов. Для регулирования напряжения срабатывания детектора можно собрать схему, подобную той, что изображена на рис. 3.21, но проще применить специальную микросхему - помимо прочего она обеспечивает дополнительную задержку срабатывания, за время которой напряжение питания имеет возможность принять свое стабильное значение. Такие микросхемы обычно дешевле, чем весь набор элементов, входящих в схему на рис. 3.21.

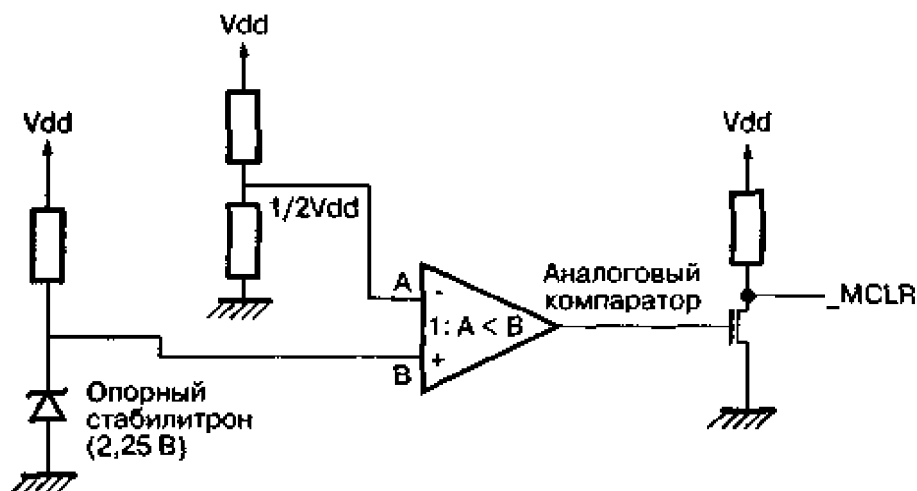


Рис. 3.21. Внешняя схема формирования сигнала сброса при падении напряжения источника питания (*brown-out detection*)

Наконец, в микроконтроллерах PICmicro может присутствовать специальный таймер PWRT (Power-up Wait Reset Timer, он упомянут в комментариях к нашей программе example.c). С его помощью микроконтроллер начинает работать не сразу, а спустя 72 мс после того, как сигнал на входе _MCLR примет неактивный (высокий) уровень. Такая задержка гарантирует, что напряжение питания микроконтроллера и всех других схем успело стабилизироваться.

В заключение обсудим подключение генератора тактовых импульсов. Именно эти импульсы управляют процессом выполнения всех операций в микроконтроллере и синхронизируют работу всех его таймеров. Существует восемь способов обеспечить тактирование микроконтроллеров PICmicro. Каждый из этих способов проявляет свои достоинства в различных ситуациях.

Проще всего применить релаксационный RC-генератор (рис. 3.22). Такая внешняя схема использовалась с первыми микроконтроллерами PIC. Здесь частота генерируемых импульсов определяется сопротивлением резистора и емкостью конденсатора. Схема очень простая, но ее стабильность невысока.

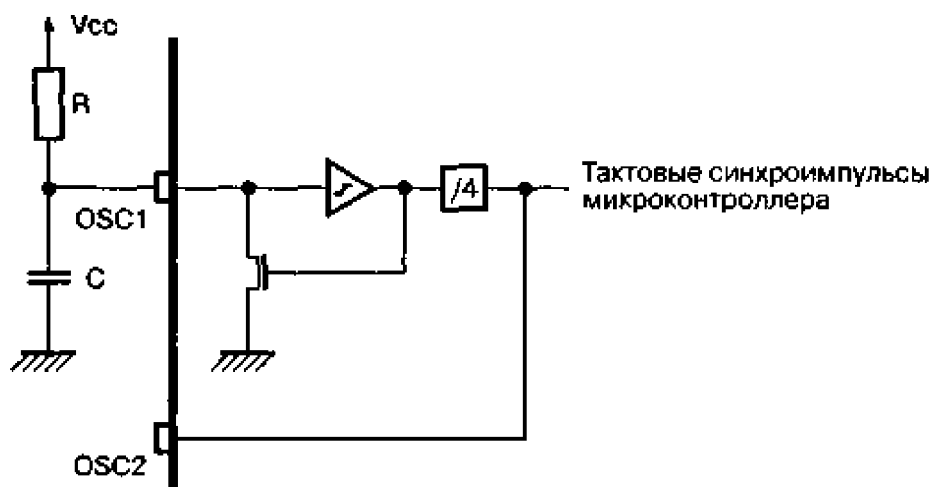


Рис. 3.22. Использование RC-генератора **тактовых** импульсов

В более поздних микроконтроллерах используется встроенный RC-генератор: регулировочным резистором (для настройки частоты). Обычно стабильность работы такого генератора составляет около 1,5%. Если этого недостаточно, то всегда можно использовать внешний генератор, для подключения которого имеются два вывода.

RC-генератор некоторых микроконтроллеров (например, PIC16F627) имеет встроенный конденсатор, но требует подключения внешнего резистора, номинал которого можно выбирать исходя из требуемой частоты генерации. Такое решение обеспечивает малую стоимость тактового генератора, но позволяет получить очень высокую стабильность частоты. Следует помнить, что резистор с одним и тем же сопротивлением при использовании разных микроконтроллеров может соответствовать различной частоте генерации.

Для повышения стабильности можно использовать кварцевые или керамические резонаторы, которые подключаются к микроконтроллеру по схеме, показанной на рис. 3.23.

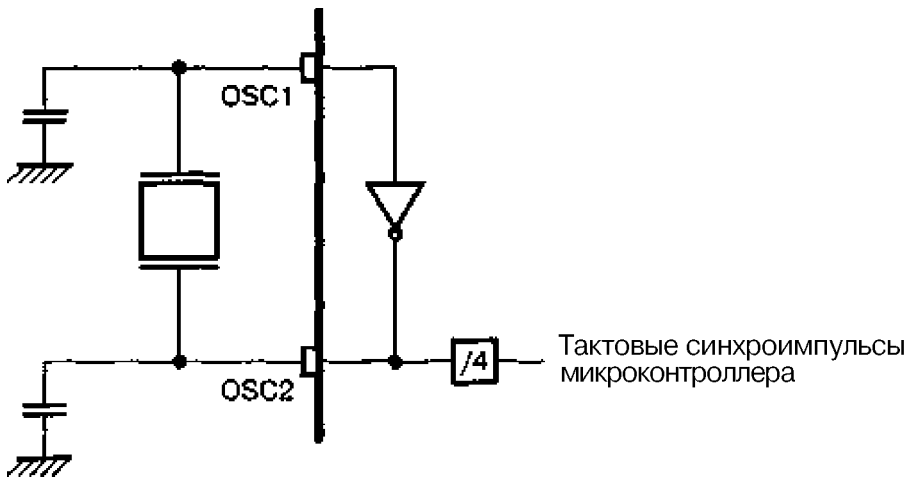


Рис. 3.23. Использование керамического резонатора для генерации тактовых импульсов

Три различных варианта резонаторов, применяемые для тактирования микроконтроллеров PICmicro, перечислены в табл. 3.4. Нужный режим задается в слове конфигурации директивой__CONFIG.

Таблица 3.4. Керамические и кварцевые резонаторы

Обозначение	Частота	Примечание
LP	0-200 кГц	Низкая потребляемая мощность (Low Power)
XT	200 кГц - 4 МГц	Обычная частота тактирования
HS	Более 4 МГц	Повышенная частота тактирования (High Speed)

Сигнал с вывода OSC2 может быть использован для тактирования других КМОП микросхем. Однако следует иметь в виду, что при подключении к генератору других

входов его частота изменится, а в некоторых случаях генерация может прекратиться.

Наконец, можно взять тактовые импульсы от внешнего источника, подключенного к входу OSC1. Неиспользуемый в этом случае вход OSC2 в некоторых микроконтроллерах PIC можно запрограммировать на ввод или вывод информации.

В новых микроконтроллерах PIC18C/Fxx существует еще один режим, основанный на применении контура фазовой автоподстройки частоты (ФАПЧ). С его помощью невысокая частота тактового генератора перед тем, как она будет использована для тактирования микроконтроллера, умножается в несколько раз. Такой способ снижает мощность, потребляемую тактовым генератором от источника питания, а также уменьшает уровень электромагнитных помех.

3.4. ОПИСАНИЕ МИКРОКОНТРОЛЛЕРА PIC16F627

Чтобы читатель мог без проблем разбирать примеры программ на языке C для микроконтроллера PIC16F627, представленные в этой книге, необходимо понимать некоторые особенности его устройства и принципа работы. Этот МК поддерживает большое число функций ввода-вывода - они характерны для многих представителей семейства микроконтроллеров PICmicro, в том числе и для более ранней модели PIC16F84. Поэтому приведенное здесь описание основных особенностей PIC16F627 будет полезно всем, кто хочет разрабатывать свои устройства на основе любого МК фирмы Microchip.

Разводка выводов микроконтроллера PIC16F627 показана на рис. 3.24. Как можно видеть, многие выводы предназначены для выполнения двух-трех различных функций.

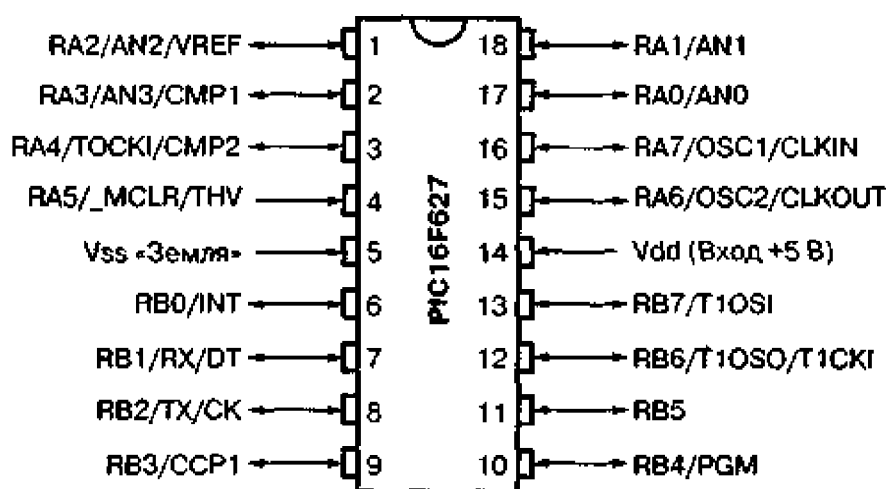


Рис. 3.24. Цоколевка микроконтроллера PIC16F627

Все МК PICmicro основаны на гарвардской архитектуре, поэтому структурная схема PIC16F627, показанная на рис. 3.25, характерна для всех микроконтроллеров фирмы Microchip. Для обеспечения высокой эффективности выполнения функций ввода-вывода процессор и периферийные устройства интегрированы на одном кристалле.

программно доступных регистров, предназначенных для управления работой этих устройств.

Регистры микроконтроллера PIC16F627 показаны на рис. 3.26. Для хранения данных, необходимых для **работы** программы, предназначены регистры общего

				Адреса регистров			
INDF *	00h	INDF *	80h	INDF *	100h	INDF *	180h
TMR0	01h	OPTION	81h	TMR0	101h	TMR0	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h		105h		185h
PORTB	06h	TRISB	86h	PORTB	106h	PORTB	186h
	07h		87h		107h		187h
	08h		88h		108h		188h
	09h		89h		109h		189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch		10Ch		18Ch
	0Dh		8Dh		10Dh		18Dh
TMR1L	0Eh	TMR1L	8Eh		10Eh		18Eh
TMR1H	0Fh		8Fh		10Fh		18Fh
T1CON	10h		90h				
TMR2	11h		91h				
T2CON	12h	PR2	92h				
	13h		93h				
	14h		94h				
CCPR1L	15h		95h				
CCPR1H	16h		96h				
CCP1CON	17h		97h				
RCSTA	18h	TXSTA	98h				
TXREG	19h	SPBRG	99h				
RCREG	1Ah	EEDATA	9Ah				
	1Bh	EEADR	9Bh				
	1Ch	EECON1	9Ch				
	1Dh	EECON2*	9Dh				
	1Eh		9Eh				
CMCON	1Fh	VRCON	9Fh				
	20h		A0h	Регистры общего назначения 48 байт	11Fh A0h		
Регистры общего назначения 96 байт		Регистры общего назначения 80 байт			14h 150h		
			EFh		16h		1EFh
		Доступ к 70H-7FH	F0h	Доступ к 70H-7FH	170h		1F0h
	7Fh		FFh		17Fh	Доступ к 70H-7FH	1FFh
Bank 0		Bank 1		Bank 2		Bank 3	

 Нереализованные ячейки памяти, читаются как 0

* Регистр **косвенной** адресации (физически не существует)

Рис. 3.26. Карта памяти микроконтроллера PIC16F627

назначения (РОН, General Purpose Registers - GPR). Компилятор PICC Lite автоматически размещает в этих регистрах все переменные, описанные в программе.

Особенность микроконтроллеров PICmicro - наличие специального конфигурационного регистра, каждый бит которого определяет один из следующих режимов:

- тип используемого тактового генератора;
- режим защиты кода программы;
- параметры сброса;
- режим работы сторожевого таймера (Watch-Dog Timer - WDT);
- режим отладки PIC16F87х.

Программист должен позаботиться о том, чтобы корректно установить состояния всех разрядов регистра конфигурации; в противном случае правильная работа микроконтроллера может быть нарушена. Указанное в программе значение автоматически записывается в этот регистр программатором.

В микроконтроллерах младшего семейства¹ у конфигурационного регистра нет собственного адреса, поэтому программист не имеет к нему непосредственного доступа и не может, скажем, в своей программе прочитать записанное в него значение. В МК среднего семейства² этот регистр всегда имеет адрес 0x02007. Однако он также недоступен для чтения.

Включаемые файлы .inc, всегда используемые в ассемблерных программах, содержат необходимые определения каждого значения, записываемого в конфигурационный регистр, поэтому в директиве __CONFIG программист может использовать имена режимов и поразрядную операцию AND.

Компилятор PICC Lite имеет псевдофункцию __CONFIG(cw), где cw - это значение (*слово конфигурации*), предназначенное для записи в конфигурационный регистр. После компиляции оно попадает в HEX-файл и используется программатором, как и все остальные содержащиеся там команды и данные. Разумеется, для этого программатор должен быть специально спроектирован для работы с микроконтроллерами PICmicro. Некоторые программаторы требуют, чтобы программист осуществил запись слова конфигурации вручную; если вы забудете сделать это, могут возникнуть проблемы.

По умолчанию все биты конфигурационного регистра установлены в 1.

Выше мы описали возможные типы генераторов, используемых для тактирования микроконтроллеров PICmicro. Если реально имеющийся в системе тактовый генератор не соответствует тому, который записан в слове конфигурации, то микроконтроллер может работать неустойчиво (если вообще будет работать).

При программировании МК разработчик должен решить, будет ли он использовать сторожевой таймер (WDT). В большинстве микроконтроллеров PICmicro единичное значение бита разрешения, устанавливаемое в регистре конфигурации по умолчанию, означает, что сторожевой таймер включен. Это приводит к тому, что ваш микроконтроллер (если не предпринять специальных действий) будет

¹ К нему относятся 12-разрядные МК PIC12 и PIC16C5х. - Прим. перев.

² К нему относятся 14-разрядные МК PIC16. - Прим. перев.

выполнять общий сброс каждые 18 мс - 2,3 с. Чтобы этого не происходило, необходимо либо запретить работу сторожевого таймера, либо обеспечить выполнение машинной команды `clrwdt` (`clear WDT` — сброс сторожевого таймера) через определенное время. Длительность интервала обычно выбирается равной половине длительности промежутка времени, которое требуется сторожевому таймеру, чтобы вызвать сброс системы¹.

Разработчик должен помнить, что у различных микроконтроллеров PICmicro существуют некоторые особенности реализации конфигурационного регистра, игнорирование которых может привести к тому, что микроконтроллер будет ненадежно испорчен. У старых МК, имеющих для хранения программ память EEPROM с ультрафиолетовым стиранием, защита кода программы выполнена в виде специального металлического экрана, который предотвращает попадание ультрафиолетовых лучей на поверхность чипа через специальное окошко, расположенное на корпусе микроконтроллера.

В новых МК защита кода осуществляется путем установки нужного значения в разряде CP (`code protection`) регистра конфигурации. В результате исключается возможность умышленного частичного стирания памяти микроконтроллера (например, переустановки бита CP), которое позволило бы прочесть записанную в МК программу с целью ее последующего «пиратского» использования.

Если при программировании вы случайно установите такую защиту, то уже и сами не сможете изменить код программы, а тем более записать в МК новую программу. Чтобы иметь возможность использовать микроконтроллер в своих дальнейших разработках, в директиве `__CONFIG` следует указывать параметр `CP_OFF` (защита выключена).

Некоторые МК PICmicro имеют возможность *низковольтного программирования* (`Low Voltage Programming - LVP`). Некорректное использование этого режима может привести к неправильной работе микроконтроллера, так как при включенном LVP порт RB4 неспособен выполнять свои обычные функции ввода-вывода.

В табл. 3.5 показано назначение различных разрядов регистра конфигурации, а в табл. 3.6 приведены четыре наиболее часто используемых значения, записываемые в этот регистр с помощью псевдофункции `__CONFIG`. Во всех четырех случаях защита кода программы и данных выключена.

Таблица 3.5. Назначение разрядов конфигурационного регистра

Номер бита	Имя	Значение
13–12	CP1:CP0	Эти разряды дублированы битами 11 и 10, которые должны иметь то же состояние. 1х — защита программы выключена; 01 — защищен диапазон адресов 0x0200–0x03FF; 00 — защищен весь диапазон адресов программы

¹ Если из-за программных ошибок или какого-либо аппаратного сбоя управляющая программа зависнет, то команда сброса сторожевого таймера не выполнится вовремя. В результате произойдет сброс, и контроллер снова продолжит выполнение своих функций. - Прим. перев.

Таблица 3.5. Назначение разрядов конфигурационного регистра (окончание)

Номер бита	Имя	Значение
9	—	Не используется
8	CPD	1 — защита памяти данных выключена; 0 — защита памяти данных включена
7	LVP	1 — включен 5-вольтовый режим программирования (RB4 используется для управления программированием); 0 — низковольтный режим программирования выключен (RB4 используется в качестве порта ввода-вывода)
6	BODEN	1 — включен детектор падения напряжения питания (BOD); 0 — детектор BOD выключен
5	MCLRE	1 — вывод RA5/ MCLR используется для подачи сигнала сброса MCLR;
3	_PWRTE	Разрешение работы таймера PWRT, обеспечивающего задержку (72 мс) отпускания сигнала сброса после включения питания: 1 — таймер PWRT включен; 0 — таймер PWRT выключен
2	WDTE	1 — сторожевой таймер (WDT) включен; 0 — сторожевой таймер (WDT) выключен
4, 1–0	FOSC2:FOSCO	Выбор типа тактового генератора: 111 — используется внешний резистор, подключаемый к выводу RA7. При этом вывод RA6 может использоваться для подачи на внешние устройства тактовых импульсов, частота которых в четыре раза меньше, чем частота тактирования (один период на каждую выполняемую машинную команду); 110 — то же, но RA6 используется для обычного ввода-вывода; 101 — внутренний RC-генератор. Вывод RA6 используется как выход тактовых импульсов (как и в случае 111); 100 — внутренний RC-генератор, но RA6 используется для обычного ввода-вывода; 011 — внешний тактовый генератор, подключаемый к выводу RA7. Вывод RA6 используется для обычного ввода-вывода; 010 — генератор HS (см. табл. 3.4); 001 — генератор XT; 000 — генератор LP

Таблица 3.6. Часто используемые значения, записываемые в конфигурационный регистр

Слово конфигурации	Примечание
0x03F70	Внутренний тактовый генератор, RA6/RA7 используются для ввода-вывода, BODEN включен, WDT выключен, сигнал сброса внешний
0x03F74	Внутренний тактовый генератор, RA6/RA7 используются для ввода-вывода, BODEN и WDT включены, сигнал сброса внешний
0x03F61	Внешний тактовый генератор XT на 4 МГц, BODEN включен, WDT выключен, сигнал сброса внешний
0x03F65	Внешний тактовый генератор XT на 4 МГц, BODEN и WDT включены, сигнал сброса внешний

Микроконтроллеры PICmicro имеют специальные регистры для управления периферийными устройствами: OPTION, TMR0, PORT и др. Регистр INTCON позволяет управлять работой системы прерываний, а так называемый *регистр флагов* хранит специальные признаки (флажки), необходимые для выполнения команд процессора и некоторых периферийных функций.

Карта адресного пространства микроконтроллеров PICmicro разделена на области - *банки* (см. рис. 3.26), и за исключением некоторых регистров (таких как INTCON) все перечисленные регистры могут быть доступны при работе только с определенными банками памяти.

При разработке программ в некоторых случаях может пригодиться знание одной особенности МК PICmicro: при чтении информации из несуществующих регистров возвращаются нулевые значения 0x000. Адреса физически не существующих регистров на карте памяти (см. рис. 3.26) выделены серым цветом. Разумеется, бесполезно пытаться записать в любой из этих регистров какое-либо ненулевое значение.

Для записи информации в периферийные устройства, подключенные к микроконтроллеру, и для чтения информации из этих устройств используются порты ввода-вывода. Каждый вывод микроконтроллера, применяемый для этих целей, может быть запрограммирован либо на ввод, либо на вывод сигналов. Нагрузочная способность в режиме выхода составляет 20 мА.

На рис. 3.27 показана структурная схема устройства, управляющего работой одной линии порта ввода-вывода.

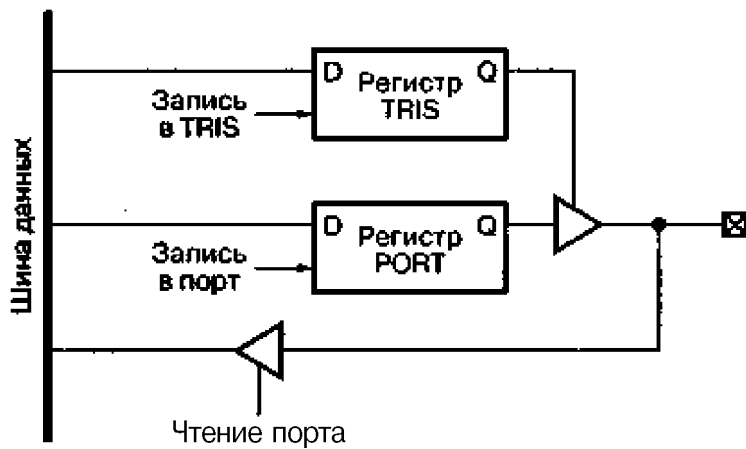


Рис. 3.27. Стандартный порт ввода-вывода микроконтроллеров *PICmicro*

Каждому выводу микроконтроллера, который может работать в режиме ввода-вывода информации, соответствует один бит специального регистра, который доступен для программирования. Эти регистры в программе именуются по принципу **R%#**; вместо знака % указывается латинская буква (А, В и т.д., в зависимости от числа имеющихся портов ввода-вывода), а вместо знака # — любая цифра от 0 до 7 (номер бита соответствующего регистра). Таким образом, восемь линий ввода-вывода соответствуют одному 8-разрядному регистру. Например, RB3 обозначает третий разряд порта В.

В некоторых компиляторах требуется указывать имена в формате PORT%# или PORT%.#; расшифровка знаков % и # та же.

Для управления портами ввода-вывода предназначены специальные регистры TRIS (для порта А управляющий регистр называется TRISA, для порта В - TRISB и т.д.). При включении питания все биты регистров TRIS устанавливаются в 1. Это означает, что все линии ввода-вывода находятся в режиме *ввода* информации. Если программист, например, запишет 0 во второй бит регистра TRISA, то линия RA2 микроконтроллера перейдет в режим *вывода* информации. Аналогичное правило действует и для всех остальных линий. Если заметить, что цифра 0 напоминает букву О (Output - вывод), а цифра 1 - букву I (Input - ввод), то легко запомнить, как программируется тот или иной режим.

Для обращения к регистрам TRIS действует то же правило именования, что и для портов ввода-вывода: все имена имеют формат TRIS%#. Например, чтобы запрограммировать вывод RB2 микроконтроллера для работы в качестве выходного и вывести на него сигнал высокого уровня, в программе на языке С следует написать:

```
TRISB2 = 0;      // Линия RB2 переведена в режим выхода.
RB2 = 1;         // На выводе RB2 установлен сигнал высокого уровня.
```

Кроме поразрядного доступа и регистры TRIS, и сами порты ввода-вывода поддерживают побайтный режим.

Схема на рис. 3.27 хорошо иллюстрирует еще одну особенность микроконтроллеров PICmicro: если какой-либо вывод запрограммирован на работу в качестве входного, то при обращении к соответствующему порту будет прочитано значение сигнала, действующее на внешнем выводе микроконтроллера, а не то, которое, возможно, было записано в этот порт ранее.

Некоторые микроконтроллеры других фирм способны поддерживать два режима, и программист может сам выбрать, какие данные будут читаться при обращении к портам ввода-вывода: те, что были записаны в этот порт, или те, что определяются сигналами, действующими на внешних выводах контроллера. А другие МК могут иметь два разных адреса для одного регистра - тогда при обращении по одному адресу можно прочесть значение, записанное в этот регистр ранее, а при обращении по другому адресу узнать, какой сигнал действует на внешнем выводе контроллера. Для МК PICmicro это не так.

Часто у разработчиков возникают проблемы при работе с выводом RA4 (рис. 3.28). Этот вывод управляется полевым КМОП транзистором с *открытым стоком* (ОС), поэтому для нормальной работы этот вывод (или несколько соединенных между собой таких же выводов с открытым **стоком**)¹ требуется подключать к положительной шине источника питания через резистор сопротивлением 1-10 кОм. В противном случае этот порт не может обеспечить вывод сигнала высокого уровня.

¹ Другие выходы, не имеющие третьего состояния или открытого стока (коллектора), соединять друг с другом нельзя ни в коем случае. — Прим. *перев.*

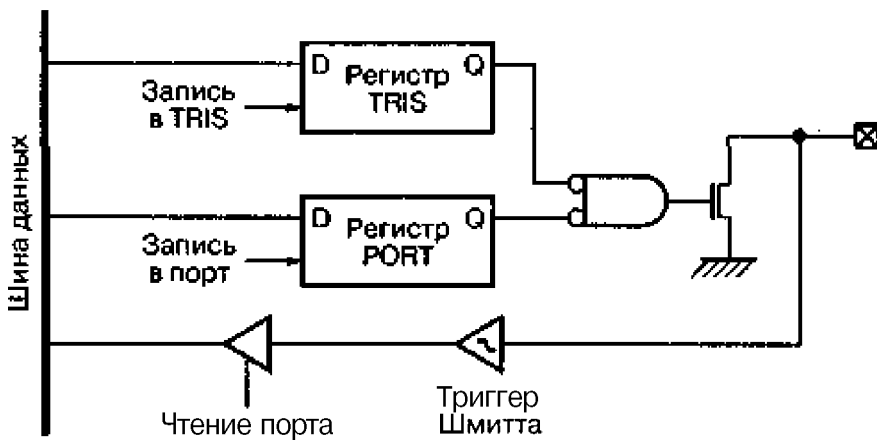


Рис. 3.28. Организация порта RA4 микроконтроллеров PICmicro

В режиме ввода информации линия RA4 проходит через специальный элемент с *гистерезисом* - так называемый *триггер Шмитта*, который имеет различные пороги переключения при переходе входного сигнала с низкого уровня на высокий и с высокого уровня на низкий (рис. 3.29). В результате обеспечивается подавление помех, величина которых не превышает расстояния между этими двумя уровнями.



Рис. 3.29. Два уровня срабатывания триггера Шмитта

Разработчику надо иметь в виду описанную особенность входа RA4: не следует пытаться подключать к нему РС-цепи, например, чтобы прочитать состояние потенциометра. В этом случае из-за наличия триггера Шмитта сигнал (0 или 1), воспринимаемый микроконтроллером при одном и том же положении движка потенциометра, будет зависеть от того, куда двигается в данный момент этот движок - в сторону увеличения или уменьшения уровня сигнала.

Выходные линии порта В микроконтроллеров PICmicro среднего семейства также имеют свою особенность: к некоторым из них подключены специальные транзисторы, «подтягивающие» (pull-up) выходное напряжение на этих линиях к напряжению положительной шины питания (рис. 3.30). Такие транзисторы (используемые в качестве управляемых сопротивлений) можно отключать с помощью специального бита `_RPBU` регистра `OPTION`. Если этот бит сброшен в нулевое состояние и внешний вывод запрограммирован на работу в качестве выходного, то для него режим «подтягивания» включен.

Изменение входного сигнала на линиях порта В может быть использовано для срабатывания системы прерываний. Обычно такой режим применяется для вывода RBO/INT, который используется в качестве входа запроса на прерывание, при условии, что установлен бит INTE регистра INTCON. Запрос вырабатывается по

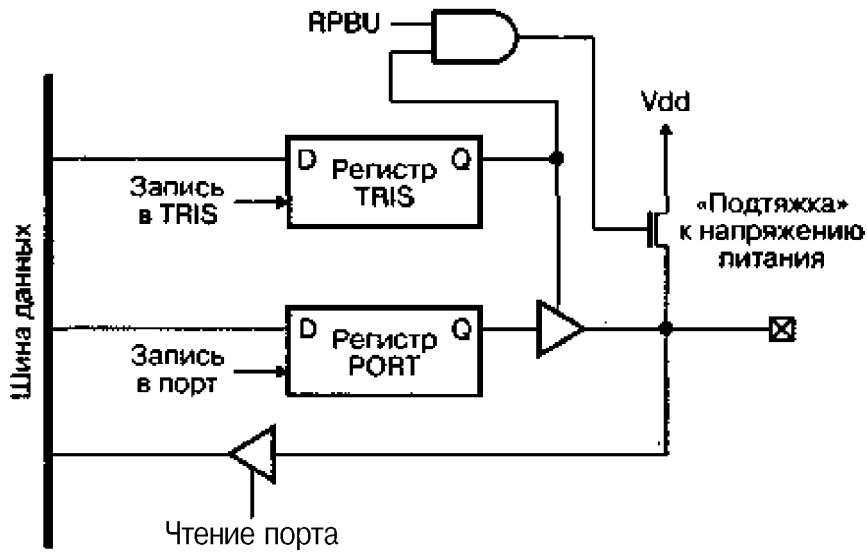


Рис. 3.30. «Подтягивание» выходного напряжения

переднему или заднему фронту импульса на этом выводе (то есть при изменении уровня напряжения с низкого на высокий или с высокого на низкий). Активный переход (положительный или отрицательный) выбирается с помощью бита INTEDG регистра OPTION. Если этот бит установлен в 1, то запрос на прерывание вырабатывается по положительному фронту входного сигнала (при изменении уровня сигнала на входе RB0/INT с низкого на высокий), а если сброшен в 0 - то по отрицательному фронту (при изменении уровня с высокого на низкий). Лично я предпочитаю использовать для формирования сигнала запроса схему, показанную на рис. 3.31.

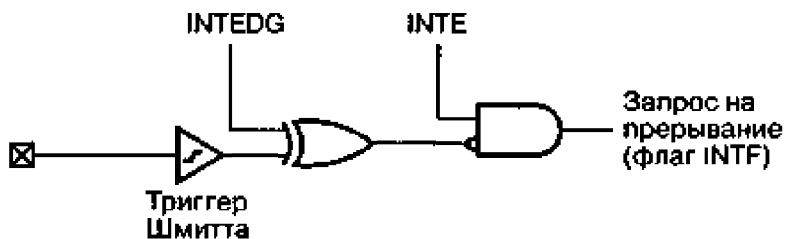


Рис. 3.31. Схема для формирования запроса на прерывание

После того как запрос на прерывание будет подтвержден процессором, необходимо сбросить флаг INTF регистра INTCON, чтобы разрешить обработку следующих запросов, которые могут поступить на вход RB0/INT. Следует помнить, что сигнал запроса на прерывание проходит через триггер Шмитта, между тем как в режиме обычного ввода-вывода линия RB0 не обладает гистерезисом. Поэтому поведение входа RB0 может несколько отличаться от работы остальных линий порта В.

Существует еще один способ формирования сигнала прерывания. Если установить в 1 бит RBIE регистра INTCON, то при любом изменении уровня сигнала на линиях RB4 - RB7 будет генерироваться запрос на прерывание. При каждом

таким запросе будет устанавливаться в единичное состояние флаг RBIF регистра INTCON. Следует сначала прочитать текущее состояние порта В, после чего можно сбросить флаг RBIF¹.

Запрос на прерывание при изменении состояния входных сигналов вырабатывается только для линий RB4 - RB7 и только если они находятся в режиме ввода информации. Когда любая из этих линий запрограммирована на вывод данных, то изменение сигнала на этом выводе уже не может служить источником запроса.

Программисту следует воздержаться от выполнения операций чтения из порта В, если его входные линии работают в качестве источников прерываний.

В этой книге мы будем использовать прерывания для реализации функций механического и электронного уровней. Это позволит нам без лишних усилий разрабатывать достаточно эффективный код для управления роботами. Уже в следующей главе мы узнаем, как программируется обработчик прерываний на языке PICC Lite.

В микроконтроллерах PIC16F627 для управления системой прерываний служит регистр INTCON, имеющий адрес 0x0B. Он доступен из любого банка памяти. Назначение битов этого регистра для всех микроконтроллеров среднего семейства практически одинаково.

Бит GIE должен быть установлен в 1, если мы хотим, чтобы процессор не игнорировал любые запросы на прерывание. На критичных участках программы, которые должны быть выполнены максимально быстро (поэтому нежелательно, чтобы процессор отвлекался на обработку прерываний), программист должен сбросить бит GIE в 0. Этот бит сбрасывается, например, в начале любой процедуры обработки прерываний. В конце процедуры обработки бит GIE снова устанавливают.

Имена других управляющих разрядов регистра INTCON также заканчиваются символом E (enable - разрешение). Если бит разрешения установлен, то прерывания соответствующего типа разрешены. Как только происходит прерывание этого типа, то устанавливается специальный флажок - другой разряд регистра INTCON, имя которого заканчивается символом F (flag). Анализ этих флагов позволяет обработчику прерываний определить источник запроса. Флажки не сбрасываются автоматически при обработке прерывания - программист должен сам позаботиться о том, чтобы записать в них нулевые значения. Но в момент очистки флага запрос на прерывание уже не должен действовать.

В табл. 3.7 показано назначение каждого бита регистра INTCON.

После получения запроса на прерывание процессор, закончив выполнение очередной команды, начинает процедуру обработки прерывания. На выполнение

¹ Для формирования сигнала запроса на прерывание при изменении уровней сигналов на входах RB4 - RB7 текущее состояние каждого из этих входов сравнивается с соответствующим битом, зашелкнутым в регистре порта В при выполнении последней операции чтения из этого порта. Если не выполнить операцию **чтения**, то записанное в регистр значение будет неопределенным и не сможет гарантировать, что следующий запрос на прерывание произойдет при первом же изменении состояния входного сигнала. Этим объясняется также рекомендация не выполнять никаких операций чтения из порта В, если разрешены прерывания по входам RB4 - RB7. — *Прим. перев.*

Таблица 3.7. Назначение разрядов регистра управления системой прерываний INTCON

Номер бита	Имя	Назначение
7	GIE	Общее разрешение прерываний
6	PEIE	Разрешение прерываний от периферийных устройств
5	TOIE	Разрешение прерываний при переполнении таймера TMRO
4	INTE	Разрешение прерываний по входу RBO/INT
3	RBIE	Разрешение прерываний при изменении сигналов на входах порта В
2	TOIF	Флаг активности прерывания от таймера TMRO
1	INTF	Флаг активности прерывания по входу RBO/INT
0	RBIF	Флаг активности прерывания при изменении сигналов на входах порта В

одной машинной команды в микроконтроллерах PICmicro требуются один-два командных цикла (каждый такой цикл длится четыре периода работы тактового генератора). С учетом еще двух циклов, необходимых для вызова процедуры обработки прерывания, получается, что время реакции на запрос (interrupt latency) составляет три-четыре командных цикла. МК PIC16F627, как и многие другие микроконтроллеры PICmicro среднего семейства, имеет только один вектор прерывания с адресом 0x00004. На рис. 3.32 показано, как происходит обработка запроса на прерывание.

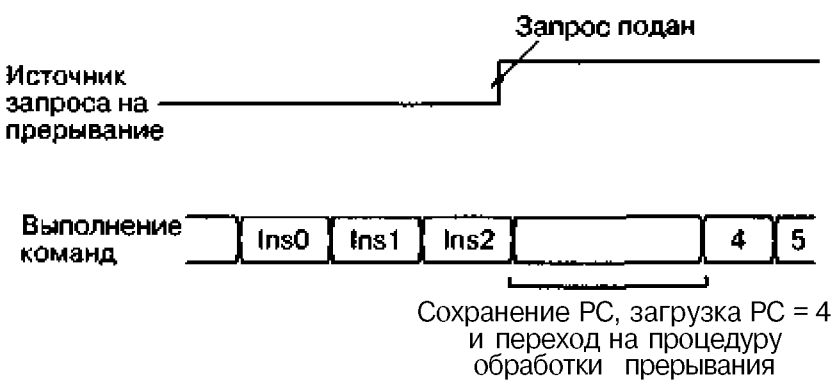


Рис. 3.32. Обработка запроса на прерывание

Итак, используя регистр INTCON, можно управлять тремя источниками запросов на прерывание. Разряды TOIE и TOIF соответствуют прерыванию от таймера TMRO (запрос формируется, когда счетчик таймера переполняется, то есть становится равным 0x0100). Вторым источником запроса может стать сигнал на входе RBO/INT. Наконец, запрос на прерывание может вырабатываться при изменении сигналов на входных линиях порта В.

Но существуют и другие возможные источники запросов на прерывание. Они управляются специальными регистрами PIE1 и PIR1. Каждый разряд регистра PIE1 (табл. 3.8) содержит бит разрешения прерываний от источника соответствующего типа, а каждый бит регистра PIR1 (табл. 3.9) - флаг активности этого прерывания.

Таблица 3.8. Назначение разрядов регистра *PIE1* микроконтроллера *PIC16F627*

Номер бита	Имя	Назначение
7	<i>EEIE</i>	Разрешение прерываний при записи в <i>EEPROM</i>
6	<i>CMIE</i>	Разрешение прерываний от компаратора
5	<i>RCIE</i>	Разрешение прерываний от <i>USART</i> при приеме
4	<i>TXIE</i>	Разрешение прерываний от <i>USART</i> при передаче
3	—	Не используется (читается 0)
2	<i>CCP1IE</i>	Разрешение прерываний от <i>CCP</i>
1	<i>TMR2IE</i>	Разрешение прерываний при переполнении таймера <i>TMR2</i>
0	<i>TMR1IE</i>	Разрешение прерываний при переполнении таймера <i>TMR1</i>

Таблица 3.9. Назначение разрядов регистра *PIR1* микроконтроллера *PIC16F627*

Номер бита	Имя	Назначение
7	<i>EEIF</i>	Флаг прерывания по окончании записи в <i>EEPROM</i> данных
6	<i>CMIF</i>	Флаг прерывания от компаратора
5	<i>RCIF</i>	Флаг прерывания от приемника <i>USART</i>
4	<i>TXIF</i>	Флаг прерывания от передатчика <i>USART</i>
3	—	Не используется (читается 0)
2	<i>CCP1IF</i>	Флаг прерывания от модуля <i>CCP</i>
1	<i>TMR2IF</i>	Флаг прерывания по переполнению таймера <i>TMR2</i>
0	<i>TMR1 IE</i>	Флаг прерывания по переполнению таймера <i>TMR1</i>

Заметим, что назначение разрядов этих регистров может быть различным у разных МК PICmicro. Поэтому за точными данными следует обращаться к документации.

Специальный регистр *PCON* может использоваться для чтения или записи информации, характеризующей частоту тактирования микроконтроллера, а также причину его последнего сброса. Назначение битов этого регистра поясняется в табл. 3.10.

Таблица 3.10. Назначение разрядов регистра *PCON*

Номер бита	Имя	Назначение
7-4	—	Не используются
3	<i>OSCF</i>	Частота тактирования: / — внутренний <i>RC</i> -генератор на 4 МГц или внешний резистор, определяющий частоту генерации; 0-37 кГц (режим низкой мощности потребления)
2	—	Не используется
1	<i>_POR</i>	0 - последний сброс произошел при включении питания; должен быть программно установлен в 7
0	<i>_BOD</i>	0 — последний сброс произошел при понижении напряжения питания; должен быть программно установлен в 1

Иногда полезно проверять состояние бита `_POR`: если он сброшен в 0, это значит, что микроконтроллер перезапустился после падения напряжения питания; следовательно, надо перезарядить аккумуляторы. Другая возможная причина - резкое возрастание тока, потребляемого двигателями, например, если робот столкнулся с каким-либо препятствием.

Нестабильное питание или большие электромагнитные наводки от других устройств могут стать причиной неправильной работы микроконтроллера. В результате не исключено зависание программы. Для борьбы с этим используется *сторожевой таймер* (Watch-Dog Timer - WDT): он формирует сигнал сброса, если микроконтроллер перестал правильно выполнять свою программу.

Обычный период работы WDT составляет 18 мс. Если за это время ни разу не выполнится команда `clrwdt`, то он вызовет сброс микроконтроллера.

Структурная схема сторожевого таймера показана на рис. 3.33. Генератор импульсов управляет работой цифрового счетчика, сигнал переполнения которого подается на управляемый делитель частоты. Это еще один счетчик, отличающийся от первого тем, что величину его модуля счета можно изменять программно. С помощью *прескалера* программист может указать, какой по счету импульс переполнения первого счетчика вызовет, наконец, генерацию сигнала сброса. Можно выбирать значение коэффициента деления от 1 до 128. С учетом этого период работы сторожевого таймера составляет от 18 мс до 2,3 с. Если сторожевой таймер вызывает сброс микроконтроллера, то сбрасывается бит `_TO` регистра состояния STATUS.

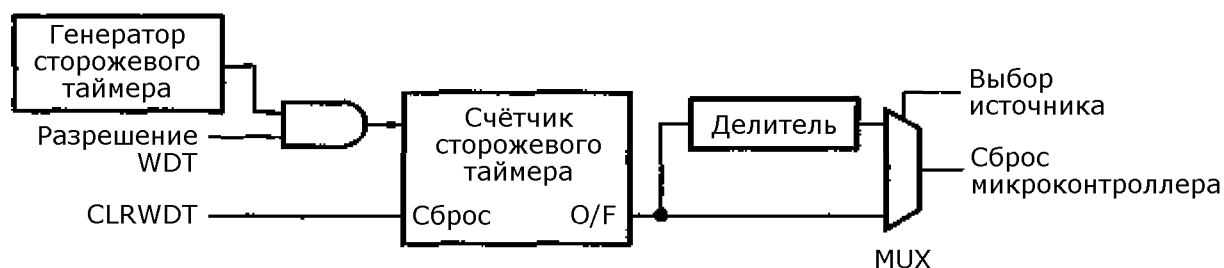


Рис. 3.33. Структурная схема сторожевого таймера в микроконтроллерах PICmicro

Желательно, чтобы команда `clrwdt` выполнялась через промежутки времени, примерно равные половине периода работы сторожевого таймера (с учетом делителя). Дело в том, что стабильность встроенного RC-генератора, от которого работает сторожевой таймер, невелика. Погрешность может достигать 20%. Это значит, что на самом деле сторожевой таймер может сработать через период времени от 14 до 22 мс (если не используется делитель). Тогда если программист обеспечит выполнение команды `clrwdt`, например, каждые 9 мс, то возможность ошибочного сброса микроконтроллера будет исключена.

Для сброса сторожевого таймера в программе на языке PICC Lite надо использовать ассемблерную вставку

```
asm("clrwdt");
```

Следует помнить, что работа сторожевого таймера разрешается или запрещается с помощью слова конфигурации, задаваемого при программировании

микроконтроллера, и эта установка не может быть изменена в программе. Если программист забудет запретить работу сторожевого таймера и не обеспечит периодическое выполнение команды его сброса, то приложение окажется неработоспособным.

В микроконтроллере PIC16F627 бит разрешения работы сторожевого таймера по умолчанию установлен в единичное состояние (работа WDT разрешена), поэтому необходимо позаботиться о том, чтобы сбросить его при использовании директивы `__CONFIG`. Если вы желаете использовать сторожевой таймер в своем приложении, я рекомендую разрешать его работу только после того, как программа будет окончательно отлажена. Иначе вам не избежать проблем при отладке.

Для формирования запросов на прерывание через заданные промежутки времени используется 8-разрядный таймер TMRO. Он имеется у большинства микроконтроллеров PICmicro. Мы будем использовать его в примерах, приведенных в следующей главе.

Программист может записать в счетчик таймера TMRO любое число, являющееся степенью двойки. Это значение будет увеличиваться на 1 при каждом импульсе, пришедшем на вход счетчика. Импульсы можно подавать от внешнего источника или от счетчика машинных циклов (длительность последних в четыре раза больше периода работы тактового генератора). Внутри микроконтроллера счетные импульсы синхронизируются с началом очередного периода времени, длительность которого равна двум командным циклам. Поэтому частота срабатывания таймера не может быть больше половины частоты выполнения команд (другими словами, не больше одной восьмой от частоты тактирования).

Структурная схема таймера показана на рис. 3.34.

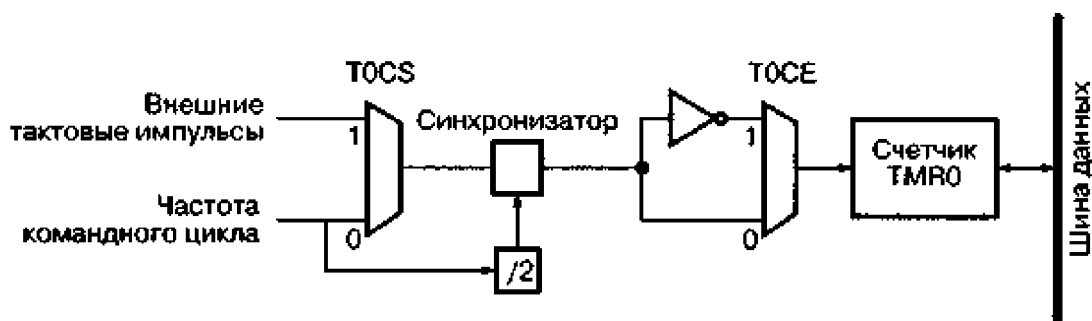


Рис. 3.34. Структурная схема таймера TMRO

Бит TOCS задает источник счетных импульсов таймера (внешний генератор или командные циклы), а бит TOCE определяет активный фронт, по которому срабатывает счетчик таймера (положительный или отрицательный). Оба бита расположены в регистре OPTION, который будет описан ниже.

Счетные импульсы от внешнего источника подаются на вывод TOSCK микроконтроллера. Этот вывод в МК PIC16F627 может также использоваться для обычного ввода-вывода. В режиме работы с таймером вход TOSCK имеет гистерезис (реализованный с помощью триггера Шмитта); это уменьшает вероятность ложных срабатываний счетчика.

Частоту входных импульсов, подаваемых на счетных вход таймера, можно понизить с помощью *прескалера* (prescaler). Это тот же самый делитель, который мы обсуждали, разбирая работу сторожевого таймера.

Работой предварительного делителя можно управлять с помощью четырех разрядов регистра OPTION. Бит PSA определяет, для каких целей используется предделитель. Если он установлен в 1, то делитель работает совместно со сторожевым таймером, а если сброшен в 0 - то с таймером TMRO.

Коэффициент деления устанавливается битами PS0, PS1 и PS2 регистра OPTION. Хранящееся в них трехразрядное двоичное число определяет степень двойки, задающую значение коэффициента деления (табл. 3.11).

На рис. 3.35 показано, как прескалер подключается к таймеру TMRO или к сторожевому таймеру WDT.

Таблица 3.11. Задание коэффициента деления с помощью разрядов PS2 – PS0 регистра OPTION

PS2 – PS0	Коэффициент деления прескалера
000	1
001	4
011	8
100	16
101	32
110	64
111	128

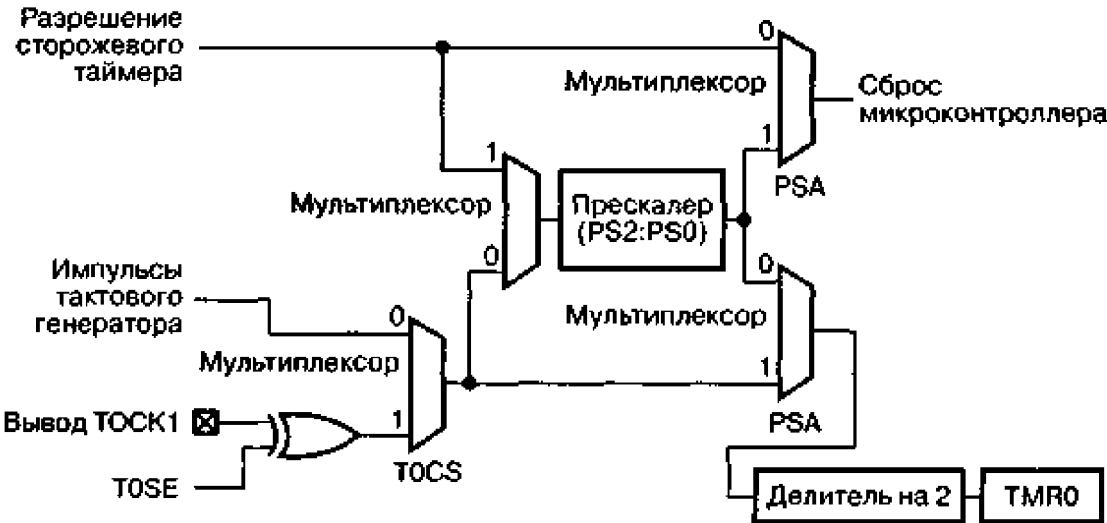


Рис. 3.35. Работа предделителя совместно с TMRO и WDT

В микроконтроллерах PICmicro младшего и среднего семейства (в том числе и в МК PIC16F627) регистр таймера TMRO располагается по адресу 0x001. Этот регистр доступен как для чтения, так и для записи. Программист должен записать в него нужное значение, а затем установить в 1 бит T0IE регистра INTCON для разрешения прерываний при переполнении таймера.

В микроконтроллерах среднего семейства (следовательно, и в МК PIC16F627) запрос на прерывание вырабатывается, когда в результате прихода очередного счетного импульса счетчик таймера переключается из состояния 0x0FF в 0x000. В результате устанавливается флаг T0IF. Если прерывания запрещены (бит GIE сброшен), то запрос игнорируется процессором, но флаг T0IF все равно устанавливается.

Перед тем как разрешить прерывания от таймера, программист должен сбросить флаг T0IF, чтобы застраховать себя от неожиданных сюрпризов. Следует помнить, что этот флаг не сбрасывается автоматически, когда процессор подтверждает запрос и переходит на выполнение процедуры обработки прерывания.

Для вычисления значения, которое следует загрузить в регистр таймера, чтобы сформировать заданную задержку во времени, программист может использовать простую формулу

$$A = 256 - C / 2,$$

где A - загружаемое в регистр таймера начальное значение; C - количество командных циклов (или число периодов внешних счетных импульсов) за время задержки. Деление на два учитывает влияние синхронизатора, который задерживает каждый счетный импульс до начала очередного периода длительностью в два командных цикла. Значение C в случае, если **предделитель** не используется, надо вычислять по формуле

$$C = \Delta t \times F / 4,$$

где Δt - длительность необходимой задержки в секундах; F - тактовая частота микроконтроллера в герцах. Деление на 4 учитывает тот факт, что каждый командный цикл длится четыре периода работы тактового генератора.

Например, если надо сформировать задержку на 160 мкс при частоте тактирования 4 МГц, то длительность задержки в командных циклах составит

$$C - \Delta t \times F / 4 = 160 \text{ мкс} \times 4 \text{ МГц} / 4 = 160 \times 10^{-6} \text{ с} \times 4 \times 10^6 \text{ Гц} / 4 = 160.$$

Следовательно, таймер необходимо инициализировать значением

$$A = 256 - 160 / 2 = 256 - 80 = 176.$$

Чтобы сформировать задержку, длительность которой превышает 512 командных циклов, придется использовать предделитель. Чтобы определить необходимый коэффициент деления, надо найти такое число, являющееся степенью двойки, при делении C на которое в остатке получается число, меньшее 512. Другими словами, надо выполнять деление числа C на 2 до тех пор, пока не получится число, меньшее 512.

Например, для задержки в 5 мс при частоте тактирования 4 МГц необходимое число командных циклов составляет

$$C = \Delta t \times F / 4 = 5 \text{ мс} \times 4 \text{ МГц} / 4 = 5 \times 10^{-3} \text{ с} \times 4 \times 10^6 \text{ Гц} / 4 = 5000.$$

Это число больше 512, поэтому делим его на 2, получаем 2500. Так как результат опять больше 512, то снова делим его на 2 и получаем 1250. Повторяя деление

еще два раза, получаем сначала 625, а затем 312,5. Потребовалась четыре раза разделить $C = 5000$ на 2, чтобы результат стал меньше 512. Два в степени 4 равно 16. Следовательно, необходимый коэффициент деления равен 16. При этом начальное значение счетчика таймера составит

$$A = 256 - 312,5 / 2 - 256 - 156,25 = 99,75.$$

Округляя в ближайшую сторону, получим 100.

Итак, для формирования задержки в 5 мс при частоте тактирования 4 МГц надо взять коэффициент деления 16 и инициализировать таймер значением 100.

Необходимо иметь в виду, что округление, выполненное нами в приведенных вычислениях, приведет к тому, что задержка окажется несколько меньше. Однако надо еще учитывать время, требуемое для срабатывания самого таймера, формирования запроса на прерывание и переключения процессора на процедуру его обработки. Кроме того, нельзя заранее сказать, сколько машинных команд потребуется компилятору PICC Lite для реализации фрагмента нашего кода. Поэтому реальная задержка всегда оказывается длиннее, чем это следует из наших расчетов.

Если требуется задержка, длительность которой должна быть выдержана с высокой точностью, то используют модуль СРР (мы обсудим его чуть позже) в режиме широтно-импульсного модулятора, чтобы сформировать импульс заданной ширины. Наконец, всегда можно подключить к микроконтроллеру внешний таймер, который вырабатывает импульсы с нужной точностью во времени. В некоторых случаях такое решение оказывается единственно правильным.

Для управления многими функциями микроконтроллера используется регистр OPTION. С его помощью задается режим работы предделителя, выбирается тип тактового генератора и устанавливается источник прерываний. Поэтому мы часто будем обращаться к этому регистру в наших программах.

В фирменной документации Microchip он называется OPTION_REG. Однако чаще всего, в том числе и в программах на PICC Lite, его именуют просто OPTION. Назначение разрядов этого регистра одинаково у всех микроконтроллеров среднего семейства (табл. 3.12).

Таблица 3.12. Регистр OPTION

Номер бита	Имя	Назначение
7	<code>_RBPU</code>	«Подтягивание» выходов порта В к положительному напряжению питания: 1 - выключено; 0 - включено
6	<code>INTEDG</code>	Активный фронт сигнала на входе <code>RB0/INT</code> , по которому вырабатывается запрос на прерывание: 1 — положительный фронт (с низкого уровня на высокий); 0 — отрицательный фронт (с высокого на низкий)
5	<code>T0CS</code>	Источник счетных импульсов для таймера <code>TMR0</code> : / - вход <code>T0CKI</code> ; 0 — командные циклы

Таблица 3.12. Регистр OPTION (окончание)

Номер бита	Имя	Назначение
4	TOSE	Активный фронт сигнала счетных импульсов таймера TMRO на входе TOSCI: 1 – отрицательный фронт; 0 - положительный фронт
3	PSA	Назначение предделителя: 1 - работает с WDT; 0 - работает с TMRO
2-0	PS2 - PSO	Выбор коэффициента деления предделителя*: 000 1:1 001 1:2 010 1:4 011 1:8 100 1:16 101 1:32 110 1:64 111 1:128

* При работе делителя с таймером TMRO из-за влияния синхронизатора реальный коэффициент деления в два раза больше указанного. - Прим. перев.

Кроме того, микроконтроллеры среднего семейства имеют специальный режим пониженного энергопотребления (sleep mode - «спящий режим»). При этом ток, потребляемый микроконтроллером, становится меньше 10 мкА. Обычно этот режим используется, если МК должен ожидать наступления какого-либо события. Наступление этого события приводит к формированию запроса на прерывание. Чтобы ввести микроконтроллер в «спящий режим», надо выполнить команду sleep. Таймеры в «спящем режиме» не работают.

Прежде чем использовать режим пониженного энергопотребления в своей разработке, надо убедиться, что эффект от этого будет заметным. На фоне значительной емкости батарей питания и большой мощности потребления других устройств выигрыш может оказаться несущественным.

Выйти из «спящего режима» можно в результате выполнения следующих событий: понижения и последующего повышения до нормального уровня напряжения питания; по сигналу сброса на входе _MCLR; по приходу запроса на прерывание; при срабатывании сторожевого таймера. Если причиной «пробуждения» микроконтроллера послужил запрос на прерывание, то следующая машинная команда выполнится, даже если обработчик прерывания не будет вызван.

Чтобы микроконтроллер перешел на обработку прерывания после выхода из «спящего режима», надо предварительно установить соответствующий бит разрешения в регистрах INTCON или PIE. Кроме того, прерывания должны быть разрешены установкой бита GIE регистра INTCON. После «пробуждения» микроконтроллер выполняет машинную команду, следующую за командой sleep, после чего переходит на процедуру обработки прерывания (вектор 0x004). Если флаг разрешения прерываний GIE сброшен в 0, то после выполнения команды, следующей за командой sleep, микроконтроллер продолжит выполнение программы.

Из-за описанной особенности процесса «пробуждения», чтобы исключить лишние операции перед выполнением обработчика прерывания, после команды sleep указывается команда пор (no operation - ничего не делать). При использовании PICC Lite соответствующий фрагмент программы выглядит следующим образом:

```
#asm
    sleep
    nop
#endasm
```

или

```
asm("sleep");
asm("nop");
```

Кроме того, многие микроконтроллеры имеют встроенную память данных EEPROM. Ее объем в МК PIC16F627 равен 128 байт. Ячейки EEPROM доступны для чтения и записи через специальные регистры. Для обращения к такой памяти можно также использовать специальные функции PICC Lite.

Некоторые микроконтроллеры, имеющие Flash-память команд (PIC16F62x и PIC16F87x), могут обращаться к этой памяти для чтения или записи. Эту возможность используют для хранения данных в энергонезависимой памяти (они не будут уничтожаться после выключения питания), а также если надо, чтобы программа могла изменить свой собственный код.

Для обращения к памяти данных EEPROM используются регистры EECON1, EECON2, EEADR и EEDATA. Регистр EEADR позволяет указать адрес нужной ячейки памяти. При выполнении операции записи в EEPROM в регистре EEDATA указывают записываемые данные, а при выполнении чтения из EEPROM в этот регистр помещаются прочитанные данные.

Регистры EECON1 и EECON2 позволяют задать тип доступа к памяти данных. Кроме того, с их помощью можно узнать о том, что операция с EEPROM уже выполнена. Из регистра EECON2 нельзя читать данные. После того как все подготовлено к записи в EEPROM, в данный регистр надо последовательно записать числа 0x055 и 0x0AA - это служит сигналом к выполнению операции. Назначение разрядов регистра EECON1 показано в табл. 3.13.

Таблица 3.13. Назначение разрядов регистра EECON1

Номер бита	Имя	Назначение
7-4	—	Не используется, читается 0
3	WRERR	Этот бит устанавливается в 1, если была ошибка записи
1	WREN	Когда этот бит установлен, запись в EEPROM разрешена
	WR	Устанавливается программистом в начале выполнения операции записи и автоматически сбрасывается после ее выполнения
0	RD	Устанавливается программистом в начале выполнения операции чтения и автоматически сбрасывается при выполнении следующей машинной команды

Для выполнения операции чтения из EEPROM служит следующий фрагмент кода:

```
EEADR = Address;      // Задали адрес ячейки EEPROM.
RD = 1;               // Начали операцию чтения.
EEPROM_data = EEDATA; // Сохранили в переменной EEPROM_data прочитанное значение.
```

Можно использовать для чтения из EEPROM встроенную функцию PICC Lite; тогда достаточно написать одну строку:

```
EEPROM_data = eeprom_read(Address);
```

Записать данные в EEPROM несколько сложнее:

```
EEADR = Address;      // Задали адрес ячейки EEPROM.
EEDATA = EEPROM_data; // Определили записываемые данные.
GIE = 0;              // Запретили прерывания.
WREN = 1;             // Разрешили запись в EEPROM.
EECON2 = 0x055;       // Эти две команды не должны быть
EECON2 = 0x0AA;       // разорваны из-за обработки прерывания.
WR = 1;              // Начали операцию записи.
GIE = 1;             // Разрешили прерывания.
while (WR == 1);      // Дождались завершения операции записи.
WREN = 0;            // Убрали сигнал разрешения записи в EEPROM.
```

Две следующие друг за другом операции записи в регистр EECON2 не могут быть прерваны процедурой обработки прерываний, запрос на которые может поступить в любой момент. Поэтому приходится запрещать обработку прерываний. Если программист нарушит последовательность подачи и снятия управляющих сигналов, то операция записи в EEPROM не будет выполнена.

Вместо того чтобы в цикле проверять состояние бита WR, ожидая сигнал успешного завершения операции записи, можно использовать прерывания. Для этого надо установить бит EEIE. После того как запись в EEPROM будет выполнена, автоматически установится флаг EEIF и будет сгенерирован запрос на прерывание.

Для выполнения записи в EEPROM также доступна встроенная функция

```
eeprom_write(Address, EEPROM_data);
```

Кроме TMRO в микроконтроллере PIC16F627 имеется еще два таймера: 16-разрядный TMR1 и 8-разрядный TMR2. Обычно они используются для работы с модулем захвата/сравнения (он будет описан ниже), но также могут применяться для формирования задержек во времени, как и TMRO.

Структурная схема таймера TMR1 показана на рис. 3.36. Этот 16-разрядный таймер имеет четыре возможных источника синхроимпульсов. Одним из таких источников может быть собственный встроенный генератор импульсов (в этом состоит особенность **TMR1**)¹. Тактирование от внутреннего генератора позволяет

¹ В документации Microchip работа от внутреннего источника импульсов частотой $F/4$, синхронизированных с системным тактовым генератором, называется *режимом таймера*. Работа от внешнего источника импульсов (при установленном бите **TMR1CS**) с включенной синхронизацией (**бит T1SYNCH сброшен**) называется *режимом синхронного счетчика*, а при установленном бите **T1SYNCH** – *режимом асинхронного счетчика*. — Прим. *перев.*

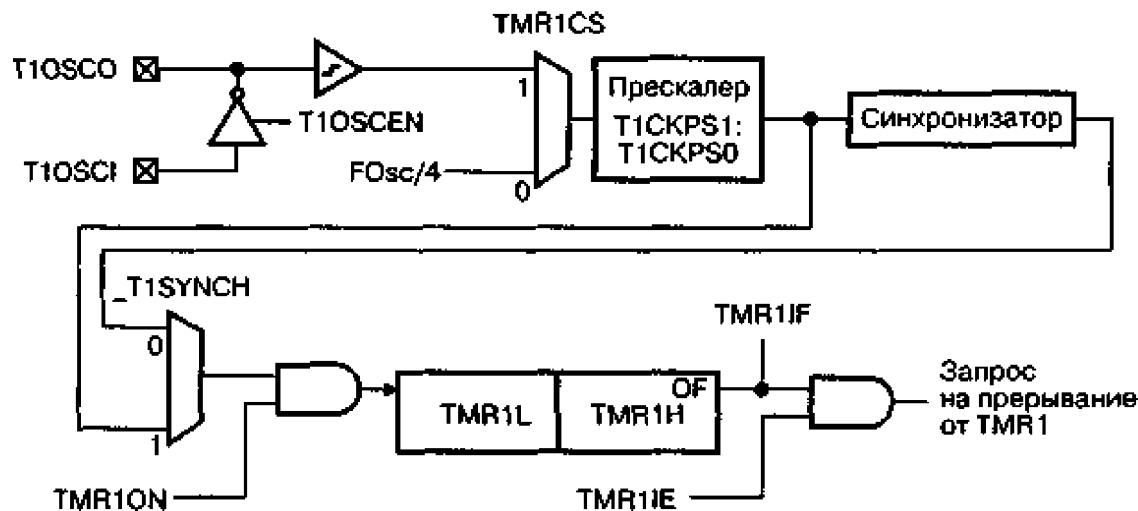


Рис. 3.36. Структурная схема таймера TMR1

таймеру продолжать работу¹, даже если микроконтроллер находится в «спящем режиме».

Для работы с таймером TMR1 предназначены два 8-разрядных регистра TMR1L и TMR1H; оба они доступны как для записи, так и для чтения. Как и в случае с TMRO, при записи в эти регистры каких-либо значений предделитель сбрасывается. Когда таймер переполняется, то устанавливается флаг TMR1F регистра PIR и генерируется запрос на прерывание. Если сброшен бит разрешения TMR1E регистра PIR, или бит разрешения прерываний GIE регистра INTCON, или бит разрешения прерываний от периферийных устройств PIR регистра INTCON, то запрос игнорируется процессором.

Для управления таймером TMR1 служит регистр T1CON. Назначение его битов показано в табл. 3.14.

Таблица 3.14. Регистр T1CON

Номер бита	Имя	Назначение
7-6	–	Не используется, читается 0
5-4	T1CPS1:T1CPS0	Выбор коэффициента деления прескалера: 11 - 1:8; 10 - 1:4; 01 - 1:2; 00 - 1:1
3	T1OSCEN	1 - разрешение работы внутреннего генератора тактовых импульсов
1	T1SYNCH	Если TMR1CS = 0, то используется синхронизация внешних тактовых импульсов с командными циклами процессора
1	TMR1CS	1 - используется керамический или кварцевый резонатор, подключенный к входам RB6 и RB7, или внешний источник, подключенный к входу RB6; 0 - тактирование командными циклами
0	TMR1ON	1 - работа таймера TMR1 разрешена; 0 - работа таймера TMR1 запрещена

¹ Только в режиме асинхронного счетчика. - Прим. перев.

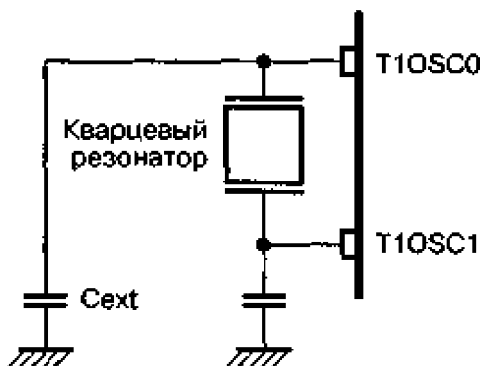


Рис. 3.37. Подключение генератора тактовых импульсов для таймера TMR1

Внешний источник тактовых импульсов обычно используют для разработки низкоскоростных приложений, работающих в реальном времени. Часто выбирают резонатор на 32,768 кГц (он применяется в электронных часах) и два внешних конденсатора емкостью 33 пФ. Можно применять резонатор на 100 или 200 кГц, но емкость конденсаторов в этом случае надо уменьшить до 15 пФ. Схема подключения резонатора и конденсаторов показана на рис. 3.37.

Если в качестве источника тактовых импульсов для TMR1 надо использовать командные циклы процессора, то бит T1SYNCH должен быть сброшен.

Если синхронизация тактовых импульсов не нужна, надо установить бит T1SYNCH; это можно сделать только в то время, пока микроконтроллер находится в «спящем режиме», так как при этом отключен главный тактовый генератор, управляющий выполнением команд, и есть гарантия, что не придет тактовый импульс, пока происходит обращение к таймеру.

Начальное значение счетчика таймера и соответствующее время задержки связаны соотношением

$$\Delta t = (65536 - A) \times P / F_1,$$

где Δt - длительность задержки в секундах; A - значение, загружаемое в регистр таймера; P - коэффициент деления прескалера; F_1 - частота тактовых импульсов в герцах (определяемая командными циклами, внутренним генератором таймера TMR1 или внешними импульсами в зависимости от используемого источника тактирования). Отсюда можно выразить значение, которым следует инициализировать счетчик таймера:

$$A = 65536 - (\Delta t \times F_1 / P).$$

При вычислении коэффициента деления P надо увеличивать его в два раза начиная с 1 до тех пор, пока число A не станет положительным (аналогично тому, как мы делали это при работе с таймером TMRO).

Структурная схема таймера TMR2 приведена на рис. 3.38. Его использование во многом аналогично описанному выше TMRO в режиме тактирования внутренними

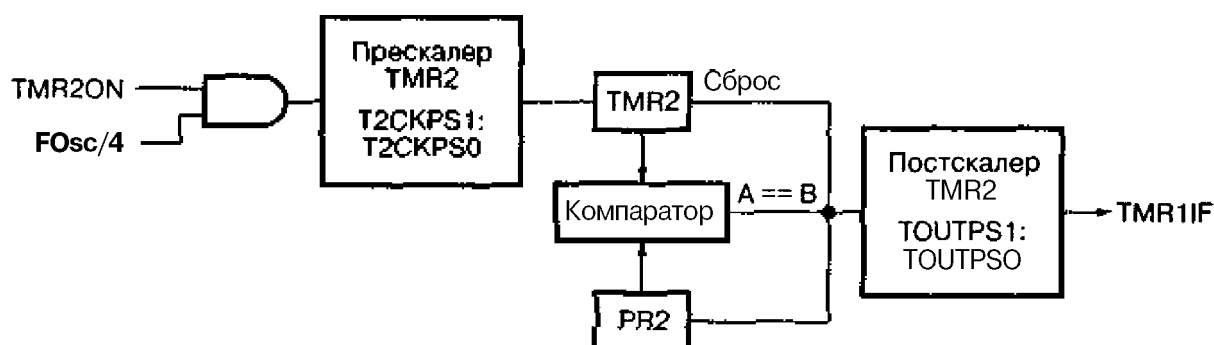


Рис. 3.38. Структурная схема таймера TMR2

синхроимпульсами. TMR2 также может работать совместно с широтно-импульсным модулятором (ШИМ, Pulse Width Modulation - PWM).

Текущее значение счетчика таймера TMR2 постоянно сравнивается с величиной, записанной в регистре PR2. В тот момент, когда оба значения совпадут, произойдет сброс таймера TMR2. Этот сигнал будет подан на модуль CCP.

Если TMR2 используется в режиме обычного таймера, в момент его переполнения происходит то же самое, что и с TMRO: увеличивается на 1 счетчик делителя (postscaler - постскалер), сигнал о переполнении которого, в свою очередь, является источником прерывания.

Для управления работой TMR2 служит регистр T2CON (табл. 3.15), доступный как для чтения, так и для записи. При выполнении операции записи обнуляются оба делителя (прескалер и постскалер).

Таблица 3.15. Регистр T2CON

Номер бита	Имя	Назначение
7	—	Не используется, читается 0
6-3	TOUTPS3:TOUTPS0	Выбор коэффициента деления постскалера: 1111 – 16:1; 1110 – 15:1; ... 0001 – 2:1; 0000 – 1:1
2	TMR2ON	1 – работа таймера TMR2 разрешена; 0 – работа таймера TMR2 запрещена
1-0	T2CKPS1:T2CKPS0	Выбор коэффициента деления прескалера: 1x – 16:1; 01 – 4:1; 00 – 1:1

Совместно с TMR2 не используется синхронизатор, так как источником тактовых импульсов для него могут являться только командные циклы процессора. Поэтому он тактируется моментом начала каждого командного цикла (а не каждым вторым командным циклом, как это было с предыдущими таймерами).

TMR2 срабатывает либо при переполнении, либо при достижении заданного в регистре PR2 значения. Формула для вычисления задержки:

$$At = P_1 \times (PR2 + 1) \times 4 / F,$$

где F - как и раньше, частота тактирования микроконтроллера в герцах, а P_1 - коэффициент деления прескалера. При нулевом значении PR2

$$At = (P_1 \times 256) \times 4 / F.$$

Обычно для формирования повторяющихся задержек я использую нулевое начальное значение счетчика таймера, а длительность задержки, задаваемую с помощью PR2, вычисляю по формуле

$$At = (P_1 \times [PR2 + 1 \mid 256]) \times 4 / (F \times P_2).$$

Для разрешения прерываний от таймера TMR2 необходимо установить бит TMR2IE регистра PIE1. При срабатывании таймера устанавливается флаг TMR2IF регистра PIR1. Благодаря своей высокой точности таймер TMR2 обычно используется при реализации асинхронных протоколов связи или для широтно-импульсной модуляции (ШИМ) сигнала.

Во многих случаях может пригодиться еще одно устройство, имеющееся в микроконтроллерах PIC16F627 и в некоторых других - это блок захвата/сравнения/ШИМ (Capture/Compare/WPM - CPP). Некоторые МК имеют два таких блока, поэтому в PIC16F627 блок CPP называется CPP1, хотя и является единственным.

Для управления модулем CPP1 предназначен регистр CPP1CON (табл. 3.16).

Таблица 3.16. Регистр CPP1CON

Номер бита	Имя	Назначение
7-6	—	Не используется, читается 0
5-4	ССР1Х:ССР1У	Два младшие бита числа СЕРСТ, загружаемого в компаратор и определяющего ширину импульса при широтно-импульсной модуляции (восемь старших разрядов находятся в регистре ССР1L)
3-0	ССР1М3: ССР1М0	Режим работы модуля CPP: 1 /хх - режим ШИМ; 1011 - режим сравнения — триггер специального события*; 1010 - режим сравнения — программное прерывание**; 1001 — режим сравнения — при срабатывании компаратора выход CPP принимает значение 0; 1000 — режим сравнения — при срабатывании компаратора выход CPP принимает значение 1; 0111 — выборка значения по положительному фронту каждого 16-го импульса; 0110 — выборка значения по положительному фронту каждого 4-го импульса; 0101 - выборка значения по положительному фронту каждого импульса; 0100 — выборка значения по отрицательному фронту каждого импульса; 00хх — модуль CPP выключен

* В режиме триггера специального события в момент срабатывания модуля CPP1 (то есть в момент совпадения значения таймера TMR1 и величины, загруженной в регистры ССР1Н: ССР1L) формируется запрос на прерывание и сбрасывается таймер TMR1. Если в микроконтроллере имеется АЦП, то в этот же момент на него подается сигнал запуска. — Прим. перев.

** В режиме программного прерывания в момент срабатывания модуля CPP 1 состояние вывода CPP 1 не изменяется. — Прим. перев.

Когда модуль CPP находится в режиме захвата (capture), то в определенные моменты времени 16-разрядное значение, хранящееся в счетчике таймера TMR1, переписывается в регистры ССР1Н:ССР1L модуля CPP. Это может происходить по переднему или заднему фронту импульсов, подаваемых на вход модуля CPP, а также по переднему фронту каждого четвертого или каждого шестнадцатого импульса. Режим задается в младших четырех битах регистра ССР1CON. Структурная схема модуля CPP в режиме захвата показана на рис. 3.39.

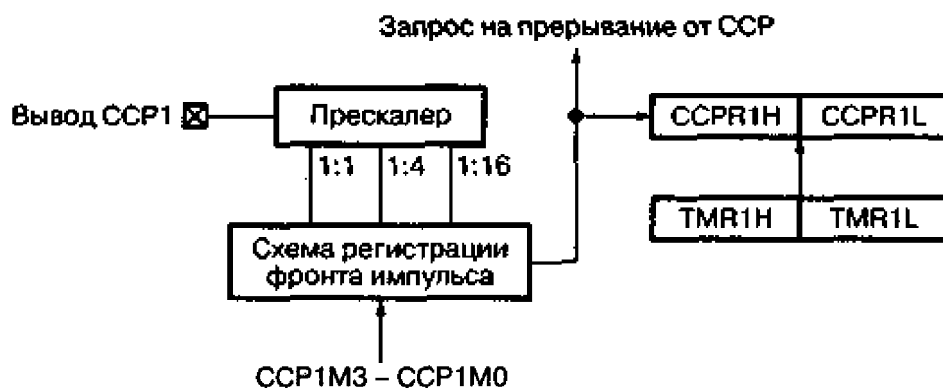


Рис. 3.39. Структурная схема модуля CCP в режиме захвата

Перед тем как разрешать работу модуля CCP в режиме захвата, надо разрешить работу таймера TMR1 (обычно в этом случае используется внутреннее тактирование таймера). Схема, срабатывающая по фронту входных импульсов (детектор фронта), на рис. 3.39 представляет собой четырехходовый мультиплексор (по числу режимов). На его входы подаются сигналы от предварительного делителя. Выбор нужного входа определяется младшими битами регистра CCP1CON. Сигнал с выбранного входа управляет срабатыванием регистров CCPR1H:CCPR1L, а также может формировать запрос на прерывание от модуля CCP.

В режиме захвата таймер TMR1 постоянно включен. По приходу на вход CCP1 активного фронта импульса (определяемого младшими битами регистра CCP1CON) происходит захват (загрузка значения из таймера в регистры модуля CCP1) и формируется запрос на прерывание. Процедура обработки прерывания должна сохранить захваченное значение счетчика таймера, так как оно будет потеряно в момент следующего срабатывания модуля CCP1.

Режим захвата часто используют для реализации функций, повторяющихся через заданные промежутки времени, или для формирования импульсов заданной ширины при широтно-импульсной модуляции. В последнем случае тактирование таймера TMR1 должно производиться от генератора достаточно высокой частоты, чтобы модуль CCP1 мог регулировать ширину импульсов с высокой точностью.

Когда модуль CCP1 работает в режиме **сравнения**¹, то значение счетчика таймера TMR1 сравнивается с величиной, предварительно загруженной в регистры CCPR1H:CCPR1L. В момент совпадения значений изменяется состояние вывода CCP1, который теперь работает в качестве выходного. Структурная схема модуля CCP1 в режиме сравнения показана на рис. 3.40. Этот режим обычно используют для того, чтобы переключать состояние какого-либо периферийного устройства с заданной задержкой во времени.

Из трех возможных режимов работы модуля CCP1, по мнению автора, наиболее полезен для разработчика режим широтно-импульсной модуляции (ШИМ, Pulse Width Modulation - PWM). Структурная схема модуля CCP1 в режиме ШИМ показана на рис. 3.41. Этот режим представляет собой своего рода комбинацию

¹ В режиме сравнения TMR1 должен работать в режиме таймера или синхронизированного счетчика; в режиме асинхронного счетчика операция сравнения не работает. - Прим. перев.

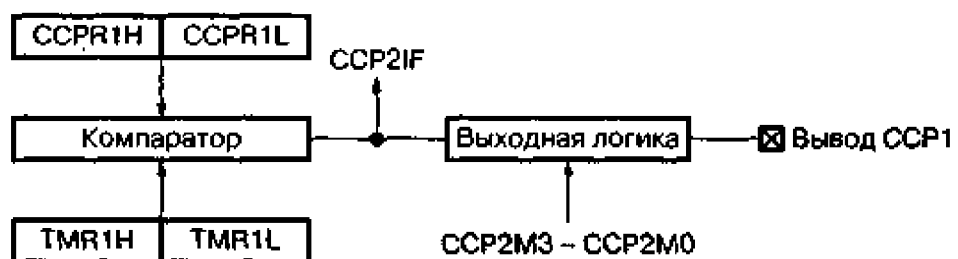


Рис. 3.40. Структурная схема модуля **CPP1** в режиме сравнения

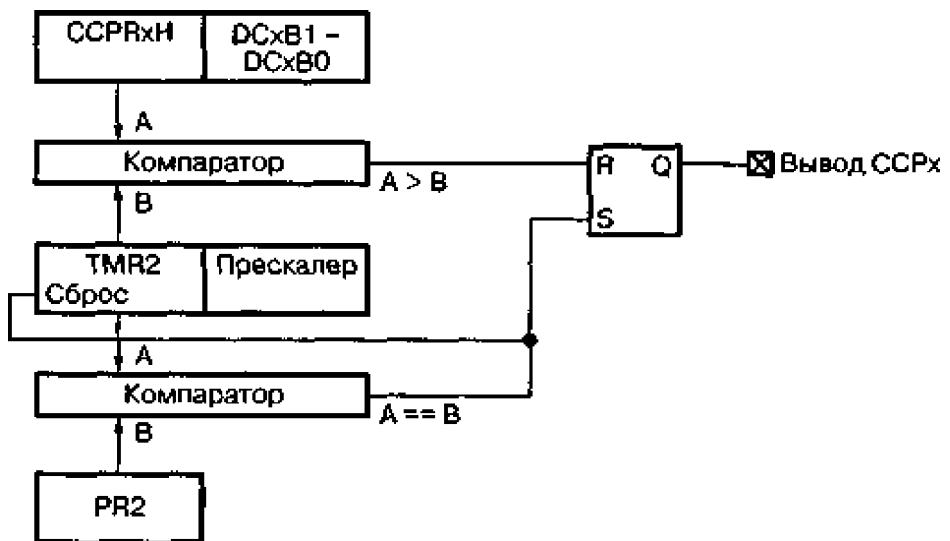


Рис. 3.41. Структурная схема модуля **CPP1** в режиме ШИМ

обычного режима работы таймера TMR2 и режима сравнения. Обычный режим позволяет осуществлять отсчет времени, а благодаря режиму сравнения формируется запрос на прерывание по истечении заданного времени.

В режиме ШИМ содержимое 8-разрядного счетчика таймера TMR2 сравнивается с содержимым регистра PR2. В момент совпадения обоих значений таймер сбрасывается, а на выводе **СРР1** устанавливается сигнал высокого уровня - начинается очередной период широтно-модулированного сигнала. В это время таймер TMR2 работает в 10-разрядном режиме - дополнительные два разряда обеспечивает предделитель с коэффициентом деления 1, 4 или 16 (относительно частоты системного тактового генератора). Это десятибитное значение сравнивается со значением, записанным в 8-разрядный регистр **CCPR1L** (младшие два разряда - в регистре **CCP1CON**). В момент совпадения обоих значений на выходе **CCP1** устанавливается сигнал низкого уровня - начинается вторая фаза периода широтно-модулированного сигнала. Она будет продолжаться до тех пор, пока снова не сравняются значения TMR2 и PR2 и не начнется новый период.

Период ШИМ-сигнала определяется по формуле

$$T = (PR2 + 1) \times P \times 4 / F,$$

где **P** - коэффициент деления прескалера таймера TMR2; **F** - частота системного тактового генератора в герцах.

Длительность импульса можно рассчитать следующим образом:

$$\Delta t = \text{ССР} \times P / F,$$

где ССР - это 10-разрядное значение, старшие восемь бит которого хранятся в регистре ССР1L, а младшие два бита - в пятом и четвертом разрядах регистра ССР1CON.

Например, мы хотим сформировать с помощью микроконтроллера PIC16F627 сигнал частотой 20 кГц (то есть с периодом 50 мс), у которого длительность импульса составляет 65%, а длительность паузы - 35% периода, при частоте системного тактового генератора 4 МГц. Во-первых, необходимо вычислить значение, которое будет загружено в регистры ССР1L и PR2. Если коэффициент деления прескалера $P = 1$, то

$$\text{PR2} = T \times F / (P \times 4) - 1 = 50 \text{ мкс} \times 4 \text{ МГц} / 4 - 1 = 50 - 1 = 49.$$

Так как 65% от 50 мкс составляет 32,5 мкс, то

$$\text{ССР} = \Delta t \times F / P = 32,5 \text{ мкс} \times 4 \text{ МГц} / 1 = 130.$$

В шестнадцатеричном виде число 130 представляется как 0x082, а в двоичном - 0010000010. Младшие два бита 10 надо записать в соответствующие разряды регистра ССР1CON, а старшие восемь битов 0010 0000, то есть число 0x020 = 32, - в регистр ССР1L. Другими словами, значение ССР1L находится как целое частное от деления ССР на 4, а остаток от деления определяет младшие два бита.

Установка нужного значения коэффициента деления производится с помощью двух младших разрядов регистра T2CON согласно табл. 3.17.

Таблица 3.17. Установка коэффициента деления прескалера TMR2

T2CKPS1:T2CKPS0	Коэффициент деления
00	1
01	4
1x	16

Широтно-импульсная модуляция часто применяется для управления работой периферийных устройств, например электродвигателей, так как ШИМ-сигнал легко может быть преобразован в аналоговый с помощью RC-цепи.

Как и многие другие микроконтроллеры, PIC16F627 имеет последовательный интерфейс ввода-вывода. Это не только позволяет подключать к МК внешнюю память и различные периферийные устройства, в частности АЦП, но и обеспечивает возможность связи МК с компьютером по протоколу RS-232.

Следует заметить, что возможности настройки модуля последовательного интерфейса весьма скудны. Они ограничены простейшей реализацией¹ протокола

¹ Сигналы интерфейса RS-232 кодируются в отрицательной логике: низкому уровню соответствует большее значение напряжения, а высокому - меньшее. Модуль USART микроконтроллеров PICmicro использует положительную логику и не имеет аппаратных устройств для инвертирования сигнала, которое приходится реализовывать с помощью дополнительных элементов. - Прим. перев.

RS-232 и минимальными функциями, обеспечивающими синхронное последовательное соединение с периферийными устройствами.

Модуль USART (универсальных синхронно-асинхронный приемопередатчик) микроконтроллеров PIC16F627 наиболее подходит для обеспечения асинхронной последовательной передачи данных.

Режим синхронной передачи (синхронизированный тактовыми импульсами) показан на рис. 3.42. Принятый сигнал защелкивается во входном регистре приемника по заднему фронту синхроимпульсов. Правила формирования сигнала передатчиком и декодирования принятого сигнала приемником будут подробно рассмотрены позже, а пока будем представлять кодирующее и декодирующее устройства на структурной схеме модуля USART в виде «черных ящиков».



Рис. 3.42. Временные диаграммы сигналов в режиме синхронной передачи

Модуль USART состоит из трех устройств: генератора синхроимпульсов (baud rate generator - BRG), блока передачи и блока приема. Структурная схема генератора показана на рис. 3.43.

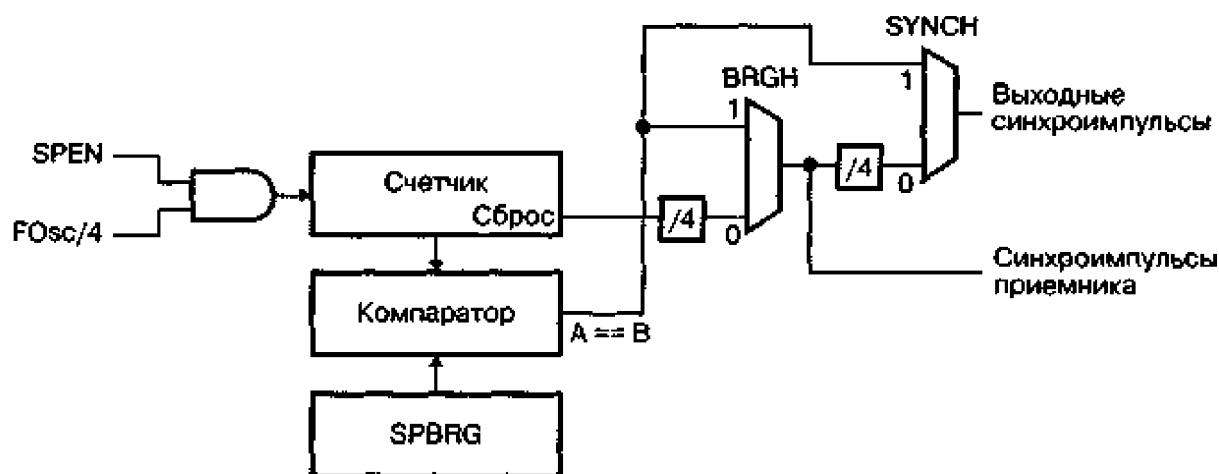


Рис. 3.43. Структурная схема блока формирования тактовых импульсов модуля USART

Значение, хранящееся в регистре SPBRG, сравнивается компаратором со значением счетчика. В момент совпадения обоих значений счетчик сбрасывается и генерируется импульс Reset - заканчивается очередной цикл работы генератора.

Работа счетчика управляется битом SPEN (Serial Port ENable - разрешение работы последовательного порта), битом SYNCH, состояние которого определяет, в каком режиме находится порт (в синхронном или асинхронном), и битом BRGH, с помощью которого устанавливается скорость передачи данных.

К сожалению, биты управления разбросаны по разным регистрам. Мы опишем их подробно чуть позже, когда разберем все три режима работы модуля USART.

В асинхронном режиме скорость передачи данных (бит в секунду) определяется по формуле

$$R = F / (16 \times 4^{1-BRGH} \times (SPBRG + 1)) .$$

Если задана скорость передачи R , то

$$SPBRG = F / (R \times 16 \times 4^{1-BRGH}) - 1.$$

Например, при частоте системного тактового генератора 4 МГц, если необходимо передавать данные со скоростью 1200 бод (то есть 1200 бит/с), при сброшенном бите BRGH понадобится загрузить в регистр SPBRG значение

$$SPBRG = 4 \text{ МГц} / (1200 \text{ с}^{-1} \times 16 \times 4^{1-0}) - 1 = 4 \times 10^6 / (1200 \times 16 \times 4) - 1 = 51,0833.$$

Так как округлять нужно до ближайшего целого, на самом деле в регистр SPBRG придется загрузить число 51; при этом фактическая скорость передачи данных составит 1201,9 бит/с, что дает ошибку в 0,16% по сравнению с заданным значением 1200 бит/с. К счастью, эта ошибка незначительна для приемника.

Блок передачи USART может пересылать за один цикл работы восемь или девять битов в синхронном или асинхронном режиме. Структурная схема блока передатчика USART приведена на рис. 3.44.

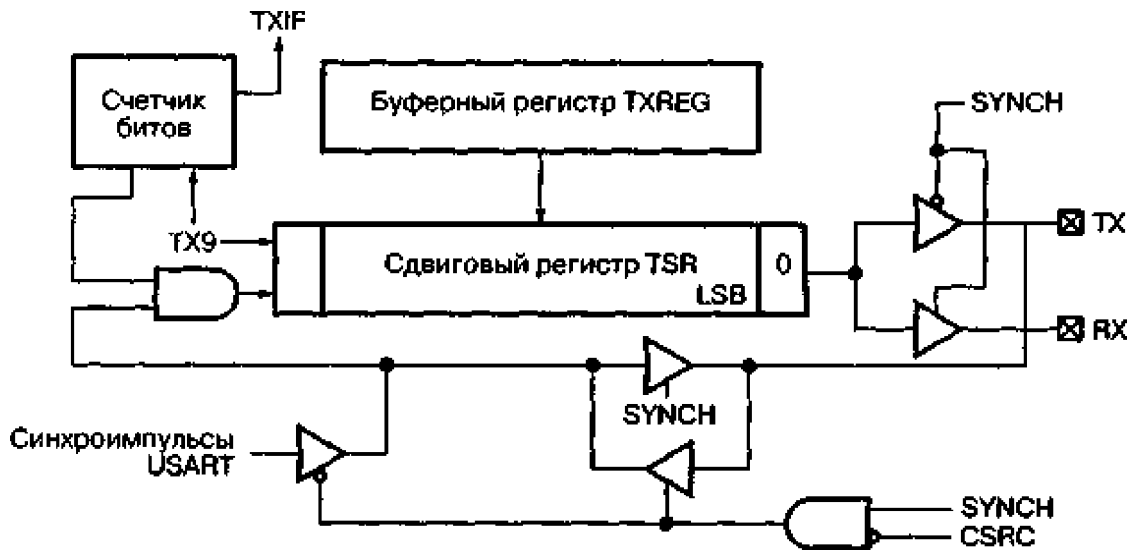


Рис. 3.44. Структурная схема блока передатчика модуля USART

Когда бит SYNCH установлен, передаваемые данные появляются на выходе RX синхронно с тактовыми импульсами, принимаемыми со входа TX или выводимыми на выход TX (синхронный режим передачи).

Восемь передаваемых двоичных разрядов хранятся в регистре TXREG, этот регистр загружается программно. Низкий уровень сигнала CSRC соответствует режиму ведомого (внешний тактовый сигнал). Как только закончится передача

предыдущего слова, данные из регистра TXREG автоматически записываются в сдвиговый регистр TSR. На это требуется один командный цикл, по завершении которого устанавливается флаг TXIF (бит 4 регистра PIR1) и формируется запрос на прерывание. Это прерывание можно разрешить/запретить установкой/сбросом бита TXIE (бит 4 регистра PIE1). Сдвиговый регистр преобразует параллельный код в последовательный: с каждым тактовым импульсом хранящая в нем информация сдвигается в сторону старших разрядов, а старший разряд передается на выход.

Буферизация передаваемых данных с помощью регистра TXREG позволяет производить запись очередного слова в регистр передатчика в любой момент, без ожидания конца передачи предыдущего слова.

В асинхронном режиме перед началом передачи очередного 8-разрядного слова формируется так называемый *стартовый бит* (он кодируется нулевым значением). Затем передаются информационные биты (начиная с младшего), после чего следует *стоповый бит*. Нулевой передаваемый информационный разряд кодируется импульсом низкого уровня, единичный - импульсом высокого уровня (NRZ-кодирование). Перед стоповым битом в зависимости от настроек может формироваться специальный *бит четности*.

Блок приемника модуля USART устроен намного сложнее, чем передатчик. Это объясняется сложностью процесса декодирования принимаемого сигнала. Структурная схема приемника показана на рис. 3.45.

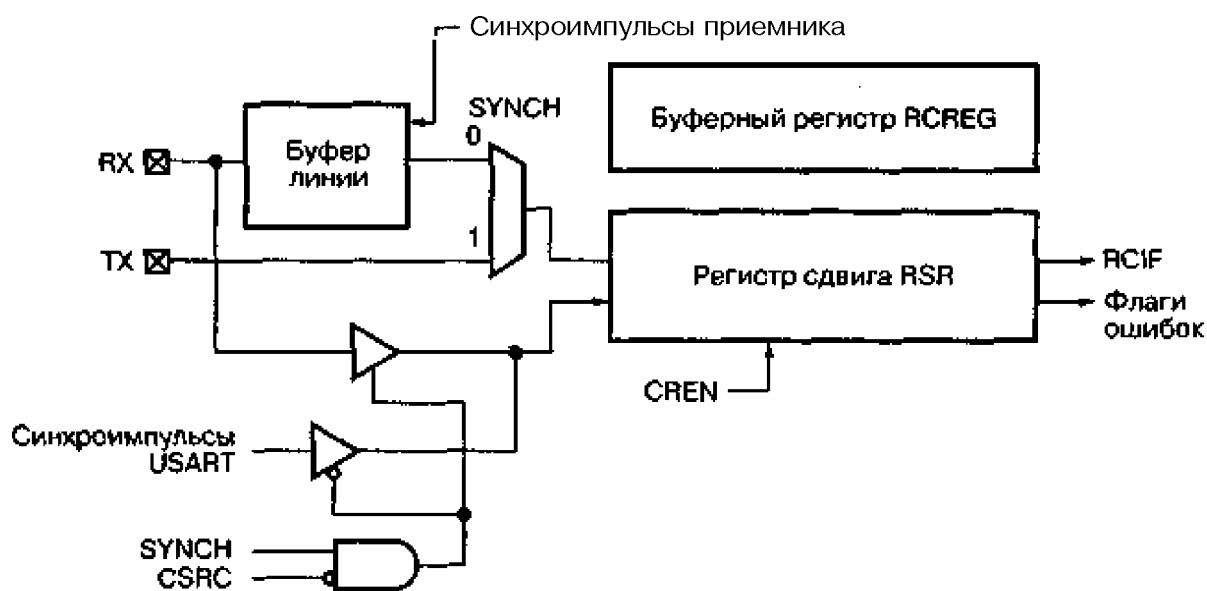


Рис. 3.45. Структурная схема блока приемника модуля USART

В режиме синхронного приема входные данные поступают на сдвиговый регистр, где преобразуются в параллельный код. При этом может использоваться либо внешний (режим ведомого), либо внутренний генератор синхроимпульсов (режим ведущего), которые будут фиксировать моменты времени, когда принимаемые данные имеют правильное значение.

В режиме асинхронного приема, когда дополнительные синхрои импульсы отсутствуют, момент фиксации данных определяется с помощью генератора синхрои импульсов приемника (Receiver Sensor Clock - RSC). Этот генератор работает на частоте, которая в 16 раз больше частоты передачи данных. Выборка текущего значения принимаемых данных со входа RX производится трижды, примерно в середине интервала, соответствующего очередному биту информации (рис. 3.46). По трем выбранным уровням входного сигнала специальный *мажоритарный элемент* формирует значение, за которое «голосует» большинство (хотя бы два) из них. Это значение подается на сдвиговый регистр RSR. Так повторяется восемь (или девять, если передавался бит четности) раз для всех разрядов передаваемого слова, после чего еще три раза происходит выборка входного сигнала для декодирования стопового бита. Синхронизация генератора приемника осуществляется по стартовым битам: приемник ожидает, пока все три выборки во время приема стартового бита не дадут нулевое значение уровня входного сигнала.

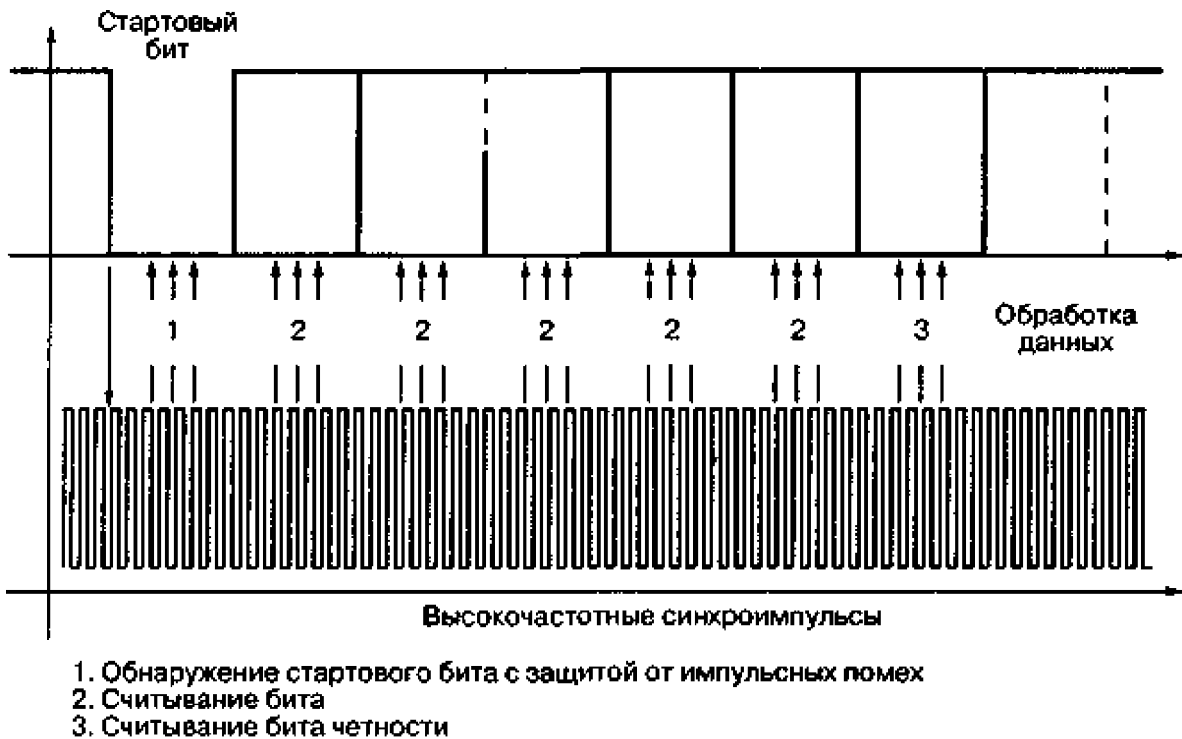


Рис. 3.46. Асинхронный прием данных

В приемнике применяется двойная буферизация данных с помощью регистра RCREG, который на самом деле представляет собой очередь (FIFO-буфер) глубиной в два уровня. Если приемник заканчивает декодирование стопового бита очередного 8-разрядного слова, а два предыдущих еще не прочитаны из регистра RCREG, то возникает ошибка переполнения - устанавливается флаг OERR (бит 1 регистра RCSTA). Сбросить этот флажок можно программно. Еще одна возможная ошибка — неправильное значение стопового бита. Ей соответствует флаг FERR (бит 2 регистра RCSTA).

Назначение разрядов регистра состояния передатчика TXSTA показано в табл. 3.18, а регистра состояния приемника RCSTA - в табл. 3.19.

Таблица 3.18. Назначение разрядов регистра TXSTA

Номер бита	Имя	Назначение
7	CSRC	Выбор источника синхронизации для синхронного режима: 1 – режим ведущего (используется внутренний генератор синхроимпульсов BRG); 0 - режим ведомого (синхронизация от внешнего генератора); в асинхронном режиме состояние этого бита значения не имеет
6	TX9	Разрешение передачи 9-го бита (четности): 1 - 9-й бит разрешен; 0 - 9-й бит запрещен (8-разрядная передача)
5	TXEN	Разрешение работы передатчика: 1 – передатчик включен; 0 – передатчик выключен
4	SYNC	Выбор режима USART: 1 - синхронный режим; 0 – асинхронный режим
3	–	Зарезервирован, читается 0
2	BRGH	Выбор скорости передачи в асинхронном режиме: 1 - высокоскоростной режим; 0 - низкоскоростной режим В синхронном режиме состояние этого бита значения не имеет
1	TRMT	Состояние передатчика: 1 – сдвиговый регистр TSR свободен; 0 – сдвиговый регистр TSR занят
0	TXD	Значение 9-го разряда (бита четности) передаваемых данных

Таблица 3. / 9. Назначение разрядов регистра RCSTA

Номер бита	Имя	Назначение
7	SPEN	Разрешение работы приемника: 1 - включен; 0 – выключен
6	RX9	Разрешение приема 9-го бита (четности): 1 - 9-й бит разрешен; 0 - 9-й бит запрещен (8-разрядный прием)
5	SREN	Разрешение однократного приема (одного байта) данных в синхронном режиме ведущего; этот бит сбрасывается после приема байта
4	CREN	Разрешение многократного приема данных
3	ADDEN	Разрешение режима детектирования адреса (имеется не у всех МК PICmicro)
2	FERR	Флаг ошибки кадровой синхронизации
1	OERR	Флаг ошибки переполнения приемного буфера
0	RX9D	Значение 9-го разряда (бита четности) принятых данных

Для инициализации режима асинхронной передачи данных служит следующий фрагмент программы:

```
SYNCH = 0;           // Установлен асинхронный режим.
BRGH = 0;           // Низкоскоростной режим передачи.
SPBRG = DataRate;   // Скорость передачи.
SPEN = 1;           // Разрешена работа USART.
TX9 = RX9 = 0;      // 8-разрядный режим передачи и приема.
TXEN = 1;           // Разрешена работа передатчика.
```

Чтобы передать один байт, достаточно двух строк кода:

```
while (TMR1 == 0);    // Ждать, пока не освободится регистр TSR.
TXREG = SendData;     // Загрузить в TXREG передаваемые данные.
```

Здесь бит TMR1 используется для определения момента освобождения сдвигового регистра TSR. Есть и другой способ: можно разрешить прерывания установкой бита TXIE. Тогда в процедуре обработки прерываний надо будет проверить состояние флага TXIF. Если он установлен, то прерывание сформировано блоком передатчика. Значит, можно загрузить в регистр TXREG очередной байт.

Для приема байта **служит** следующий фрагмент программы:

```
while (RXIF == 0);    // Ждать, пока будет принят очередной байт.
ReceiveData = RCREG;   // Получить принятые данные.
RXIF = 0;             // Сбросить флаг прерывания.
```

Для работы с аналоговыми сигналами порта А микроконтроллера PIC16F627 предназначены два *компаратора*. С их помощью можно узнать, превышает ли входное напряжение заданный порог. Величина порога задается внутренним или внешним источником опорного напряжения.

При изменении состояния компаратора может формироваться запрос на прерывание. Выходное значение компаратора можно записать в выходной порт микроконтроллера.

Работа компаратора поясняется на рис. 3.47. Если напряжение на входе «+» оказывается больше, чем на входе «-», то на **выходе** компаратора формируется сигнал высокого уровня, в противном случае - сигнал низкого уровня. Для управления работой обоих компараторов предназначен регистр CMCON (табл. 3.20). Состояние выходов компараторов можно узнать по **состоянию** битов C1OUT и C2OUT этого регистра.

Все возможные комбинации состояний битов CM2:CM0 и CIS расшифрованы в табл. 3.21.

При включении питания, когда биты CM2:CM0 = 000, входы RA3 - RA0 находятся в режиме ввода аналоговой информации и не могут быть использованы для обычного цифрового ввода-вывода. Если надо переключить эти выходы в режим цифрового ввода-вывода, то необходимо установить все биты CM2:CM0, записав значение 0x007 в регистр CMCON.

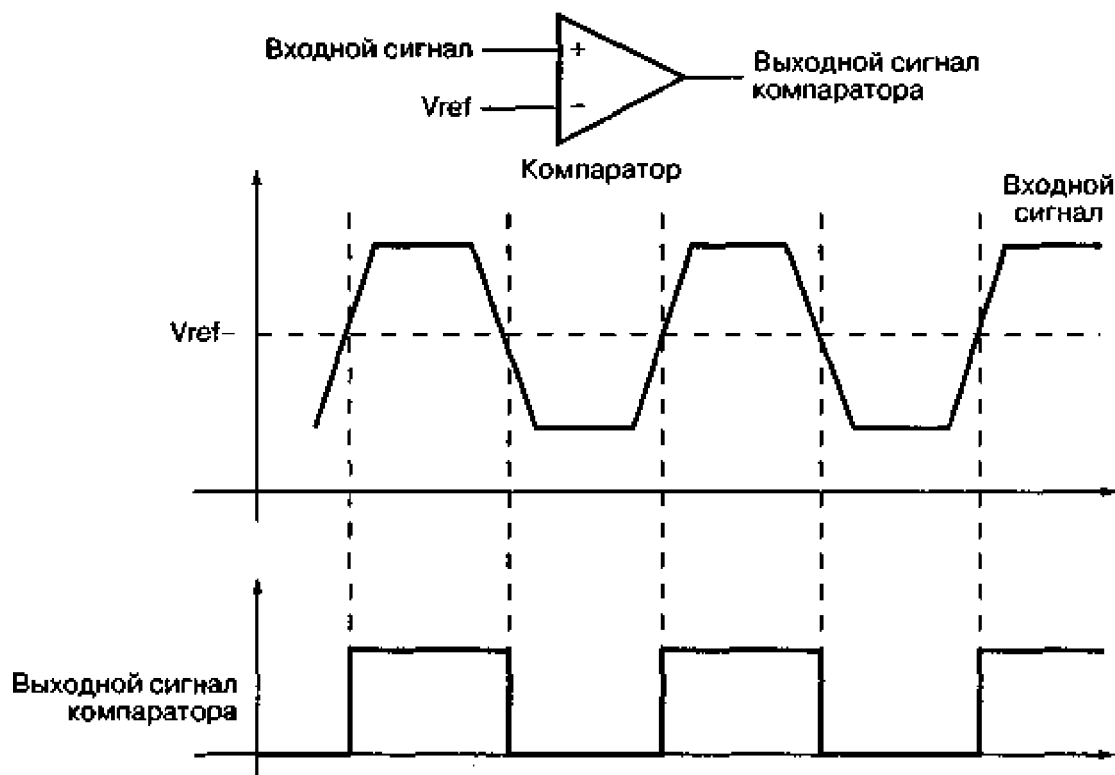


Рис. 3.47. Принцип работы компаратора

Таблица 3.20. Регистр CMCON

Номер бита	Имя	Назначение
7	C2OUT	Выход второго компаратора
6	C1OUT	Выход первого компаратора
5	C2VINV	Если этот бит установлен, то выход второго компаратора инвертируется
4	C1VINV	Если этот бит установлен, то выход первого компаратора инвертируется
3	CIS	Переключение входов компараторов
2-0	CM2:CM0	Режим работы компараторов

Таблица 3.21. Режимы работы компараторов

CM	CIS	Вход «+» компаратора 1	Вход «-» компаратора 1	Вход «+» компаратора 2	Вход «-» компаратора 2	Выход компаратора 1	Выход компаратора 2
000	X	RA0	RA3 (1)	RA2	RA1	C1OUT = 0	C2OUT = 0
001	0	RA2	RA0	RA2	RA1	C1OUT	C2OUT
00 J	1	RA2	RA3	RA2	RA1	C1OUT	C2OUT
0JO	0	Vref	RA0	Vref	RA1	C1OUT	C2OUT
010	1	Vref	RA3	Vref	RA2	C1OUT	C2OUT
011	X	RA2	RA0 (2)	RA2	RA1	C1OUT	C2OUT
100	X	RA3	RA0	RA2	RA1	C1OUT	C2OUT
101	X	GND	GND (3)	RA2	RA1	C1OUT = 0	C2OUT

Таблица 3.21. Режимы работы компараторов (окончание)

CM	CIS	Вход «+» компаратора 1	Вход «-» компаратора 1	Вход «+» компаратора 2	Вход «-» компаратора 2	Выход компаратора 1	Выход компаратора 2
ПО	X	RA2	RA0	RA2	RA1	C1OUT = = RA3 (4)	C2OUT = = RA4 (5)
111	X	GND (6)	0	0			

Примечания:

(1) При CM2:CM0 – 000 выводы RA3 - RA0 не могут быть использованы для обычного ввода-вывода.

(2) Вывод RA3 может быть использован для обычного ввода-вывода.

(3) Выводы RA0 и RA3 могут быть использованы для обычного ввода-вывода.

(4) Вывод RA3 является выходом первого компаратора.

(5) Вывод RA4 является выходом (с открытым стоком) второго компаратора.

(6) Выводы RA3 – RA0 могут быть использованы для обычного ввода-вывода.

Vref – внутренний источник опорного напряжения.

Запрос на прерывание от компараторов формируется, если состояние любого из них изменяется¹. При этом устанавливается флаг CMIF. Для разрешения обработки запроса на прерывание необходимо установить биты разрешения CMIE, PEIE и GIE. Если любой из этих трех битов сброшен, то прерывание не генерируется, но флаг CMIF устанавливается в любом случае. Программа обработки прерывания должна, во-первых, прочесть текущее значение регистра CMCON, а во-вторых, сбросить флаг CMIF.

Для формирования опорного напряжения можно использовать внутренний источник. Для этого в микроконтроллере имеется специальный четырехразрядный цифро-аналоговый преобразователь (ЦАП). Опорное напряжение Vref встроенного источника задается с помощью регистра VRCON (табл. 3.22).

Таблица 3.22. Регистр VRCON

Номер бита	Имя	Назначение
7	VREN	Разрешение работы внутреннего источника опорного напряжения: 1 - разрешено; 0 - запрещено
6	VROE	Разрешение подачи напряжения Vref на вход RA2: 1 — разрешено; 0 - запрещено
5	VRR	Выбор диапазона напряжений Vref: 1 - нижний диапазон; 0 - верхний диапазон
4	-	Не реализовано, читается как 0
3-0	VR3:VR0	Выбор значения напряжения Vref

¹ Возможна и программная установка флага CMIF; в этом случае моделируется возникновение прерывания от модуля компараторов независимо от того, изменилось ли состояние любого из них. - Прим. перев.

При установленном бите VRR регистра VRCON (нижний диапазон опорных напряжений) выходное напряжение внутреннего источника определяется по формуле

$$V_{ref} = V_{dd} \times (Vrcon \& 0x00F) / 24,$$

а при сброшенном бите VRR (верхний диапазон)

$$V_{ref} = V_{dd} \times (8 + (Vrcon \& 0x00F)) / 32.$$

Здесь Vdd - это напряжение питания микроконтроллера; Vrcon - число, записанное в регистр VRCON; & - поразрядная операция AND. Заметим, что максимальное опорное напряжение нижнего диапазона составляет 15/24 (то есть чуть меньше 2/3) от напряжения питания Vdd, а максимальное напряжение верхнего диапазона - 3/4 от напряжения питания.

3.5. ПРОГРАММАТОР EL CHEAPO

Для программирования микроконтроллеров PICmicro существует множество различных устройств. Дело в том, что принцип работы программатора достаточно прост и его несложно сделать самому. Самый простой программатор состоит всего из нескольких элементов, а наиболее сложные коммерческие модели могут стоить несколько тысяч долларов.

Для того чтобы вы могли изготовить устройства, описанные в следующих двух главах книги, я разработал несложную схему, предназначенную для программирования микроконтроллера PIC16F627. Вам потребуется лишь около часа, чтобы реализовать эту конструкцию, а стоимость используемых элементов не превысит нескольких долларов. Я назвал свой программатор El Cheapo¹.

Несомненное преимущество микроконтроллеров PICmicro — возможность *внутрисхемного программирования* (In-Circuit Serial Programming - ICSP). Благодаря этому разработчик не должен использовать в своей конструкции дорогие разъемы для размещения микроконтроллера, из которых его можно было бы оперативно извлекать для того, чтобы вставить в программатор. Также не нужны сложные схемы, обеспечивающие перепрограммирование МК.

По протоколу ICSP микроконтроллер может быть запрограммирован уже после того, как все его выводы будут припаяны к печатной плате. Кроме того, разработчик может после этого в любой момент изменить программу, если захочет исправить какую-нибудь ошибку или добавить новые функциональные возможности. Благодаря простоте ICSP можно продавать незапрограммированные устройства - тогда пользователь сможет выбирать программу по своему усмотрению.

Выводы микроконтроллера, участвующие в процессе внутрисхемного программирования, показаны в табл. 3.23.

¹ Схема программатора и программное обеспечение, необходимое для его работы, доступны на сайте издательства www.dmk.ru, а также на сайте автора этой книги www.myke.com. - Прим. *перев.*

Таблица 3.23. Выводы микроконтроллеров PICmicro, используемые для внутрисхемного программирования

Сигнал	12C5xx	16C50x	МК среднего семейства		
			корпус с 18 выводами	корпус с 28 выводами	корпус с 40 выводами
1 - Vpp	4 - <code>_MCLR</code>	4 - <code>_MCLR</code>	4 - <code>_MCLR</code>	1 - <code>_MCLR</code>	1 - <code>_MCLR</code>
2 - Vdd	1 - Vdd	1 - Vdd	14 - Vdd	26 - Vdd	11, 32 - Vdd
3 - GND	8 - Vss	14 - Vss	5 - Vss	8, 21 - Vss	21, 31 - Vss
4 - DATA	7 - GP0	13 - RB0	13 - RB7	28 - RB7	40 - RB7
5 - CLOCK	6 - GP1	12 - RB1	12 - RB6	27 - RB6	39 - RB6

Для записи программы в микроконтроллер или чтения ранее записанной программы необходимо перевести МК в режим программирования, подав напряжение от 12 до 14 В на вывод `_MCLR/Vpp`. Затем в течение нескольких миллисекунд необходимо поддерживать на линиях Data и Clock сигнал низкого уровня. После этого данные можно записывать или считывать в последовательном коде, используя линию Data. Вход Clock используется для тактирования.

Вход Vdd при напряжении 5 В потребляет ток от 20 до 50 мА, поэтому для питания подойдет микросхема 78L05 или простейший самодельный стабилизатор напряжения на одном стабилитроне. Этот вопрос уже обсуждался в начале главы (см. рис. 3.18 и 3.19).

Для коммутации напряжений Vpp и Vdd в схеме программатора можно использовать транзисторные ключи. Микроконтроллеры PICmicro имеют встроенную цепь, которая обеспечивает подачу напряжения низкого уровня на вход `_MCLR/Vpp` в то время, пока этот вывод отключен от внешнего источника напряжения. Временные диаграммы сигналов для инициализации режима программирования показаны на рис. 3.48.

Когда напряжение на выводе `_MCLR/Vpp` достигает 12 В, сбрасывается внутренний программный счетчик микроконтроллера. Он используется для формирования

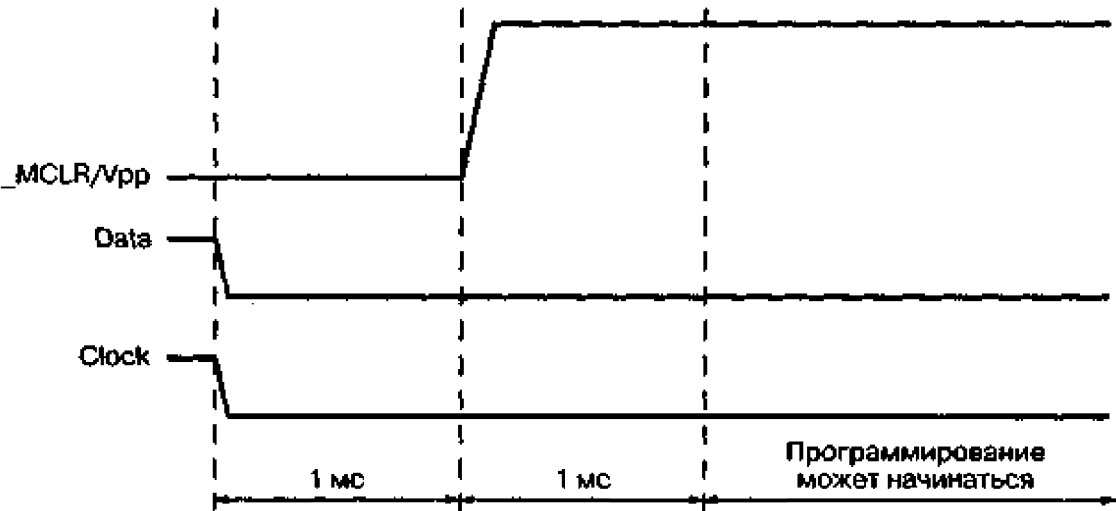


Рис. 3.48. Инициализация режима программирования

текущего адреса программной памяти. По адресу 0x02000 находится специальное слово идентификации ID, а по адресу 0x02007 - слово конфигурации (см. табл. 3.5).

Запись программы в микроконтроллер производится по синхронному последовательному протоколу, то есть побитно, младшим байтом вперед. Фиксация данных на линии Data производится по заднему фронту синхроимпульсов Clock, то есть в середине очередного такта (рис. 3.49). Минимальная длительность одного периода синхроимпульсов составляет 200 нс. При программировании используются 6-разрядные команды (табл. 3.24).

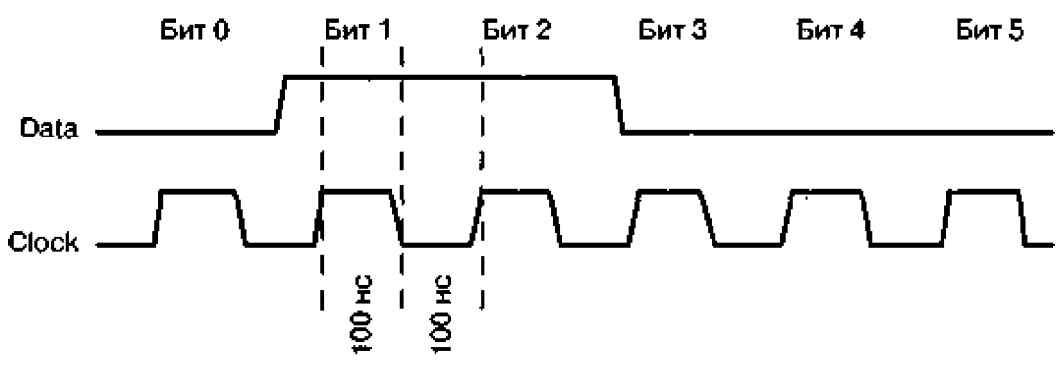


Рис. 3.49. Временные диаграммы сигналов при передаче 6-разрядных команд программатора

Таблица 3.24. Команды режима программирования

Команда	Код	Данные
Загрузка слова конфигурации	000000	07FFE
Загрузка команд в <i>память</i> программ	000010	0, 14-разрядная команда, 0
Загрузка данных в память <i>данных</i>	000011	Байт данных, сдвинутый на 1 разряд влево
Считывание команд из памяти программ	000100	0, 14-разрядная команда, 0
Считывание данных из памяти данных	000101	Байт данных, сдвинутый на 1 разряд влево
Инкремент программного счетчика	000110	
<i>Начало</i> программирования	001000	
Стирание всей памяти программ	001001	
Стирание всей памяти данных	0.01011	

Программируемые данные (то есть коды команд, записываемых в программную память, или значения, записываемые в память данных) начинают передаваться спустя 1 мкс после подачи соответствующей 6-разрядной команды. Коды команд при программировании сдвигаются влево на один разряд; в результате получается, что 14-разрядный код слева и справа дополняется до шестнадцати разрядов нулевыми битами.

Цикл программирования Flash-памяти микроконтроллеров выглядит следующим образом:

1. Посылается команда 0b000010 загрузки в память программ, после которой следует сдвинутый на 1 бит влево код очередной команды.

2. Посылается команда начала программирования 0b001000.
3. Выдерживается пауза 10 мс.

Аналогично программируются байты идентификации и конфигурации. Для этого посылается команда 0b000000 и следующий за ней байт 0x07FFE; в результате в программном счетчике устанавливается адрес 0x02000. После этого описанные три пункта повторяются каждый раз для очередного байта идентификации или конфигурации. Для увеличения программного счетчика на 1 в конце очередного цикла подается команда 0b000110.

Еще не запрограммированные ячейки Flash-памяти содержат единичные значения во всех битах, поэтому при программировании нужные биты устанавливаются в нулевое состояние. Для стирания программы можно опять записать во все биты единичные значения. Это можно сделать автоматически, используя специальную команду стирания 0b001001 для программной памяти и команду 0b001011 для памяти данных. Но я предпочитаю использовать способ, который позволяет стереть и память команд, и память данных даже в тех случаях, когда в слове конфигурации установлен бит защиты. Для этого надо выполнить следующие действия:

1. Подать напряжение Vpp, удерживая низкий уровень на выводах Data и Clock.
2. Выполнить команду загрузки слова конфигурации (0b0000000 и 0x07FFE).
3. Семь раз выполнить инкремент программного счетчика (команда 0b000110) для получения адреса регистра конфигурации 0x02007.
4. Послать команду 0b0000001.
5. Послать команду 0b0000111.
6. Послать команду начала программирования 0b0000000.
7. Выдержать паузу 10 мс.
8. Послать команду 0b0000001.
9. Послать команду 0b0000111.

Здесь используются две недокументированные команды PICmicro - 0b0000001 и 0b0000111.

После выполнения этих девяти действий память программ и память данных будет стерта и подготовлена для последующего программирования.

В процессе программирования после записи в программную память микроконтроллера очередной команды необходимо увеличивать значение программного счетчика на 1, выполнив команду 0b000110. Если какие-либо адреса не содержат кодов команд или содержат код 0x03FFF, можно пропустить шаг записи команды, просто выполнив инкремент программного счетчика.

После окончания программирования надо на короткое время снять напряжение Vpp, а затем подать его снова и прочитать только что записанную программу с целью проверки ее правильности (этап *верификации*). Если верификация прошла удачно, надо повторить ее при минимальном напряжении питания Vdd (около 4,5 В), снова предварительно убрав, а затем подав напряжение Vpp. В третий раз процесс верификации повторяется при максимальном напряжении питания (около 5,5 В).

Некоторые программаторы (включая El Cheapo, а также программатор PIC-Start Plus фирмы Microchip) не могут изменять напряжение питания в процессе верификации. Это допустимо при макетировании устройств или для радиолюбительских конструкций, но при серийном производстве верификация на предельных режимах питания необходима.

Принципиальная схема программатора El Cheapo показана на рис. 3.50. Он подключается к персональному компьютеру через параллельный порт и управляется с помощью программы, которая работает в операционной системе Microsoft Windows.

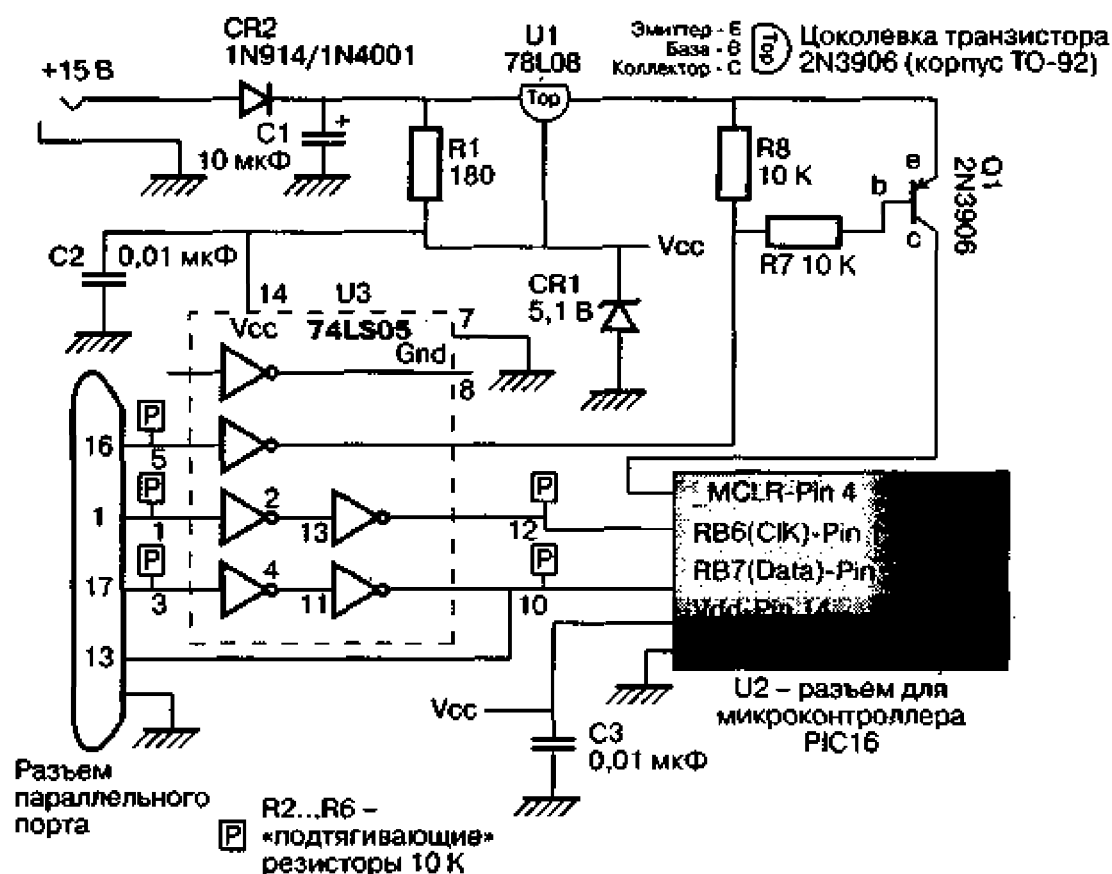


Рис. 3.50. Принципиальная схема программатора El Cheapo

Первая версия этой схемы была опубликована автором четыре года назад; здесь представлена шестая версия. В последней модификации программатор может быть использован с любым микроконтроллером PICmicro, который поддерживает режим программирования ICSP. Большинство модификаций схемы программатора и управляющей его работой программы было направлено как раз на то, чтобы он мог работать без переделок на любых персональных компьютерах и с практически любыми МК PICmicro.

В табл. 3.25 перечислены все элементы, использованные в схеме программатора. Все необходимые элементы нетрудно достать в любом радиомагазине.

Приведенную на рис. 3.50 схему можно условно разделить на четыре блока. Первый - это источник питания. Он может быть выполнен на основе специальной микросхемы или на стабилитроне (см. рис. 3.18, 3.19) и обеспечивает питание

Таблица 3.25. Элементы, использованные в схеме программатора El Cheapo¹

Позиционное обозначение	Элемент	Примечание
U1	78L08 (KP142EH8A или П)*	Микросхема стабилизатора напряжения (9 В)
U2	18-контактный разъем DIP	Разъем для подключения к микроконтроллеру
U3	74LS05 (K155ЛН2)*	Буфер-инвертор, имеющий выход с открытым коллектором
Q1	2N3906 (КТ361П)*	Биполярный p-n-p транзистор. Может быть заменен на BC557 (КТ361Д, КТ3107)*
CR1	5,1 В; 0,5 Вт	Любой стабилитрон на 5,1 В соответствующей мощности (например, КС407П)*
CR2	1N914 или 1N4001	Может быть использован любой плоскостной кремниевый диод на ток свыше 50 мА (например, КД521)*
a	10 мкФ	Оксидный (электролитический) конденсатор на 16 В
C2, C3	0,01 мкФ	Конденсатор любого типа
R1	180 Ом, 1 Вт	Резистор
R2 - R8	10 кОм, 0,25 Вт	Резисторы
Л	Разъем 2,5 мм	Для подачи напряжения питания
a	Разъем DB-25F	Разъем-розетка для параллельного порта
	Источник питания + 14 В, 250 мА	
	Кабель с разъемами DB-25F на обоих концах	Для подключения программатора к персональному компьютеру

¹ В скобках указаны добавленные при переводе отечественные аналоги. - Прим. перев.

Vdd для программатора, а также формирует напряжение программирования Vpp для микроконтроллера. Напряжение программирования должно составлять по крайней мере **13 В**. Чтобы сформировать такое напряжение с помощью 9-вольтовой микросхемы 78L08, используется дополнительный стабилитрон на 5,1 В, с которого одновременно снимается напряжение Vdd, необходимое для работы программатора. Входной диод CR2 используется для защиты от неправильного включения напряжения питания.

На вход источника питания необходимо подавать напряжение **+15 В**. Для этого можно использовать обычный сетевой AC/DC адаптер.

В документации Microchip для различных микроконтроллеров PICmicro указывается значение тока, потребляемого по входу Vdd в режиме программирования, не превышающее 40 мА. Кроме этого, необходимо учесть небольшой ток потребления микросхемы U3. При общем токе 60 мА на резисторе R1 = 180 Ом будет падать напряжение около 10 В. При этом резистор будет рассеивать более половины всей потребляемой мощности. Поэтому необходимо использовать резистор мощностью 1 Вт.

оминал резисторов R2-R6 зависит от типа использованных **инверторов**, для микросхем серии 155 его рекомендуется уменьшить до 5,1 кОм. - Прим. перев.

Если микроконтроллер не подключен, то весь предназначенный для него ток (40 мА) течет через стабилитрон. При этом на нем рассеивается мощность около 0,3 Вт. Поэтому необходимо использовать стабилитрон, рассчитанный на мощность по крайней мере 0,5 Вт.

Источник питания на 5 В рассчитан таким образом, что при коротком замыкании на выходе ток через него не превысит 60 мА. Поэтому во время подключения или отключения микроконтроллера нет необходимости выключать и снова включать источник питания.

Обычно радиолюбители предпочитают самостоятельно разрабатывать источники питания для своих конструкций и не упустят возможности переделать указанную схему. Однако я настоятельно рекомендую оставить ее без изменений - она и без того достаточно проста и эффективна.

Второй блок нашей схемы обеспечивает интерфейс с компьютером. Для связи программатора с персональным компьютером используются инвертирующие буферные элементы (микросхема U3). Я испробовал множество разных вариантов, но указанная схема обладает наилучшими характеристиками при работе с различными компьютерами. Выходы микросхемы 74LS05 имеют открытый коллектор, поэтому необходимо использовать «подтягивающие» резисторы сопротивлением около 10 кОм, включая их между выходом микросхемы и положительной шиной питания. Аналогичные резисторы подключены и к линиям параллельного порта персонального компьютера.

Для подключения к параллельному порту компьютера применяется стандартный кабель с розеткой DB-25F. На рис. 3.50 указаны контакты разъема параллельного порта. Заметим, что программатор использует вывод GND параллельного порта - иначе было бы невозможно подавать все управляющие сигналы. Не рекомендуется брать слишком длинный кабель: его длина не должна превышать 10 футов (около 3 м).

Прежде чем подключать программатор к компьютеру, желательно проверить правильность распайки кабеля. Все жилы одного разъема должны быть подключены к одноименным выводам другого, как показано в табл. 3.26.

Третий блок - это стабилизатор программирующего напряжения V_{pp} . Он выполнен на одном элементе микросхемы U3, выход которого через резистор R7 подключен к р-п-р транзистору Q1. Эта простая схема реализует программное включение и выключение напряжения V_{pp} при токе до 50 мА.

Последний блок - это разъем для подключения к микроконтроллеру. На рис. 3.50 указаны номера используемых выводов МК PIC16F627 (и любых других МК среднего семейства, имеющих корпус с 18 выводами).

При разработке печатной платы не забудьте обеспечить свободный доступ к разъему, предназначенному для установки микроконтроллера. Не следует забывать также про разъем подключения кабеля и источника питания.

Макетирование программатора заняло у меня не более получаса. Однако вы поступите разумно, если будете планировать график работ без излишней спешки. Я заметил, что результат всегда получается более качественным, если отводишь на работу больше времени, чем требуется на самом деле.

Электронное приложение к этой книге содержит инсталлятор программы, предназначенной для работы с описанным программатором. Там вы найдете подробные инструкции по ее установке.

Таблица 3.26. Распайка кабеля для подключения программатора к параллельному порту персонального компьютера

Разъем 1	Разъем 2
1	1
2	2
3	3
...	...
25	25

На рис. 3.51 показан внешний вид окна программы. В правой части окна содержатся необходимые подсказки и пошаговые инструкции по сборке схемы программатора и по ее наладке (**Build/Test Instructions**). По завершении теста необходимо нажать кнопку **Debug End** (Закончить отладку).

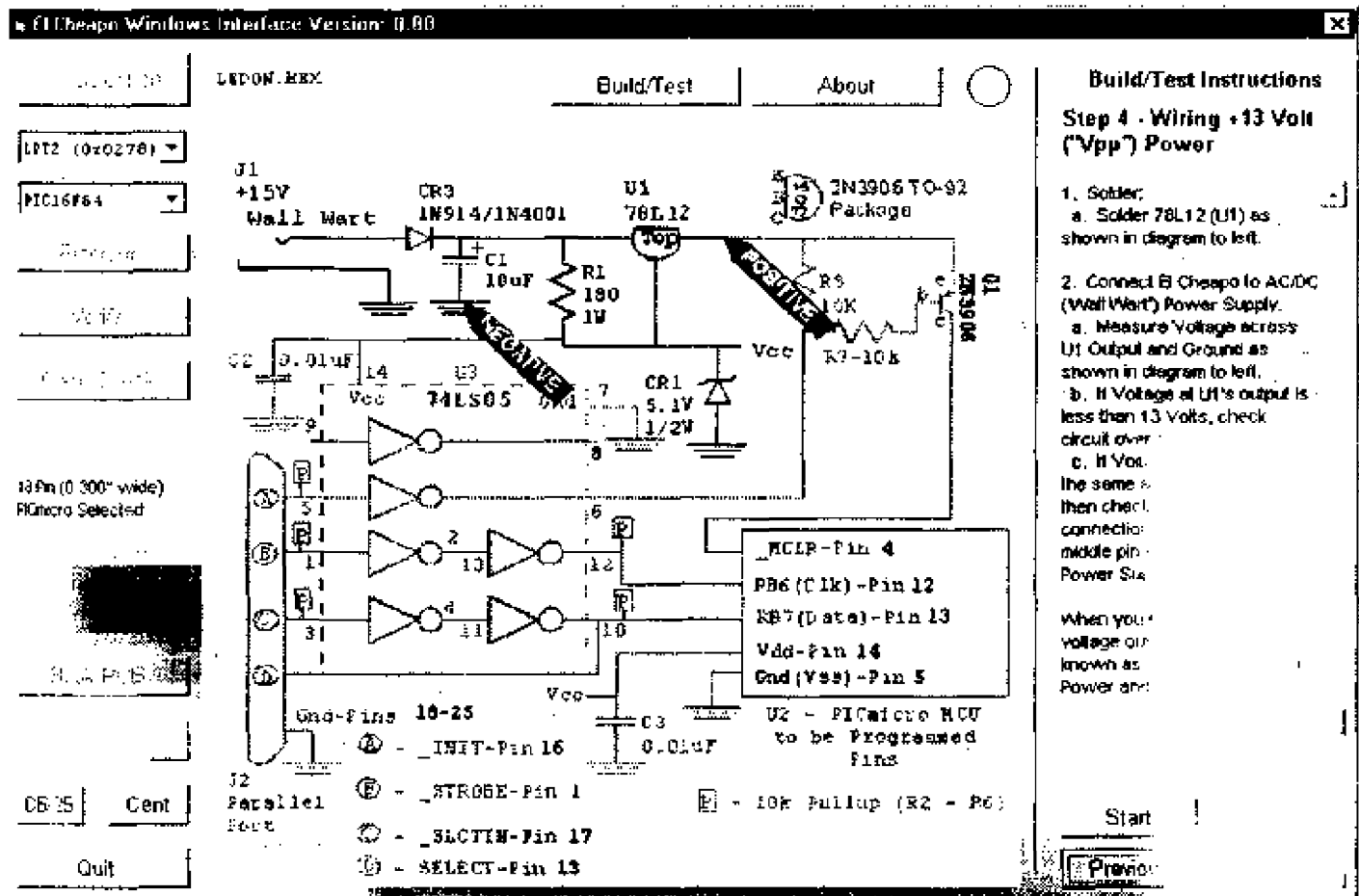


Рис. 3.51. Интерфейс программы El Cheapo

После этого кружок в верхнем правом углу окна окрасится в желтый цвет. Если цвет останется черным, необходимо повторить все этапы настройки, чтобы найти причину неисправности.

Процесс стирания содержимого памяти микроконтроллера перед его программированием занимает не более 15 с. Хотя многие программаторы справляются с этой задачей несколько быстрее, El Cheapo более устойчив в работе.

Первое **очевидным** улучшение конструкции программатора состоит в использовании **ZIF-сокета** (Zero Insertion Force - с «нулевым» усилием соединения). Это предохранит микроконтроллер от повреждений, которые нередко случаются, когда микросхему пытаются вставить в **DIP-разъем** или вытащить из него при помощи отвертки.

Во-вторых, можно заметить, что благодаря простоте схемы программатора его можно встроить в целевую схему. При этом напряжение программирования 13–14В можно взять от источника, уже, возможно, имеющегося в устройстве, или от дополнительной батарейки напряжением 9 В. В последнем случае «минус» батарейки надо подключить к линии Vcc, а «плюс» - к эмиттеру транзистора Q1.

При разработке устройств на основе микроконтроллера желательно не занимать выводы RA5/_MCLR, RA6 и RA7, резервируя их для режима программирования. Ведь еще остается не так уж мало выводов, которые можно использовать. Зато не придется извлекать микроконтроллер из целевой схемы, чтобы подключить его к программатору. В большинстве случаев это решение можно считать оптимальным.

ГЛАВА 4

ПОДКЛЮЧЕНИЕ

К МИКРОКОНТРОЛЛЕРУ

ПЕРИФЕРИЙНЫХ УСТРОЙСТВ

В этой главе мы рассмотрим способы подключения различных периферийных устройств к микроконтроллеру. Здесь будут приведены примеры программ, реализующих функции механического и электронного уровней. Надеюсь, что эти примеры пригодятся вам при разработке собственных устройств. Хотя все описанные конструкции построены на основе микроконтроллера PIC16F627, при использовании других контроллеров PICmicro среднего семейства в большинстве случаев никаких переделок не потребуется. То же касается и других использованных в схемах элементов.

Перед тем как начать читать эту главу, вам необходимо установить на свой компьютер программу MPLAB IDE, компилятор PICC Lite и программный интерфейс программатора El Cheapo. Если вы уже провели тестирование программатора, описанного в конце предыдущей главы, то вам не составит **труда** повторить любое из описанных здесь устройств.

Как уже упоминалось, первоначально приведенные в этой книге примеры (электронные схемы и программное обеспечение) проектировались на основе микроконтроллера PIC16F84. Этот МК не поддерживает некоторых расширенных функций ввода-вывода, которые имеются у МК PIC16F627. Тем не менее адаптация примеров к новому микроконтроллеру не вызвала особых затруднений. Некоторые фрагменты программ стали проще, но основные участки кода совершенно не изменились.

В описанных здесь примерах функции верхнего (биологического) уровня выполняются в основной программе, которая прерывается каждую 1 мс процедурой обработки прерываний от таймера, чтобы выполнить необходимые функции нижних двух уровней. Благодаря использованию основного таймера TMR0, имеющегося во всех МК, основной код программы в минимальной степени зависит от модели микроконтроллера и используемой периферии. Поэтому при модернизации или замене какого-либо внешнего устройства достаточно только переписать ту часть обработчика прерываний, которая работает с этим устройством. Основной код программы при этом не изменится.

Некоторые приведенные в книге конструкции могут вас позабавить, но они так же, как и все остальные, иллюстрируют способы решения проблем, возникающих

у разработчика автоматических устройств при реализации функций механического и электронного уровней. Везде, насколько возможно, я основывал предлагаемые решения на использовании прерываний, предоставляя основному коду программы принимать решения верхнего уровня, не отвлекаясь на работу с периферией.

Я надеюсь, что приведенные принципиальные схемы окажутся полезными при разработке ваших устройств, а примеры программ помогут научиться надлежащим образом проводить разграничение между разными уровнями абстракции при создании управляющего программного обеспечения.

Во всех описанных схемах используются только широко распространенные электронные компоненты. Разумеется, при необходимости вы всегда сможете найти эквивалентную замену любому из них.

4.1. АППАРАТНЫЕ ИНТЕРФЕЙСЫ

Довольно легко разработать схему на основе микроконтроллера, если требуется подключить к нему всего один источник входного сигнала и одно исполнительное устройство. Управляющая программа в этом случае тоже несложна. Трудности начинаются, когда входных и выходных устройств оказывается несколько. Бывает, что программа при добавлении новых периферийных устройств усложняется настолько, что не только отказываются должным образом исполнять свои обязанности новые блоки, но и старые перестают работать.

Причины многих неудач заключаются в неправильном планировании. Уже на начальных стадиях проекта необходимо тщательно распределить выполняемые функции и предусмотреть возможность их как можно менее трудоемкой модернизации. Разумеется, очень трудно спланировать все заранее, тем более, в настоящее время темпы развития промышленных технологий настолько стремительны, что многие интерфейсы и компоненты становятся доступны уже после начала работ над проектом. Но вам проще будет внести надлежащие изменения, если вы заранее подготовитесь к этому.

Первая вещь, которая поможет вашим разработкам, - это осознание того, что реализация функций управления электромеханическими устройствами (механический уровень) должна быть отделена от реализации функций управления другими электронными устройствами и интерфейсами (электронный уровень). В свою очередь, оба эти слоя должны разрабатываться отдельно от основной управляющей программы (биологический уровень). Только при таком подходе можно добиться относительной независимости между программным и аппаратным обеспечением.

Для достижения еще большей гибкости при настройке программного обеспечения, рассчитанного на работу с различным оборудованием, желательно создать для управляющей программы надлежащую операционную среду. В идеале это должна быть настоящая операционная система реального времени (Real Time Operating System - RTOS). К сожалению, при использовании большинства имеющихся микроконтроллеров (в том числе и PIC16F627) функционирование такой

операционной системы в полной мере невозможно. Но в любом случае следует стараться разделять свой проект на отдельные задачи, каждая из которых затем может быть модернизирована без особого ущерба для других.

При использовании микроконтроллеров, таких как PIC16F627, важно при проектировании определить последовательность операций, с помощью которых будут реализованы через встроенную в микроконтроллер систему прерываний необходимые функции механического и электронного уровней. Необходимо назначить для каждой такой функции свою процедуру обработки прерывания или обособленный фрагмент общего обработчика, чтобы разные интерфейсы как можно меньше взаимодействовали друг с другом. Основывая свой проект на микроконтроллере PIC16F627, разработчик должен хорошо ориентироваться в том, какие источники запросов на прерывания и в каких случаях можно использовать.

Чтобы обеспечить максимальную совместимость с другими микроконтроллерами PICmicro я обычно реализую управление последовательностью выполняемых операций с помощью таймера TMR0. В большинстве случаев период срабатывания таймера рекомендуется выбирать равным 1 мс (точнее, 1024 мкс). Такой выбор обеспечивает достаточно быструю реакцию управляющей программы на внешние воздействия и в то же время предоставляет коду верхнего уровня необходимое время для принятия решений. Разумеется, в отдельных случаях промежутки между прерываниями приходится уменьшать.

Так как длительность командного цикла в микроконтроллерах равна четырем периодам работы системного тактового генератора, то при частоте тактирования 4 МГц задержка длительностью 1024 мкс эквивалентна 1024 командным циклам, поэтому приходится использовать **предделитель (прескалер)**. В инициализирующей части программы разработчик должен разрешить работу таймера TMR0 и предделителя, а также обработку прерываний. После этого запрос на прерывание будет вырабатываться каждый раз, когда счетчик таймера при пересечении границы между 0x0FF и 0x100 будет сбрасываться в нулевое состояние.

Хотя на фоне быстродействия современных компьютеров робот, имеющий миллисекундную задержку реакции на входные воздействия, может показаться слишком медлительным, по сравнению с человеческой реакцией этот показатель все равно впечатляет.

Текст обработчика прерываний в наших программах будет выглядеть примерно так:

```
void interrupt timer_int(void){ // Обработчик прерываний от таймера.
    if (TimerOverflow) {      // Если таймер сработал,
        TimerInterrupt = Reset; // то сбросить флаг прерывания
        RTC++;                // и выполнить инкремент счетчика
                               // реального времени.
        // Здесь можно разместить дополнительный код
        // для обработки прерываний от таймера.
    } // Конец условного оператора.
    // Здесь надо разместить код для обработки других прерываний.
} // Конец обработчика прерываний.
```

Приведенный образец оформления процедуры обработки прерываний практически не зависит от того, какой именно микроконтроллер вы используете.

В главной программе с помощью значения счетчика реального времени RTC, устанавливаемого с помощью обработчика прерываний от таймера, можно реализовать выполнение различных операций, требующих точно выдержанных задержек во времени. Например, если некоторая операция должна длиться 5 мс, то соответствующий фрагмент главной программы выглядит следующим образом:

```
Final = RTC + 6;           // Запомнить время окончания операции.
while(Final != RTC);       // Ждать не менее 5 мс.
```

Здесь к текущему значению счетчика прибавляется число, на 1 большее величины требуемой задержки в миллисекундах. В результате обработчик прерываний от таймера выполнится по крайней мере шесть раз. Это гарантирует, что длительность задержки не окажется меньше 5 мс. В худшем случае она может составить почти 6 мс. Таким образом, реальная длительность задержки лежит в диапазоне 5-6 мс и зависит от того, в какой момент времени выполнен оператор присваивания¹.

Вставляя необходимый код обработки там, где указано в комментариях, вы можете реализовать выполнение достаточно сложных операций механического и электронного уровней. Если вся операция не может быть выполнена внутри обработчика сразу (например, из-за того, что приходится выдерживать заданные временные промежутки), всегда можно сделать так, что при каждом вызове обработчика будут выполняться отдельные части общей задачи. Чуть позже я продемонстрирую несколько примеров. Во-первых, вы узнаете, как можно с помощью обработчика прерываний, выполняемого каждую 1 мс, реализовать широтно-импульсную модуляцию с шагом 30 мс сигнала частотой 32 Гц. Во-вторых, будет показано, как с помощью отдельной процедуры обработки прерываний, использующей текущее значение счетчика реального времени, реализуется программа для дистанционного управления роботом. Несмотря на то что обе процедуры работают с одними и теми же аппаратными ресурсами, между собой они напрямую не взаимодействуют.

При проектировании периферийных интерфейсов я рекомендую использовать модель *конечного автомата*. Тогда управление последовательностью отдельных операций, каждая из которых выполняется при очередном вызове обработчика прерываний и реализует часть общей задачи, удобно осуществлять с помощью таблицы переходов или оператора выбора switch. При очередном вызове обработчика по значению соответствующей переменной определяется текущее состояние автомата, и по этому состоянию осуществляется переход на тот фрагмент кода, который реализует необходимые действия.

В текст описанного выше шаблона процедуры обработки прерываний можно вставить дополнительные фрагменты кода для реализации необходимых реакций на другие прерывания, формируемые, например, при изменении состояния входного

¹ От предыдущего запроса на прерывание до следующего проходит 1 мс, между шестью вызовами обработчика пять миллисекундных промежутков, к этому еще надо прибавить длительность промежутка времени, который прошел между выполнением оператора присваивания и первым запросом на прерывание от таймера. Ясно, что этот промежуток длился не более 1 мс. - Прим. перев.

порта; в результате прихода сигнала запроса на линию RBO/INT; при записи в EEPROM память данных и т.п.

Так как система прерываний у большинства микроконтроллеров PICmicro является одновекторной, то при формировании запроса от любого источника управление передается на одну и ту же процедуру обработки. Поэтому каждый раз приходится определять, что послужило причиной возникновения запроса, чтобы **уяснить**, какие действия необходимо выполнять. В нашем примере это делается помощью условного оператора, проверяющего состояние флага прерывания от таймера. Аналогично определяются остальные возможные источники запроса. Таким образом, добавление обработчика прерываний от какого-либо определенного устройства никак не влияет на обработку прерываний от других устройств.

Чуть позже мы увидим, что обработка прерываний от блока инфракрасного приемника сигналов дистанционного **управления** нисколько не мешает обработке прерываний от таймера, а наоборот, использует результат его работы для вычисления длительности задержек между событиями.

При проектировании программного обеспечения следует помнить, что все операции по обработке прерываний должны выполняться как можно быстрее. Ведь во время обработки одного запроса все другие прерывания запрещены, и остальным периферийным устройствам приходится ждать своей очереди. Если вы будете слишком долго работать с одним устройством, то рискуете так и не заметить запросы, пришедшие от других. Кроме того, не следует забывать, что для выполнения основной программы, реализующей функции биологического уровня, также необходимо время.

4.2. ШАБЛОН ПРОГРАММЫ НА ЯЗЫКЕ C

Прежде чем начать детальное обсуждение примеров, рассмотрим шаблон приложения, на основе которого мы будем строить большинство наших программ. Этот шаблон разработан специально для того, чтобы облегчить возможность безболезненного добавления новых функций механического и электронного уровней. Приведенный ниже код можно найти в электронном **приложении**¹ к этой книге в папке Code\Template:

```
«include <pic.h>
//
// Шаблон программы управления роботом для компилятора PICC Lite.
//
// Таймер установлен на срабатывание каждые 1024 мкс.
//
// 28.03.2002 - Адаптировано для микроконтроллеров PIC16F627/PIC16F84.
// 23.01.2002 - Разработано Майклом Предко.
//
```

¹ Электронное приложение к этой книге можно скачать с сайта издательства www.dmk.ru. Тексты программ в электронном приложении приведены в том виде, какой они имели в английском оригинале. - *Прим. перев.*

```

// Примечания:
// Микроконтроллер PIC16F84/PIC16F627 работает на частоте 4 МГц.
// Микроконтроллер PIC16F627 использует внутренний тактовый генератор.
// Используется внешний сигнал сброса _MCLR.

//
// Слово конфигурации:
#if defined (_16F84)
    #warning PIC16F84 selected // Выбран микроконтроллер PIC16F84.

    __CONFIG(0x03FF1);    // Конфигурация PIC16F84:
                          // тактовый генератор XT,
                          // таймер PWRT включен,
                          // сторожевой таймер выключен,
                          // защита кода отключена.

#elif defined(_16F627)
    #warning PIC16F627 with internal oscillator selected
                          // Выбран микроконтроллер PIC16F627.

    _CONFIG(0x03F70);    // Конфигурация PIC16F627:
                          // внутренний тактовый генератор,
                          // RA6/RA7 используются для ввода-вывода,
                          // используется внешний сигнал сброса,
                          // таймер PWRT включен,
                          // сторожевой таймер выключен,
                          // защита кода отключена,
                          // детектор падения напряжения питания включен.

#else
    #error Unsupported PICmicro MCU selected
                          // Ошибка: неправильно выбран тип микроконтроллера.
#endif

// Глобальные переменные:
volatile unsigned int RTC = 0;    // Счетчик реального времени.

// Обработчик прерываний от таймера:
void interrupt tmr0_int(void)
{
    if (TOIF) {    // Если запрос на прерывание поступил от таймера TMRO,
        TOIF = 0;    // то сбросить флаг прерывания
        RTC++;    // и выполнить инкремент счетчика времени.

        // Здесь можно описать дополнительные действия
        // для обработки прерываний от таймера.

    }    // Конец условного оператора.

    // Здесь можно разместить код для обработки прерываний
    // от других источников.

}    // Конец обработчика прерываний.

// Главная программа:
void main(void)
{
    TMRO = 0;    // Начальное значение счетчика
                // реального времени.

```



```

OPTION = 0x0D1;          // Использовать предделитель совместно
                          // с таймером TMR0, коэффициент деления 4.

TOIE = 1;                // Разрешить прерывания от таймера.
GIE = 1;                 // Разрешить обработку прерываний.

// Здесь надо выполнить инициализацию периферийных устройств.

while (1 == 1) {         // Бесконечный цикл.

    // Здесь надо разместить код для реализации
    // функций биологического уровня.

}                          // Конец оператора while.

)                          // Конец главной программы.

```

Поясним назначение каждой строки, чтобы вы до конца разобрались в том, как строится приложение при использовании компилятора PICC Lite.

В первой строке к программе подключается заголовочный файл `pic.h`, в котором определена информация, специфичная для микроконтроллеров PICmicro. В частности, в нем содержатся прототипы библиотечных функций, названия регистров и имена их отдельных разрядов.

Как уже упоминалось, компилятор PICC Lite может выполнять трансляцию программ только для микроконтроллеров PIC16C84, PIC16F84, PIC16F84A и PIC16F627. Следует помнить, что компиляция программ для других микроконтроллеров PICmicro может завершиться неудачно без каких бы то ни было предупреждающих сообщений.

Затем в нашем шаблоне содержится информация об используемых аппаратных средствах. Эта часть не является обязательной, но я рекомендую всегда пояснять в начале программы, какие особенности микроконтроллеров и периферийных устройств используются в приложении.

Далее указана директива условной компиляции `#if defined(...)`. Она выполняется во время компиляции программы, а не во время ее выполнения, в отличие от обычного условного оператора `if`. Здесь мы используем различные значения слова конфигурации в зависимости от **того**, для какого микроконтроллера компилируется программа.

Кроме этого, используется еще одна директива - `ttwarning`: с ее помощью выводится предупреждающее сообщение о типе микроконтроллера, для которого в данный момент производится компиляция. В интегрированной среде MPLAB разработчик должен указать микроконтроллер того же самого типа. Об этом не следует забывать, особенно если новый проект создается на основе старого, который был рассчитан на работу с другим микроконтроллером.

Если указанный тип не совпадает ни с одним из двух описанных вариантов (если используется МК PIC16C84), то выдается сообщение об ошибке - для этого предназначена директива `#error`.

Для отсчета времени в программе служит беззнаковая 16-разрядная переменная RTC (real time counter - счетчик реального времени). Ее значение увеличивается на 1 каждый раз, когда очередной раз выполняется код обработчика прерываний от

таймера, а происходит это каждые 1024 мкс. Эта переменная используется для реализации задержек заданной длительности.

Переменная RTC является *глобальной*. Она объявлена в программе вне какой бы то ни было функции и поэтому может быть использована для чтения и/или записи внутри любой из них, в том числе и в процедуре обработки прерываний. Модификатор `volatile` указывает, что компилятор не будет оптимизировать доступ к этой переменной, пытаясь подставлять во все выражения ее предыдущее значение, ведь оно может в любой момент (незаметно для главной программы) измениться в результате выполнения обработчика прерываний.

Переменная RTC описана как беззнаковая (`unsigned`), поэтому ее значение может быть в диапазоне от 0 до 65535 (0xFFFF). Напомним, что возможные значения обычных переменных типа `int` лежат в диапазоне от -32768 до +32767.

Код процедуры обработки прерываний располагается отдельно от остальной программы, в микроконтроллерах PICmicro он начинается с адреса 0x0004. Главная программа и вызываемые функции размещаются в конце памяти программ.

Функция, описанная как обработчик прерываний (с модификатором `interrupt`), будет вызываться каждый раз, когда произойдет какое-нибудь прерывание. Поэтому в первую очередь требуется определить источник запроса. Для этого мы проверяем, установлен ли **флаг** T0IF прерывания от таймера. Если он установлен, значит произошло переполнение таймера и надо выполнить инкремент счетчика RTC. В противном случае прерывание было запрошено другим устройством, поэтому никаких действий со счетчиком реального времени выполнять не следует.

Если в нашем приложении требуется обрабатывать запросы и других источников, то соответствующий код надо поместить после условного оператора `if`¹. В начале каждого фрагмента обработчика, относящегося к своему источнику прерываний, следует использовать условный оператор, проверяющий состояние соответствующего флага. При этом не стоит забывать о необходимости сбрасывать этот флаг до того, как завершится обработка прерывания.

Что касается шаблона основной программы, то он предельно прост. Здесь инициализируется значение коэффициента деления прескалера и разрешается его работа совместно с таймером TMR0. Устанавливается начальное значение счетчика таймера (что гарантирует выполнение 1024 командных циклов, прежде чем первый раз будет сгенерировано прерывание от таймера). Если этого не сделать, то начальное значение счетчика окажется неопределенным, а в результате первый запрос может быть сформирован еще до того, как будет **проинициализирован** контроллер прерываний.

Затем следует выполнить инициализацию периферийных устройств. Соответствующий код зависит от того, какие устройства используются.

¹ В этом случае функцию `tmr0_int` логичнее назвать как-нибудь по-другому. - Прим. *перев.*

В конце главной программы указан оператор цикла `while` с условием продолжения, которое всегда выполняется (`1 == 1`). Это приводит к бесконечному циклу, который будет закончен только при выключении микроконтроллера или при подаче сигнала сброса. Внутри этого бесконечного цикла выполняется код, реализующий функции биологического уровня.

Компилятор PICC Lite всегда размещает функцию `main` в конце памяти программ. Поэтому машинные команды, соответствующие последнему оператору главной программы, оказываются записанными в последних ячейках памяти. После выполнения этих команд программный счетчик PC снова обнуляется, и программа продолжает выполняться с самого начала. Поэтому даже без явного использования оператора цикла можно добиться заикливания главной программы.

Как мы увидим в дальнейшем, реализация приложений на основе представленного шаблона не должна вызывать каких-либо затруднений. Вы будете удивлены тем, с какой легкостью удастся добавлять к проектам новые функции.

4.3. МАКЕТИРОВАНИЕ УСТРОЙСТВ НА ОСНОВЕ МИКРОКОНТРОЛЛЕРОВ PICMICRO

Работая над книгой, я ни на секунду не пожалел о том, что выбрал для своих проектов микроконтроллер PIC16F627. Вы увидите, что с этим МК очень легко работать, а современные инструментальные средства MPLAB IDE и PICC Lite превращают эту работу в истинное удовольствие, позволяя реализовывать достаточно сложные устройства, нисколько не уступающие по своим возможностям коммерческому программному обеспечению.

Но сначала перечислим необходимый набор инструментов, который пригодится для реализации любой из описанных в этой книге конструкций. Вам потребуется рабочий стол для размещения персонального компьютера (включая монитор, клавиатуру и мышь); программатор для загрузки программ в микроконтроллер; место для макетирования устройств (защищенное от воздействия статического электричества); цифровой измерительный прибор (мультиметр). По возможности следует располагать компьютер и программатор вблизи от того места, где вы будете выполнять макетирование своих устройств.

Итак, чтобы повторить конструкции, приведенные в этой главе, вам потребуются:

- персональный компьютер;
- программатор для микроконтроллера PIC16F627;
- цифровой мультиметр;
- источник питания +5 В;
- макетная плата;
- набор монтажных проводов;

- кусачки;
- приспособление для зачистки проводов;
- остроносые плоскогубцы;
- небольшая отвертка с плоским наконечником;
- два (или больше) микроконтроллера PIC16F627.

Персональный компьютер должен иметь:

- процессор не ниже Pentium II, 200МГц¹;
- 64 Мб оперативной памяти;
- видеоадаптер SuperVGA, поддерживающий разрешение экрана не ниже 1024x768, объем видеопамати по крайней мере 2 Мб;
- мышь;
- параллельный порт для подключения программатора;
- последовательный порт (используется в некоторых конструкциях).

На вашем жестком диске должны быть установлены:

- операционная система Microsoft Windows9x/NT/2000/XP;
- Web-браузер (MS Internet Explorer или Netscape Navigator);
- Adobe Acrobat Reader (для чтения документации в формате PDF);
- связь с Internet;
- интегрированная среда разработки программ MPLAB;
- компилятор PICC Lite.

При выборе программатора (если вы почему-либо не желаете использовать тот, что описан в предыдущей главе) имейте в виду, что большинство программаторов, рассчитанных на работу с микроконтроллерами PIC16C84/PIC16F84/PIC16F84A должны без всяких модификаций работать и с PIC16F627.

Если желаете поэкспериментировать с другими микроконтроллерами PICmicro, рекомендую приобрести программатор PICSTART Plus фирмы Microchip. Он постоянно модифицируется, чтобы поддерживать все новые и новые типы микроконтроллеров PICmicro.

На рис. 4.1 показан внешний вид простой платы для макетирования электронных схем. Отверстия печатной платы должны быть рассчитаны на установку корпусов микросхем DIP (dual inline package - корпус с двухрядным расположением выводов).

Если вы имеете некоторый опыт работы с высокочастотными схемами, то знаете, что разъемы универсальных макетных плат имеют значительную паразитную емкость и не рассчитаны на работу с высокочастотными сигналами (больше 10 МГц). К счастью, все описанные здесь устройства работают на более низких частотах.

¹ Здесь автор явно завышает требования к аппаратному обеспечению. Все описанные в книге программы работают даже на Pentium-133 с памятью 16 Мб. - *Прим. перев.*

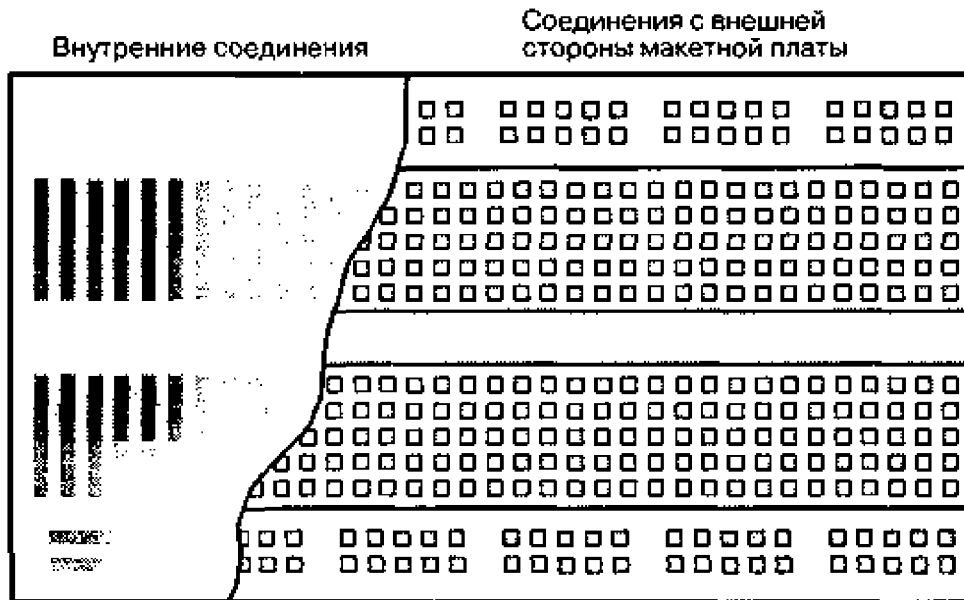


Рис. 4.1. Плата для макетирования

Однако я не советую использовать макетные платы в окончательных конструкциях. Они недолго выдержат вибрацию двигателей, используемых для перемещения робота; к тому же на них собирается очень много пыли, которая может нарушить работу устройства.

Отвертка с плоским наконечником может использоваться для извлечения микросхем из специально предназначенных для них панелек. Но еще лучше применять разъем ZIF (с «нулевым» усилием), который может оказаться весьма дорогим, но позволяет производить демонтаж микросхем без особых хлопот и, что самое главное, без риска повредить их выводы.

Отвертка также может пригодиться для подачи сигнала сброса на вход `_MCLR` микроконтроллера. С ее помощью можно быстро закоротить этот вывод на массу (вывод `Vss`), не применяя для этого никаких дополнительных проводов. Но, чтобы воспользоваться этим способом, вы должны предварительно соединить вывод `_MCLR` с положительной шиной питания (вывод `Vdd`) через резистор сопротивлением около 10 кОм (если, как обычно, соединить их непосредственно друг с другом, то при соединении вывода `_MCLR` с массой произойдет короткое замыкание).

Иногда потребуется по крайней мере два одинаковых микроконтроллера PIC-16F627, чтобы иметь возможность оценить влияние некоторых изменений в управляющей программе на поведение робота. Разумеется, для этого хватило бы одного МК, но в этом случае вы не смогли бы наблюдать особенности их работы одновременно. Кроме того, следует ожидать, что рано или поздно ваши эксперименты закончатся повреждением одного из микроконтроллеров - тогда наличие запасной микросхемы позволит не делать перерыва для похода в магазин.

Все описанные в книге устройства могут быть проверены с помощью цифрового мультиметра. Разумеется, вы можете также использовать логический пробник, осциллограф, логический анализатор или даже внутрисхемный эмулятор. Это поможет вам лучше понять работу приведенных здесь электронных схем.

Для большинства устройств требуется один источник напряжения питания +5 В. Потребляемый ток не превышает 100 мА. Если бы не двигатели или ультразвуковой дальномер, то во всех конструкциях можно было бы использовать 9-вольтовый элемент питания с интегральным стабилизатором 78L05 или четыре никель-кадмиевых батарейки, включенных последовательно.

Не забывайте выключать источник питания во время подключения или отключения микросхем (в том числе и самого микроконтроллера). Разумеется, для этого достаточно вытащить одну из батареек, но лучше использовать специальный выключатель, установленный в разрыве положительной шины питания.

Для выполнения соединений между элементами схемы следует использовать провод минимальной длины. Это облегчит поиск ошибок в схеме, а также уменьшит влияние электромагнитных наводок.

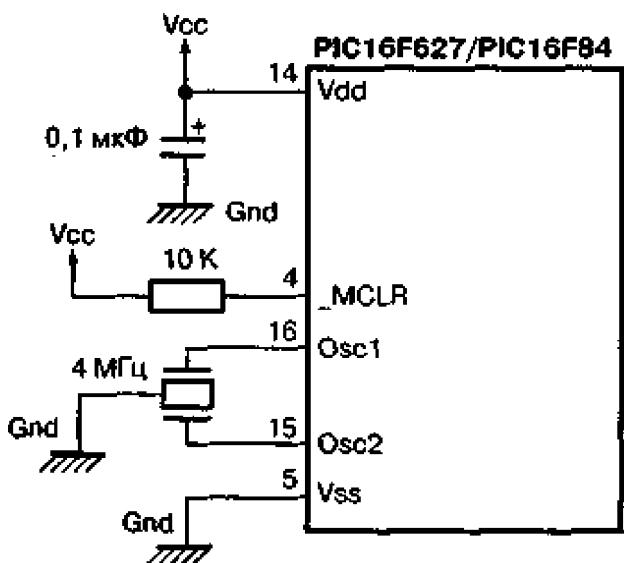


Рис. 4.2. Базовая схема подключения микроконтроллера **PIC 16F84/PIC16F627**

На рис. 4.2 показана базовая схема, на основе которой будут выполнены все описанные в книге конструкции. В большинстве случаев неважно, какой микроконтроллер используется: PIC16F84 или PIC16F627. Мы будем особо отмечать случаи, когда нам придется использовать функции, имеющиеся только у PIC16F627. Печатная плата и размещение компонентов показаны¹ на рис. 4.3.

В схеме могут быть использованы любые конденсаторы емкостью 0,1 мкФ. Рекомендую вам по возможности брать танталовые конденсаторы, рассчитанные на напряжение 16 В или более².

Я предпочитаю располагать обе шины питания (Vcc и Gnd) у каждого края макетной платы.

Обратите внимание, что одноименные шины на разных концах платы соединены друг с другом. При подключении элементов следует стараться, чтобы провода питания были минимально возможной длины и чтобы разноименные провода ни в коем случае не пересекались.

На рис. 4.3 также показано место размещения микроконтроллера, обозначенное особым образом (при помощи отрезка монтажного провода). Этот способ стоит применять во всех случаях, когда не используются специальные панельки для микросхем.

¹ Здесь и далее все рисунки с изображениями макетных плат, как и в оригинале, на 10-15% меньше натуральной величины. Читатель без труда может вычислить реальные размеры, используя тот факт, что расстояние между соседними выводами микросхем должно составлять 2,5 мм. - *Прим. перев.*

² Здесь и далее, если кварцевый резонатор не имеет встроенных конденсаторов, то следует использовать два внешних конденсатора емкостью по 15 пФ каждый, как показано на рис. 3.23. - *Прим.*

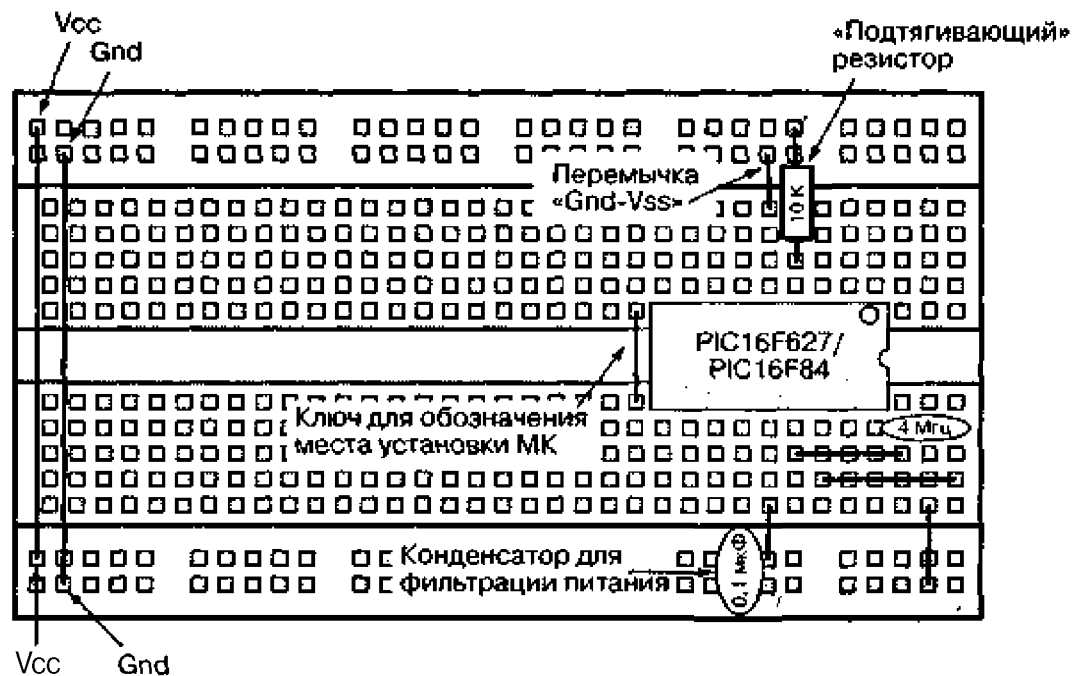


Рис. 4.3. Рисунок печатной платы и размещение компонентов базовой схемы

4.4. МЕЖПРОЦЕССОРНЫЕ КОММУНИКАЦИИ

Можно перечислить несколько причин, по которым возникает необходимость в обмене информацией между процессорами (одним из которых или сразу обоими может оказаться микроконтроллер). Самые распространенные из них:

- отладка программ для микроконтроллера;
- мониторинг операций, выполняемых роботом;
- реализация интерфейсных функций с помощью отдельного микроконтроллера.

Существует несколько способов загрузки и размещения программ для управления работой автоматических устройств. Самый очевидный - это программирование самого микроконтроллера, но может также использоваться метод, типичный для персональных компьютеров, когда в памяти микроконтроллера располагается только специальная программа-загрузчик, а основное программное обеспечение размещается во внешней оперативной памяти. В последнем случае программа обычно загружается в память с использованием какого-нибудь последовательного интерфейса, например RS-232, работой которого управляет специальная программа-монитор. Заметим, что способ загрузки и хранения программы не влияет на методы ее проектирования, поэтому описанные в этой книге приемы разработки программного обеспечения будут действенны в любом случае.

Обычно при использовании микроконтроллеров PIC16F84 или PIC16F627 управляющее устройство размещается внутри робота, а не снаружи. Для внутрисхемного программирования микроконтроллера необходимо, чтобы выводы `_MCLR`, `RB6` и `RB7` во время загрузки программы в память микроконтроллера были отключены от основной схемы и подключены к программатору. Для этого разумно использовать переключатель, установленный на плате. Кроме того, желательно, чтобы программирующее напряжение подавалось от какого-либо внешнего источника.

Связь микроконтроллера с компьютером необходима для загрузки программ, а также для работы внутрисхемного эмулятора (In-Circuit Emulator - ICE). Обычно в этом случае используется стандартный последовательный протокол RS-232. Кроме широкой распространенности, еще одно преимущество его применения при отладке автоматических устройств заключается в том, что он использует достаточно высоковольтные цифровые уровни, устойчивые к помехам, возникающим при включении-отключении двигателей. Если в некоторых случаях затруднительно обеспечить проводное подключение мобильного робота к компьютеру, то последовательный протокол обмена информацией можно реализовать с использованием радиосвязи.

Что касается мониторинга состояния робота, то я бы рекомендовал воздержаться от применения компьютеров в этом случае. Хорошо продуманное размещение нескольких светодиодных индикаторов на самом роботе может оказаться полезнее, чем компьютерный дисплей. Кроме того, подобное решение проблемы исключает необходимость обеспечения связи робота с компьютером. Жидкокристаллический дисплей для этих целей менее пригоден, так как при настройке подвижного механизма большую часть времени он будет находиться под углом к вам, и вы не сможете прочесть то, что на нем отображается.

Заметим, что индикация текущего состояния программы — не то же самое, что и ее отладка. Ведь в первом случае вы не можете на время приостановить выполнение программы и тем более взять на себя функции по управлению роботом, как в процессе пошаговой отладки.

Наконец, связь между контроллерами может потребоваться для управления различными периферийными устройствами. Часто при этом используется интерфейс PC (синхронный последовательный протокол) или RS-485 (обладающий отличной устойчивостью к помехам).

Следует воздержаться от искушения заставить микроконтроллер делать сразу две вещи - выполнять какую-либо операцию с периферийным устройством и обмениваться информацией с компьютером или другим микроконтроллером. В противном случае либо передаваемая информация окажется некорректной (в результате выполнения операции с периферийным устройством изменится текущее состояние), либо при передаче произойдет потеря или искажение данных из-за того, что микроконтроллеру придется отвлекаться и окажутся нарушенными длительности промежутков времени между моментами изменения сигналов, определяемые протоколом обмена.

В результате многочасовых экспериментов над своими устройствами я выработал следующие общие рекомендации по методам реализации межпроцессорного обмена информацией:

- скорость обмена не критична для большинства приложений, поэтому обычно вполне достаточно 115,2 Кбит/с в случае использования RS-232 или несколько больше для Ethernet и других сетевых протоколов. Но даже при таких скоростях не следует стремиться к максимальному использованию пропускной способности канала связи. На более низких скоростях передачи улучшается помехоустойчивость, снижается потребляемая мощность и уменьшаются электромагнитные наводки;

- связь с локальными периферийными устройствами должна иметь более высокий приоритет, чем с удаленными;
- при реализации межпроцессорной связи следует стремиться к максимальной надежности и помехоустойчивости. Все соединения должны быть надежно пропаяны или монтаж накруткой должен быть высокого качества, чтобы снизить сопротивление соединительных проводов. Ненадежная связь может стать причиной многочасовых безуспешных попыток найти неисправность в программе.

4.5. РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА RS-232

Интерфейс RS-232 получил широкое распространение; он достаточно надежен и прост в реализации. Поэтому в большинстве микроконтроллеров есть встроенные средства, обеспечивающие связь с другими устройствами на основе данного интерфейса. При этом используется кодирование сигналов по методу NRZ, который уже упоминался в главе 3. Программное обеспечение для реализации последовательного обмена информацией с персональными компьютерами широко доступно, соответствующие функции поддерживаются интерфейсами прикладных программ (API) всех известных операционных систем.

Обычно применяется протокол 8-N-1, что означает 8-разрядные данные, без бита четности (No parity), один стоповый бит. Проверка на четность не проводится, поскольку современные компьютеры в большинстве случаев позволяют осуществить достаточно надежную передачу данных на разумных скоростях.

На рис. 4.4 показано, как обычно реализуется последовательная связь с использованием интерфейса RS-232.

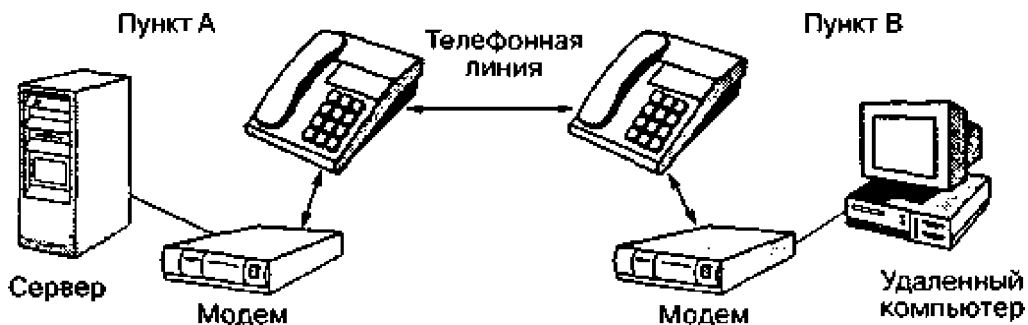


Рис. 4.4. Передача данных между двумя компьютерами с использованием модемов

Согласно стандарту на интерфейс RS-232, по исполняемым функциям различают два типа оборудования. *Оконечное оборудование обработки данных* (Data Terminal Equipment, DTE) — это компьютеры, сканеры, принтеры, плоттеры и другие устройства, передающие и/или принимающие данные. *Оборудование передачи данных* (Data Communication Equipment, DCE) - это модем и другие устройства, обеспечивающие прием и/или передачу данных.

К сожалению, стандарт RS-232 разрабатывался достаточно давно, когда аппаратные средства были довольно примитивны и не слишком надежны, поэтому некоторые недостатки этого интерфейса достались нам в наследство с тех старых времен.

Прежде всего, это цифровые уровни, кодирующие двоичные состояния. Они отличаются от стандартных уровней ТТЛ или КМОП микросхем. Логическая единица передается напряжением в диапазоне от -5 до -12 В, а логический ноль - от +5 до +12 В (рис. 4.5). Так как при передаче наблюдается затухание сигнала, то на приемной стороне порог переключения снижен до -3 и +3 В соответственно.

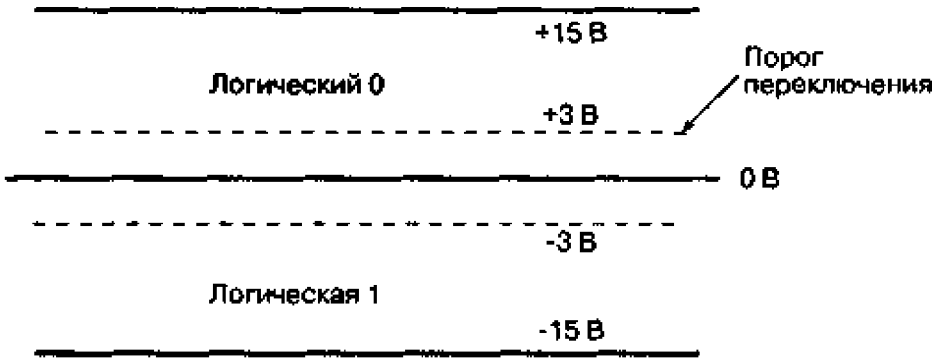


Рис. 4.5. Стандартные логические уровни сигналов интерфейса RS-232

В результате возникает необходимость использования дополнительных аппаратных средств, с помощью которых на передающей стороне уровни сигналов, вырабатываемые в микроконтроллере, переводятся в уровни интерфейса RS-232, а на приемной стороне осуществляют обратное преобразование. Чуть ниже мы опишем несколько способов решения этой проблемы. Наша задача облегчается тем, что многие реализации интерфейса RS-232 в современных компьютерах могут работать со стандартными уровнями пятивольтовой ТТЛ/КМОП логики.

На задней панели вашего персонального компьютера вы можете увидеть стандартный 25- или 9-контактный D-образный разъем-вилку последовательного порта для подключения внешних устройств. Назначение контактов разъемов обоих типов показано на рис. 4.6.

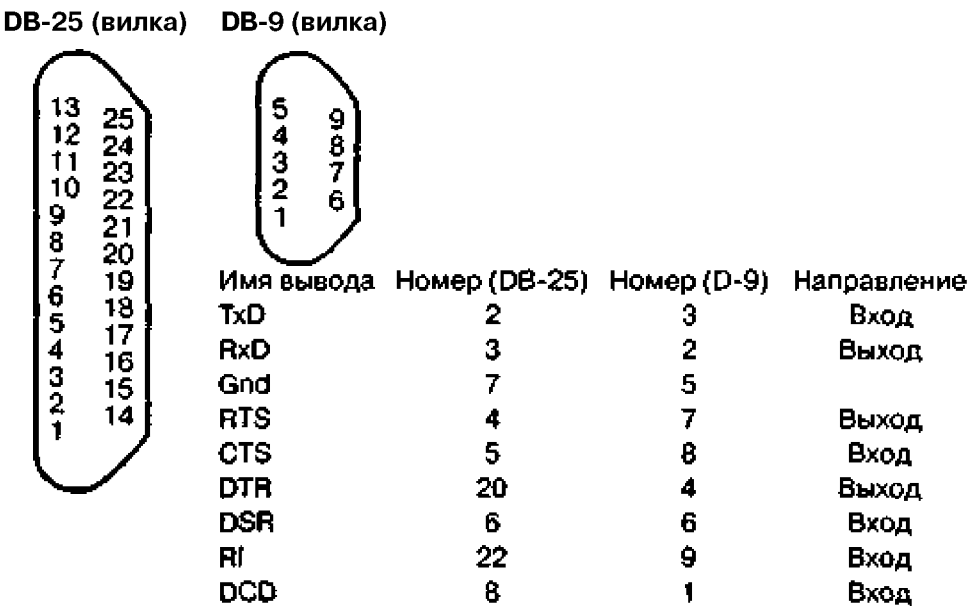


Рис. 4.6. Контакты разъемов последовательного порта DB-25 и D-9

В большинстве современных устройств (и не только радиолюбительских) линии подтверждения не используются. Они были добавлены в стандарт, когда средства межкомпьютерных коммуникаций еще не отличались высокой надежностью. Сейчас в большинстве случаев достаточно трех линий, показанных на рис. 4.7.

Проверка того, что коммуникационный кабель вставлен в разъем, и имитация сигналов подтверждения обеспечиваются простым замыканием двух пар линий: **DTR** (Data Terminal Ready - готовность DTE к работе) и **DSR** (Data Set Ready - готовность DCE к работе), а также **RTS** (Request to Send - запрос на передачу) и **CTS** (Clear to Send - готовность DCE к приему данных). Линии **DCD** (Data Carrier Detected - несущая обнаружена) и **RI** (Ring Indicator - индикатор вызова) при этом не используются.

Так как линии запроса/подтверждения соединены, то сигналы подтверждения готовности к приему данных вырабатываются автоматически, простым дублированием сигналов запроса, поэтому программное обеспечение может теперь не заботиться о реализации протокола *квитирования* (handshaking).

Обращаем ваше внимание на то, что DTE и DCE должны иметь общую «землю» (Ground - Gnd), иначе приемник не сможет распознать передаваемые логические уровни. Однако сигнальная «земля» ни в коем случае не должна быть соединена с «землей» источника питания или корпусом устройства. В противном случае через эту линию будет протекать слишком большой ток. Даже если результатом не явится поломка интерфейсных устройств, то наверняка заметно сместятся принимаемые электрические уровни¹, что приведет к искажению информации.

Чтобы избежать возможных неприятностей, перед **включением** питания с помощью омметра следует убедиться, что сопротивление между нулевой шиной источника питания и сигнальной «землей» достаточно велико (сотни килоом).

Стандартные скорости обмена через интерфейс RS-232 составляют 300, 2400 или 9600 бит/с. Передача информации на малых скоростях использовалась раньше для телетайпов, потому что механическим устройствам требовалось много времени на печать одного символа. Нетрудно заметить, что стандартные скорости отличаются друг от друга в число раз, являющееся степенью двойки. Например, 9600 - это пятикратно увеличенное вдвое число 300.

Такое соглашение удобно для аппаратной реализации приемопередатчиков **USART**. Для изменения скорости передачи или приема в этом случае достаточно всего лишь изменить рабочий коэффициент делителя тактовой частоты. Если ваш микроконтроллер не имеет встроенного модуля **USART**, то всегда можно воспользоваться его программной реализацией, имеющейся в стандартной библиотеке

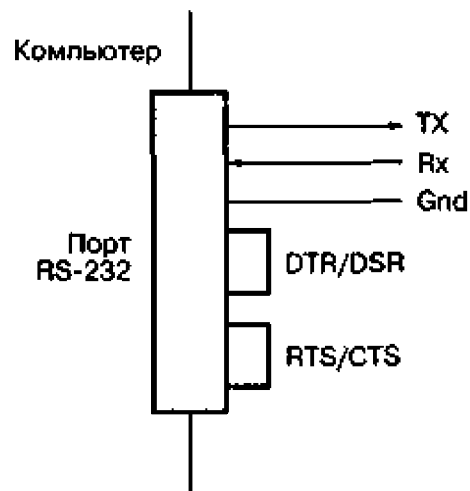


Рис. 4.7. Передача информации по трем линиям интерфейса RS-232

¹ При большом токе согласно закону Ома на соединительных проводах падает заметное напряжение. - Прим. перев.

компилятора PICC Lite. Может показаться, что самостоятельная программная реализация последовательного протокола связи - не такая уж простая задача, но дело существенно облегчается благодаря тому, что в алгоритме вычисления задержки, требуемой для передачи одного бита информации, мистическим образом возникает число 13.

Если для определения задержки, необходимой для передачи одного бита, вы захотите найти число, обратное к величине стандартной скорости передачи данных, то обнаружите, что искомый период, выраженный в микросекундах, делится на число 13 (с небольшой погрешностью). А это означает, что любая стандартная скорость передачи, определенная в интерфейсе RS-232, может быть реализована уже при частоте тактирования, не превышающей 1 МГц.

Рассмотрим пример. Пусть частота системного тактового генератора составляет 20 МГц и требуется вести передачу со скоростью 9600 бит/с. Тогда для определения числа командных циклов, в течение которых будет передаваться каждый бит информации, надо выполнить следующие действия:

1. Разделить 10^6 на скорость передачи, чтобы найти период в микросекундах. Для скорости 9600 бит/с получим около 104 мкс/бит.
2. Разделить найденный период на 13. Для нашего примера получим число 8.

Если тактовый генератор работает на частоте 20 МГц, то период тактовых импульсов равен 50 нс; следовательно, один командный цикл длится $50 \times 4 = 200$ нс. Значит, за 1 мкс выполняется пять командных циклов. Тогда количество командных циклов, которые пройдут за время передачи одного бита (то есть за 104 мкс), равно¹:

$$5 \text{ командных циклов/мкс} \times 13 \times 8 \text{ мкс/бит} = \\ = 520 \text{ командных циклов/бит} = 520 \times 4 \text{ тактов/бит} = 1280 \text{ тактов/бит.}$$

Но почти то же самое получается и без использования операции деления:

$$20 \text{ тактов/мкс} \times 13 \times 5 \text{ мкс/бит} = 1300 \text{ тактов/бит.}$$

Для реализации интерфейса RS-232 в ваших конструкциях обычно удобнее всего использовать 9-контактный разъем и стандартный кабель. Это избавляет от необходимости следить за правильной распайкой кабеля. Соединение двух пар контактов запроса/подтверждения, как показано на рис. 4.7, желательно выполнять внутри самого устройства, а не внутри кабеля или, тем более, со стороны персонального компьютера. В этом случае и кабель, и последовательный порт компьютера не требуют какой-либо переделки и их можно будет использовать для работы с другими устройствами.

Теперь обсудим способы согласования уровней интерфейса RS-232 и обычных ТТЛ/КМОП микросхем. При этом нам не потребуется дополнительный источник

¹ Если отвлечься от мистики, то формула для вычисления периода передачи одного бита (в командных циклах) довольно проста: $n = F / (4V)$, где F - тактовая частота, Гц; V - скорость передачи, бит/с. Что касается числа 13, то предоставляем читателю самому проверить, появляется ли оно в выкладках при других тактовых частотах и на других стандартных скоростях передачи. - *Прим. перев.*

питания для формирования напряжения ± 12 В, так как большинство устройств прекрасно работает при обычном для цифровых микросхем напряжении +5 В.

Первый, наиболее распространенный способ преобразования уровней основан на использовании микросхемы MAX232 (рис. 4.8).

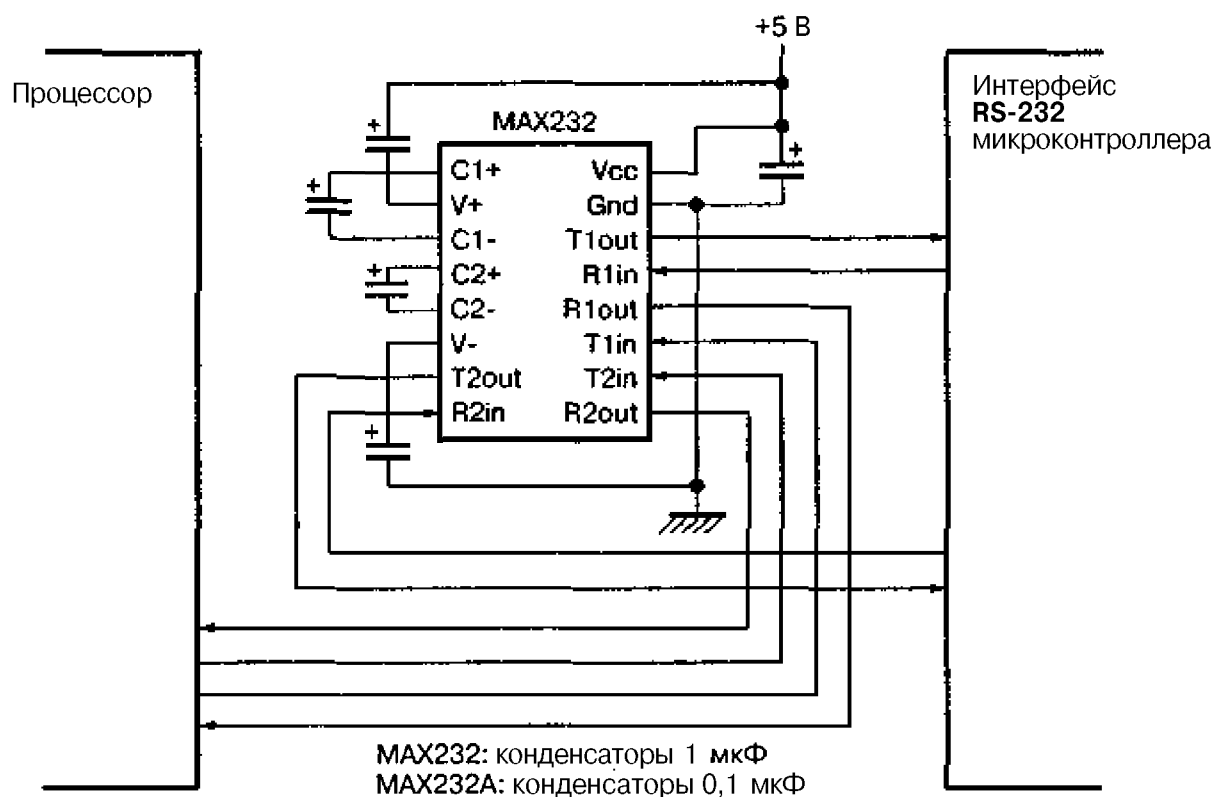


Рис. 4.8. Согласование уровней с помощью микросхемы MAX232

Эта микросхема имеет встроенный преобразователь для формирования напряжения ± 12 В и идеально подходит для реализации трехпроводной последовательной передачи данных. Сигнальная «земля» в этом случае совпадает с «землей» источника питания микросхем.

Кроме MAX232, имеется большое число аналогичных микросхем других производителей, предназначенных для выполнения этой же функции; они дают возможность осуществлять преобразование большего числа сигналов (включая линии квитирования). Многие из них удобны тем, что не требуют подключения внешних конденсаторов. Разумеется, за эти достоинства, как обычно, приходится платить.

Другой способ преобразования логических уровней интерфейса RS-232 в стандартные уровни микросхем ТТЛ/КМОП заключается в использовании инвертора сигнала. Соответствующая схема приведена на рис. 4.9. Недостаток этой схемы - возможность работы только в *полудуплексном* режиме, то есть в каждый момент передача информации может вестись только в одном направлении.

Когда внешнее устройство хочет передать информацию компьютеру, оно посылает данные в виде импульсов и пауз между ними. При высоком уровне сигнала на выходе TX транзистор закрыт и на входе RX действует сигнал низкого уровня,

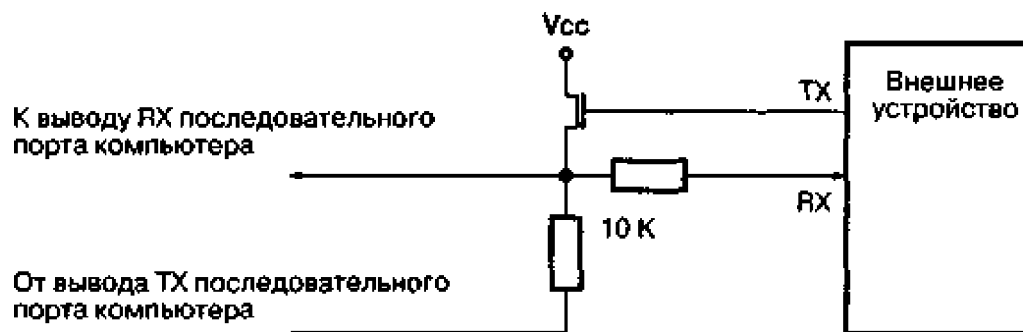


Рис. 4.9. Подключение периферийного устройства к последовательному порту

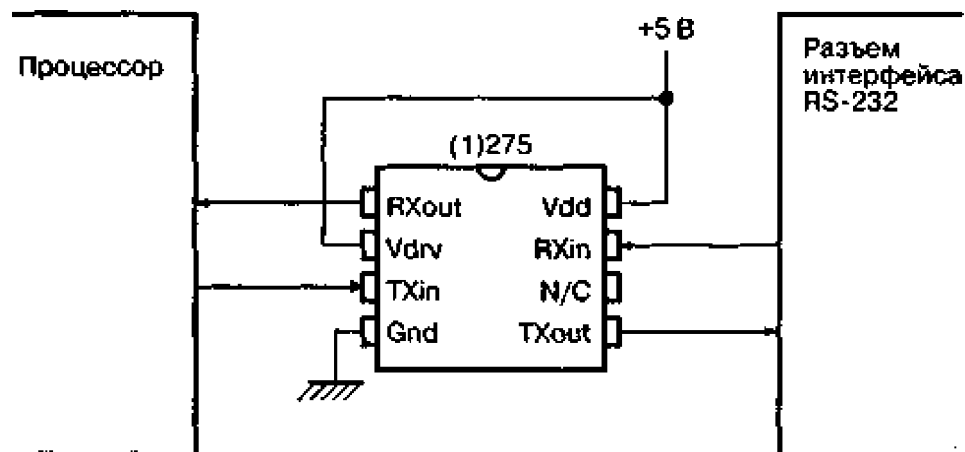


Рис. 4.10. Сопряжение уровней с помощью микросхемы DS(1)275

а при низком уровне выходного сигнала транзистор открывается и на вход подается сигнал высокого уровня.

Описанный способ достаточно прост и недорог; к тому же транзистор потребляет небольшой ток. Однако он требует инвертирования входного сигнала, принимаемого микроконтроллером, а большинство МК не имеют для этого встроенных аппаратных средств. Как следствие, часто используют микросхему DS(1)275 фирмы Dallas Semiconductor, которая аналогична предыдущей схеме на навесных компонентах, но имеет еще и встроенный инвертор (рис. 4.10).

4.6. ПРОГРАММА HYPERTERMINAL

Для того чтобы посылать данные от компьютера к микроконтроллеру и принимать их, требуется специальная программа - *эмулятор терминала*, - которая обеспечивает выполнение следующих функций:

- эмуляцию телетайпа (TTY) и стандартного терминала ANSI (American National Standards Institute — Американский национальный институт стандартов);
- изменение скорости передачи/приема данных;
- передачу/прием данных в формате 8-N-1;
- моноширинные шрифты для отображения информации на экране;
- изменение настроек последовательного порта;

- выбор метода квитиования;
- сохранение настроек;
- передачу/прием текстовых файлов.

Автор рекомендует использовать для этих целей программу HyperTerminal фирмы Hilgraeve Inc. Она имеется в комплекте стандартной поставки большинства версий операционной системы Microsoft Windows и поддерживает все необходимые стандартные функции. Эту программу можно свободно скачать с сайта www.hilgraeve.com. После инсталляции программы необходимо правильно ее сконфигурировать.

Запустите программу HyperTerminal¹. Внешний вид окна программы показан на рис. 4.11.

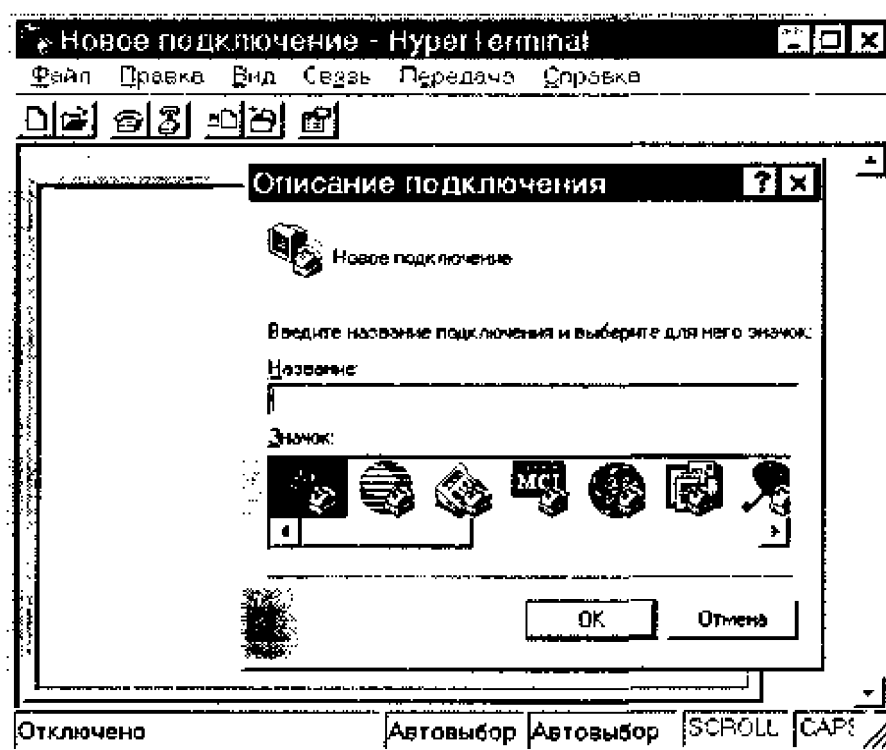


Рис. 4.11. Окно программы *HyperTerminal*

Далее отключите текущее соединение (которое могло установиться автоматически при запуске программы), выбрав пункты меню **Call** ⇒ **Disconnect** (Связь ⇒ Завершить)². Затем выполните команды **File** ⇒ **Properties** (Файл ⇒ Свойства) - на экране появится диалоговое окно настройки параметров (рис. 4.12).

¹ Обычно для этого достаточно выполнить команды Пуск ⇒ Программы ⇒ Стандартные ⇒ Связь ⇒ HyperTerminal (Start ⇒ Programs ⇒ Accessories ⇒ Media ⇒ HyperTerminal). Если появится диалоговое окно для ввода параметров нового соединения, закройте его, нажав кнопку Отмена (Cancel). - Прим. перев.

² Русские варианты названий пунктов меню могут варьироваться в зависимости от версий программы. - Прим. перев.

Перейдите на вкладку Settings (Настройка) и убедитесь, что параметр Terminal Keys (клавиши терминала) активизирован, в поле выбора Emulation (Эмуляция терминала) выбран пункт ANSI, в поле ввода Telnet Terminal ID (Терминал Telnet) также записано значение ANSI, а в поле Backscroll buffer lines (Размер буфера) указано значение 500 строк. Все необходимые настройки показаны на рис. 4.13.

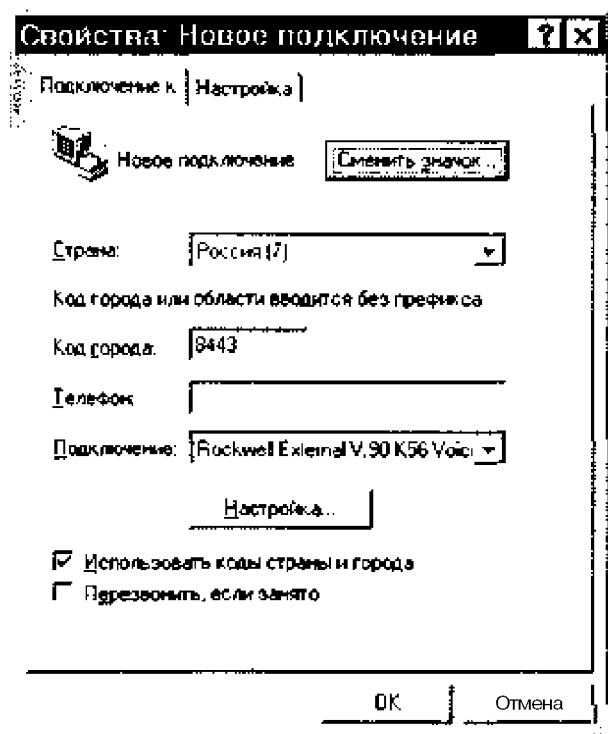


Рис. 4.12. Окно настройки параметров программы *HyperTerminal*

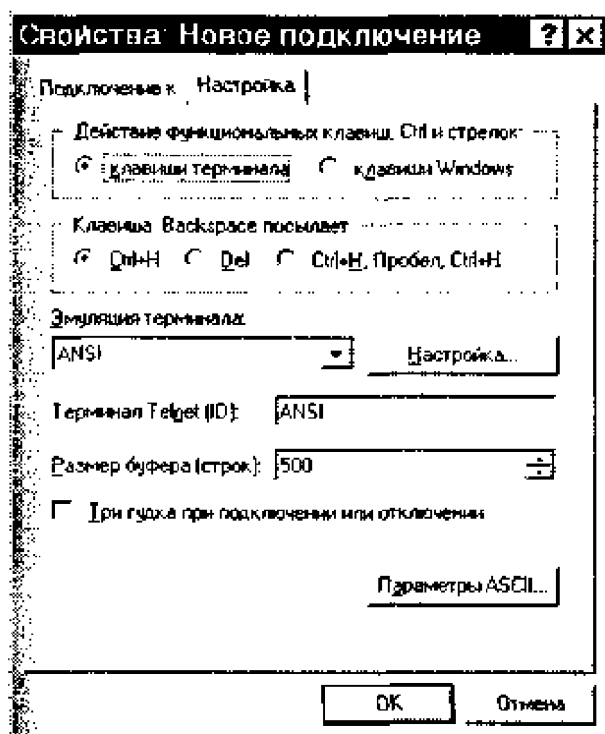


Рис. 4.13. Настройки терминала

Нажав кнопку Terminal Setup (Настройка), можно изменить вид курсора. После нажатия кнопки ASCII Setup (Параметры ASCII) можно задать форматы передаваемых и принимаемых данных. Эти настройки зависят от используемых протоколов обмена, поэтому пока оставим те режимы, что установлены по умолчанию.

Теперь перейдите на вкладку Connect To (Подключение к) и в раскрывающемся списке Connect Using (Подключаться через) выберите используемое соединение (рис. 4.14). Необходимо указать номер последовательного порта, к которому подключено ваше устройство. Кроме одного из имеющихся на вашем компьютере последовательных портов можно указать модем или сетевой адрес (сокеты) для подключения по протоколу TCP/IP.

После выбора соединения надо нажать кнопку Configure (Настройка). В появившемся диалоговом окне (рис. 4.15) надо установить скорость передачи (Bits per second - количество битов в секунду), количество информационных (Data Bits) и стоповых битов (Stop Bits), наличие бита четности (Parity) и метод управления потоком (Flow Control). Мы договорились использовать формат 8-N-1, поэтому надо указать 8 информационных битов, 1 стоповый, отключить проверку на четность. Также необходимо выключить управление потоком (так как мы

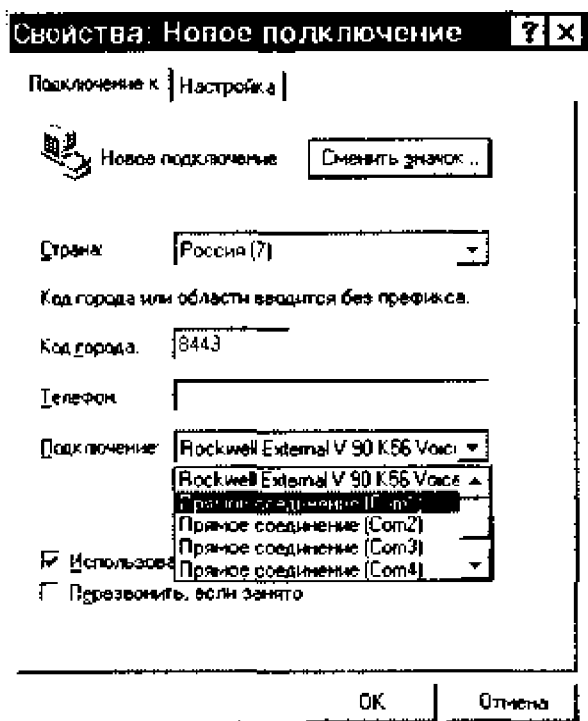


Рис. 4.14. Настройки параметров соединения

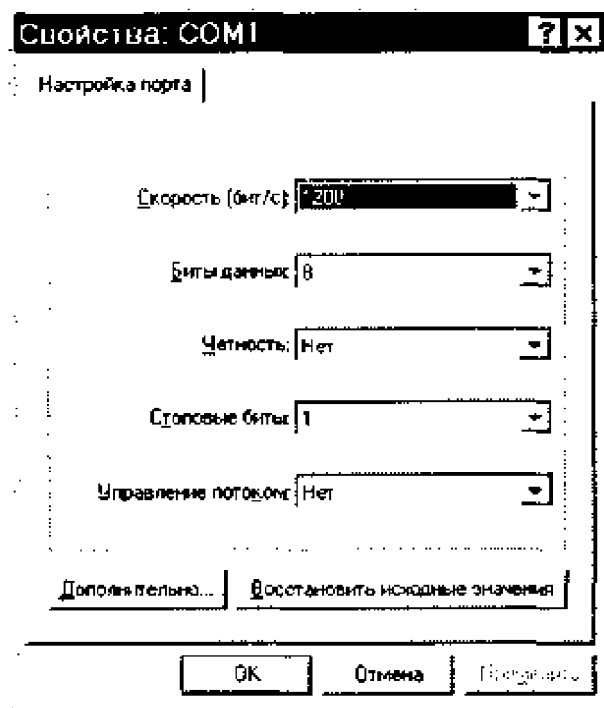


Рис. 4.15. Установки параметров последовательного порта

реализуем соединение по трем проводам без использования сигналов квитирования). В наших проектах мы будем выбирать скорость передачи 1200 бит/с.

Далее нажмите кнопку **Advanced** (Дополнительно) и снимите галочку **Use FIFO Buffers** (Использовать буферы FIFO).

Теперь можно подтвердить все настройки, нажав два раза подряд кнопку **OK**.

После этого надо выбрать шрифт, который будет использоваться для отображения информации на терминале. Для этого предназначены команды меню **View** $\Rightarrow$ **Font** (Вид $\Rightarrow$ Шрифт). Желательно выбрать какой-нибудь моноширинный шрифт, например Courier New (рис. 4.16).

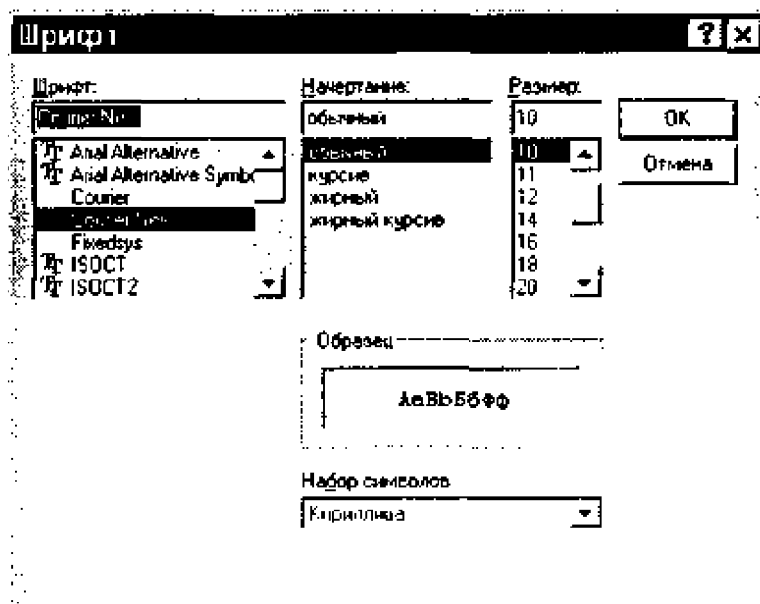


Рис. 4.16. Выбор шрифта

Наконец, желательно сохранить все настройки. Выполните команды **File** $\Rightarrow$ **Save As** (Файл $\Rightarrow$ Сохранить как). Укажите нужную папку, а в качестве имени файла выберите что-нибудь информативное, например:

DIRECT COM 1 - 1200 bps

и нажмите на кнопку **Save** (Сохранить).

Теперь, если установить соединение, выполнив команды **Call** $\Rightarrow$ **Call** (Вызов $\Rightarrow$ Вызов), то при каждом обращении к клавиатуре через последовательный порт вашего компьютера внешнему устройству будет посылаться ASCII-код нажатой клавиши. А данные, приходящие от внешнего устройства, будут сразу отображаться на экране терминала.

Кроме того, вы можете передать целый файл - это позволяет легко реализовать различные тестовые ситуации, избавляя вас от необходимости каждый раз вводить необходимые данные вручную. Для этого может быть использована команда меню **Send File** (Передача $\Rightarrow$ Отправить файл), но предпочтительна команда **Send Text File** (Передача $\Rightarrow$ Отправить текстовый файл), поскольку в последнем случае данные передаются точно в том виде, как если бы вы вводили их с клавиатуры, включая специальные символы перевода строки, возврата каретки и конца файла.

4.7. РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА RS-232 ДЛЯ СВЯЗИ МИКРОКОНТРОЛЛЕРА С ПЕРСОНАЛЬНЫМ КОМПЬЮТЕРОМ

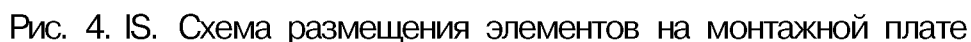
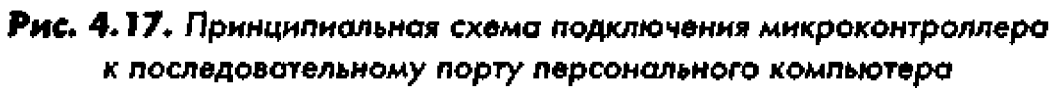
В нашем первом проекте мы реализуем передачу данных между микроконтроллером и персональным компьютером, используя для этого приемопередатчик USART, встроенный в микроконтроллер. Для согласования уровней интерфейса RS-232 и стандартной ТТЛ/КМОП логики можно применить любой из способов, описанных в предыдущем разделе.

Так как МК PIC16F84 не имеет модуля USART, а мы не хотим пока прибегать к его программной реализации, имеющейся в стандартной библиотеке функций компилятора PICC Lite, то в этом проекте МК PIC16F627 не может быть заменен на PIC16F84.

Мы будем предполагать, что микроконтроллер подключен к компьютеру с помощью стандартного кабеля, имеющего разъем-вилку на одном конце и разъем-розетку на другом (этот кабель широко применяется для подключения к последовательному порту персональных компьютеров различных периферийных устройств).

Если для преобразования уровней используется микросхема MAX232, то схема нашего устройства принимает вид, показанный на рис. 4.17. Схема размещения элементов на монтажной плате показана на рис. 4.18.

Контакты 2, 3 и 5 разъема D-9 последовательного порта должны быть припаяны к проводникам монтажной платы. При этом желательно использовать провода трех разных цветов (красный, зеленый или голубой для контактов 2 и 3, черный - для контакта 5).



Наша цель заключается в том, чтобы показать, каким образом можно осуществить обмен данными между микроконтроллером и персональным компьютером. Поэтому пока не будем нагружать наше устройство какой-либо интеллектуальной обработкой информации. Допустим, мы всего лишь хотим, чтобы микроконтроллер получал коды символов, набираемых на клавиатуре компьютера, переводил их в верхний регистр и отправлял назад, чтобы они отображались на экране эмулятора терминала.

Таблица 4.1. Перечень элементов, использованных в схеме на рис. 4.17

Обозначение	Элемент	Примечание
U1	PIC16F627	Микроконтроллер
U2	MAX232 или MAX232A	Микросхема преобразователя уровней сигналов
R1	10 кОм; 0,25 Вт	Для «подтягивания» напряжения на выводе MCLR до напряжения положительной шины питания
C1	0,1 мкФ	Для фильтрации напряжения питания микроконтроллера
C2 – C6	1 или 0,1 мкФ	Для микросхемы MAX232 требуются конденсаторы емкостью 1 мкФ, а для микросхемы MAX232A – 0,1 мкФ
XTAL1	Керамический резонатор на 4 МГц, имеющий встроенный конденсатор	Для генератора тактовых импульсов микроконтроллера
	Материалы	Макетная плата, монтажные провода, источник питания +5 В

Приведенный ниже текст программы можно найти в электронном приложении к этой книге в файле Code\Serial\serial.c:

```
#include <pic.h>

// 22 апреля 2002.
// Передача данных через последовательный порт компьютера.
// Введенная строка символов передается микроконтроллеру после
// нажатия клавиши Enter.
// Принимаемые символы микроконтроллер переводит в верхний регистр
// и посылает обратно на компьютер.
//
// Используется модуль USART микроконтроллера PIC16F627.
// Для согласования уровней служит микросхема MAX232.
//
//
// Используемые аппаратные средства:
// Микроконтроллер PIC16F627,
// внешний тактовый генератор работает на частоте 4 МГц.
// Вывод RB1 - приемник последовательных данных,
// вывод RB2 - передатчик последовательных данных.
//

// Глобальные переменные:
unsigned int RTC = 0;           // Счетчик реального времени.

char StringOut[27] = "\r\nHello\r\n\0"
//                               1 2 345678 9 0 1234567890123456

unsigned char CharOutIndex = 0;
unsigned char CharInIndex = 3;
unsigned char SingleCharFlag = 0;           // Для передачи одного символа
                                           // строки CharOutIndex.

// Слово конфигурации:
#if defined(_16F627)
    <warning PIC16F627 with external XT oscillator selected
```

```

    // Выбран МК PIC16F627 с внешним кварцевым генератором.
__CONFIG(0x03F61);    // Слово конфигурации МК PIC16F627:
                      // внешний кварцевый генератор;
                      // RA6/RA7 используются для ввода-вывода;
                      // внешний сигнал сброса;
                      // таймер PWRT включен;
                      // сторожевой таймер выключен;
                      // защита кода отключена;
                      // детектор падения напряжения питания включен.
#else
    #error Unsupported PICmicro MCU selected
    // Неправильно выбран тип микроконтроллера.
#endif

// Обработчик прерываний:
void interrupt tmr0_int(void)
{
    unsigned char temp;

    if (TOIF) {    // Если запрос на прерывание был от таймера:
        TOIF = 0;    // Сбросить флаг прерывания
        RTC++;    // и выполнить инкремент счетчика.
    }    // Конец обработчика прерываний от таймера.

    if (TXIF) {    // Если запрос на прерывание получен от передатчика:
        TXIF = 0;    // Сбросить флаг прерывания.
        if ((temp = StringOut[CharOutIndex]) == '\0')
            if (SingleCharFlag)    // Только отправить обратно?
                SingleCharFlag = 0;
            else    // Нет, новая строка.
                StringOut[2] = '\n';
        else {
            TXREG = temp;    // Вывод очередного символа.
            CharOutIndex++;    // Передвинуть указатель
                                // к следующему символу.
        }
    }

    > // Конец обработчика прерываний от передатчика.

    if (RCIF) {    // Если запрос на прерывание получен от приемника:
        if ((temp = RCREG) < 0x020) // Получен управляющий символ?
            if (temp == 0x008)    // Это символ Backspace?
                if (CharInIndex > 3) {
                    StringOut[CharOutIndex = CharInIndex] = temp;
                    StringOut[CharInIndex + 1] = ' ';
                    StringOut[CharInIndex + 2] = temp;
                    StringOut[CharInIndex + 3] = '\0';
                    CharInIndex--;
                    TXIF = 1;
                    SingleCharFlag = 1;
                } else;
            else if (temp == 0x00D) { // Enter - конец строки.

```

```

        StringOut[CharInIndex++] = '\\';
        StringOut[CharInIndex++] = '\\r';
        StringOut[CharInIndex++] = '\\n';
        StringOut[CharInIndex] = '\\0';
        CharInIndex = 3;
        CharOutIndex = 0;
        TXIF = 1;
    } else;
else {
    // Обычные символы ASCII.
    if ((temp >= 'a') && (temp <= 'z'))
        temp -= ('a' - 'A');
    if (CharInIndex < 23) { // Оправить назад.
        StringOut[CharOutIndex = CharInIndex] = temp;
        StringOut[++CharInIndex] = '\\0';
        // Символ конца строки.
        TXIF = 1;
        SingleCharFlag = 1;
    }
}

RCIF = 0; // Сбросить флаг прерывания.

} // Конец обработчика прерываний от приемника.

} // Конец обработчика прерываний.

// Главная программа:
void main(void)
{
    unsigned char temp; // Служебная переменная.
    OPTION = 0x0D1; // Предделитель работает с TMRO,
    // коэффициент деления равен 4.

    TMRO = 0; // Начальный сброс таймера TMRO.
    TOIE = 1; // Разрешить прерывания от таймера TMRO.
    GIE = 1; // Разрешить обработку прерываний.

    SPBRG = 51; // Скорость передачи 1200 бит/с
    // при тактовой частоте 4 МГц.
    TXEN = 1; // Разрешить работу модуля USART.
    CREN = 1;
    SPEN = 1;

    PEIE = 1; // Разрешить прерывания от периферийных устройств.

    temp = RCREG;
    RCIF = 0; // Разрешить прерывания от приемника USART.
    RCIE = 1;

    TXIF = 1; // Вызвать прерывание от передатчика USART.
    TXIE = 1;

    while (1 == 1) { // Бесконечный цикл.
    }
} // Конец главной программы.

```

Возможно, вы не нашли в этой программе того, что ожидали увидеть. Мы же договорились, что микроконтроллер будет получать ASCII-код символа, переводить его

в верхний регистр и отправлять обратно компьютеру. Поэтому логично было бы написать нечто вроде следующей программы¹:

```
#include <pic.h>

«if defined(_16F627)

    #warning PIC16F627 Selected
    __CONFIG(0x03F61);

«else

    #error Unsupported PICmicro MCU selected

#endif

void main(void)                // Получение строки символов,
                                // преобразование их к верхнему регистру
                                // и отправка назад.

unsigned char temp;

SP8RG = 51;                    // 1200 бит/с на частоте 4 МГц.
TXEN = 1;                      // Разрешить работу модуля USART.
CREN = 1;
SPEN = 1;

RCIF = 0;                      // Очистить флаг прерывания от приемника.
while (1 == 1)                 // Бесконечный цикл.
    if (RCIF) {                 // Ждем сигнал от приемника.
        temp = RCREG; // Получили код символа.
        if ((temp >= 'a') && (temp <= 'z'))
            temp -= ('a' - 'A'); // Перевели символ в верхний регистр.
        TXREG = temp; // Записали его в буфер передатчика.
        RCIF = 0;      // Сбросили флаг приема
                        // для ожидания следующего символа.
    }
} // Конец главной программы.
```

Эта программа прекрасно справляется с поставленной задачей, но она не может работать совместно с другими задачами, которые мы захотим добавить к нашему проекту. Поэтому следует использовать такой метод программирования, который позволил бы вести отдельную разработку различных интерфейсов и безболезненно добавлять к проекту новые функции.

Поэтому все операции, связанные с реализацией последовательной передачи данных через интерфейс RS-232, а также все другие функции электронного или механического уровней мы будем выполнять не в главной программе, а внутри процедуры обработки прерываний от таймера, которые вырабатываются каждую **миллисекунду**². Подчеркну, что такой метод программирования должен использоваться во всех

¹ Программа исправлена при переводе. В оригинале автор использует директиву __CONFIG внутри функции main (поэтому компиляция заканчивается сообщением об ошибке), а также забывает закрыть фигурную скобку для оператора if. По видимому, те программы в книге, которые не содержатся в электронном приложении, не проверялись на компьютере. — *Прим. перев.*

² Строго говоря, в описанной программе прерывания генерируются не только по запросу таймера, который срабатывает каждые 1024 мкс, но и после приема очередного символа, представленного 8-разрядным кодом ASCII, и по окончании передачи очередного символа. Поэтому название функции tmr0_int не слишком корректно. — *Прим. перев.*

случаях, даже если решаемая задача на первый взгляд вообще не требует обработки каких-либо прерываний.

Для того чтобы программа могла как можно быстрее реагировать на приход очередного символа, используется прерывание от приемника модуля USART, а для записи очередного символа в буфер передатчика - прерывание от передатчика USART, которое вырабатывается каждый раз по окончании передачи предыдущего символа. Такой метод работы с данными мы будем применять и в других наших проектах, например для управления жидкокристаллическим дисплеем.

Если бы мы использовали только прерывания от модуля USART, то программа могла бы выглядеть следующим образом¹:

```
#include <pic.h>

#if defined(_16F627)

    <warning PIC16F627 Selected
    __CONFIG(0x03F61);

#else

    #error Unsupported PICmicro MCU selected

#endif

void interrupt serial_int(void) // Обработчик прерываний.
{
    unsigned char temp;
    if (RCIF) { // Если получен Запрос на прерывание:
        temp = RCREG; // Получить код символа.
        if ((temp >= 'a') &&(temp <= 'z'))
            temp -= ('a'-'A'); // Преобразовать к верхнему регистру.
        TXREG = temp;
        RCIF = 0; // Сбросить флаг прерывания.
    }
} // Конец процедуры обработки прерывания.

// Главная программа:
void main(void) {

    SPBRG = 51; // 1200 бит/с на частоте 4 МГц.
    TXEN = 1; // Разрешить работу USART.
    CREN = 1;
    SPEN = 1;

    RCIF = 0; // Сбросить флаг прерывания.
    while (1 == 1); // Бесконечный цикл.
} // Конец главной программы.
```

Эта программа выглядит намного лучше. Здесь интерфейсная часть вынесена за пределы главной программы и располагается в обработчике прерываний.

¹ Программа исправлена при переводе. В оригинале автор не только повторяет ошибку предыдущей программы, используя директиву __CONFIG внутри функции main, но также располагает обработчик прерываний внутри главной программы. Разумеется, компилятор PICC Lite, как и все остальные компиляторы языка C, не допускает вложенных определений функций. - Прим. перев.

Недостаток программы заключается в том, что она может передавать в основную программу (или принимать из основной программы) только один символ за один вызов обработчика.

На первый взгляд все выглядит правильно, но представьте, что нам при получении от компьютера какой-либо строковой команды, например

Send Left Light Sensor Value

(Узнать состояние левого светового датчика)

надо выполнить эту команду, отправив компьютеру через последовательный порт один байт, характеризующий освещенность этого датчика, например

251

Для реализации этой функции необходимо дождаться приема всей строки, чтобы выяснить, какую команду требуется выполнить, и только после этого передавать ее в главную программу для исполнения.

Лучший способ обмена данными между интерфейсной частью программы и биологическим кодом - использование стандартных для языка C строк в формате ASCIIZ, оканчивающихся нулевым байтом. В этом случае ограничитель строки - символ NUL с кодом 0x000 - играет роль флага, который позволяет управлять работой программы. При получении очередного байта в интерфейсной части программы производится его сравнение с нулем, и если принятый байт отличен от нуля, то ожидается приход следующего символа, а если равен нулю, то вся полученная строка передается в основную программу.

Когда данные передаются микроконтроллеру с помощью эмулятора терминала, то ввод очередной строки пользователь завершает нажатием клавиши Enter. В результате генерируется символ с кодом 0x00D (CR - возврат каретки). Поэтому при получении данных микроконтроллер должен заменить символы с кодом 0x00D нулевыми байтами. Тогда фрагмент кода для обработки прерывания от приемника модуля USART будет выглядеть примерно так:

```
if (RCIF) {           // Если запрос на прерывание был от приемника:
    temp = RCREG;
    RCIF = 0;
    if (temp < 0x020) { // Это управляющий символ?
        if ((temp == 0x008) && (StringInIndex != 0))
            // Это символ Backspace?
            StringInIndex--; // Стираем предыдущий символ.
        else
            if (temp == 0x00D) { // Символ возврата каретки.
                StringIn[StringInIndex] = '\0';
                StringInFlag = 1; // Прием строки закончен.
            } else ;           // Нет других управляющих символов.
    } else
        // Обычные символы ASCII.
        StringIn[StringInIndex++] = temp;
}
```

Этот фрагмент уже почти не отличается от того, что мы использовали в нашей программе. Теперь данные принимаются построчно. Разница только в том, что обратно на компьютер они все еще отправляются отдельными символами.

Вы, вероятно, заметили, что в приложение включена возможность стирания предыдущего принятого символа нажатием клавиши **Backspace**. Эта функция весьма полезна, если вы набираете данные для отправки микроконтроллеру на клавиатуре компьютера. При этом обязательно рано или поздно возникнет необходимость исправить уже введенный символ¹.

Наконец, внесем последние исправления в наш код, чтобы реализовать построчную отправку символов на компьютер. С этой целью мы используем тот факт, что после отправки предыдущего байта, когда освобождается регистр TXREG, автоматически устанавливается флаг прерывания от передатчика TXIF. Поэтому для отправки всех остальных символов при обработке каждого прерывания от передатчика USART надо проверять, не достигнут ли конец строки, и если нет, то записывать в регистр TXREG новый символ для отправки:

```
if (TXIF) {           // Если регистр передатчика пуст:
    if ( ( temp = Message[MessageOffset++] ) != 0 )
        TXREG = temp;      // Очередной символ.
    else
        TXREGEmptyFlag = 1; // Достигнут конец строки.
}
```

Тогда в главной программе для отправки строки достаточно использовать следующие четыре оператора:

```
Message = StringToOutput; // Указатель на отправляемую строку.
MessageOffset = 0;
TXREGEmptyFlag = 0;        // Передатчик теперь занят.
TXIF = 1;                  // Генерируем запрос на прерывание.
```

Как только флаг прерывания от передатчика TXIF установлен в 1, происходит переход на процедуру обработки прерывания (если предварительно установлены флаги разрешения обработки прерываний). В обработчике этот флаг сбрасывается, а затем устанавливается автоматически, когда передача очередного байта завершается. В результате все остальные символы будут отправлены автоматически. Когда указатель MessageOffset достигнет конца строки (нулевого байта), будет установлен программный флаг TXREGEmptyFlag, указывающий, что больше передавать нечего². Теперь мы уже почти вплотную подошли к тому, что написано в нашем приложении. Не хватает только обработки принятых символов, то есть преобразования их к верхнему регистру, но в этом вы без труда разберетесь сами.

Внимательно проанализировав код приложения, вы обнаружите, что в начале работы на компьютер посылается строка приветствия "Hello". Чтобы сделать это, достаточно двух команд:

```
TXIE = 1;           // Разрешить прерывания от передатчика.
TXIF = 1;           // Сгенерировать прерывание от передатчика.
```

¹ Эмулятор терминала отправляет данные побайтно, не дожидаясь конца строки. — *Прим. перев.*

² В этом примере автор для простоты нигде не проверяет состояние флага TXREGEmptyFlag. Реальное приложение должно это делать. — *Прим. перев.*

В отладчике MPLAB можно увидеть, что в результате компилятор добавит к программе две машинные команды.

Для тестирования нашего устройства надо запустить программу HyperTerminal и установить соединение на скорости 1200 бит/с. Если теперь мы включим питание микроконтроллера, то увидим на экране терминала приветствие. На рис. 4.19 показан внешний вид окна программы HyperTerminal после того, как в ответ на приветствие пользователь нажал пять клавиш ABCDE, а затем клавишу Enter. Как и положено, микроконтроллер вернул все пять символов.

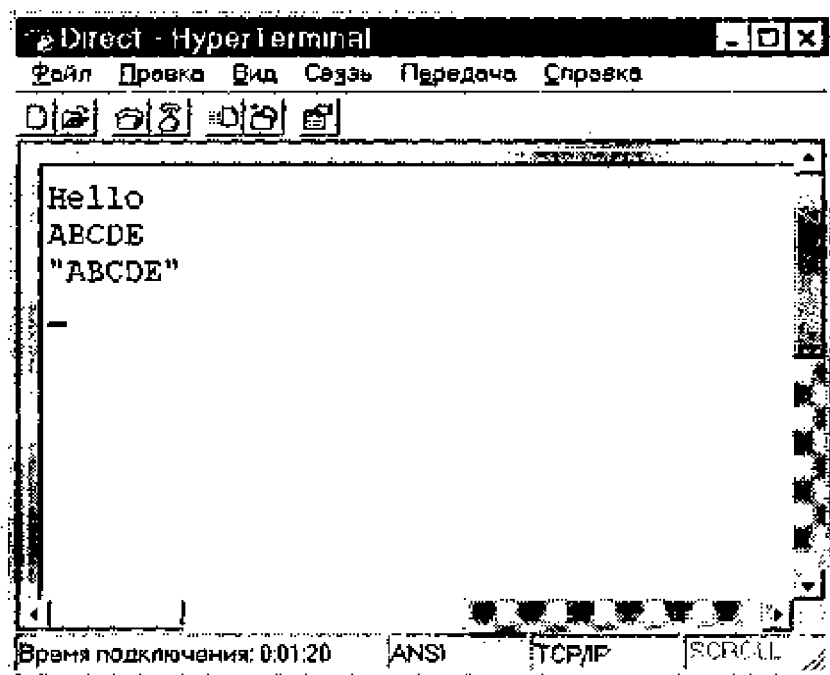


Рис. 4.19. Внешний вид окна программы HyperTerminal во время тестирования устройства

При разработке кода электронного уровня я советую по возможности всегда передавать данные в виде строк в формате ASCIIZ. Чуть позже в этой главе мы будем использовать их для отображения информации на жидкокристаллическом дисплее и для реализации дистанционного управления роботом с помощью инфракрасных лучей. При этом, как и в нашем первом примере, обработка информации будет производиться не по отдельным символам, а целыми блоками.

Но прежде чем перейти к описанию следующего устройства, необходимо сделать три замечания по только что рассмотренному.

Первое касается соединительных проводов. Можно, конечно, подвешивать их к потолку, но это не самое оптимальное решение проблемы. Провода любой длины рано или поздно оказываются слишком короткими и ограничивают площадь, доступную для передвижения робота; кроме того, провода могут намотаться на ось колеса. Наилучшее решение - использование стандартных радиопередающих/приемных модулей для реализации беспроводного интерфейса RS-232.

Второе замечание относится к скорости передачи данных. В нашем примере мы установили ее равной 1200 бит/с. Разумеется, по сегодняшним меркам это очень мало. Не удивлюсь, если ваш домашний Ethernet работает на скорости 100 Мбит/с, а это

на пять порядков больше, чем 1200 бит/с. Но прежде чем пытаться протестировать работу нашего устройства на более высоких скоростях, следует убедиться, что оно работает на низких. Кроме того, при невысоких скоростях передачи снижаются электромагнитные наводки и повышается надежность канала связи, что позволяет работать с более длинными соединительными проводами и более простой аппаратурой передачи/приема данных.

Более высокие скорости требуются только при **большом** объеме передаваемых данных (например, если ваш робот оснащен телекамерой). Что касается рассматриваемого устройства, вы все равно не сможете нажимать клавиши чаще, чем несколько раз в секунду.

Последнее замечание - о системе команд, которыми компьютер обменивается с роботом. Конечно, быстрее и проще передавать закодированные короткие команды (не длиннее одного байта), но я все же рекомендую использовать удобочитаемые строковые команды. Даже если в дальнейшем вы захотите разработать графический интерфейс пользователя (Graphical User Interface - GUI), с помощью которого закодированные короткие команды будут переводиться в формат, удобный для восприятия, предлагаемый мной подход упростит процесс отладки ваших приложений. Ведь еще до разработки специальных программ вы сможете проводить тестирование своего устройства с помощью простого эмулятора терминала, используя обычные строковые команды.

4.8. ДВУНАПРАВЛЕННЫЙ СИНХРОННЫЙ ИНТЕРФЕЙС

Интерфейс RS-232 использует *асинхронный* протокол передачи данных, потому что приемник не получает какого-либо сигнала тактирования в явном виде.

На рис. 4.20 показаны временные диаграммы сигналов при *синхронной* передаче данных. Здесь приемник фиксирует данные на линии Data по переднему или заднему фронту синхроимпульсов Clock.



Рис. 4.20. Синхронная передача данных

Имеется большое число различных синхронных последовательных протоколов. Многие из них широко примеряются, и для их реализации доступны необходимые аппаратные средства. Недостаток этих интерфейсов состоит в том, что при подключении нескольких устройств они требуют использования как минимум одной дополнительной управляющей линии для выбора активного устройства, которое в настоящий момент должно передавать или получать информацию.

Этот недостаток отсутствует у интерфейса PC. Он был первоначально разработан фирмой Philips в конце 1970-х годов специально для того, чтобы обеспечивать

такой способ подключения периферийных устройств к микропроцессорам, который не требовал бы использования традиционных шин адреса, данных и управления, а кроме того, позволял бы нескольким микропроцессорам работать с одними и теми же периферийными устройствами (**multimastering**).

Интерфейс PC использует всего две линии - они именуются SCL (Serial Clock) и SDA (Serial Data). Первая предназначена для передачи синхроимпульсов (они формируются тем устройством, которое в настоящий момент передает данные), а вторая — для передачи самих данных и команд, управляющих этим процессом. Обе линии имеют открытый коллектор (как и во многих других случаях, когда необходимо, чтобы к одной линии могло подключаться несколько различных устройств), поэтому требуют подключения «подтягивающих» резисторов сопротивлением 1-10 кОм.

Для примера на рис. 4.21 показана структурная схема устройства управления стереосистемой.

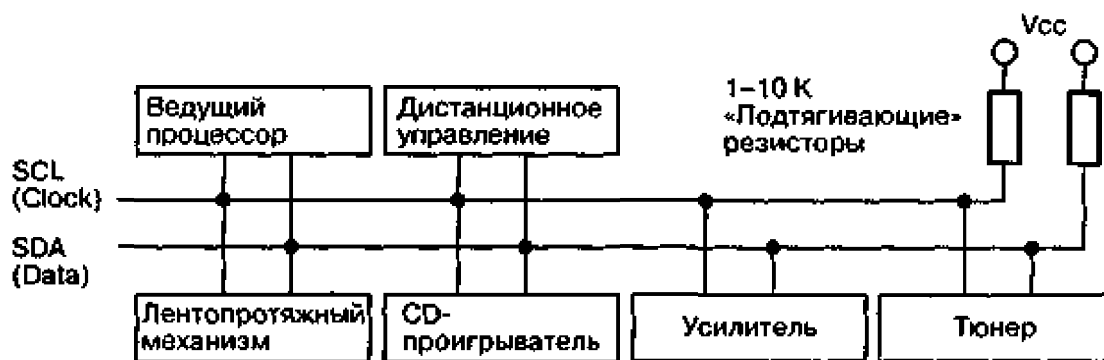


Рис. 4.21. Устройство управления стереосистемой на основе интерфейса PC

В обмене информацией по шине PC всегда принимают участие два устройства - *ведущее* (master, задатчик) и *ведомое*. Ведущее устройство вырабатывает синхроимпульсы, а принимать или передавать данные может как задатчик, так и ведомое устройство.

Пока ни одно устройство не начало передачу данных, благодаря «подтягивающим» резисторам на обеих линиях шины PC действует напряжение высокого уровня. Если какое-либо устройство собирается начать передачу данных, оно сначала проверяет, свободна ли шина. Ведь в каждый момент времени ведущим на шине может быть только одно устройство. Напряжение высокого уровня на линии SCL показывает, что шина пока свободна.

Перед началом процесса передачи задатчик устанавливает напряжение низкого уровня сначала на линии SDA а затем на линии SCL (рис. 4.22). В процессе передачи данных такое состояние линий невозможно, поскольку сигнал на линии SDA не должен изменяться во время действия тактового импульса на линии SCL.

Затем начинается передача данных от ведущего устройства к ведомому (slave) или наоборот, но в любом случае источником синхроимпульсов является задатчик. Данные фиксируются приемником по заднему фронту синхроимпульсов.

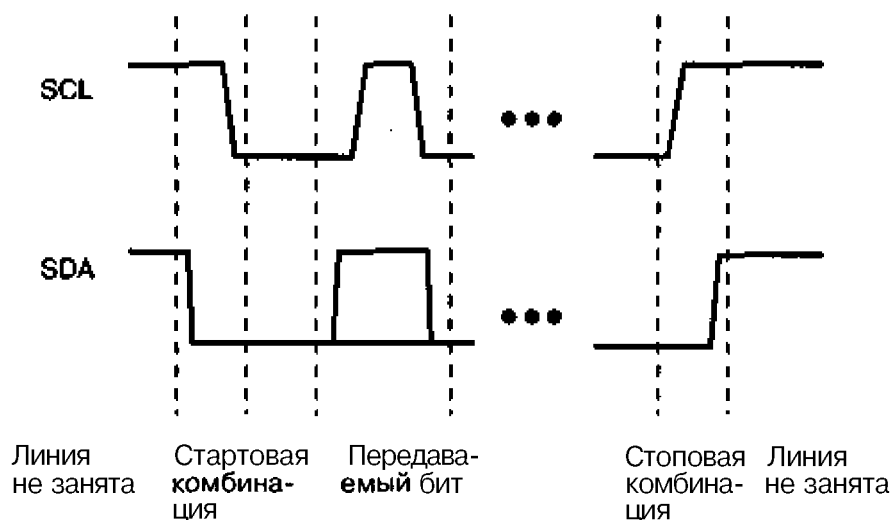


Рис. 4.22. Начало и конец передачи данных по интерфейсу PC

В конце передачи ведущее устройство прекращает генерацию синхроимпульсов; в результате на линии SCL благодаря «подтягивающему» резистору устанавливается напряжение высокого уровня, после этого отключается передатчик, из-за чего устанавливается высокий уровень на линии SDA — иными словами, повторяется ситуация, обратная той, что наблюдалась перед началом передачи.

В отличие от интерфейса RS-232, передача данных производится начиная со старшего бита; при этом используются обычные логические уровни микросхем ТТЛ/КМОП. После передачи последнего (восьмого) бита каждого байта во время действия очередного синхроимпульса передатчик отключается от линии SDA чтобы дать возможность приемнику подтвердить получение данных. Для этого приемник должен выставить на линии SDA сигнал низкого уровня. Перед посылкой очередного бита сигнал низкого уровня действует на обеих линиях. Временные диаграммы на рис. 4.23 иллюстрируют процесс передачи одного байта данных по интерфейсу PC.

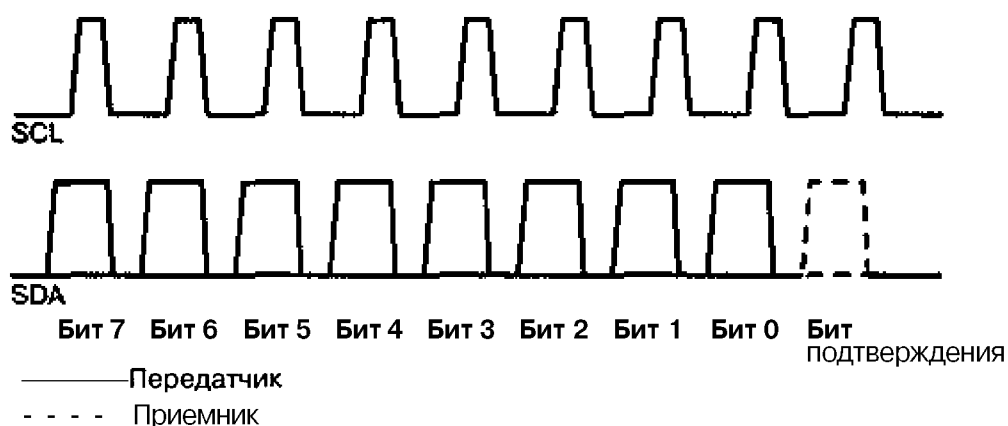


Рис. 4.23. Передача данных по интерфейсу I²C

В некоторых случаях бит подтверждения передается высоким уровнем сигнала, даже если прием прошел успешно. Это показывает, что обмен закончен и передатчик (обычно являющийся либо ведущим устройством, либо **здатчиком**, который не

должен сам начинать операцию обмена) может подготовиться к получению следующего запроса¹.

Минимальная скорость передачи по интерфейсу PC ничем не ограничена. И передатчик, и приемник могут при необходимости замедлять процесс обмена на неопределенное время. Задатчик делает это, удерживая сигнал низкого уровня на линии SCL после приема или передачи предыдущего бита. Ведомое устройство может замедлить работу задатчика, удерживая сигнал на линии SCL на низком уровне после приема или передачи очередного бита (увидев это, задатчик не сможет выставить на линии SCL следующий синхроимпульс).

Но существуют две максимальные скорости передачи. В так называемом *стандартном* режиме это 100 Кбит/с (частота синхроимпульсов 100 кГц), а в *быстром* режиме - 400 Кбит/с (частота синхроимпульсов 400 кГц). На рис. 4.24 показаны минимальные временные задержки для обоих режимов (все значения указаны в микросекундах).

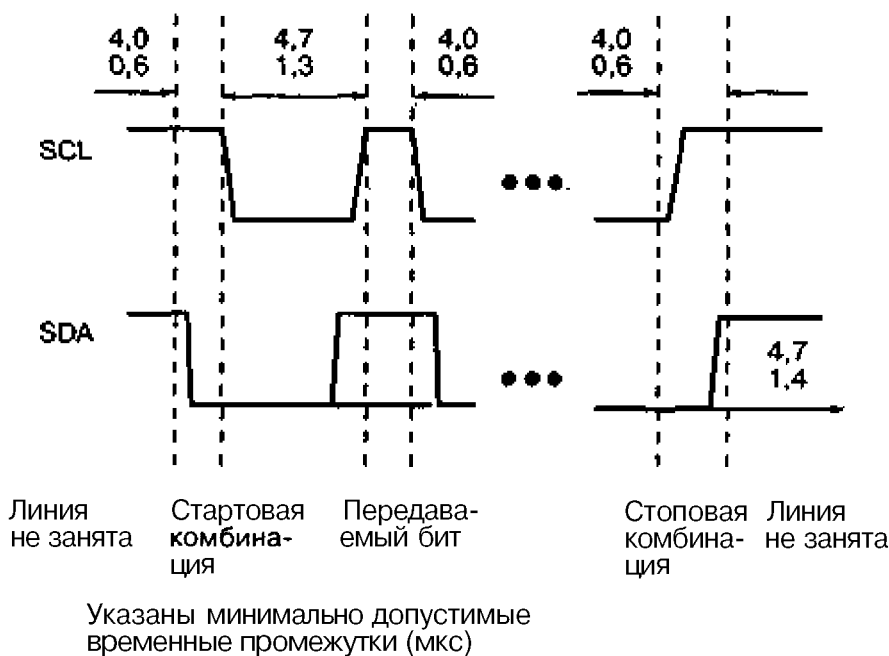


Рис. 4.24. Минимальные временные задержки
при двух режимах передачи данных по интерфейсу PC

На рис. 4.25 показан формат команд, используемых для управления процессом передачи данных по интерфейсу PC.

Адрес получателя данных задается семью битами. Старшие четыре бита адреса определяют тип устройства, а оставшиеся три младших бита указывают, какому именно устройству (из восьми возможных) этого типа предназначена посылаемая информация.

¹ Этот режим используется, когда микроконтроллер запрашивает данные у какого-либо периферийного устройства. В этом случае микроконтроллер является приемником данных. Если вместо бита подтверждения микроконтроллер выставит сигнал высокого уровня, то ведомое устройство «поймет», что следующую порцию данных пересылать не нужно. - Прим. *перев.*

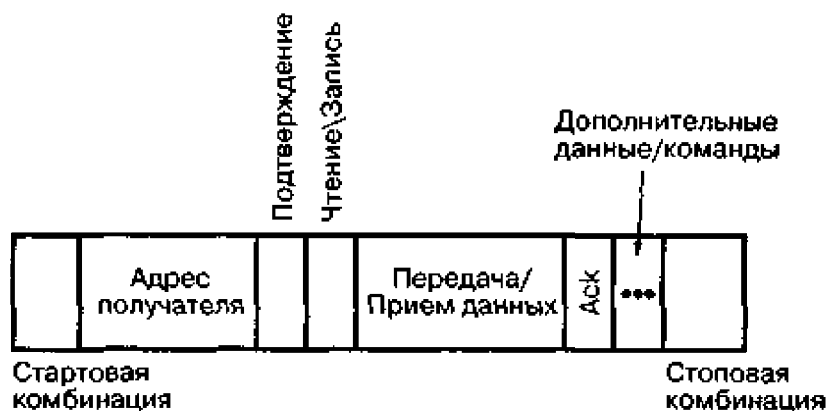


Рис. 4.25. Формат управляющих команд интерфейса РС

В некоторых случаях требуется чуть усложнять протокол обмена. Например, при чтении информации из памяти EEPROM (или записи данных в память) датчик должен сначала установить стартовую последовательность, чтобы переслать адрес нужной ячейки памяти, а затем снова выполнить стартовую последовательность, чтобы теперь уже считать данные из памяти (или записать их).

Для того чтобы ведущими на шине могли быть различные устройства, необходим какой-либо *протокол разрешения коллизий* (конфликтов). Коллизия возникает, когда два устройства, одновременно проверив состояние шины и обнаружив, что она пока свободна, начинают передачу данных.

Конфликты разрешаются благодаря тому, что на линии с открытым коллектором подача сигнала высокого уровня реализуется на самом деле простым отключением активного устройства (вспомните о «подтягивающих» резисторах). В этом случае побеждает всегда то устройство, которое выставило сигнал низкого уровня. Тогда второе устройство, «увидев», что действующий на линии уровень напряжения не совпадает с тем, который оно пытается установить, «понимает», что на шине активен еще один **датчик**, и на время отключается, чтобы дать ему возможность беспрепятственно закончить обмен информацией.

Реализация интерфейса I²C с помощью микроконтроллеров PICmicro весьма проста. Однако из-за программной его реализации трудно достичь высоких скоростей передачи. Даже максимальная скорость стандартного режима (100 Кбит/с) может оказаться недостижимой.

Я считаю, что программная реализация интерфейса I²C все же является наилучшим решением, если кроме микроконтроллера на шине не может быть других **датчиков**. Ведь в этом случае не требуется синхронизировать его работу с какими-либо быстрыми устройствами, в которых используется аппаратная реализация этого интерфейса.

4.9. УСТРОЙСТВА ИНДИКАЦИИ

Программные симуляторы, такие как MPLAB, могут дать разработчику некоторую уверенность в том, что его устройство работает правильно. К сожалению, они оказываются бессильны при необходимости смоделировать сложные операции

вывода-вывода. В ряде случаев вы не можете с достаточной точностью задать входные сигналы, а следовательно, и проверить, как на них реагирует микроконтроллер. Для более точного моделирования различных ситуаций необходим внутрисхемный эмулятор, но он требует непосредственного подключения отлаживаемой схемы к компьютеру; в результате соединительные провода оказываются еще одной причиной того, что попытки точно смоделировать поведение робота в реальной ситуации терпят неудачу.

В некоторых случаях решение проблемы оказывается довольно простым: для индикации текущего состояния робота и управляющей программы можно использовать светодиоды и/или звуковые излучатели. Часто более изощренные методы индикации (вроде жидкокристаллического дисплея) оказываются менее эффективными.

При добавлении световых или звуковых индикаторов к своему проекту (их можно было бы назвать устройствами обратной связи - ведь они помогают пользователю или разработчику понять, что происходит с роботом, застрявшим в дальнем углу комнаты) важно помнить, что неумелое их использование может обернуться именно теми проблемами, которые мы пытались решить с их помощью. Ведь световой или звуковой сигнал, вырабатываемый этими индикаторами, может воздействовать на свето- или звукочувствительные датчики робота, мешая их нормальной работе.

Для индикации различных ситуаций можно использовать комбинации светодиодов разного цвета, а также звуковой зуммер, вырабатывающий разное количество гудков или генерирующий сигналы различной высоты.

Тем не менее я не рекомендую использовать сигналы разной тональности или различной длительности, так как не у всех людей идеальный слух или хорошее чувство ритма. Проще подавать сигналы при помощи разного количества гудков одинаковой длительности и высоты. Однако учтите: если число гудков превышает 3, всегда найдется кто-нибудь, кто ошибется при счете.

Если вы не желаете добавлять к своему устройству световую или звуковую сигнализацию, так как боитесь, что из-за этого он станет больше походить на игрушку, примите во внимание, что после отладки программы вы всегда можете отключить ее.

4.10. СВЕТОДИОДНЫЕ ИНДИКАТОРЫ

Светодиодные индикаторы - это наиболее распространенные устройства вывода информации в современной электронике. Они недороги, и их крайне легко использовать в разных ситуациях. Они могут различаться по форме и цвету; на их основе могут быть выполнены числовые и текстовые дисплеи.

Для использования в схемах нам потребуется знать некоторые электрические характеристики светодиодов. Прежде всего, не следует забывать, что они, как и обычные диоды, проводят ток только в одном направлении. Современные светодиоды начинают светиться уже при токе 5 мА, хотя многим (особенно самым ярким) требуется для этого не менее 20 мА.

При подключении к микроконтроллеру анод светодиода обычно соединяют с положительной шиной питания, а катод соединяют с нужным выводом микроконтроллера через ограничивающий резистор, как показано на рис. 4.26. В этом случае светодиод загорается, если на выходе микроконтроллера действует сигнал низкого уровня.

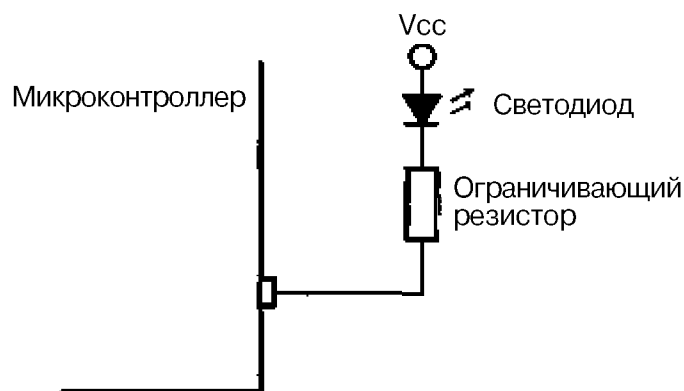


Рис. 4.26. Подключение светодиодного излучателя к микроконтроллеру/

Если подключить светодиод между выводом микроконтроллера и нулевой шиной питания, то может оказаться, что выходной ток микроконтроллера (например, i8051) будет слишком мал, чтобы обеспечить достаточное свечение светодиода.

Падение напряжения на светодиоде в прямом включении обычно составляет около 2 В, что заметно отличается от аналогичного показателя у простого кремниевого диода (0,6-0,7 В). Это напряжение необходимо знать при выборе сопротивления ограничивающего резистора. Обычно для работы с 5-вольтовыми сигналами я использую резисторы сопротивлением 470 Ом¹.

Для изменения яркости свечения можно использовать широтно-импульсную модуляцию (ШИМ) выходного сигнала микроконтроллера. Это проще, чем пытаться управлять сопротивлением резистора. Мы рассмотрим соответствующий пример чуть позже в этой главе.

Для отображения цифр и некоторых букв часто используется семисегментный светодиодный индикатор (рис. 4.27). В таких индикаторах катоды или аноды всех светодиодов соединены друг с другом и имеют один общий вывод. Поэтому различают светодиодные индикаторы с *общим катодом* или с *общим анодом*.

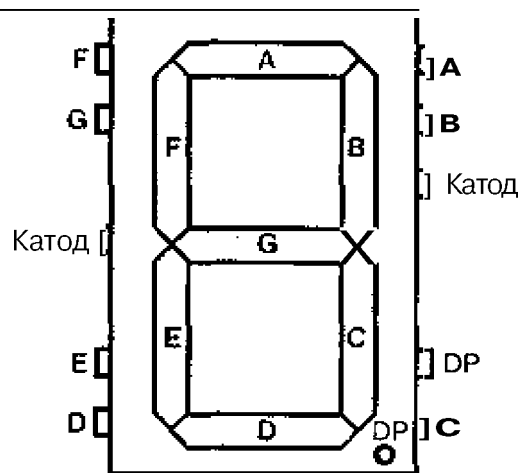


Рис. 4.27. Семисегментный светодиодный индикатор

¹ Формула расчета сопротивления ограничивающего резистора: $R = (U_{cc} - U_{пр}) / I_{ном}$, где U_{cc} - напряжение питания; $U_{пр}$ - падение напряжения на светодиоде в прямом включении; $I_{ном}$ - номинальный ток, необходимый для свечения светодиода. Таким образом, при $U_{cc} = 5$ В, $U_{пр} = 2$ В и $R = 470$ Ом ток через светодиод равен 6,4 мА. — Прим. перев.

Кроме того что при этом уменьшается количество внешних выводов, такое соединение позволяет легко реализовать управление группой одинаковых индикаторов, как показано на рис. 4.28. Здесь четыре светодиодных индикатора с общим катодом подключены к одним и тем же выводам микроконтроллера, а с помощью дополнительных выводов осуществляется их включение и выключение.

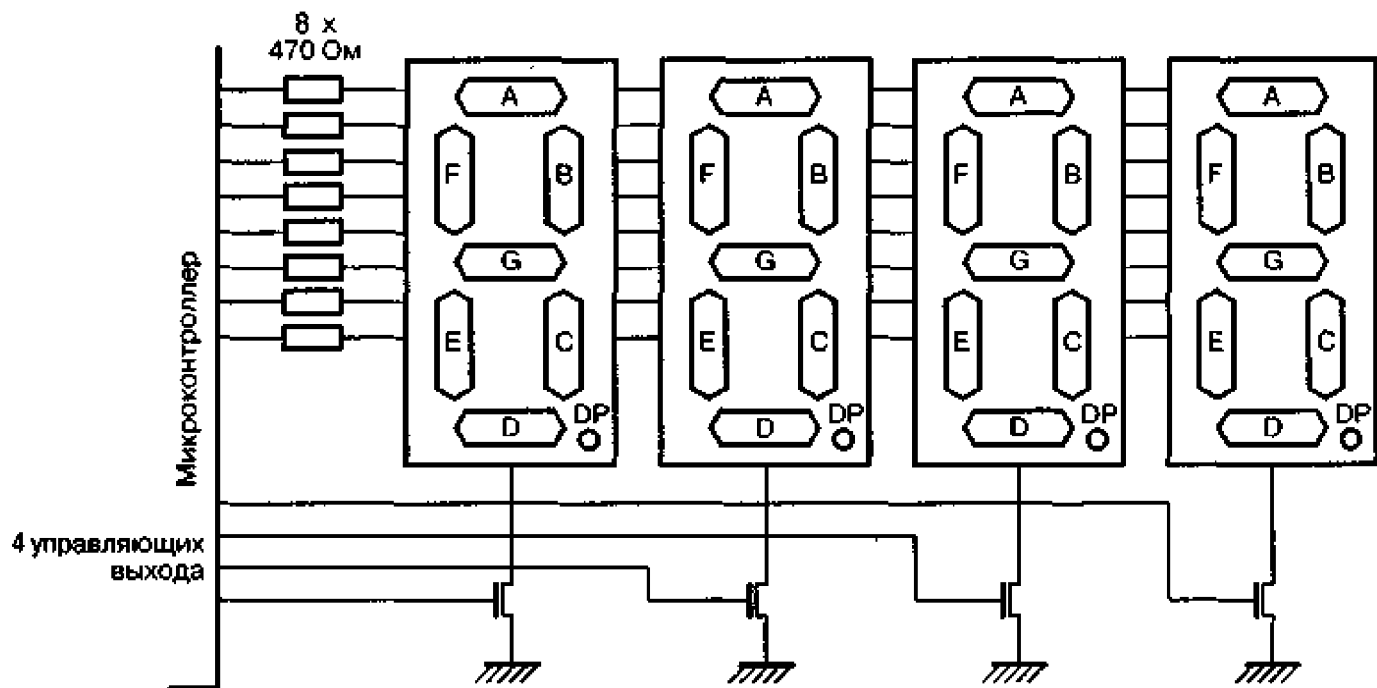


Рис. 4.28. Подключение **семисегментного** светодиодного индикатора к микроконтроллеру

Индикаторы включаются по очереди. В каждый момент времени на информационные выходы микроконтроллера выводятся данные, предназначенные для одного индикатора; в это время все остальные отключены, для чего на затворы соответствующих транзисторов подается сигнал низкого уровня. Чтобы не было заметно мерцания светодиодов, каждый из них должен включаться и выключаться по крайней мере 100 раз в секунду. Если микроконтроллер управляет четырьмя семисегментными индикаторами, то длительность работы каждого из них составляет не более 25% всего времени. Увеличение числа индикаторов приводит к уменьшению той доли времени, в течение которой ток течет через светодиоды.

Для управления индикаторами хорошо подходит наш стандартный метод реализации интерфейсных функций с помощью прерываний от таймера, генерируемых каждую 1 мс. Соответствующий фрагмент обработчика прерываний:

```
// Управление четырьмя семисегментными светодиодными индикаторами:
ControlPin[OldRTC] = 0;    // Выключить текущий индикатор.
OldRTC = RTC & 3;         // Младшие два бита используются
                           // для адресации индикаторов.
DataOut = Data[OldRTC];    // Данные для текущего индикатора.
ControlPin[OldRTC] = 1;    // Разрешить вывод нового символа.
```

Здесь предполагается, что DataOut - это порт ввода-вывода, например PORTB микроконтроллера PIC16F627. ControlPin - это массив, состоящий из четырех одноразрядных элементов, которые используются для включения соответствующих

индикаторов (в приложении указано, как на языке С описать битовую структуру). Массив Data заполняется в главной программе и определяет информацию, отображаемую на индикаторах.

При выполнении этой программы каждый индикатор включается примерно 250 раз в секунду, поэтому глаз человека не может заметить мигания светодиодов.

4.11. УПРАВЛЕНИЕ СВЕТОДИОДНЫМ ИНДИКАТОРОМ

Рассмотрим пример схемы и управляющей программы, которая демонстрирует пример подключения светодиодных индикаторов к микроконтроллеру. Допустим, мы хотим, чтобы светодиод, подключенный к выводу RB1 микроконтроллера PIC16F627, зажигался примерно раз в секунду. Здесь мы, как и в предыдущем примере, имеем дело с функциями электронного уровня.

Схема подключения показана на рис. 4.29, а в табл. 4.2 перечислены используемые в ней элементы.

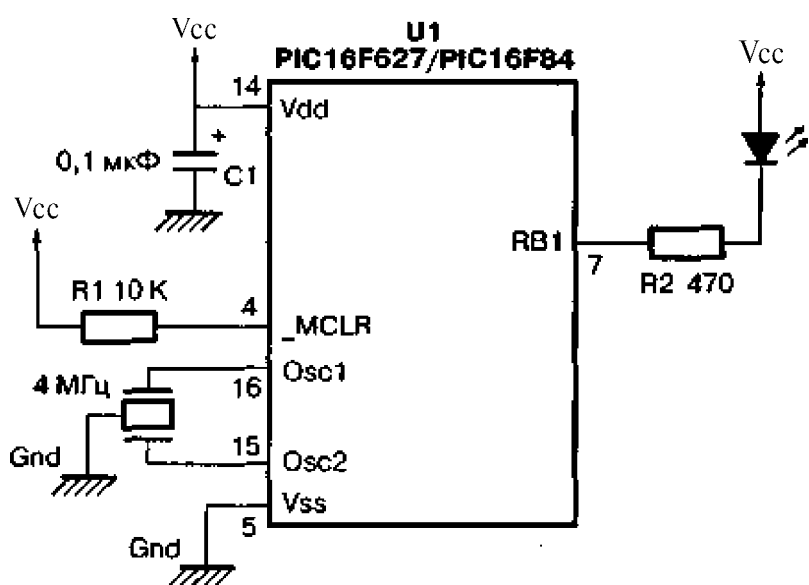


Рис. 4.29. Микроконтроллер управляет светодиодом

Таблица 4.2. Перечень элементов, используемых в схеме на рис. 4.29

Обозначение	Элемент	Примечание
U1	PIC16F627 или PIC16F84	Микроконтроллер. Можно использовать отечественный светодиод красного свечения АЛ370, номинал резистора R2 при этом необходимо снизить до 300 Ом
CR1	Светодиод	Любой видимого диапазона
R1	10 кОм; 0,25 Вт	Для «подтягивания» напряжения на выводе _MCLR до напряжения положительной шины питания
R2	470 Ом; 0,25 Вт	Для ограничения тока через светодиод
C1	0,1 мкФ	Для фильтрации напряжения питания микроконтроллера
XTAL1	Керамический резонатор на 4 МГц, имеющий встроенный конденсатор	Для генератора тактовых импульсов микроконтроллера PIC16F84. Для PIC16F627 не требуется
	Материалы	Макетная плата, монтажные провода, источник питания +5 В

Схема размещения элементов приведена на рис. 4.30. От рис. 4.3 она отличается только наличием светодиода CR1 и резистора R2. При монтаже обратите внимание на правильную полярность подключения светодиода.

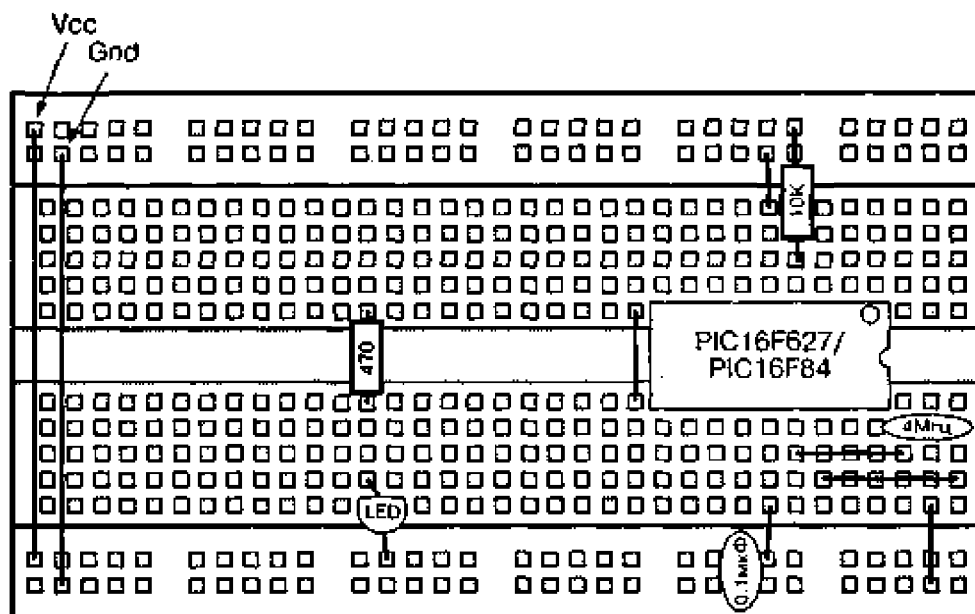


Рис. 4.30. Схема размещения элементов

В случае применения микроконтроллера PIC16F627 из схемы надо исключить керамический резонатор; в слове конфигурации для этого МК мы указываем, что используется внутренний тактовый генератор.

Управляющая программа лишь в нескольких местах отличается от нашего шаблона. Вы можете найти ее в файле Code\Ledflash\ledflash.c:

```
#include <pic.h>

// Включение светодиода каждую секунду.
//
// Используется прерывание от таймера TMR0,
// генерируемое каждую 1 мс.
//
// 9 апреля 2002 - последняя модификация.
// 28 марта 2002 - адаптировано для МК PIC16F627/PIC16F84.
// 23 января 2002 - разработано Майклом Предко.
//
// Используемое аппаратное обеспечение:
// микроконтроллер PIC16F84/PIC16F627,
// частота тактирования 4 МГц,
// МК PIC16F627 использует внутренний тактовый генератор,
// внешний сигнал сброса _MCLR,
// светодиод подключен к выводу RB1.
//

// Слово конфигурации:
#if defined (_16F84)
    #warning PIC16F84 selected
    _CONFIG(0x03FF1); // Для МК PIC16F84:
                       // кварцевый тактовый генератор,
                       // таймер PWRT включен,
```

```

        // сторожевой таймер выключен,
        // защита кода отключена.
#elif defined(_16F627)
    «warning PIC16F627 with internal oscillator selected
    __CONFIG(0x03F70);    // Для МК PIC16F627:
                          // внутренний тактовый генератор,
                          // RA6/RA7 используются для ввода-вывода,
                          // внешний сигнал сброса,
                          // таймер PWRT включен,
                          // сторожевой таймер выключен,
                          // защита кода отключена,
                          // детектор BODEN включен.
#else
    #error Unsupported PICmicro MCU selected
#endif

// Глобальные переменные и константы:
volatile unsigned int RTC = 0;    // Счетчик реального времени.

static bit trisLEO $ (unsigned) &TRISB*8+1;
                                // Биты управления светодиодом.
static bit      LED @ (unsigned) &PORTB*8+1;

const int LEDon  = 0;            // Включить светодиод.
const int LEDoff = 1;            // Выключить светодиод.

// Обработчик прерываний:
void interrupt tmr0_int(void)
{
    if (TOIF) {                // Обработчик прерываний от таймера TMR0.
        TOIF = 0;              // Сбросить флаг прерывания от таймера TMR0.
        RTC++;                  // Инкремент счетчика реального времени.

        // Здесь можно разместить дополнительный код
        // для реализации функций электронного и механического уровней,
        // которые надо выполнять каждую 1 мс.

        if ( ( RTC % 512 ) == 0 )
            LED = LED " 1;
            // Переключать состояние светодиода каждые 512 мс.
        }                    // Конец обработчика прерываний от таймера TMR0.

    // Здесь можно разместить другие обработчики прерываний.
} // Конец обработки прерываний.

// Служебная подпрограмма:
void enableLED(int LEDstate)    // Установить состояние вывода LED
{                                // в соответствии с LEDstate.
    LED = LEDoff;                // Сначала светодиод выключен.
    if (LEDstate)
        trisLED = 0;            // Вывод МК, управляющий состоянием
                                // светодиода, теперь работает как выходной.
    else
        trisLED = 1;            // Вывод МК, управляющий состоянием
                                // светодиода, теперь работает как входной.
}

```

```
// Главная программа:

void main(void)
{
    TMRO = 0;           // Начальный сброс таймера TMRO.
    OPTION = 0x0D1;     // Предделитель будет работать с TMRO,
                        // коэффициент деления равен 4.
    TOIE = 1;          // Разрешить прерывания от таймера TMRO.
    GIE = 1;           // Разрешить обработку всех прерываний.

    // Здесь можно разместить дополнительный код для управления
    // периферийными устройствами.

    enableLED(1);       // Вначале светодиод должен гореть.

    while (1 == 1) {    // Бесконечный цикл.

        // Здесь можно разместить код,
        // реализующий функции биологического уровня.

    }                  // Конец биологического кода.
}                    // Конец главной программы.
```

По сравнению с описанным в начале главы шаблоном программы, здесь внесены следующие изменения:

- добавлены комментарии, относящиеся к управлению состоянием светодиода;
- описаны те биты аппаратных регистров, которые используются для управления светодиодом, и объявлены константы, служащие для переключения его состояния;
- для определения момента переключения светодиода в противоположное состояние используется операция взятия остатка % от деления текущего значения счетчика реального времени на 512. Это приводит к тому, что состояние светодиода изменяется каждые 512 мс. Напомним, что записываемое в выходной порт значение защелкивается во внутреннем регистре и изменяет фактическое состояние вывода только в том случае, если этот вывод сконфигурирован для работы в качестве выходного;
- для управления состоянием вывода, к которому подключен светодиод, используется функция `enableLED`. Если надо включить светодиод, эта функция должна быть вызвана с ненулевым параметром - тогда вывод, к которому подключен светодиод, переводится в режим выходного и состояние светодиода (горит или не горит) устанавливается в зависимости от последнего записанного в порт значения. Если надо потушить светодиод, данная функция вызывается с нулевым значением параметра; тогда вывод, к которому подключен светодиод, переводится в режим входного. В этом случае любая записываемая в порт информация не может включить светодиод;
- начальное состояние светодиода задается при старте программы вызовом функции

```
enableLED (1);
```

Эта программа убедительно демонстрирует, как необходимо разделять выполнение функций электронного и биологического уровней. Нигде в главной программе, содержащей биологический код, мы не обращаемся к аппаратным регистрам

микроконтроллера. Непосредственное управление светодиодом производится в процедуре обработки прерываний.

Возможно, вам захочется упростить предложенную программу. Для этого нет необходимости изменять обработчик прерываний, а в текст главной программы надо добавить строки:

```
LED = LEDoff;           // Сначала светодиод выключен.
trisLED = 0;           // Вывод LED является выходным.
while (1 == 1) {
    if ((RTC % 512) == 0) {      // Прошло 0,5 с?
        LED = LED ^ 1;         // Да, переключить светодиод.

        while ( ( RTC % 512 ) == 0 ) ;
                                // Ждать изменения счетчика реального времени
    }
}
```

Этот код проще понять, и он несколько короче. Но представьте, что будет, если мы должны управлять многими внешними устройствами? Мы хотим иметь возможность разрабатывать и отлаживать различные интерфейсы независимо друг от друга, поэтому первоначальный вариант лучше подходит для нашей цели.

Еще один вопрос, который может возникнуть у внимательного читателя, - почему состояние светодиода изменяется каждые 512, а не 500 мс. Вы можете попробовать изменить это значение и проверить, какой длины машинный код будет сгенерирован компилятором PICC Lite. Он окажется почти в два раза длиннее! Дело в том, что при включенном режиме оптимизации компилятор PICC Lite использует эффективные алгоритмы для упрощения машинного кода.

Без оптимизации результат компиляции фрагмента программы

```
if ( ( RTC % 500 ) == 0 )
    LED = LED ^ 1;
```

выглядит примерно так:

1. Вычислить остаток от деления RTC на 500 и запомнить результат в служебной переменной temp.
2. Если temp равно нулю, то инвертировать LED.

Выполнение первого пункта требует использования библиотечной функции для реализации операции деления. Дело в том, что в системе команд микроконтроллеров PICmicro отсутствует машинная команда для выполнения этой операции, поэтому компилятор вынужден заменять ее на вызов заранее подготовленной подпрограммы, которую в таком случае необходимо добавить к машинному коду программы. Это увеличивает объем программы и время ее выполнения.

Но если делитель является степенью двойки (как, например, число 512), то можно использовать формулу

$$A \% B = A \& (B - 1) \text{ при } B = 2^n.$$

Тогда приведенный фрагмент программы на языке C может быть представлен в виде

```
if ( ( RTC & 511 ) == 0 )
    LED = LED ^ 1;
```


Другими словами, для проверки того, делится ли заданное число на 512 (или любую другую степень двойки), компилятор при включенной оптимизации может заменить длинную операцию деления и получения остатка на короткую поразрядную операцию AND с битовой маской, которая находится вычитанием единицы из константы, задающей значение делителя.

Этот вариант при компиляции генерирует более короткий код и к тому же не требует никаких дополнительных подпрограмм. Разумеется, пока рассматриваемые программы достаточно просты и нет особой необходимости экономить программную память, но разработчик может столкнуться с проблемами, если захочет добавить к данному примеру более сложные функции. Поэтому в любом случае следует выбирать самое эффективное решение.

После успешной компиляции нашего приложения вызовите на Рабочий стол интегрированной среды MPLAB окно **Stopwatch** и загрузите файл конфигурации окна просмотра P16F627.WAT, который находится в папке Code\Procwat. Создайте еще одно окно **Watch** для просмотра значения 16-разрядного счетчика реального времени RTC (при желании произведите необходимые настройки самостоятельно или же воспользуйтесь файлом Code\Ledflash\RTC.WAT). Внешний вид окна MPLAB после всех необходимых приготовлений показан на рис. 4.31.

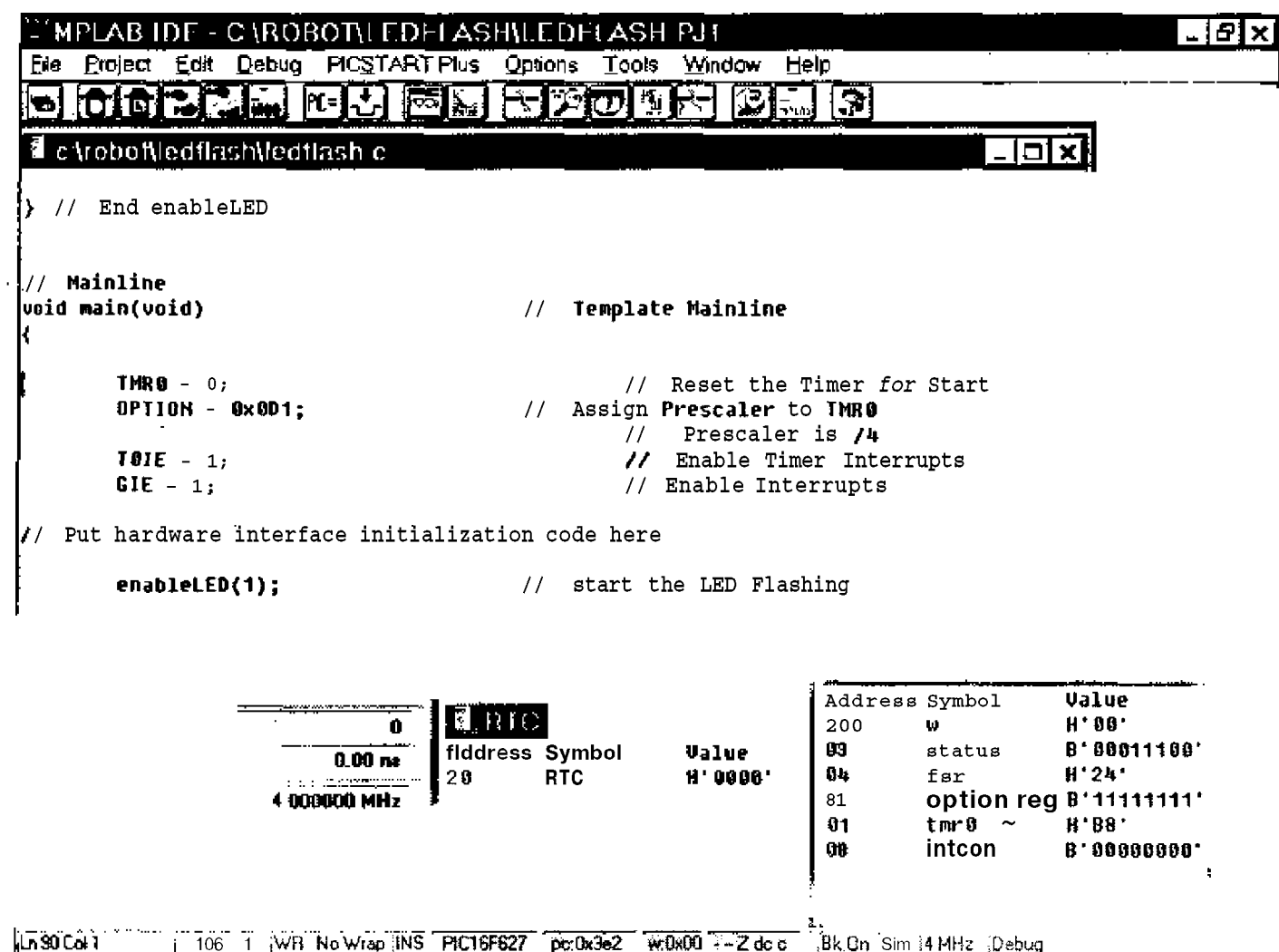


Рис. 4.31. Окно MPLAB перед началом моделирования

Теперь надо шелкнуть мышью где-нибудь внутри окна с исходным текстом программы на языке C (чтобы оно получило фокус ввода) и выполнить сброс микроконтроллера. Напомним, что для этого можно использовать команды главного меню **Debug** $\Rightarrow$ **Run** $\Rightarrow$ **Reset** (Отладка $\Rightarrow$ Старт $\Rightarrow$ Сброс) или нажать клавишу F6. В результате первая выполняемая строка нашей программы подсветится, и все будет готово для начала пошаговой отладки. Один шаг отладки выполняется с помощью команды **Debug** $\Rightarrow$ **Run** $\Rightarrow$ **Step** (Отладка $\Rightarrow$ Старт $\Rightarrow$ Шаг) или при нажатии клавиши F7. С каждым выполненным шагом подсветка перемещается на следующую строку программы. Когда выполнение дойдет до вызова функции `enableLED`, мы увидим, что подсветка перемещается в начало программы, показывая процесс выполнения всех операторов этой функции, а затем снова возвращается в основную программу.

В конце концов мы попадем в бесконечный цикл `while (1 == 1)`, и единственное, что будет меняться на экране при нажатии F6, - это значение счетчика командных циклов в окне **Stopwatch** и значение счетчика таймера в окне просмотра содержимого регистров микроконтроллера. Чтобы продолжить процесс отладки в менее скучном режиме, шелкните правой кнопкой мыши по строке

```
TOIF = 0;
```

в самом начале процедуры обработки прерываний от таймера. На экране появится контекстное меню (рис. 4.32), в котором надо выбрать команду **Break Point(s)** (Точка (точки) останова). В результате соответствующая строка будет выделена красным цветом.

Задав точку останова, можно продолжить выполнение программы в свободном (не пошаговом) режиме. Для этого воспользуйтесь командой меню **Debug** $\Rightarrow$ **Run** $\Rightarrow$ **Run** (Отладка $\Rightarrow$ Выполнить $\Rightarrow$ Выполнить) или просто нажмите клавишу F9. Программа продолжит работу в автоматическом режиме, но затем будет остановлена перед тем, как выполнить строку, подсвеченную красным цветом.

Когда я проводил отладку программы, то останов произошел во время 1087-го командного цикла, то есть на 1,09-й миллисекунде. Это чуть больше, чем можно было бы ожидать. Разница объясняется тем, что перед инициализацией контроллера прерываний и таймера проходит некоторое время. После того как я снова нажал клавишу F9, следующий останов программы пришелся на 2111-й командный цикл, что ровно на 1024 циклов больше, чем в первый раз. Это означает, что выполнение прерываний от таймера TMRO моделируется правильно¹.

Уберите точку останова, опять же шелкнув правой кнопкой мыши и выбрав команду **Break Point(s)**.

Теперь поставьте точку останова в строке, изменяющей состояние светодиода:

```
LED = LED ^ 1;
```

Нажмите клавишу F9 и ждите, пока выполнение не дойдет до контрольной точки. После этого в окне **Watch** проверьте состояние порта PORTB. Оно будет подсвечено синим цветом. Нажав F7, выполните еще один шаг; только теперь

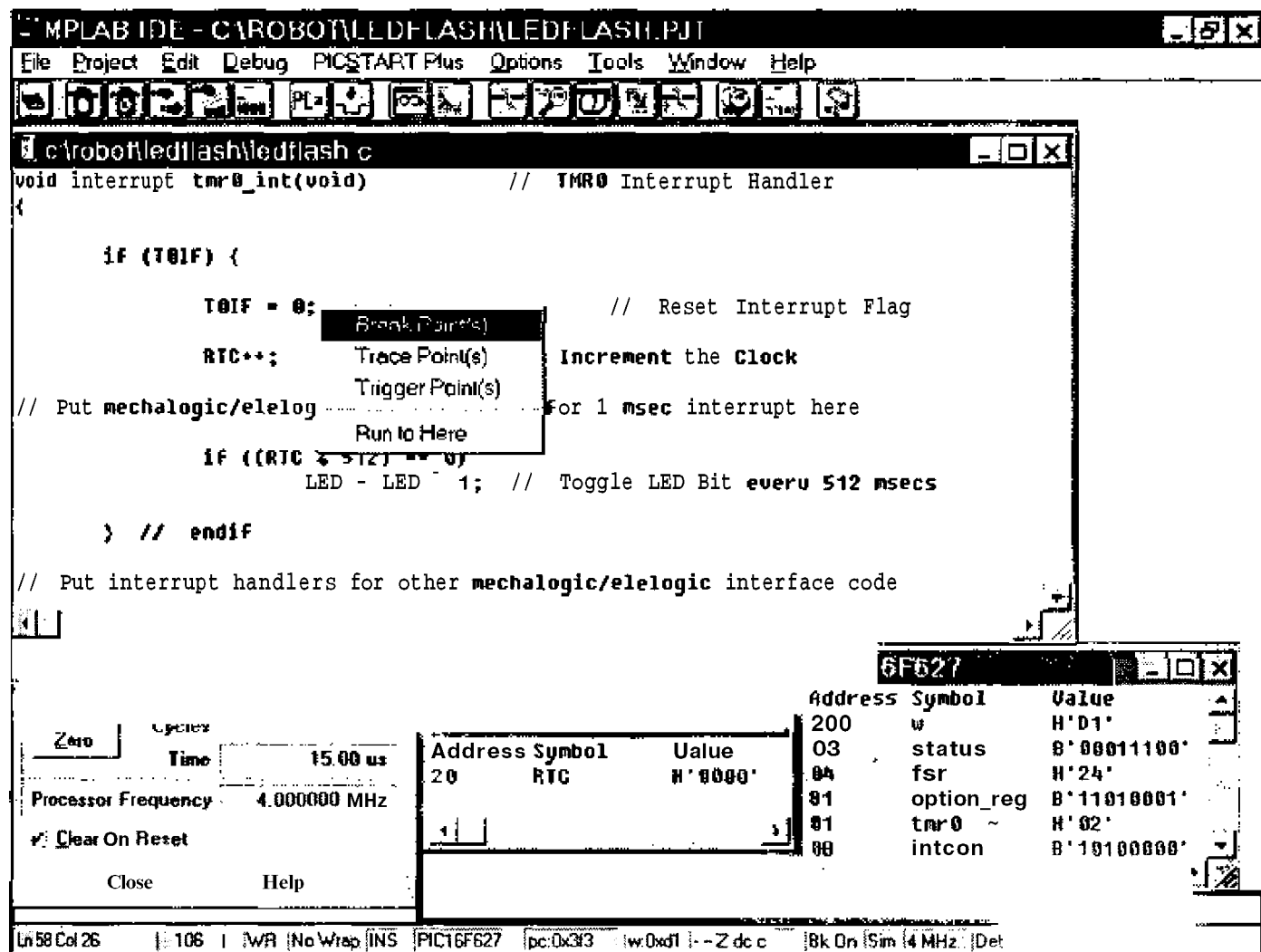


Рис. 4.32. Указание точек останова

PORTB выделится красным. Значит, в результате выполнения последнего шага значение регистра изменилось. В данном случае первый его бит (считая с нуля) изменил свое состояние с 1 на 0. Для нашего устройства это значит, что светодиод загорелся.

В моих экспериментах загорание светодиода произошло на 524368-м командном цикле. При использовании микроконтроллера PIC16F627, работающего на тактовой частоте 4 МГц, момент срабатывания таймера на самом деле может на 5-10% отличаться от заданного в программе значения. Разумеется, если индикация предназначена для человека, то такая погрешность не имеет особого значения.

Теперь вам уже известна приблизительно половина тех сведений, которые необходимо знать для отладки приложений. Мы обсудили самые необходимые вещи, а чуть позже, при рассмотрении следующих конструкций, описанных в этой главе, затронем и другие важные вопросы.

Автору повезло. Приращение счетчика командных циклов между прерываниями обычно на один цикл в ту или другую сторону отличается от точной величины 1024, поэтому в наших экспериментах оно никогда не принимало четного значения. - Прим. перев.

Убедившись в правильности программы, пора проверить работу схемы. Я советую всегда проводить моделирование программы с помощью встроенного в MPLAB симулятора, прежде чем начинать программирование настоящего микроконтроллера. Но еще перед началом проектирования программы желательно собрать саму схему. Дело в том, что в процессе сборки вы можете для удобства или по необходимости изменить некоторые внешние соединения, например подключить светодиод к другому выводу микроконтроллера. В нашем простом примере для таких изменений нет особого повода, но в сложных случаях, когда к микроконтроллеру подключается большое количество периферийных устройств, всегда лучше иметь возможность маневра.

После загрузки программы в микроконтроллер отключите его от программатора и вставьте в схему. Включите питание и убедитесь, что светодиод мигает приблизительно один раз в секунду.

Если что-то не так, вам придется внимательно проверить правильность всех соединений и наличие необходимых уровней напряжения на выводах микроконтроллера. Обратите внимание на полярность включения светодиода.

Не разбирайте устройство после тестирования, оно потребуется нам для следующего проекта.

4.12. ИСПОЛЬЗОВАНИЕ ШИРОТНО-ИМПУЛЬСНОЙ МОДУЛЯЦИИ ДЛЯ УПРАВЛЕНИЯ АНАЛОГОВЫМИ УСТРОЙСТВАМИ

Микроконтроллеры PICmicro, как и большинство других цифровых микросхем, не слишком-то приспособлены для непосредственного управления аналоговыми устройствами. Особые трудности возникают, если выходное устройство потребляет слишком большой ток. Самый простой способ решения проблемы заключается в использовании последовательности импульсов одинаковой амплитуды и частоты, но изменяющейся длительности (ширины), то есть так называемой *широотно-импульсной модуляции* (ШИМ, Pulse Width Modulation - PWM). Обычно именно этот способ применяется для управления двигателями постоянного тока или для вывода аналоговых сигналов нужной величины.

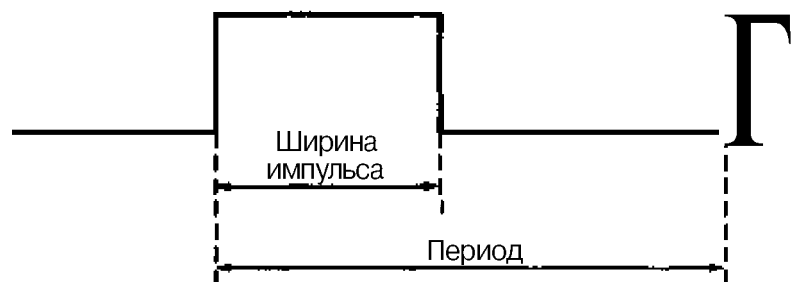


Рис. 4.33. ШИМ-сигнал

Пример сигнала, подвергнутого широтно-импульсной модуляции, представлен на рис. 4.33.

Для генерации **ШИМ-сигнала** можно использовать следующий код:

```
Period = PWMPeriod;          // Параметры выходного сигнала: длительность
On = PWMOn;                   // периода и длительность импульса.

while ( 1 == 1 ) {           // Бесконечный цикл.
    PWM = ON;                 // Подать очередной импульс.
    for ( i = 0; i < On; i++ ); // Держать высокий уровень в течение
                                // времени, заданного значением On.
    PWM = OFF;                // Закончить импульс.
    for ( ; i < PWMPeriod; i++ ); // Ждать до конца периода.
}
```

Здесь PWM - имя вывода (то есть бита выходного порта), с которого должен сниматься **ШИМ-сигнал**. Это простая программа, но ее трудно совместить с какими-либо другими задачами, которые, возможно, должен выполнять робот. Поэтому, как обычно, нам придется вынести функцию электронного уровня в процедуру обработки прерываний от таймера:

```
void interrupt tmr0_int (void)      // Обработчик прерываний.
{
    .....
    if ( PWM == ON ) {              // Если сейчас высокий уровень, то
        PWM = OFF;                 // сделать его низким
        TMRO = PWMPeriod - PWMOn;  // и установить таймер,
    } else {
        PWM = ON;                  // в противном случае сделать уровень высоким
        TMRO = PWMOn;              // и установить таймер.
    }
    .....
}                                     // Конец обработчика прерываний.
```

Здесь мы изменяем значение счетчика таймера, чтобы следующий запрос на прерывание возник в тот момент, когда надо будет переключить уровень выходного сигнала на противоположный. Выглядит все очень неплохо, но этот код никуда не годится! Мы монополизировали аппаратный ресурс (таймер TMRO), исключив для других задач возможность работы с ним.

Программа, приведенная в следующем разделе, будет построена немного по-другому, и тогда мы покажем, как лучше работать с таймером для решения нашей задачи.

Если ШИМ-сигнал используется для управления двигателем, то при выборе его частоты следует убедиться, что она лежит за пределами слышимого диапазона, то есть превышает 20 кГц, иначе при работе робот будет издавать не слишком приятные звуки (с другой стороны, следует иметь в виду, что не воспринимаемые человеком ультразвуковые колебания могут оказывать неблагоприятное воздействие на животных, например на собак).

Если выбрать очень высокую частоту, то некоторые устройства (да и сам микроконтроллер) могут оказаться не в состоянии реагировать на слишком частые переключения.

Для всех рассмотренных в книге примеров есть возможность уменьшить период срабатывания таймера TMRO, чтобы можно было генерировать более высокочастотный сигнал, но при этом могут возникнуть проблемы с другими периферийными устройствами. В итоге вы даже можете решить, что разумнее было бы разработать какую-либо специальную схему, работой которой можно было бы управлять с помощью микроконтроллера и которая позволяла бы формировать высокочастотный ШИМ-сигнал. Но лучше всего использовать для этих целей специальный модуль ШИМ, имеющийся во многих микроконтроллерах PICmicro среднего семейства.

Другое очевидное решение задачи заключается в том, чтобы работать на частотах, которые лежат ниже слышимого диапазона. Многие производители используют частоты 30-60 Гц. Однако в таком режиме допустимо управлять только двигателями малой мощности, рабочий ток которых не превышает 500 мА, иначе могут возникнуть проблемы с вибрацией, вызываемой из-за низкой частоты включения-выключения двигателей. Низкочастотные ШИМ-сигналы может генерировать даже PIC16F84, не имеющий для этого специальных аппаратных средств.

4.13. УПРАВЛЕНИЕ ЯРКОСТЬЮ СВЕТОДИОДНОГО ИНДИКАТОРА

На основе нашего предыдущего проекта в этой главе мы продемонстрируем, как микроконтроллер может управлять аналоговыми устройствами. Допустим, имеются три светодиода, и мы хотим независимо задавать яркость свечения каждого из них.

Схема устройства показана на рис. 4.34, схема размещения элементов на макетной плате - на рис. 4.35, а перечень использованных элементов приведен в табл. 4.3.

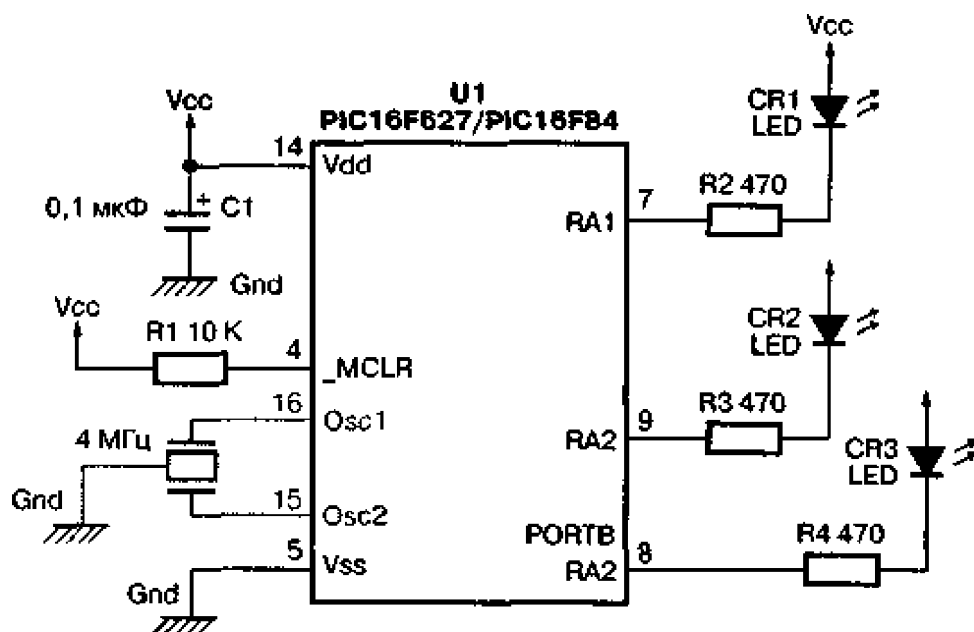


Рис. 4.34. Принципиальная схема устройства управления яркостью свечения индикаторов

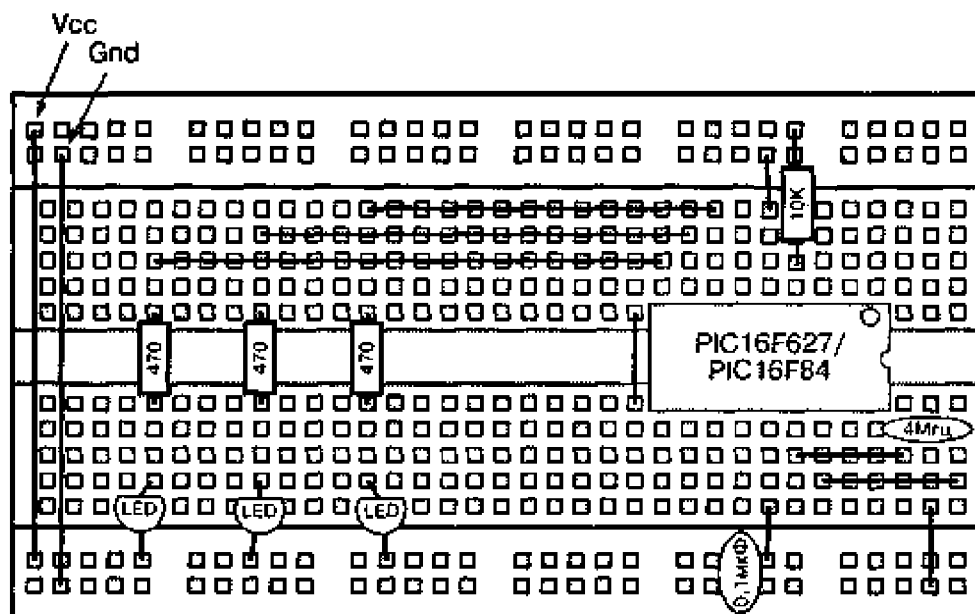


Рис. 4.35. Схема размещения элементов устройства управления яркостью свечения индикаторов

Таблица 4.3. Перечень использованных элементов

Обозначение	Элемент	Примечание
U1	PIC16F627 или PIC16F84	Микроконтроллер
CR1 - CR3	Светодиоды	Любые видимого диапазона
R1	10 кОм; 0,25 Вт	Для «подтягивания» напряжения на выводе MCLR до напряжения положительной шины питания
R2 - R4	470 Ом; 0,25 Вт	Для ограничения тока через светодиоды
C1	0,1 мкФ	Для фильтрации напряжения питания микроконтроллера
XTAL1	Керамический резонатор на 4 МГц, имеющий встроенный конденсатор	Для генератора тактовых импульсов микроконтроллера PIC16F84. Для PIC16F627 не требуется
Материалы		Макетная плата, монтажные провода, источник питания +5 В

Для подключения трех светодиодов мы отвели выводы RB1 - RB3. Надеюсь, вы еще не разобрали макет предыдущего устройства, так как наш новый проект отличается от него всего лишь двумя новыми светодиодами. Для управления яркостью их свечения мы будем использовать широтно-модулированный сигнал частотой 32 Гц. Это очень низкая частота, особенно если учесть, что микроконтроллер PIC16F627 имеет встроенные аппаратные средства для генерации сигналов достаточно высокой частоты. Но способы их применения мы рассмотрим в наших следующих проектах.

Заметим, что возможность независимой генерации сразу нескольких ШИМ-сигналов важна для управления скоростью вращения разных электродвигателей. В этом случае можно управлять направлением движения и избежать проскальзывания колес. К сожалению, микроконтроллер PIC16F627 имеет всего один модуль ШИМ и не может генерировать несколько независимых широтно-модулированных сигналов. Поэтому такую функцию приходится реализовывать программно. Зато

при этом остается возможность использовать модуль ШИМ для других целей, например для инфракрасного детектора столкновений.

Но вернемся к нашему проекту. Сначала я реализовал управление только одним светодиодом CR1, подключенным к выводу RB1 микроконтроллера. Как и в предыдущих наших программах, функция электронного уровня реализована в процедуре обработки прерываний от таймера.

Приведенную ниже программу вы найдете в электронном приложении к этой книге в файле **Code\Ledpwm\ledpwm.c**:

```
#include <pic.h>

// Управление яркостью свечения светодиодного индикатора,
// подключенного к выводу RB1, с помощью ШИМ-сигнала.
//
// Используется прерывание от таймера TMR0, генерируемое каждую 1 мс.
// Частота ШИМ-сигнала 32 Гц.
//
// 28 марта 2002 - адаптировано для PIC16F627/PIC16F84.
// 23 января 2002 - разработано Майком Предко.
//
// Используемое аппаратное обеспечение:
// микроконтроллер PIC16F84/PIC16F627,
// тактовая частота 4 МГц,
// МК PIC16F627 использует внутренний тактовый генератор,
// внешний сигнал сброса _MCLR,
// светодиод подключен к выводу RB1.

// Слово конфигурации:
#if defined (_16F84)
    #warning PIC16F84 selected
    _CONFIG(0x03FF1);    // Для МК PIC16F84:
                        // Кварцевый тактовый генератор,
                        // таймер PWRT включен,
                        // сторожевой таймер выключен,
                        // защита кода отключена.
#elif defined(_16F627)
    #warning PIC16F627 with internal oscillator selected
    __CONFIG(0x03F70);  // Для МК PIC16F627:
                        // внутренний тактовый генератор,
                        // RA6/RA7 используются для ввода-вывода,
                        // внешний сигнал сброса,
                        // таймер PWRT включен,
                        // сторожевой таймер выключен,
                        // защита кода отключена,
                        // детектор BODEN включен.
#else
    #error Unsupported PICmicro MCU selected
#endif

// Глобальные переменные и константы:
volatile int  RTC = 0;    // Счетчик реального времени.
char PWMCycle;           // Счетчик циклов (от 0 до 29).
```



```

char PWMDuty;           // Длительность импульса.
volatile int  PWMLoop;   // Счетчик периодов ШИМ-сигнала.

static bit trisLED @ (unsigned) &TRISB*8+1;
                        // Биты управления выводом RB1,
static bit LED @ (unsigned) &PORTB*8+1;

const int LEDon = 0;     // Включить светодиод.
const int LEDoff = 1;    // Выключить светодиод.

// Обработчик прерываний:
void interrupt tmr0_int(void)
{
    if (TOIF) {          // Обработчик прерываний от таймера TMR0.
        TOIF = 0;        // Сбросить флаг прерывания.

        RTC++;           // Инкремент счетчика реального времени.

        // Здесь можно разместить дополнительный код
        // для обработки прерываний от таймера.

        switch(PWMDuty) { // Проверить на крайние значения.
            case 0:        // Если задана нулевая ширина импульсов,
                LED = LEDoff; // то держать низкий уровень.
                break;
            case 29:       // Если ширина импульсов равна периоду,
                LED = LEDon;  // то держать высокий уровень.
                break;
            default:        // В противном случае:
                // если значение счетчика циклов
                // меньше заданной ширины импульса,
                if (PWMCycle <= PWMDuty)
                    LED = LEDon;
                // то держать высокий уровень,
                else // в противном случае уровень низкий.
                    LED = LEDoff;
        } // Конец оператора switch.

        if (++PWMCycle == 30) { // Конец текущего периода?
            PWMCycle = 0;       // Да, сброс счетчика циклов
            PWMLoop++;          // и инкремент счетчика периодов.
        }
    }

    // Здесь можно разместить код для обработки других прерываний.
} // Конец обработки прерываний.

// Служебная подпрограмма:
void enableLED(int LEDstate) // Установить состояние светодиода
{                             // в соответствии со значением LEDstate.

    LED = LEDoff;             // Сначала светодиод не горит.
    if (LEDstate) {
        PWMCycle = 0; // Обнулить счетчик циклов.
        PWMDuty = 29; // Сначала ширина импульсов равна периоду.
        PWMLoop = 0;  // Счетчик периодов равен 0.
    }
}

```

```

        trisLED = 0;    // Вывод управления светодиодом
                        // переведен в режим выходного.
    > else
        trisLED = 1;    // Вывод управления светодиодом
                        // переведен в режим входного,
                        // светодиод потушен.

    } // Конец служебной подпрограммы.

// Главная программа:
void main(void)
{
    OPTION = 0x0D1;      // Предделитель работает с таймером TMRO,
                        // коэффициент деления равен 4.
    TMRO = 0;            // Начальный сброс таймера TMRO.
    TOIE = 1;           // Разрешить прерывания от таймера.
    GIE = 1;             // Разрешить обработку прерываний.

    // Здесь можно разместить код для инициализации
    // других периферийных устройств.

    enableLED(1);        // Включить светодиод
                        // и выполнить инициализацию параметров.

    while (1 == 1) (      // Бесконечный цикл (биологический код).

        // Здесь можно разместить код для реализации
        // функций биологического уровня.

        // Алгоритм изменения ширины импульсов во времени:
        if (PWMLoop == 2) {          // Каждые две секунды.
            if (PWMDuty == 0)
                PWMDuty = 29;        // Максимальная ширина импульсов.
            else
                PWMDuty--;            // Уменьшить ширину импульсов.
            PWMLoop = 0;              // Сбросить счетчик периодов.
        }

    }

} // Конец главной программы.

```

В этой программе один период выходного сигнала делится на 30 подынтервалов. Длительность периода ШИМ-сигнала выбрана равной 32 Гц, поэтому длительность одного подынтервала - около 1 мс, то есть в точности равна промежутку между срабатываниями таймера TMRO.

Ширина импульсов задается значением переменной PWMDuty, оно может быть от 0 (все время низкий уровень сигнала, импульсы отсутствуют, светодиод потушен) до 29 (ширина импульсов равна периоду, все время высокий уровень управляющего сигнала, светодиод горит с максимальной яркостью).

Каждый раз при срабатывании таймера производится инкремент счетчика циклов PWMCycle, его текущее значение сравнивается с величиной PWMDuty и по результатам сравнения формируется низкий или высокий уровень управляющего сигнала. Как только счетчик циклов становится больше 29, он обнуляется, и начинается новый период.

Для изменения яркости свечения светодиодного индикатора с течением времени используется счетчик периодов `PWMLoop`. При запуске программы устанавливается максимальная яркость (`PWMDuty = 29`), затем она постепенно уменьшается (`PWMDuty--`), пока не достигнет нулевой отметки. После этого счетчик периодов обнуляется, и все повторяется с самого начала.

После выполнения моделирования работы этой программы с помощью симулятора MPLAB и загрузки ее в микроконтроллер можно проверить, что светодиод чуть уменьшает яркость свечения каждые 2 с, пока не погаснет совсем, после чего вспыхивает с максимальной яркостью. Благодаря инерционности человеческого глаза мы не замечаем мигания светодиода, которое происходит с частотой 32 Гц. При уменьшении той доли **времени**, в течение которого ток протекает через светодиод, нам кажется, что он горит менее ярко.

Испытав описанный проект в работе, я добавил к микроконтроллеру еще два светодиода и, разумеется, внес необходимые изменения в управляющую программу. Можно предложить по крайней мере четыре способа добавления новых независимых выходов для генерации **ШИМ-сигналов**, и я испытал все четыре.

Первый светодиод, подключенный к выводу **RB1** микроконтроллера, как и прежде, должен был уменьшать яркость свечения каждые 2 с. Второй светодиод, подключенный к выводу **RB2**, должен был делать то же самое, но с периодом 5 с. Наконец, третий светодиод, подключенный к выводу **RB3**, должен был увеличивать свою яркость каждые 7 с.

Первый (*экстенсивный*) способ добавления новых независимых выходов к нашему устройству заключается в том, что в процедуре обработки прерываний от таймера дописывается код, относящийся к новому ШИМ-генератору, который в точности дублирует тот, что уже имеется для реализации предыдущего. Соответствующим образом измененный текст программы для управления двумя светодиодами вы найдете в файле `Code\Ledpwm\ledpwm2.c`, а для управления тремя светодиодами - в файле `Code\Ledpwm\ledpwm3.c`.

Но это не слишком изящное решение. Гораздо привычнее в этом случае было бы использование массивов. Как известно, индексы элементов массивов в языке C нумеруются с 0, поэтому, чтобы не менять нумерацию светодиодов и избежать лишних вычислений, в программе проще указать размер массивов на 1 больше требуемого и не использовать нулевой элемент.

В этом случае текст обработчика прерываний от таймера выглядит следующим образом:

```
for (i = 1; i < 4; i++) {           // Повторять для трех светодиодов.
    switch(PWMDuty[i]) {           // Проверка на крайние значения.
        case 0:                   // Нулевая ширина импульсов.
            writeLED(i, LEDoff);
            break;
        case 29:                  // Ширина импульсов равна периоду.
            writeLED(i, LEDon);
            break; ,
        default:                  // В противном случае:
            if (PWMCycle[i] <= PWMDuty[i])
                writeLED(i, LEDon);
    }
```

```

        else
            writeLED(i, LEDoff);
    }        // Конец оператора switch.

    if (++PWMCycle[i] == 30) {    // Конец периода?
        PWMCycle[i] = 0;        // Да.
        PWMLoop[i]++;           // Инкремент счетчика периодов.
    }
}        // Конец цикла.

```

Как видим, здесь первоначальный текст лишь немного изменен, чтобы можно было работать с элементами массивов. Но еще здесь фигурирует вызов некоей функции `writeLED`. На самом деле это не функция, а макрос:

```

#define writeLED(bit, value)
    ( (PORTB) = (PORTB & ~(1<< (bit)) | (value * (1<< bit)) ) )

```

Параметрами макроса являются номер разряда регистра и флаг, который указывает, надо ли этот разряд сбросить в 0 или установить в 1. Сначала нужный бит сбрасывается с помощью поразрядной операции AND, а затем в результате выполнения поразрядной операции OR с подготовленной битовой маской принимает нужное значение.

Аналогичные макросы можно использовать и для чтения отдельных разрядов регистра:

```

#define readLED(bit) (PORTB & (1 << (bit)))

```

Описанный способ наращивания количества выходов будем называть *программным циклом*.

Если вы выполните моделирование измененной программы с помощью симулятора, то, вероятно, удивитесь тому, какого количества командных циклов потребует изменение состояния разрядов порта PORTB. Возможно, это из-за неэффективной реализации нашего макроса? Чтобы решить эту проблему, придется манипулировать битами явно:

```

void writeLED(int LEDbit, int value)    // Запись указанного бита.
{
    switch(LEDbit) {
        case 1:
            LED1 = value;
            break;
        case 2:
            LED2 = value;
            break;
        case 3:
            LED3 = value;
            break;
    }
}

```

Назовем этот способ *циклом с переключателем*. Соответствующим образом измененный код программы можно найти в файлах `Code\Ledpwm\ledpwm2b.c` и `Code\Ledpwm\ledpwm3b.c`.

Еще один способ заключается в том, чтобы не отказываться от макросов, но изменить способ доступа к отдельному биту регистра. Возможно, компилятор PICC Lite не слишком эффективно реализует операцию умножения на $(1 \ll \text{bit})$, которая была указана в нашем предыдущем макросе, поэтому можно попытаться использовать условный оператор:

```
#define writeLED(bit, value)
    if (value == 0) PORTB &= ~(1 << bit); else PORTB |= (1 << bit);
```

Здесь мы сначала проверяем значение параметра `value`, а затем либо сбрасываем, либо устанавливаем нужный бит. Полный текст программы вы найдете в файле `Code\Ledpwm\ledpwm3c.c`. Для работы с двумя светодиодами достаточно только уменьшить на единицу количество повторений цикла `for`, заменив 4 на 3.

Назовем этот способ *модифицированным программным циклом*.

В табл. 4.4 проведено сравнение всех четырех способов наращивания числа независимых ШИМ-сигналов по четырем показателям: размеру обработчика прерываний, размеру главной программы, размеру всего приложения и количеству циклов, требуемых для обработки каждого запроса на прерывание.

Таблица 4.4. Сравнительная характеристика различных способов наращивания количества программных модулей ШИМ

Имя программы	Количество светодиодов	Метод	Размер обработчика прерывания	Размер главной программы	Размер всего приложения	Количество циклов, требуемых для обработки прерываний
<code>ledpwm</code>	1	Базовый код	63	44	107	36
<code>ledpwm2</code>	2	Копирование кода	86	80	166	50
<code>ledpwm3</code>	3	Копирование кода	109	116	225	64
<code>ledpwm2a</code>	2	Программный цикл	179	140	319	151
<code>ledpwm3a</code>	3	Программный цикл	179	152	331	214
<code>ledpwm2b</code>	2	Цикл с переключателем	116	210	326	157
<code>ledpwm3b</code>	3	Цикл с переключателем	116	256	372	223
<code>ledpwm2c</code>	2	Модифицированный программный цикл	133			147
<code>ledpwm3c</code>	3	Модифицированный программный цикл	133	102	235	210

При использовании экстенсивного способа (копирование кода) размер обработчика прерываний увеличивается на 23 машинных команды при каждом добавлении одного выхода, а количество командных циклов, требуемых на обработку прерывания, увеличивается на 14. Как и следовало ожидать, размер обработчика при использовании программного цикла не зависит от количества выходов, но время его выполнения увеличивается на 63 командных цикла на каждый добавляемый выход. Цикл с переключателем также не увеличивает размер обработчика при

добавлении новых светодиодов, но требует 66 циклов на каждый новый выход. Как видим, наш первый вариант макроса не слишком-то плох: явное обращение к отдельным разрядам регистра менее эффективно. Наконец, при использовании программного цикла с модифицированным макросом размер обработчика снова не зависит от количества светодиодов, а время его выполнения увеличивается на 63 для каждого нового светодиода.

Модификация макроса почти не повлияла на время выполнения обработчика прерываний, но позволила заметно уменьшить размер кода.

На рис. 4.36 показано, как растет размер обработчика прерываний при увеличении числа программно реализованных модулей ШИМ. Как видим, простое копирование кода при не слишком большом количестве выходов эффективнее всех остальных методов. Поскольку немногие роботы имеют количество двигателей, большее **трех-четырёх**, то для управления ими можно рекомендовать наш самый первый метод. Только если количество независимых **ШИМ-сигналов** превышает шесть, копирование становится менее эффективным, чем программный цикл, с точки зрения размера обработчика прерываний.

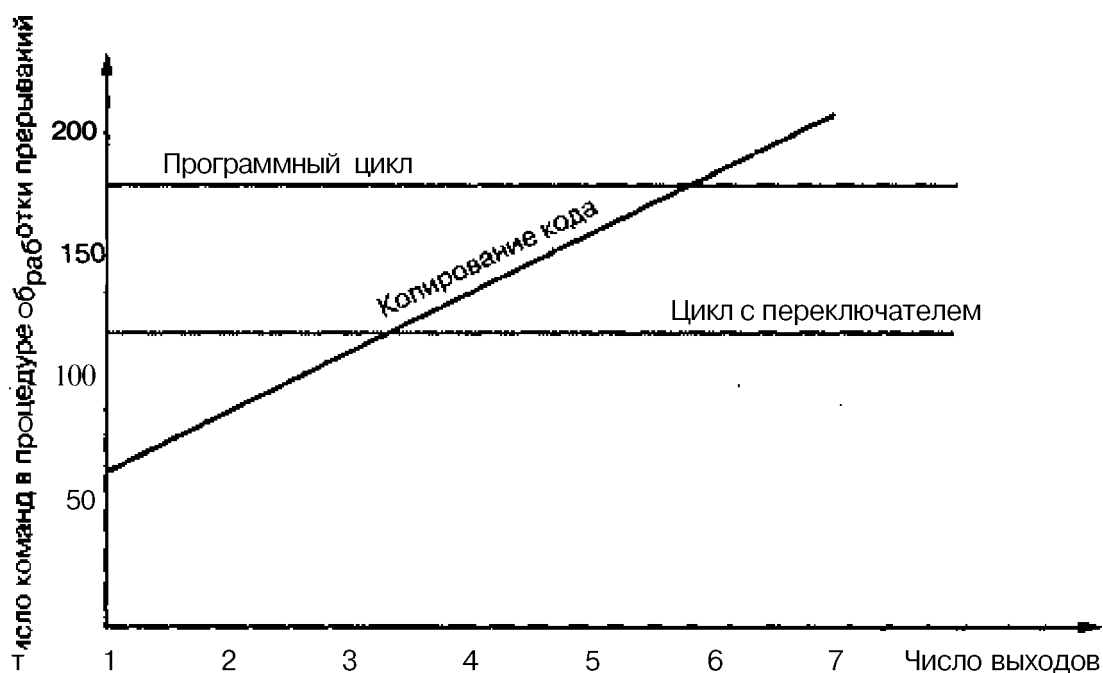


Рис. 4.36. Рост размера обработчика прерываний при увеличении количества программно реализованных модулей ШИМ

На рис. 4.37 показано, как при добавлении новых выходов растет время выполнения обработчика прерываний (в командных циклах). Как видим, здесь опять преимущество за методом копирования, так как наклон прямой в случае его применения меньше, чем для всех остальных методов.

Результаты проведенного тестирования не слишком-то изменили мое мнение: я по-прежнему предпочитаю для выполнения похожих действий использовать программные циклы. Дело в том, что при копировании кода и его исправлении (которое в этом случае сводится к приписыванию числовых индексов к именам переменных)

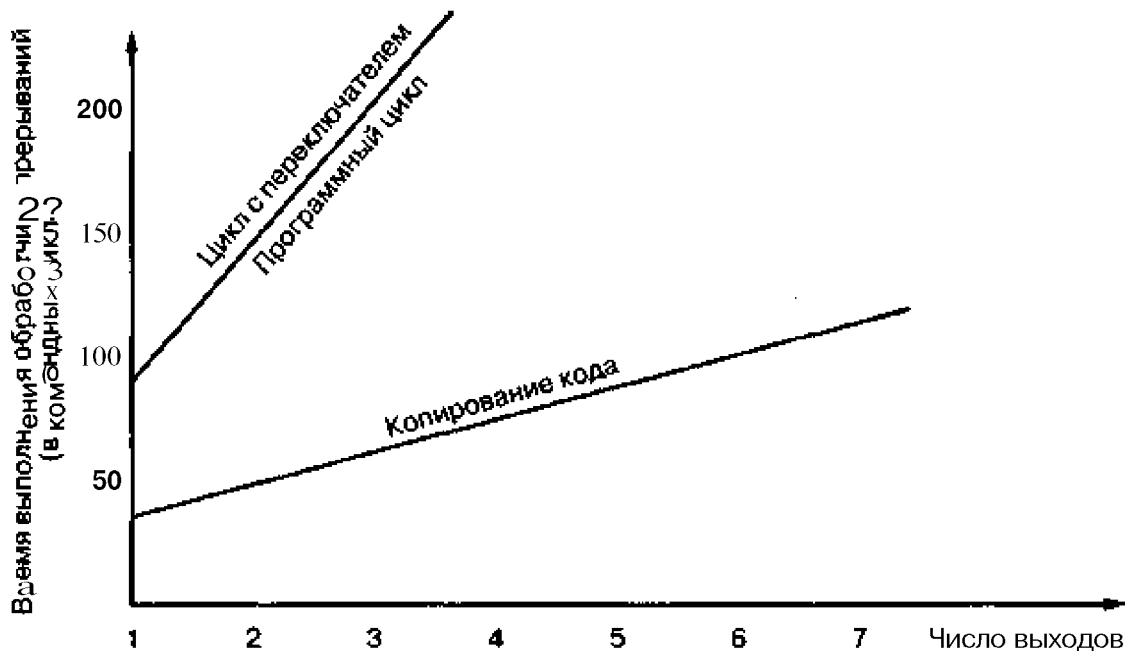


Рис. 4.37. Рост времени выполнения обработчика прерываний при увеличении количества программно реализованных мод/лей ШИМ

легко допустить ошибку, которую потом будет не так-то просто заметить и исправить. Кроме того, многократное копирование похожих участков кода проигрывает программному циклу с точки зрения простоты и легкости понимания программы.

Итак, тестирование было проведено с целью выяснить, как изменяются размер кода и скорость его выполнения при добавлении новых периферийных устройств к микроконтроллеру. Должен признаться, что результаты оказались совершенно противоположными тому, чего я ожидал, приступая к эксперименту. Но здесь рассматривалась несколько искусственная ситуация с одинаковыми периферийными устройствами. В реальной жизни они чаще оказываются разными, поэтому соответствующие фрагменты обработчика прерываний в принципе не могут быть объединены в одном программном цикле.

Итак, можно сделать вывод, что при использовании микроконтроллера PIC16F627 (или PIC16F84) и компилятора PICC Lite наиболее эффективный метод добавления новых выходных устройств - это простое размещение в обработчике прерываний от таймера соответствующих фрагментов кода, напрямую работающих с добавляемыми устройствами. Разумеется, для какого-нибудь другого микроконтроллера или другого компилятора результаты аналогичного эксперимента могут оказаться совершенно другими.

4.14. ИСПОЛЬЗОВАНИЕ ПЬЕЗОЭЛЕКТРИЧЕСКИХ ИЗЛУЧАТЕЛЕЙ И ЗВУКОВЫХ ДИНАМИКОВ

Надо сказать, что микроконтроллеры мало приспособлены к тому, чтобы управлять звуковыми выходными устройствами. Большинство МК не имеют аппаратных средств для выполнения операции умножения, не работают с числами в формате с плавающей точкой и не могут непосредственно оперировать даже целыми

числами, если их разрядность превышает 8 бит. Архитектура микроконтроллеров оптимизирована для работы с цифровой информацией в реальном времени и не рассчитана на вывод аналоговой информации.

Несмотря на это, не так уж сложно добавить к нашему устройству простой звуковой сигнализатор. Разумеется, возможности индикации в этом случае сведутся всего лишь к нескольким гудкам или щелчкам, количество которых и будет сигнализировать о текущем состоянии робота. Как уже говорилось, такое решение имеет многие преимущества по сравнению с использованием жидкокристаллического дисплея.

Схема подключения **пьезоизлучателя** (или звукового динамика) к микроконтроллеру показана на рис. 4.38. Благодаря конденсатору на динамик подается напряжение только в момент изменения сигнала на выходе микроконтроллера (постоянная составляющая отфильтровывается).

Эпюры напряжений на динамике и на выводе микроконтроллера показаны на рис. 4.39. Внешний конденсатор и внутренняя емкость и индуктивность звукового

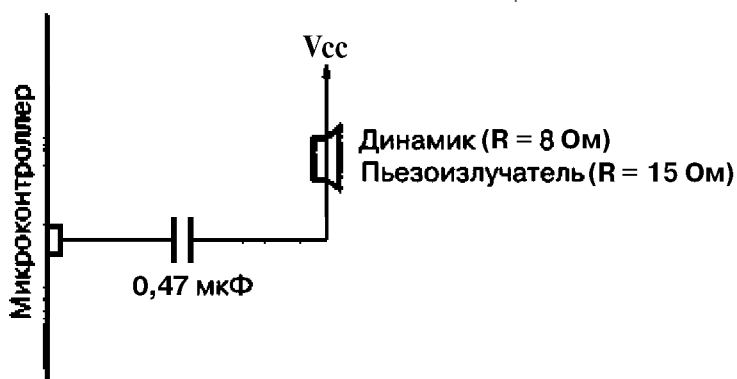


Рис. 4.38. Схема подключения звукового индикатора к **микроконтроллеру**

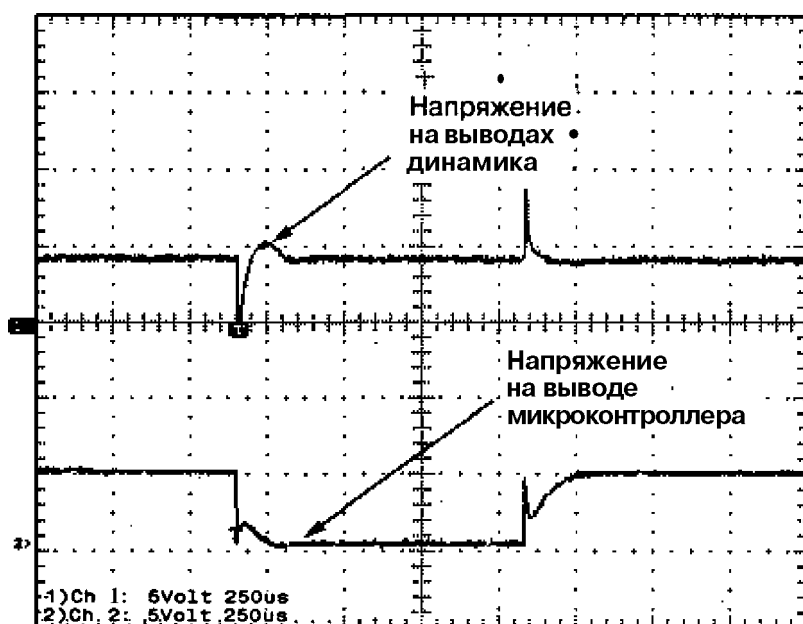


Рис. 4.39. Осциллограммы сигналов на звуковом излучателе и на выводе **микроконтроллера**

излучателя приводят к тому, что цифровой сигнал на выходе микроконтроллера искажается.

Реализовать подачу сигнала нужной частоты в выходной порт, к которому подключен звуковой индикатор, можно либо с помощью аппаратного модуля ШИМ, либо с помощью прерываний от таймера. При этом на каждый период выводимого колебания должно приходиться два вызова обработчика, чтобы сформировались два всплеска противоположной полярности.

Если прерывания от таймера генерируются каждую 1 мс, то частота сигнала составит 500 Гц - получится нечто среднее между нотами «си» и «до». Нельзя сказать, что этот звук слишком неприятен, но он заметно отличается от чистого «ля» (440 Гц), обычно используемого в электронных приборах.

Разумеется, мы могли бы изменить период срабатывания таймера TMRO, но это отразится и на работе всех остальных интерфейсов, которые мы договорились реализовывать с помощью прерываний.

4.15. УСТРОЙСТВО ЗВУКОВОЙ СИГНАЛИЗАЦИИ

Как было показано в предыдущем разделе, не слишком трудно заставить микроконтроллер генерировать звуковые сигналы. Как и раньше, для реализации функций нижних уровней используются прерывания от таймера. В этом разделе мы рассмотрим законченное приложение для звуковой сигнализации и, кроме этого, обсудим еще один важный вопрос: как в коде биологического уровня можно реализовать задержки указанной длительности.

Схема устройства мало отличается от двух предыдущих, только вместо светодиода к выводу микроконтроллера подключены конденсатор и звуковой динамик (рис. 4.40). Размещение элементов на макетной плате показано на рис. 4.41, а список используемых элементов содержится в табл. 4.5.

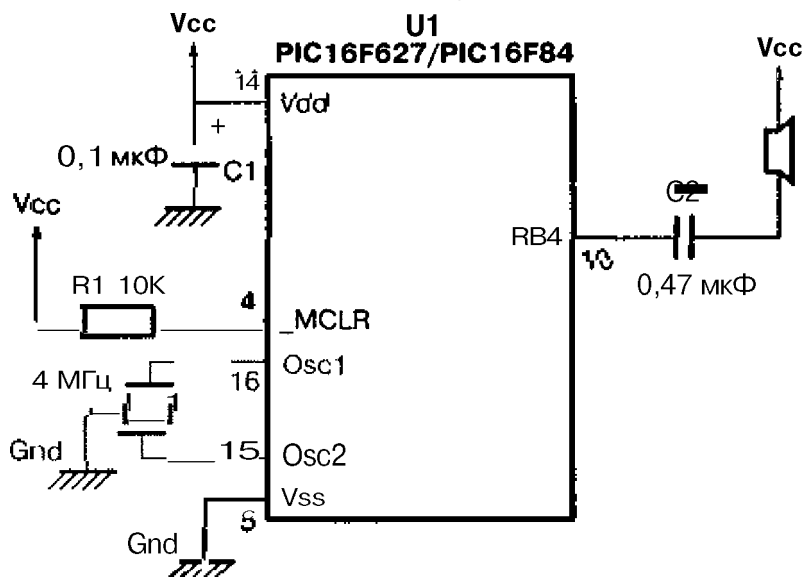


Рис. 4.40. Принципиальная схема устройства звуковой сигнализации

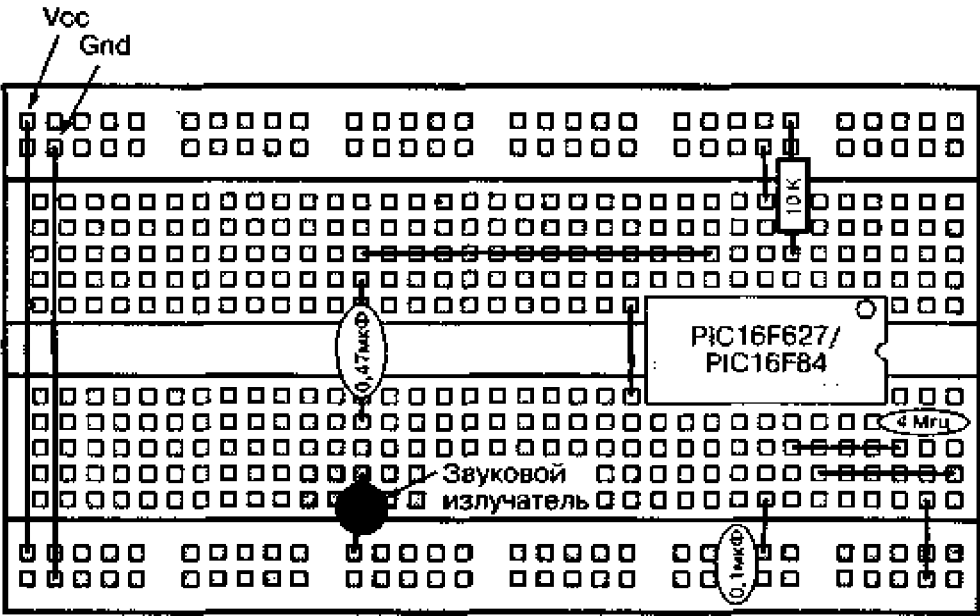


Рис. 4.41. Схема размещения элементов на макетной плате

Таблица 4.5. Список используемых элементов

Обозначение	Элемент	Примечание
UI	PIC16F627 или PIC16F84	Микроконтроллер
R1	10 кОм; 0,25 Вт	Для «подтягивания» напряжения на выводе MCLR до напряжения положительной шины питания
C1	0,7 мкФ	Для фильтрации напряжения питания микроконтроллера
C2	0,47 мкФ	Для подключения динамика; можно использовать конденсатор любого типа
SPKR	15-омный пьезоэлектрический преобразователь	
XTAL1	Керамический резонатор на 4 МГц, со встроенными конденсаторами	Для генератора тактовых импульсов микроконтроллера PIC16F84. Для PIC16F627 не требуется
Материалы		Макетная плата, монтажные провода, источник питания +5В

Исходный текст управляющей программы можно найти в файле Code\Beeper\beeper.c:

```
#include <pic.h>

// Звуковая сигнализация.
// Звуковой динамик включается на 1 с, а затем выключается на 1 с.
//
// Используются прерывания от таймера TMRO, генерируемые каждую 1 мс.
// Частота сигнала составляет около 500 Гц.
//
// Используемое аппаратное обеспечение:
//     микроконтроллер PIC16F84/PIC16F627,
//     тактовая частота 4 МГц,
//     МК PIC16F627 использует внутренний тактовый генератор,
```

```

//      внешний сигнал сброса _MCLR,
//      светодиод подключен к выводу RB4
//      через конденсатор емкостью 0,47 мкФ.

// Слово конфигурации:
#if defined (_16F84)
    #warning PIC16F84 selected
    _CONFIG(0x03FF1);    // Для МК PIC16F84:
                        // кварцевый тактовый генератор,
                        // таймер PWRT включен,
                        // сторожевой таймер выключен,
                        // защита кода отключена.
#elif defined(_16F627)
    #warning PIC16F627 with internal oscillator selected
    _CONFIG(0x03F70);    // Для МК PIC16F627:
                        // внутренний тактовый генератор,
                        // RA6/RA7 используются для ввода-вывода,
                        // внешний сигнал сброса,
                        // таймер PWRT включен,
                        // сторожевой таймер выключен,
                        // защита кода отключена,
                        // детектор BODEN включен.
«else
    #error Unsupported PICmicro MCU selected
#endif

// Глобальные переменные:
volatile unsigned int RTC = 0;           // Счетчик реального времени.
unsigned char BeeperFlag = 0;           // Флаг разрешения звучания.

static bit trisBeeper @ (unsigned) &TRISB*8+4;
                                           // Биты управления выводом RB4.
static bit Beeper @ (unsigned) &PORTB*8+4;

// Обработчик прерываний:
void interrupt tmr0_int(void)
{
    if (TOIF) {           // Обработчик прерываний от таймера.
        TOIF = 0;         // Сброс флага прерываний.
        RTC++;            // Инкремент счетчика реального времени.

        // Здесь можно разместить другой код
        // для обработки прерываний от таймера.

        if (BeeperFlag) // Переключить состояние звукового динамика.
            Beeper = 1;

    } // Конец обработчика прерываний от таймера.

    // Здесь можно разместить код
    // для обработки других прерываний.

} // Конец обработчика прерываний.

void Dlay(unsigned int msecs)    // Задержка на указанное число миллисекунд.

```

```

{
unsigned int DlayEnd;

    DlayEnd = RTC + msec + 1;

    while (DlayEnd != RTC);
} // Конец функции задержки.

void enableBeeper(void)          // Разрешить работу звукового динамика.
{
    trisBeeper = 0;              // Вывод, к которому подключен динамик,
                                // перевести в режим выходного.

} // Конец функции разрешения работы динамика.

// Главная программа:
void main(void)
{
    TMRO = 0;                    // Начальный сброс таймера.
    OPTION = 0x0D1;              // Предделитель работает с TMRO,
                                // коэффициент деления равен 4.

    TOIE = 1;                    // Разрешить прерывания от таймера.
    GIE = 1;                     // Разрешить обработку прерываний.

    // Здесь можно разместить код для инициализации
    // других периферийных устройств.

    enableBeeper();              // Разрешить работу звукового динамика.

    while (1 == 1) {              // Бесконечный цикл.

// Здесь можно разместить код для реализации
// функций биологического уровня.

        BeeperFlag = 1;          // Включить звуковой динамик
        Dlay(1000);              // на 1 секунду.

        BeeperFlag = 0;          // Выключить динамик
        Dlay(1000);              // на 1 секунду.

    }

} // Конец главной программы.

```

Как и прежде, таймер срабатывает каждую миллисекунду. Если флаг разрешения работы звукового динамика установлен, то состояние выходного порта изменится на противоположное. В результате генерируется сигнал частотой примерно 500 Гц. Так продолжается секунду, после чего флаг разрешения работы динамика сбрасывается, и в течение еще одной секунды он молчит.

В этой программе нам впервые понадобилась функция задержки. Здесь она реализована с помощью счетчика реального времени. Замечу, что к количеству миллисекунд, в течение которых длится задержка, была прибавлена 1 мс, чтобы в любом случае время ожидания не оказалось чуть меньше заданной длительности (этот вопрос уже обсуждался раньше).

Однако необходимо заметить, что такую функцию задержки внутри кода, реализующего функции верхнего (биологического) уровня, надо очень внимательно

использовать в реальных приложениях, так как во время ее выполнения не могут выполняться другие операции верхнего уровня. Мы еще вернемся к этой проблеме в главе 5.

4.16. ИСПОЛЬЗОВАНИЕ ЖИДКОКРИСТАЛЛИЧЕСКОГО ДИСПЛЕЯ

*Жидкокристаллические дисплеи*¹ часто используются для вывода информации о текущем состоянии управляющей программы и для показа данных, получаемых от входных датчиков. К сожалению, для считывания показаний на жидкокристаллическом экране часто приходится бегать вслед за роботом по всей комнате. Более удобны в этом отношении *вакуумные люминесцентные индикаторы*; многие из них по своему внешнему интерфейсу аналогичны жидкокристаллическим.

Для подключения ЖКИ к процессору (микроконтроллеру) обычно используется специальный контроллер (чаще всего - микросхема 44780 фирмы Hitachi).

Хотя некоторые радиолюбители неохотно применяют контроллеры такого типа в своих конструкциях, почему-то считая, что для них трудно найти хорошую документацию (к тому же такие контроллеры обычно не слишком дешевы), я использую их уже много лет - и вам рекомендую. Для управления ЖКИ в большинстве случаев достаточно всего трех (а то и двух) дополнительных линий; система их команд достаточно проста, а что касается высокой стоимости, то всегда можно найти старое устройство с еще работающим жидкокристаллическим дисплеем.

У большинства ЖКИ, которые рассчитаны на работу с контроллером, совместимым с микросхемой 44780, имеется 14 внешних выводов, расположенных на расстоянии 0,1 дюйма (2,5 мм) друг от друга. Назначение этих выводов описано в табл. 4.6.

Таблица 4.6. Цоколевка жидкокристаллических дисплеев

Вывод	Обозначение	Назначение
1	Gnd	«Земля» (общий)
2	Vcc	Положительное напряжение питания
3	Contrast	Регулировка контрастности
4	R/S	Команда/Выбор регистра
5	R/W	Чтение/_Запись
6	E	Тактовые импульсы
7-14	Data	Данные: D0 - 7, ..., D7 - 14

Запись информации происходит в параллельном коде по фронту тактовых импульсов E. Можно не только записывать данные в регистры ЖКИ, но и читать (их; для выбора режима доступа используется линия R/W. Типичные временные диаграммы сигналов показаны на рис. 4.42.

¹ Также распространено название *жидкокристаллические индикаторы* (ЖКИ). - Прим. перев.

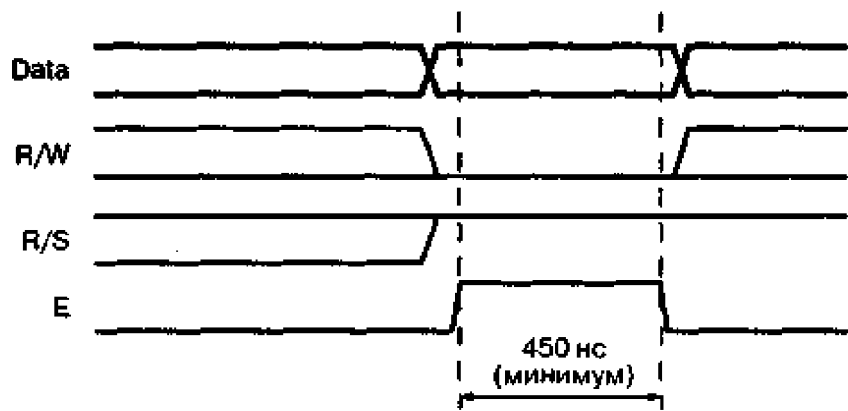


Рис. 4.42. Временные диаграммы сигналов ЖКИ

Здесь показано, как ASCII-код символа записывается в регистр ЖКИ. В коде ASCII символы кодируются одним байтом, все 8 бит которого можно записать в регистры ЖКИ одновременно или в два приема, группами по 4 бита (их называют *тетрадами*). Сначала записываются старшие 4 бита, а затем младшие. Стробирование данных осуществляется сигналом E (Enable - разрешение).

Разработчик должен решить, какой из двух режимов записи он будет использовать в своем проекте: 4-разрядный или 8-разрядный. Для реализации первого требуется всего шесть внешних выводов (используются только старшие четыре вывода данных), а для последнего - десять, зато в этом случае скорость работы выше.

Специальная управляющая линия R/S предназначена для указания типа информации, которая в данный момент подается на выводы данных. При высоком уровне сигнала R/S на выводы Data можно подавать ASCII код символа, который должен отображаться в текущей позиции курсора (если напряжение на линии R/W имеет низкий уровень), или прочесть код символа, который отображается в данный момент в этой позиции (при высоком уровне сигнала R/W). При низком уровне сигнала на линии R/S на выводы Data подается код команды (R/W = 0) или с этих выводов читается текущее состояние ЖКИ.

Команды контроллера ЖКИ 44780 (и его многочисленных аналогов) приведены в табл. 4.7.

Таблица 4.7. Команды контроллера 44780

R/S	R/W	D7	D6	D5	D4	D3	D2	D1	D0	Команда
4	5	14	13	12	11	10	9	8	7	Входы/выходы данных
0	0	0	0	0	0	0	0	0	1	Очистка дисплея
0	0	0	0	0	0	0	0	1	-	Перемещение курсора в верхний левый угол (Home)
0	0	0	0	0	0	0	1	ID	S	Указание направления перемещений курсора
0	0	0	0	0	0	1	DE	C	B	Включение/выключение дисплея/курсора
0	0	0	0	0	1	SC	RL	-	*	Смещение курсора/сдвиг изображения
0	0	0	0	1	DL	N	F	-	-	Выбор режимов работы

Таблица 4.7. Команды контроллера 44780 (окончание)

R/S	R/W	D7	D6	D5	D4	D3	D2	D1	D0	Команда
0	0	0	1	A	A	A	A	A	A	Выбор позиции курсора в области программируемых символов
0	0	1	A	A	A	A	A	A	A	Выбор позиции курсора на экране дисплея
0	1	BF	A	A	A	A	A	*	*	Прочесть флаг занятости
1	0	D	D	D	D	D	D	D	D	Записать код символа
1	1	D	D	D	D	D	D	D	D	Прочитать код символа

Примечания:

Звездочкой отмечены биты, состояние которых безразлично.

ID - если этот бит установлен, то после записи очередного символа курсор перемещается в следующую позицию.

S - если этот бит установлен, то после записи очередного символа изображение на дисплее сдвигается.

DE - **включить** (1) или **выключить** (0) экран.

C - **включить** (1) или **выключить** (0) курсор.

B - **включить** (1) или **выключить** (0) мигание курсора.

SC - **включить** (1) или **выключить** (0) режим сдвига изображения на экране.

RL - направление сдвига курсора/экрана: **вправо** (1) или **влево** (0).

DL - разрядность данных: **8-разрядные** (1) или **4-разрядные** (0).

N - число строк на экране: **одна** (0) или **две** (1).

F - **размер** символов: **5×10** (1) или **5×7** (0).

BF - этот бит устанавливается, если **ЖКИ** занят выполнением операции.

A - адрес.

D - данные.

Режим чтения кода символа, находящегося в текущей позиции курсора, полезен в тех случаях, когда включена прокрутка изображения. Флаг занятости (Busy Flag - BF) используют, чтобы узнать, завершено ли выполнение предыдущей команды. Этот флаг сбрасывается, если команда уже выполнена.

В большинстве конструкций, в которых используется ЖКИ, вывод R/W подсоединен к общей шине питания, так как обычно нет необходимости использовать режим чтения. Это упрощает схему устройства, потому что в режиме чтения выводы Data становятся выходными и интерфейсная часть схемы должна обеспечить переключение направления передачи данных. Следует учесть, что если вывод R/W заземлен, то уже нет возможности прочитать состояние флага занятости, поэтому после подачи текущей команды приходится делать задержку перед началом следующей, причем длительность этой задержки следует выбирать, исходя из максимально возможного времени выполнения команд.

На выполнение команд ЖКИ затрачивает разное время. Например, для стирания экрана и/или перемещения курсора в начальную (левую верхнюю) позицию требуется около 4,1 мс, а на многие другие команды достаточно и 160 мкс. Вообще говоря, скорость выполнения команд может заметно отличаться у различных ЖКИ, поэтому при составлении программы желательно рассчитывать на самый

медленный вариант. В противном случае при замене дисплея придется заново программировать микроконтроллер.

Что касается режимов отображения символов, то обычно используемый размер символов — 5×7 , что соответствует сброшенному биту F при подаче команды $001xxFxx$.

Перед тем как подавать команды или записывать данные, необходимо проинициализировать ЖКИ. Если используется 8-разрядный режим, то алгоритм инициализации выглядит следующим образом:

1. Ждать не менее 15 мс после включения питания.
2. Записать в ЖКИ команду $0x030$ и ждать 5 мс, пока она не выполнится.
3. Опять записать в ЖКИ команду $0x030$ и ждать 160 мкс.
4. В третий раз подать команду $0x030$ и ждать 160 мкс (или пока не сбросится флаг занятости).
5. Установить режимы:
 - подать команду выбора режимов работы и тем самым установить разрядность данных, число строк и размер символов;
 - подать команду $0x008$, чтобы выключить экран;
 - подать команду $0x001$, чтобы стереть изображение на экране;
 - установить направление перемещения курсора/экрана;
 - подать команду включения экрана и, если необходимо, курсора.

Для инициализации 4-разрядного режима необходимо:

1. Ждать не менее 15 мс после включения питания.
2. Записать в ЖКИ команду $0x03$ и ждать 5 мс, пока она не выполнится.
3. Опять записать в ЖКИ команду $0x03$ и ждать 160 мкс.
4. В третий раз подать команду $0x03$ и ждать 160 мкс (или пока не сбросится флаг занятости).
5. Установить режимы:
 - подать команду $0x02$ для установки курсора в начальную позицию;
 - подать команду $0x028$ для выбора 4-битного режима, после чего все последующие команды и данные будут передаваться в два приема;
 - установить число строк и размер символов;
 - подать команду $0x00/0x08$ (то есть команду $0x008$), чтобы выключить экран;
 - подать команду $0x00/0x01$ (то есть команду $0x001$), чтобы стереть изображение на экране;
 - установить направление смещения курсора/экрана;
 - подать команду включения экрана и, если необходимо, курсора.

Запись данных производится так же, как и запись команд, только в первом случае на линии R/S должно быть напряжение высокого уровня. Во время инициализации ЖКИ мы задали режим смещения курсора/экрана, поэтому каждый раз после записи очередного символа либо курсор, либо все изображение на

экране будет смещаться в нужную сторону - налево или направо. Обычно биты SC и RL устанавливают в 1, то есть ЖКИ работает в режиме телетайпа, когда текст пишется слева направо, а ранее написанный текст уходит за пределы экрана.

Можно производить запись символов в различные позиции жидкокристаллического дисплея. В табл. 4.8 приведены форматы различных ЖКИ, совместимых с контроллером 44780.

Таблица 4.8. Форматы наиболее распространенных жидкокристаллических дисплеев

Формат	Верхняя левая позиция	Девятый символ верхней строки	Начало второй строки	Начало третьей строки	Начало четвертой строки	Наличие встроенного контроллера
8x1	0	—	—	—	—	Нет
16x1	0	0x040	—	—	—	Нет
16x1	0	8	—	—	—	Да
8x2	0	—	0x040	—	—	Нет
10x2	0	8	0x040	—	—	Да
16x2	0	8	0x040	—	—	Да
20x2	0	8	0x040	—	—	Да
24x2	0	8	0x040	—	—	Да
30x2	0	8	0x040	—	—	Да
40x2	0	8	0x040	—	—	Да
6x4	0	8	0x040	0x020	0x060	Да
10x4	0	8	0x040	0x020	0x060	Да

Для включения курсора (он обычно реализован в виде знака подчеркивания) надо установить бит C. Что касается режима мигания символов (бит V), я не советую его использовать, так как в результате начинается мигание всей прямоугольной области, занятой символом, а это выглядит не слишком привлекательно.

Если прокрутка экрана выключена и задано смещение курсора вправо, то при записи очередного символа курсор автоматически передвигается в соседнюю правую позицию, а по достижении правого края очередной строки производится переход на начало следующей. Однако, когда будет заполнена нижняя правая позиция, надо подать команду очистки дисплея, чтобы можно было продолжить вывод информации.

При подаче команды установки позиции курсора на экране дисплея семь младших разрядов (биты A в табл. 4.7) используются для указания номера позиции, на которую должен переместиться курсор. Нужный номер можно узнать с помощью табл. 4.8. Используя семь битов, можно адресовать до 128 различных позиций, что более чем достаточно для существующих типов жидкокристаллических дисплеев.

Коды символов, отображаемые дисплеем 44780 (и совместимыми с ним), в большинстве случаев совпадают с обычным кодом ASCII. Но есть и отличия.

Например, не реализован символ «обратный слэш» (\), имеющий ASCII-код 0x05C. Что касается управляющих символов с кодами от 0x000 до 0x01F, то они не распознаются ЖКИ как управляющие, а соответствуют различным символам (часто это японские иероглифы).

ЖКИ 44780 могут отображать восемь различных символов, которые запрограммировал пользователь; им соответствуют коды от 0x000 до 0x007. Для переключения в режим программирования надо дать команду установки курсора в область программируемых символов, указав нужный адрес в памяти знакогенератора (Character Generator RAM - CGRAM), для этого предназначены биты А в поле команды (см. табл. 4.7). После этого надо последовательно выполнить запись восьми байтов, каждый из которых определяет вид одной горизонтальной линии в изображении программируемого символа (начиная с верхней).

Перемещая позицию курсора по области CGRAM, следует помнить, что одному символу в памяти знакогенератора соответствует восемь последовательно расположенных байтов. Поэтому адрес первого символа равен 0x000, второго - 0x008, третьего - 0x010 и т.д. Таким образом, вся память CGRAM имеет объем 64 байт, что соответствует восьми программируемым символам. Обычно все восемь символов программируют за один раз (поэтому используют команду 0x040, что соответствует установке курсора на нулевую ячейку памяти знакогенератора), после чего производят запись всех 64 байтов.

Для регулировки контрастности отображаемых символов надо иметь возможность изменять напряжение на выводе 3. Обычно для этого используется регулируемый делитель напряжения - потенциометр (рис. 4.43). Изменяя положение движка переменного резистора, можно задать любое напряжение в диапазоне от 0 до Vcc. Для разных ЖКИ одно и то же напряжение на выводе 3 приводит к различным результатам.

Для упрощения схемы включения жидкокристаллических индикаторов часто используют сдвиговый регистр, с помощью которого количество подключаемых выводов можно уменьшить до трех или даже двух. Пример схемы для двухпроводного интерфейса показан на рис. 4.44.

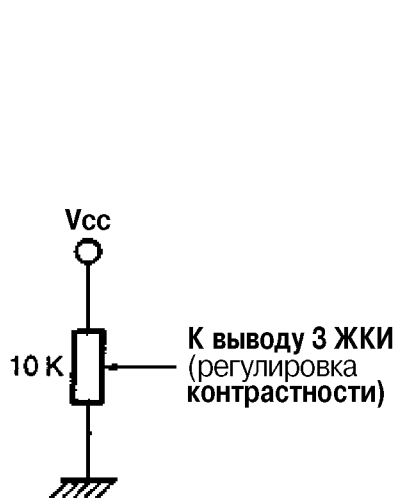


Рис. 4.43

Регулировка контрастности
отображаемых символов

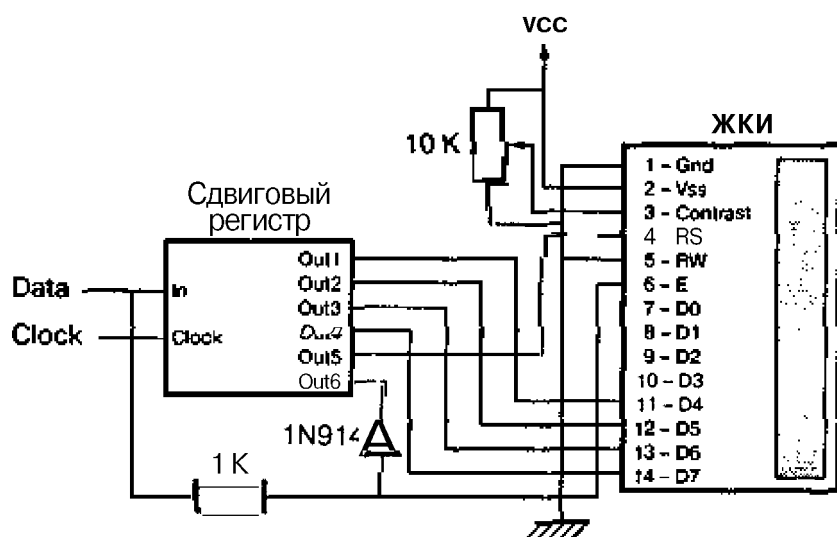


Рис. 4.44. Двухпроводной интерфейс

для подключения ЖКИ к микроконтроллеру

В качестве сдвигового регистра можно использовать микросхему **74LS174**¹, подключенный к старшему выходу регистра, и резистор сопротивлением 1 кОм, включенный между входом Data и выводом E жидкокристаллического дисплея. Эти два элемента образуют соединение «Монтажное И».

Приведенная схема подключения ЖКИ позволяет свести к минимуму количество используемых выводов микроконтроллера, освобождая их для других целей.

Я предпочитаю использовать микросхему **74LS174** вместо специализированного сдвигового регистра с последовательным входом/параллельным выходом, как раз предназначенного для применения в подобных случаях. Следует иметь в виду, что защелкивание входных данных в микросхеме **74LS174** происходит по переднему фронту тактовых импульсов.

На рис. 4.45 показаны временные диаграммы сигналов в обсуждаемой схеме.

Перед тем как начать запись данных, надо очистить регистр, установив все его выходы в нулевое состояние. Затем на вход Data подается бит 1, который обеспечит подачу строба E после заполнения регистра. После этого записываются бит R/S и, наконец, четыре бита данных (или команды), начиная со старшего. Для защелкивания в регистре очередного бита, действующего на входе Data, подается импульс на линию Clock, с передним фронтом которого бит данных записывается в младший (на рис. 4.45 - верхний) разряд регистра, а все остальные разряды сдвигаются в сторону старшего (на рис. 4.45 — вниз). После записи последнего (младшего) бита на 6-м выходе регистра устанавливается строб E; к этому времени все остальные выходные сигналы регистра примут правильные значения, которые и будут записаны в ЖКИ².

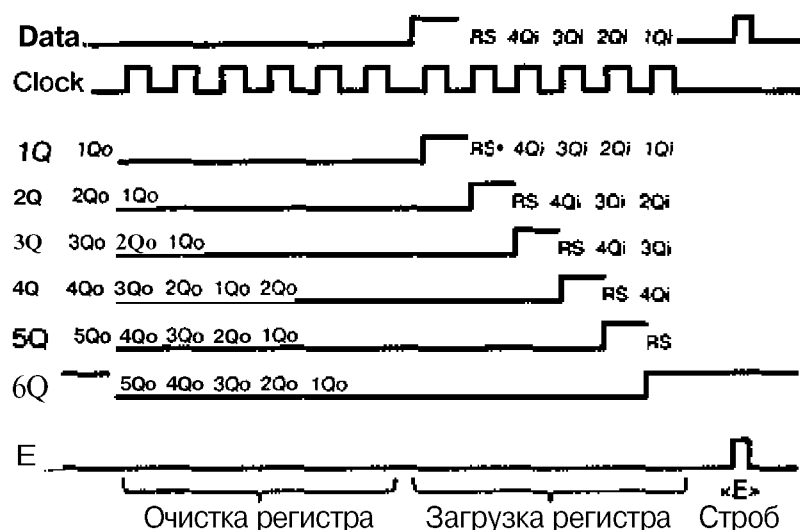


Рис. 4.45. Временные диаграммы сигналов двухпроводного интерфейса

Применяется также трехпроводная схема подключения ЖКИ к микроконтроллеру. Ее преимущество заключается в том, что нет необходимости очищать регистр побитно каждый раз перед началом записи очередного полубайта. В результате скорость записи возрастает более чем вдвое.

¹ Отечественный аналог - **K555TM9**. - Прим. перев.

² Подача строба E осуществляется автоматически после заполнения регистра, для этого предназначен диод 1N914. Можно использовать любой диод, например, КД521. - Прим. перев.

При программировании микроконтроллера следует помнить, что импульс E должен действовать не менее 450 нс. Нет необходимости запрещать обработку прерываний, пока микроконтроллер производит запись в регистр (если, разумеется, процедура обработки прерываний не изменяет состояния тех выводов микроконтроллера, к которым подключены линии Data и Clock нашей схемы). Заметим, что для подключения линий Data и Clock нельзя использовать выходы с открытым стоком.

В документации указывается, какой фронт сигнала E является активным: передний (положительный) или задний (отрицательный). Если тактирование производится задним фронтом, то в самом начале записи очередного полубайта вместо 1 на вход Data надо подавать 0.

**4.17. ВЫВОД ИНФОРМАЦИИ
НА ЖИДКОКРИСТАЛЛИЧЕСКИЙ ДИСПЛЕЙ**

Здесь мы разработаем программу, управляющую выводом информации на жидкокристаллический дисплей. Для подключения дисплея к микроконтроллеру будем использовать простой двухпроводной интерфейс, аналогичный рассмотренному в предыдущем разделе.

В нашей схеме жидкокристаллический дисплей будет единственным периферийным устройством, но для использования того же способа подключения в реальных конструкциях не придется переделывать ни схему, ни управляющую программу.

Принципиальная схема представлена на рис. 4.46, размещение элементов на макетной плате показано на рис. 4.47, а список элементов приведен в табл. 4.9.

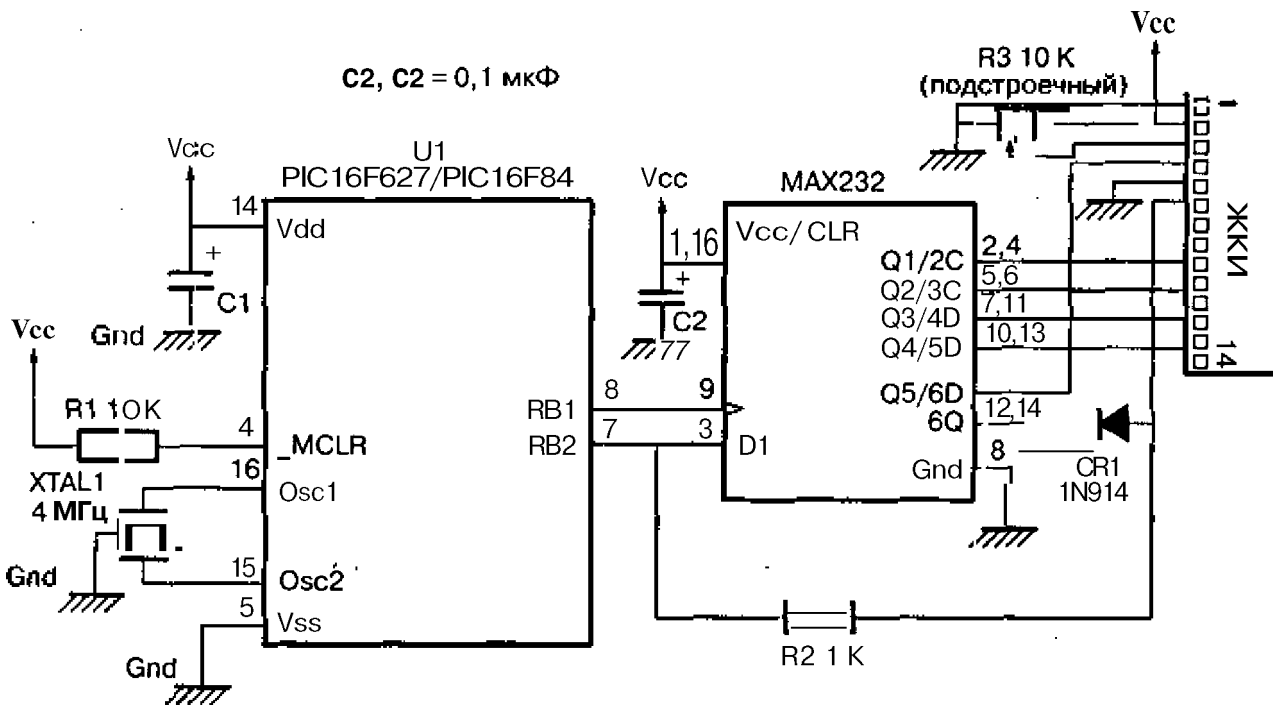


Рис. 4.46. Схема подключения ЖКИ к микроконтроллеру

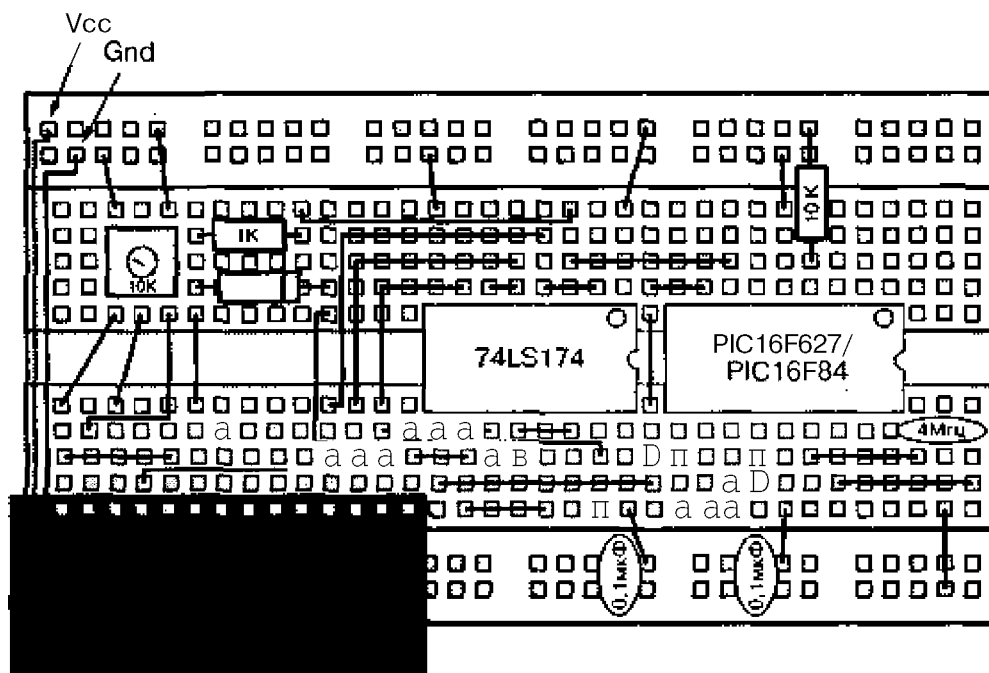


Рис. 4.47. Схема размещения элементов на макетной плате

Таблица 4.9. Список использованных элементов

Обозначение	Элемент	Примечание
U1	PIC16F627 или PIC16F84	Микроконтроллер
U2	74LS174 (K555TM9)*	Сдвиговый регистр
CR1	1N914 (KD521)*	Любой кремниевый диод
R1	10 кОм; 0,25 Вт	Для «подтягивания» напряжения на выводе <u>MCLR</u> до напряжения положительной шины питания
R2	1 кОм; 0,25 Вт	
K3	10 кОм, подстроечный	Для регулирования контрастности
C1, C2	0,1 мкФ	Для фильтрации напряжения питания микросхем
ЖКИ	Жидкокристаллический дисплей (14 выводов), совместимый с Hitachi 44780	
XTAL1	Керамический резонатор на 4 МГц, со встроенными конденсаторами	Для генератора тактовых импульсов микроконтроллера PIC16F84
Материалы		Макетная плата, монтажные провода, источник питания +5 В

* В скобках указаны отечественные аналоги, добавленные при переводе. - Прим. перев.

В качестве сдвигового регистра можно использовать микросхемы 74LS174, 74LS374, 74LS373, 74LS573, 74LS574¹.

По сравнению с вышеописанными конструкциями здесь появились новые элементы: диод и резистор, образующие соединение «Монтажное И», а также подстроечный резистор для регулировки контрастности. Кроме того, на плате при-

¹ Отечественные аналоги K555TM9, ИР23, ИР22, ИР33. - Прим. перев.

шлось разместить жидкокристаллический дисплей. Чтобы можно было сэкономить на соединительных проводах, дисплей располагается вблизи подстроечного резистора, используя те же шины Vcc и Gnd. Монтаж получился довольно плотным (по сравнению с предыдущими схемами), но все сделано для того, чтобы не затенять разводку соединений габаритными элементами.

Текст управляющей программы вы найдете в файле Code\Lcd\lcd.c:

```
#include <pic.h>

// 27 января 2002 - вывод на жидкокристаллический дисплей
// приветствия: "Hello World!".
//
// Для подключения дисплея к микроконтроллеру используется
// двухпроводной интерфейс на основе сдвигового регистра 74LS174.
//
// Используемые аппаратные средства:
// микроконтроллер PIC16F84/PIC16F627,
// тактовая частота 4 МГц,
// используется керамический резонатор,
// RB1 - линия Clock (тактирование),
// RB2 - линия Data (данные).
//

// Глобальные переменные и константы:
int RTC = 0; // Счетчик реального времени.

int Delay; // Длительность задержки.

static volatile bit Clock @ (unsigned)&PORTB*8+1;
static volatile bit ClockTRIS 9 (unsigned)&TRISB*8+1;
static volatile bit Data 9 (unsigned)&PORTB*8+2;
static volatile bit DataTRIS @ (unsigned)&TRISB*8+2;

char Message[13] = "Hello World!"; // Строка сообщения,
// выводимого на дисплей.

// Слово конфигурации:
#if defined (_16F84)
    «warning PIC16F84 selected
    __CONFIG(0x03FF1); // Для МК PIC16F84:
                        // кварцевый тактовый генератор,
                        // таймер PWRT включен,
                        // сторожевой таймер выключен,
                        // защита кода отключена.
#elif defined(_16F627)
    «warning PIC16F627 with internal oscillator selected
    __CONFIG(0x03F61); // Для МК PIC16F627:
                        // внешний тактовый генератор XT,
                        // RA6/RA7 используются для ввода-вывода,
                        // внешний сигнал сброса,
                        // таймер PWRT включен,
                        // сторожевой таймер выключен,
                        // защита кода отключена,
                        // детектор BODEN включен.
#else
    «errorUnsupported PICmicro MCU selected
#endif

// Обработчик прерывания:
```

```

void interrupt tmr0_int(void)
{
    if (TOIF) {          // Обработчик прерывания от таймера TMRO.
        TOIF = 0;        // Сбросить флаг прерывания от таймера TMRO.
        RTC++;           // Инкремент счетчика реального времени.

        // Здесь можно разместить дополнительный код для обработки
        // прерываний от таймера TMRO.

    } // Конец обработчика прерываний от таймера TMRO.

    // Здесь можно разместить дополнительные обработчики прерываний.
} // Конец обработчика прерываний.

// Служебные подпрограммы:
LCDNybble(char Nybble, char RS) {          // Запись полубайта в ЖКИ.
    unsigned int i;

    Data = 0;                             // Очистка сдвигового регистра.
    for (i = 0; i < 6; i++) {              // Повторить шесть раз.
        Clock = 1; Clock = 0;              // Строб Clock
                                           // для записи нулевых разрядов.
    }

    Data = 1;                             // Строб E.
    Clock = 1; Clock = 0;                  // Строб Clock.

    Data = RS;                             // Бит R/S.
    Clock = 1; Clock = 0;

    for (i = 0; i < 4; i++) {              // Записать 4 бита.
        if ((Nybble & 0x008) != 0)
            Data = 1;
        else
            Data = 0;
        Clock = 1; Clock = 0;              // Строб Clock.
        Nybble = Nybble << 1;              // Сдвиг перед записью
                                           // нового полубайта.
    }

    Data = 1; Data = 0;                    // Строб E.
} // Конец подпрограммы LCDNybble.

LCDByte(char Byte, char RS) {              // Запись байта в ЖКИ.
    int LBDlay;

    LCDNybble((Byte >> 4) & 0x00F, RS);    // Старшая тетрада.
    LCDNybble(Byte & 0x00F, RS);           // Младшая тетрада.

    if ((Byte < 4) && (RS == 0))            // Выясняем тип команды.
        LBDlay = RTC + 6;                  // Задержка на 5 мс.
    else
        LBDlay = RTC + 2;                  // Задержка на 1 мс.
    while(LBDlay != RTC);                  // Закончить выполнение задержки.
} // Конец подпрограммы LCDByte.

// Главная программа:

```

```

void main(void)
{
    int i;

    OPTION = 0x0D1;          // Предделитель работает с таймером TMRO,
                             // коэффициент деления равен 4.

    TMRO = 0;                // Начальный сброс таймера.
    TOIE = 1;                // Разрешить прерывания от таймера.
    GIE = 1;                 // Разрешить обработку прерываний.

    // Здесь можно выполнить инициализацию других периферийных устройств.

    Clock = 0; Data = 0;     // На обеих линиях двухпроводного интерфейса
                             // установлены сигналы низкого уровня.

    ClockTRIS = 0;           // Оба вывода, к которым подключен
    DataTRIS = 0;            // интерфейс ЖКИ, переведены
                             // в режим выходных.

    Dlay = RTC + 20;         // Шаг 1 - ждать более 15 мс
    while (Dlay != RTC);     // после включения питания.

    LCDNybble(0x003, 0);     // Шаг 2 - полубайт для инициализации
    Dlay = RTC + 6;          // и задержка на 5 мс.
    while (Dlay != RTC);

    LCDNybble(0x003, 0);     // Шаг 3 - полубайт
    Dlay = RTC + 1;          // для инициализации и задержка
    while (Dlay != RTC);     // более 160 мкс (1 мс).

    LCDNybble(0x003, 0);     // Шаг 4 - в третий раз повторяем
    Dlay = RTC + 1;          // инициализирующий полубайт
    while (Dlay != RTC);     // и задержку.

    LCDNybble(0x002, 0);     // Шаг 5 - устанавливаем
    Dlay = RTC + 1;          // курсор в начальную позицию.
    while (Dlay != RTC);     // Задержка.

    LCDByte(0x028, 0);       // Шаг 6 - устанавливаем режим
                             // (4-битные данные, одна строка,
                             // размер символов 5x7).

    LCDByte(0x008, 0);       // Шаг 7 - выключение дисплея.

    LCDByte(0x001, 0);       // Шаг 8 - очистка экрана.

    LCDByte(0x006, 0);       // Шаг 9 - режим смещения
                             // (сдвиг курсора включен,
                             // смещение изображения выключено).

    LCDByte(0x00E, 0);       // Шаг 10 - включение дисплея.

    for (i = 0; i < 12; i++) // Вывод сообщения "Hello World!".
        LCDByte(Message[i], 1);

    while (1 == 1) {         // Бесконечный цикл.

        // Здесь можно разместить код верхнего уровня.

    }

} // Конец главной программы.

```


Приведенный текст программы не должен вызвать затруднений, так как он мало отличается от наших предыдущих приложений. Как и прежде, для реализаций функций электронного уровня мы используем прерывания от таймера TMRO, генерируемые каждую миллисекунду.

Но загружать эту программу в микроконтроллер было бы преждевременно. Здесь мы допустили грубую ошибку, разместив часть кода, работающего с дисплеем, в главной программе, то есть там, где должны располагаться операторы, реализующие функции биологического уровня. В результате программа будет работоспособна только в том случае, если жидкокристаллический дисплей является единственным периферийным устройством. Обеспечить возможность выполнения этим приложением каких-либо других функций будет затруднительно.

Решить проблему можно, используя модель *конечного автомата*. Каждый раз при вызове обработчика прерываний мы будем проверять текущее состояние конечного автомата и выполнять действия, необходимые на данном этапе.

Измененный текст программы вы найдете в файле Code\Lcd\lcd2.c:

```
#include <pic.h>

// 17 февраля 2002 - программа управления жидкокристаллическим
// дисплеем, измененная в соответствии с правилами "трех уровней".
//
// Для подключения дисплея к микроконтроллеру используется
// двухпроводной интерфейс на основе сдвигового регистра 74LS174.
//
// Используемые аппаратные средства:
// микроконтроллер PIC16F84/PIC16F627,
// тактовая частота 4 МГц,
// используется керамический резонатор,
// RB1 - линия Clock (тактирование),
// RB2 - линия Data (данные).
//

// Глобальные переменные и константы:
int RTC = 0; // Счетчик реального времени.
volatile char LCDDlay = 20; // Длительность задержки.
volatile char LCDState = 1; // Номер текущего состояния
// конечного автомата.

static volatile bit Clock @ (unsigned)&PORTB*8+1;
static volatile bit ClockTRIS @ (unsigned)&TRISB*8+1;
static volatile bit Data @ (unsigned)&PORTB*8+2;
static volatile bit DataTRIS @ (unsigned)&TRISB*8+2;

char * MessageOut; // Указатель на строку выводимого сообщения.
volatile char MessageOuti = 0; // Смещение, указывающее позицию
// выводимого символа.

char Message[13] = "Hello World!"; // Текст сообщения.
char Message2[11] = "\376\3002nd Line";

// Слово конфигурации:
#if defined (_16F84)
#warning PIC16F84 selected
__CONFIG(0x03FF1); // Для МК PIC16F84:
// кварцевый тактовый генератор,
```

```

        // таймер PWRT включен,
        // сторожевой таймер выключен,
        // защита кода отключена.
#elif defined(_16F627)
    #warning PIC16F627 with internal oscillator selected
    _CONFIG(0x03F61);    // Для МК PIC16F627:
        // внешний тактовый генератор XT,
        // RA6/RA7 используются для ввода-вывода,
        // внешний сигнал сброса,
        // таймер PWRT включен,
        // сторожевой таймер выключен,
        // защита кода отключена,
        // детектор BODEN включен.
#else
    #error Unsupported PICmicro MCU selected
#endif

// Служебные подпрограммы:
LCDNybble(char Nybble, char RS) {    // Запись полубайта в ЖКИ.

    unsigned int i;

    Data = 0;                        // Очистка сдвигового регистра.
    for (i = 0; i < 6; i++) {        // Повторить шесть раз.
        Clock = 1; Clock = 0;        // Строб Clock
        // для записи нулевых разрядов.
    }

    Data = 1;                        // Строб E.
    Clock = 1; Clock = 0;            // Строб Clock.

    Data = RS;                        // Бит R/S.
    Clock = 1; Clock = 0;

    for (i = 0; i < 4; i++) {        // Записать 4 бита.
        if ((Nybble & 0x008) != 0)
            Data = 1;
        else
            Data = 0;
        Clock = 1; Clock = 0;        // Строб Clock.
        Nybble = Nybble << 1;        // Сдвиг перед записью
        // нового полубайта.
    }

    Data = 1; Data = 0;              // Строб E.
} // Конец подпрограммы LCDNybble.

LCDByte(char Byte, char RS) {        // Запись байта в ЖКИ.
    int LBDlay;

    LCDNybble((Byte >> 4) & 0x00F, RS); // Старшая тетрада.
    LCDNybble(Byte & 0x00F, RS);        // Младшая тетрада.

    // Обратите внимание, что текст подпрограммы изменился.
} // Конец подпрограммы LCDByte.

LCDInit()                            // Инициализация ЖКИ.
{

```

```

Clock = 0; Data = 0; // На обеих линиях двухпроводного интерфейса
                      // установлены сигналы низкого уровня.

ClockTRIS = 0;      // Оба вывода, к которым подключен
DataTRIS = 0;       // интерфейс ЖКИ, переведены в
                      // режим выходных.

} // Конец подпрограммы инициализации.

LCDOut(char * const LCString) // Вывод строки на ЖКИ.
{
    while (LCDState); // Ждать, когда можно будет начать запись.

    MessageOut = LCString; // Загрузка строки для вывода.
    LCDState = 100;        // Начинаем запись строки на ЖКИ.
} // Конец подпрограммы LCDOut.

// Обработчик прерываний:
void interrupt tmr0_int(void)
{
    char temp;

    if (TOIF) { // Обработчик прерываний от таймера TMRO.
        TOIF = 0; // Сбросить флаг прерываний от таймера TMRO.
        RTC++;    // Инкремент счетчика реального времени.

        // Здесь можно разместить дополнительный код
        // для обработки прерываний от таймера TMRO.

        // Конечный автомат:
        switch(LCDState) { // В зависимости от текущего состояния:
            case 1: // Начать инициализацию ЖКИ.
                if (--LCDDlay == 0)
                    LCDState++;
                break; // Ждать 20 мс.

            case 2: // Шаг 2.
                LCDNybble(0x003, 0);
                LCDDlay = 5;
                LCDState++;

            case 3: // Ждать выполнения команды.
                if (--LCDDlay == 0)
                    LCDState++;
                break;

            case 4: // Шаг 3.
                LCDNybble(0x003, 0);
                LCDState++;
                break;

            case 5: // Шаг 4.
                LCDNybble(0x003, 0);
                LCDState++;
                break;

            case 6: // Шаг 5.
                LCDNybble(0x002, 0);
                LCDState++;
                break;
        }
    }
}

```

```

case 7:
    LCDByte(0x028, 0);    // Шаг 6.
    LCDState++;
    break;
case 8:
    LCDByte(0x008, 0);    // Шаг 7.
    LCDState++;
    break;
case 9:
    LCDByte(0x001, 0);    // Шаг 8.
    LCDState++;
    LCDDlay = 5;          // Ждать 5 мс.
    break;
case 10:
    // Ждать выполнения команды.
    if (--LCDDlay == 0)
        LCDState++;
    break;
case 11:
    LCDByte(0x006, 0);    // Шаг 9.
    LCDState++;
    break;
case 12:
    LCDByte(0x00E, 0);    // Шаг 10.
    LCDState = 0;         // Все готово.
    break;
case 100:
    // Вывод сообщения MessageOut.
    switch (temp = MessageOut[MessageOuti++]) {
        case '\0':        // Конец сообщения?
            LCDState = 0;
            // ЖКИ теперь свободен.
            MessageOuti = 0;
            // Возврат индекса
            // на начало строки.
            break;
        case '\f':         // Очистка экрана.
            LCDByte(0x001, 0);
            LCDState++;
            LCDDlay = 5;
            break;
        case 254:          // Перед записью команды.
            if ((temp =
                MessageOut[MessageOuti++]) == 0)
                LCDState = 0;
            else {
                if (temp < 4) {
                    LCDState++;
                    LCDDlay = 5;
                } // endif
                LCDByte(temp, 0);
            }
            break;
        default:            // Все другие символы.
            LCDByte(temp, 1);
    } // Конец внутреннего оператора switch.
    break;
case 101:
    // Задержка.

```

```

        if (--LCDDelay == 0)
            LCDState--;
        break;
    } // Конец внешнего оператора switch.

} // Конец оператора if.

// Здесь можно разместить другие обработчики прерываний.
} // Конец обработчика прерываний.

// Главная программа:
void main(void)
{
    OPTION = 0x0D1;        // Предделитель работает с таймером TMRO,
                           // коэффициент деления равен 4.

    TMRO = 0;              // Начальный сброс таймера TMRO.
    TOIE = 1;              // Разрешение прерываний от таймера TMRO.
    GIE = 1;               // Разрешение обработки прерываний.

    LCDInit();             // Инициализация порта, к которому подключен ЖКИ.

    // Здесь можно проинициализировать другие периферийные устройства.

    LCDOut(Message);       // Передача строки для вывода на дисплей.

    LCDOut(Message2);      // Вторая строка сообщения.

    while (1 == 1) {       // Бесконечный цикл.

        // Здесь можно разместить код биологического уровня.

    }

} // Конец главной программы.

```

Как видим, текст подпрограммы LCDNybble не изменился, а из подпрограммы LCDByte исчезли все вызовы функции задержки.

Появилась новая подпрограмма LCDOut, предназначенная для вывода строки на жидкокристаллический дисплей. В этой подпрограмме после того, как конечный автомат закончил запись предыдущей строки, выполняется копирование указателя на новую строку. Заметим, что при раннем вызове функции LCDOut существует опасность зависания главной программы, поэтому читатель, возможно, захочет добавить к приведенному тексту еще одну подпрограмму для опроса текущего состояния конечного автомата, управляющего процессом записи строк на жидкокристаллический дисплей:

```

Int LCDPoll ()            // Опрос состояния конечного автомата.
{
    if (LDState)
        return 0;        // ЖКИ занят или еще не проинициализирован.
    else
        return 1;        // Можно начинать запись следующей строки.
}

```

Эта функция возвращает 1, если дисплей пока не проинициализирован или еще не закончена запись предыдущей строки. Теперь вызов LCDOut можно поместить в условный оператор, который обращается к функции LCDPoll.

Обратите внимание еще на несколько отличий от предыдущей программы. Во-первых, здесь мы используем строки в формате ASCIIZ, то есть применяем нулевой байт в качестве терминатора строки. Во-вторых, строка сообщения теперь содержит управляющие коды. Символ \f (form feed) используется для перехода на новую строку, что для нашего однострочного дисплея эквивалентно очистке экрана. Перед записью таких команд необходимо послать в контроллер ЖКИ байт с кодом 254 (то есть 0x0FE). Этот байт воспринимается большинством жидкокристаллических дисплеев как указатель того, что за ним последует не ASCII-код символа, а код команды (как вы помните, первые 32 символа, соответствующие обычным управляющим кодам, не являются таковыми для жидкокристаллических дисплеев, поэтому приходится использовать какой-то условный сигнал для подачи команд управления экраном).

В программе на языке С самый простой способ подачи таких команд - предвзрять каждый управляющий код символом \376 (как нетрудно проверить, это восьмеричная запись числа 254). Например, команда очистки экрана, эквивалентная команде 0x001, запишется в виде строки из двух символов: \376\001.

В табл. 4.10 приведены некоторые часто используемые команды.

Таблица 4.10. Строковый формат некоторых команд ЖКИ

Команда	Строка
Очистка экрана	\376\001
Перемещение курсора в начальную позицию экрана	\376\002
Выключение дисплея	\376\010
Включение дисплея, курсор скрыт	\376\014
Включение дисплея, видимый курсор	\376\016
Перемещение курсора на позицию, адрес которой — 0x040	\376\300

Допустим, мы хотим очистить экран, вывести в двух строках сообщение о том, что робот движется вперед:

```
Robot Moving
Forward
```

и спрятать курсор. Для этого в программе на языке С надо использовать строку, содержащую управляющие символы:

```
"\fRobot Moving\376\300Forward\376\014\0"
```

Возможно, вы полагаете, что метод проектирования, основанный на вынесении интерфейсных функций в процедуру обработки прерываний, эффективен только с точки зрения расширяемости программ, но требует большего объема памяти микроконтроллера. Пора развеять это заблуждение. Если вы сравните размер машинного кода, полученного в результате компиляции обоих вариантов нашей программы, то будете немало удивлены. Хотя исходный текст программы lcd2 на языке С занимает больше места и содержит дополнительные функции (в частности, добавлены новые ветви оператора выбора для обработки управляющих символов), размер

машинного кода после всех изменений получается даже чуть меньше, чем в результате компиляции программы `lscd`.

Разумеется, вы захотите использовать описанные в этой главе методы работы с периферийными устройствами при проектировании реальных приложений, в которых должны будут уживаться друг с другом разные интерфейсы. Поэтому за основу следующих проектов будет взята описанная в этом разделе конструкция, чтобы можно было продемонстрировать, как работает предложенный метод в ситуации, когда новые функциональные возможности добавляются к уже работающему устройству. Для экономии места в следующих программах будут приводиться только заголовки тех функций, которые обеспечивают вывод информации на жидкокристаллический дисплей, а пропущенные фрагменты кода — обозначаться многоточием.

4.18. ДАТЧИКИ

При слове «датчик» я всегда вспоминаю телевизионный сериал «Star Trek», в котором эти таинственные устройства могли обнаруживать вещество, энергию и, конечно, различные формы жизни.

Датчики, используемые в робототехнике, могут служить источником самой различной информации об окружающей среде, но сам алгоритм обработки этой информации определяется не датчиком, а кодом верхнего (биологического) уровня.

В следующих нескольких разделах я продемонстрирую примеры подключения различных датчиков к микроконтроллерам и приведу текст соответствующих программ. Существует множество различных способов реализации описанных функций; в других книгах этой серии многие вопросы разобраны более подробно. Здесь же я ставил себе задачу познакомить читателя с датчиками различного типа и рассмотреть основные методы программирования. Многие входные устройства — например, устройства ввода навигационной информации от спутника (Global Positioning System - GPS) или от обычного компаса, приборы измерения наклона, видеокамеры и др. — при этом вынужденно остались за пределами нашего внимания.

Несмотря на огромное количество самых разнообразных устройств ввода информации, общий принцип работы с ними остается неизменным. В любом случае вы должны опираться на рассматриваемый в этой книге метод «трех уровней».

Для упрощения отладки устройств разумно разместить поблизости от каждого входного датчика какое-либо устройство индикации, которое бы подключалось к нему непосредственно или через специальный переключатель и обеспечивало бы обратную связь, предоставляя разработчику необходимую информацию о состоянии датчика. Вы сэкономите много часов, если поверите, что входные сигналы часто не соответствуют тому, что вы о них думаете.

Разумеется, при этом не следует забывать, что в результате работы индикаторов может проявиться другая — паразитная — обратная связь, в результате действия которой входные датчики будут воспринимать информацию не столько от

окружающей среды, сколько от выходных устройств, которые сигнализируют об их собственном состоянии. Кроме того, следует добиваться того, чтобы ваше присутствие как можно меньше влияло на поведение робота. Во многих случаях ваше устройство должно реагировать только на окружающую обстановку и игнорировать любые формы жизни вблизи себя.

4.19. МЕХАНИЧЕСКИЕ ДАТЧИКИ

Подвижные автоматические устройства должны иметь возможность определять присутствие объектов на своем пути и вокруг себя, например, для того, чтобы выбрать маршрут движения или избежать столкновения. Самый простой способ реализации этой возможности - использование механических контактов. Вообще говоря, я считаю, что робот с контактными усиками выглядит несколько неуклюже, да и в большинстве случаев трудно обеспечить достаточную надежность их работы, но иногда без них просто не обойтись.

Принцип работы контактных детекторов поясняется на рис. 4.48. Длинный отрезок тонкой гибкой проволоки при прикосновении к какому-либо предмету приводит в действие микропереключатель. Длина усика и, соответственно, расстояние, на котором срабатывает такой датчик, должна быть достаточной для того, чтобы робот успел изменить направление движения или остановиться. Кроме того, в данной конструкции проволока играет роль рычага, позволяющего даже малому усилию, приложенному со стороны длинного его конца, вызвать переключение механических контактов.

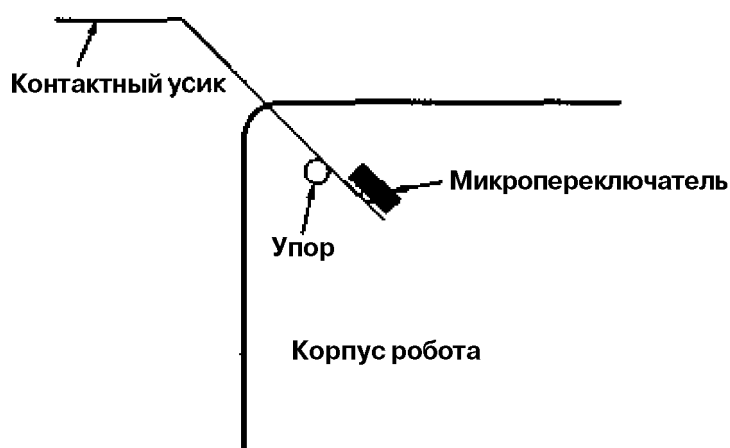


Рис. 4.48. Контактный детектор столкновений

Подобные детекторы обычно устанавливаются спереди, но в некоторых случаях они не мешают по бокам и сзади, чтобы предотвратить возможные столкновения с предметами, когда робот осуществляет маневр, резко изменяя направление движения или разворачиваясь.

Следует иметь в виду, что при срабатывании механического переключателя его контакты в течение нескольких первых миллисекунд входят в соприкосновение друг с другом сотни раз, генерируя шумовой сигнал, показанный на рис. 4.49. Это

явление называется *дребезгом контактов*. Поэтому приходится применять специальные методы подавления дребезга. В противном случае может возникнуть ситуация, когда робот, правильно прореагировав на первый импульс, будет вынужден повторить реакцию и на следующие паразитные импульсы, которые на самом деле не являются результатом повторного срабатывания датчика.

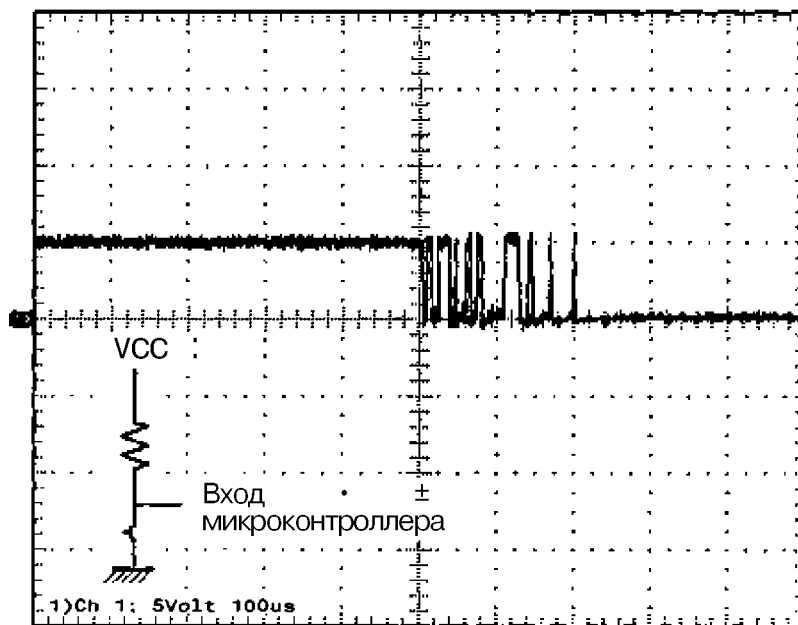


Рис. 4.49. Так выглядит дребезг механических **контактов** на экране осциллографа

Обычно при подавлении эффекта дребезга механических контактов считают, что их состояние не может измениться за время около 20 мс, то есть все изменения, происходящие за этот промежуток времени после первого срабатывания, не учитывают. Вносимая в результате работы этого алгоритма задержка обычно вполне допустима. Можно предложить для подавления дребезга следующий фрагмент кода:

```
while (1=1) {           // Бесконечный цикл
    ...                // Выполнение операций до срабатывания переключателя.
    if (Button == Press){ // Ждем, чтобы замкнувшиеся контакты остались
                        // в том же состоянии в течение 20 мс.
        Debounce = 0;
        while ( ( Debounce != 20msecs ) && ( Button == Press ) )
            for (Debounce=0; (Button==Press) && (I<20msecs); Debounce++);

        // Здесь надо разместить код
        // для реагирования на замыкание контактов.

        Debounce = 0;      // Ждем, чтобы разомкнувшиеся контакты остались
                        // в том же состоянии в течение 20 мс.
        while ( ( Debounce != 20msecs ) && ( Button == Release ) )
            for (Debounce=0; (Button==Press) && (I<20msecs); Debounce++);
    } // Конец оператора if.
```

```

...      // Выполнение операций после срабатывания переключателя.
}      // Конец оператора while.

```

Если вы сделаете контактный детектор не слишком удачно, то может оказаться, что он срабатывает при движении робота. При использовании микропереключателя от ложных срабатываний может защитить **небольшая** пружина, удерживающая контакты в разомкнутом состоянии. Если вместо микропереключателя ваша схема представляет собой пару механических контактов (один из которых подсоединен к общей шине питания, а второй - к «подтягивающему» резистору, подключенному к положительной шине питания), то использование описанной выше процедуры подавления дребезга приведет к тому, что ваш робот не будет реагировать на любые кратковременные удары. Кроме того, вам может не понравиться, что противодребезговая процедура после выполнения фрагмента кода, который производит необходимые действия после замыкания контактов, ждет, пока они снова разомкнутся.

Чтобы избежать всех этих проблем, желательно использовать логический флажок, устанавливая его на период времени около 20 мс, в течение которых мы игнорируем все изменения состояния контактов, а для отсчета времени использовать таймер микроконтроллера. Тогда противодребезговый фрагмент кода в главной программе будет выглядеть следующим образом:

```

while (1 == 1) {                                     // Бесконечный цикл.
...      // Выполнение операций до срабатывания переключателя.
if (ButtonPressFlag) {                             // Ждать замыкания контактов.
    // Здесь надо разместить код
    // для реагирования на замыкание контактов.

} // Конец оператора if.
// Выполнение операций после срабатывания переключателя.
} // Конец оператора while.

```

Соответствующая процедура обработки прерываний от таймера TMR0:

```

void interrupt tmr0_int (void)
{
    if (TOIF) {                                     // Если запрос был от таймера TMR0.
        TOIF = 0;                                  // Сбросить флаг прерывания.
        RTC++;                                     // Инкремент счетчика реального времени.
        ...                                       // Здесь можно разместить код для реализации
                                                // других действий, которые необходимо
                                                // выполнять каждую миллисекунду.
        if (Button == Press){ // Если контакты замкнулись.
            if (ButtonReleaseCounter != 0) { // В первый раз?
                ButtonPressCounter = 0;
                ButtonReleaseFlag = 0;
            } else
                if (++ButtonPressCounter >= 20) {
                    ButtonPressFlag = 1;
                    ButtonPressCounter = 19;
                }
        }
    }
}

```

```

        > else ;
    } else {
        // Если контакты разомкнулись.
        if (ButtonPressCounter != 0) { // В первый раз?
            ButtonReleaseCounter = 0;
            ButtonPressFlag = 0;
        } else
            if (++ButtonReleaseCounter >= 20) {
                ButtonReleaseFlag = 1;
                ButtonReleaseCounter = 19;
            } else;
    }
}

> // Конец обработчика прерываний.

```

Здесь при замыканий контактов флаг `ButtonReleaseFlag` сбрасывается и начинается отсчет времени (каждую миллисекунду выполняется инкремент счетчика `ButtonPressCounter`), который продолжается, пока контакты не будут разомкнуты. При очередном вызове обработчика, если контакты все еще находятся в замкнутом состоянии, а флаг `ButtonReleaseFlag` сброшен, устанавливается флаг `ButtonPressFlag`; это значение используется в главной программе и сигнализирует о том, что в данный момент выполняется противодребезговая задержка.

После размыкания контактов флаг `ButtonReleaseFlag` сбрасывается в нулевое состояние и снова делается задержка на 20 мс, только теперь с каждой миллисекундой выполняется инкремент счетчика `ButtonReleaseCounter`.

Следует помнить, что контактные усики, которые вы используете в своих конструкциях, должны быть металлическими, и их обязательно следует заземлять. Дело в том, что при движении робота в результате трения различных материалов друг о друга образуется статический заряд, который в непроводящем материале (в случае использования, например, пластмассовых усиков) накапливается до тех пор, пока при очередном касании какого-либо объекта не вызовет электрический пробой, а это может привести к повреждению электронных схем.

Наряду с микропереключателями в качестве детекторов касания можно использовать и другие приспособления:

- заземленное металлическое кольцо, которое при столкновении робота с каким-либо объектом деформируется и переключает уровень сигнала на одном из расположенных по периметру робота металлических контактов, соединенных через «подтягивающий» резистор с положительной шиной питания. Такой способ хорош, когда требуется определить, в каком именно месте произошло касание робота о внешний объект. Но если кольцо имеет достаточную массу, то из-за инерции при резкой остановке или в начале движения оно может вызывать ложные срабатывания детекторов;
- резиновая трубка вокруг робота, которая позволяет при касании изменить внутреннее давление, действующее на контакты. В такой конструкции трудно обеспечить включение датчика, зато трубка работает в качестве амортизатора и заметно снижает чувствительность к различным вибрациям и толчкам, возникающим при движении робота;

- датчик останова двигателя также может быть использован в качестве детектора столкновений. Если робот при своем движении столкнулся с каким-либо объектом, то он останавливается, что приводит к торможению двигателя. Однако при этом через обмотку двигателя течет значительный ток, создавая риск повреждения робота (из-за сильного столкновения) или электронной схемы, которая управляет скоростью вращения двигателя. К тому же данный метод снижает срок службы батарей питания.

Следует заметить, что при использовании контактных датчиков разработчик должен потратить много времени на проведение кропотливых экспериментов, прежде чем ему удастся подобрать правильную длину и форму контактного усика. В моих многочисленных экспериментах даже после того, как конструкция датчика, наконец, приближалась к некоторому идеалу, все рушилось, когда робот на полной скорости сталкивался с каким-либо предметом или кто-нибудь из детей хватал робота за проволоочный усик.

Когда есть такая возможность, я предпочитаю использовать неконтактные способы обнаружения объектов, например обсуждаемые ниже инфракрасный детектор или ультразвуковой измеритель расстояния. Тем не менее многие разработчики рассматривают контактный метод как самый надежный: ведь при использовании любого бесконтактного датчика всегда существует опасность не заметить какой-либо объект, который не отражает сигнал, излучаемый роботом.

4.20. ПОДАВЛЕНИЕ ДРЕБЕЗГА КОНТАКТОВ

Одним из первых соображений, которые побудили меня предложить использование прерываний от таймера для реализации функций электронного и механического уровней, была как раз мысль о том, что такой подход позволит без излишних хлопот и траты процессорного времени подавлять влияние дребезга механических переключателей, часто применяемых в различных автоматических устройствах в качестве источников входных сигналов. Если в нашем распоряжении имеется процедура, которая выполняется каждую миллисекунду, то реализация необходимой задержки, требуемой для подавления дребезга, существенно упрощается. В качестве примера приведем фрагмент обработчика прерываний от таймера:

```
if (!Pin)           // Если на выводе микроконтроллера
                    // сигнал низкого уровня,
                    // соответствующий замкнутым контактам,
    if (ButtonPress < 20)
        ButtonPress++;
    else;           // то подавляем дребезг при замыкании контактов.

else               // Если сигнал высокого уровня
                    // (то есть контакты разомкнуты),
    if (ButtonPress != 0)
        ButtonPress--;
    else;           // то подавляем дребезг при размыкании контактов.
```

Здесь используется условный оператор, который при замыкании контактов начинает отсчет времени, выполняя инкремент счетчика **ButtonPress**, пока его значение не достигнет 20 (это означает, что контакты оставались замкнутыми в течение 20 мс), после чего значение счетчика больше не изменяется. Но как только контакты разомкнутся (уровень сигнала на выводе микроконтроллера станет высоким), начинается обратный отсчет - каждую миллисекунду значение счетчика уменьшается на 1, пока не достигнет 0.

В этом случае код верхнего уровня будет выглядеть следующим образом:

```
if (ButtonPress == 0)
{
    // Выполняем действия, соответствующие разомкнутым контактам.
}
else
    if (ButtonPress == 20)
    {
        // Выполняем действия, соответствующие замкнутым контактам.
    }
    else
    {
        // Еще длится противодребезговая задержка.
    }
```

Оба приведенных фрагмента должны повториться в программе по одному разу для каждого имеющегося в конструкции механического переключателя.

Чтобы проиллюстрировать этот способ подавления дребезга в реальном приложении, я собрал схему, показанную на рис. 4.50.

За основу взят наш предыдущий проект с жидкокристаллическим дисплеем. Единственное, что добавлено к прошлой схеме, - резистор, светодиод и кнопка.

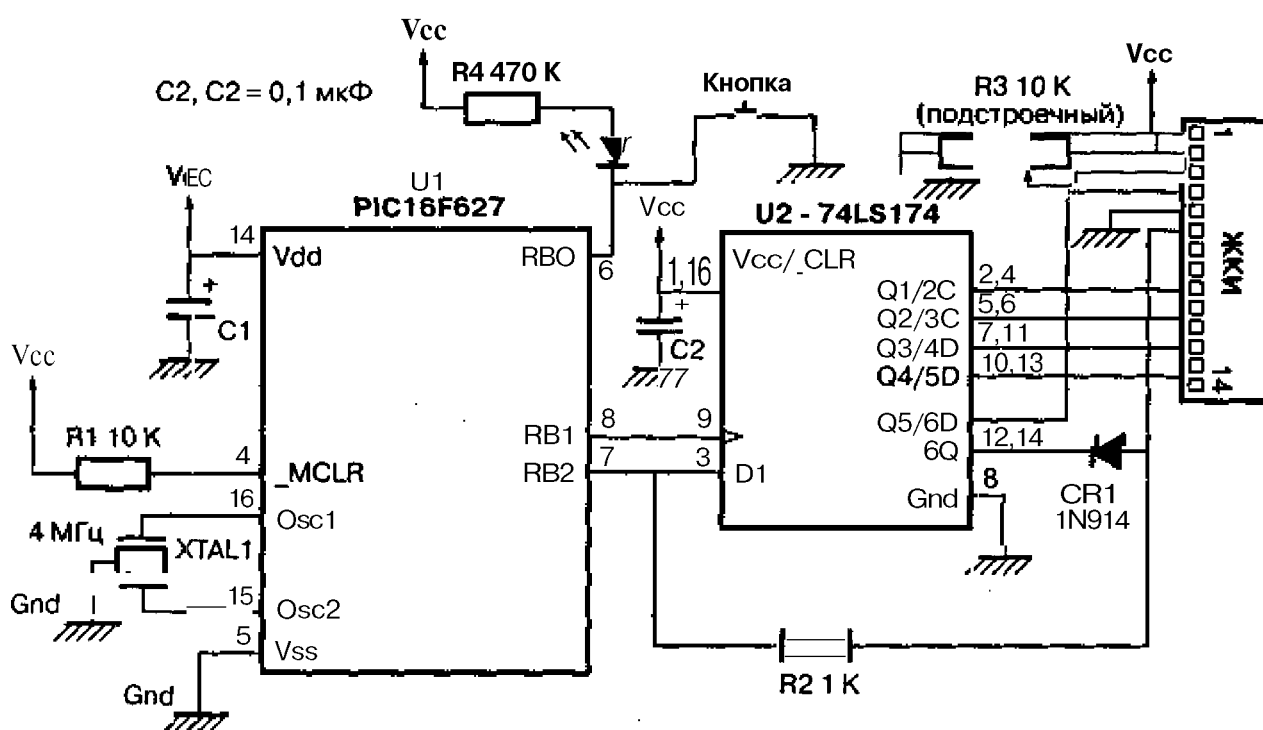


Рис. 4.50. Схема с механическими переключателями

Разумеется, ни дисплей, ни светодиод не нужны для работы противодребезговой процедуры, но я хотел показать, как различные интерфейсные модули работают в одном приложении.

Мы будем использовать дисплей для индикации состояния кнопки. Когда кнопка отжата, на выводе RBO микроконтроллера благодаря «подтягивающему» резистору действует сигнал высокого уровня. В этом случае дисплей будет выключен. При нажатии кнопки начинает работать противодребезговая процедура, и на дисплее высвечивается надпись "DEBOUNCING". Спустя 20 мс, если кнопка все еще нажата, появляется надпись "PRESSED". В главной программе опрашивается значение счетчика ButtonPressed, и в зависимости от этого выбирается одно из трех возможных действий.

Когда ваше устройство заработает, то вы увидите, что надпись "DEBOUNCING" появляется на краткий промежуток времени после нажатия и после отпускания кнопки.

Схема размещения элементов на макетной плате показана на рис. 4.51, который практически совпадает с рис. 4.47.

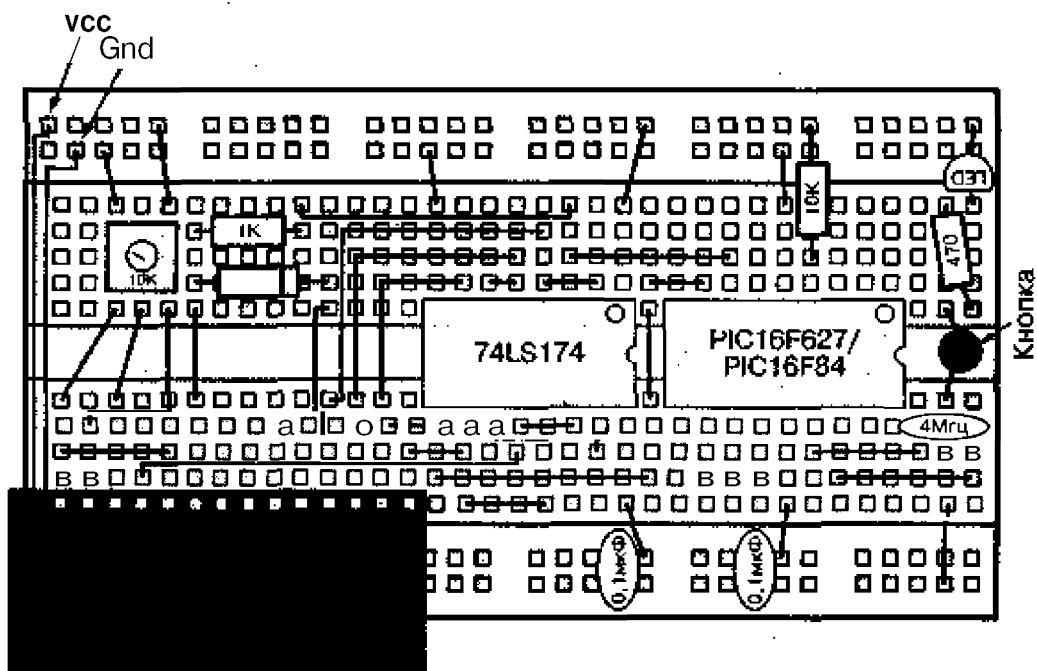


Рис. 4.51. Схема размещения элементов

Желательно взять самый дешевый микропереключатель, чтобы проверить работу противодребезговой процедуры в наиболее неблагоприятных условиях.

Список используемых элементов приведен в табл. 4.11.

Таблица 4.11. Список используемых элементов

Обозначение	Элемент	Примечание
U1	PIC16F627 или PIC16F84	Микроконтроллер
U2	74LS174 (K555TM9)*	Сдвиговый регистр
CR1	1N914 (KД521)*	Любой кремниевый диод

Таблица 4. / /. Список используемых элементов (окончание)

Обозначение	Элемент	Примечание
CR2	Светодиод	Любого типа
R1	10 кОм; 0,25 Вт	Для «подтягивания» напряжения на выводе <u>MCLR</u> до напряжения положительной шины питания
R2	1 кОм; 0,25 Вт	
R3	10 кОм, подстроечный	Для регулирования контрастности
R4	470 Ом; 0,25 Вт	
C1, C2	0,1 мкФ	Для фильтрации напряжения питания микросхем
	Микропереключатель	С нормально разомкнутыми контактами
ЖКИ	Жидкокристаллический дисплей (14 выводов), совместимый с Hitachi 44780	
XTAL1	Керамический резонатор на 4 МГц, со встроенными конденсаторами	Для генератора тактовых импульсов микроконтроллера
	Материалы	Макетная плата, монтажные провода, источник питания +5 В

** В скобках указаны отечественные аналоги, добавленные при переводе. -Прим. перев.*

Приведенный ниже текст программы вы найдете в файле Code\Whisker\whisker.c:

```
#include <pic.h>

// 19 апреля 2002 - подавление дребезга контактов.
//
// Для индикации состояния кнопки используются светодиод
// и жидкокристаллический дисплей, подключенный к микроконтроллеру
// через двухпроводной интерфейс на основе сдвигового регистра 74LS174.
//
// Используются прерывания от таймера TMR0.
//
// Замечания по аппаратным средствам:
// Микроконтроллер PIC16F627/PIC16F84 работает на частоте 4 МГц.
// Используется внешний тактовый генератор.
//
// Подключение выводов микроконтроллера:
// RBO - кнопка (с внешним "подтягивающим" резистором
// (внутренние "подтягивающие" резисторы порта PORTB
// микроконтроллера не используются);
// RB1 - линия Clock интерфейса жидкокристаллического дисплея;
// RB2 - линия Data интерфейса жидкокристаллического дисплея;
//
// Глобальные переменные и константы:
int RTC = 0; // Счетчик реального времени.
volatile char LCDDlay = 20; // Длительность задержки.
volatile char LCDState = 1; // Номер текущего состояния
// конечного автомата.
```

```

static volatile bit Clock      @ (unsigned)&PORTB*8+1;
static volatile bit ClockTRIS  @ (unsigned)&TRISB*8+1;
static volatile bit Data       @ (unsigned)&PORTB*8+2;
static volatile bit DataTRIS   @ (unsigned)&TRISB*8+2;
char * MessageOut; // Указатель строки выводимого сообщения.
volatile char MessageOuti = 0; // Смещение, указывающее позицию
                                // выводимого символа.

char Message1[2] = "\f";      // Очистка дисплея.
char Message2[12] = "\fdebouncing";
char Message3[9] = "\fPRESSED";

volatile char ButtonPress = 0; // Счетчик
                                // для противодребезговой процедуры.

// Слово конфигурации:
#if defined (_16F84)
    #warning PIC16F84 selected
    _CONFIG(0x03FF1); // Для МК PIC16F84:
                        // кварцевый тактовый генератор,
                        // таймер PWRT включен,
                        // сторожевой таймер выключен,
                        // защита кода отключена.

#elif defined(_16F627)
    #warning PIC16F627 with external oscillator selected
    __CONFIG(0x03F61); // Для МК PIC16F627:
                        // внешний тактовый генератор XT,
                        // RA6/RA7 используются для ввода-вывода,
                        // внешний сигнал сброса,
                        // таймер PWRT включен,
                        // сторожевой таймер выключен,
                        // защита кода отключена,
                        // детектор BODEN включен.

#else
    #error Unsupported PICmicro MCU selected
#endif

// Служебные подпрограммы:
LCDNybble(char Nybble, char RS) // Запись полубайта в ЖКИ.

...

LCDByte(char Byte, char RS)     // Запись байта в ЖКИ.

...

LCDInit()                      // Инициализация ЖКИ.

...

LCDOut(char * const LCDString) // Вывод строки на ЖКИ.

...

// Обработчик прерываний:
void interrupt tmr0_int(void)
{
    char temp;

```



```

if (TOIF) { // Обработчик прерываний от таймера TMRO.
    TOIF = 0; // Сбросить флаг прерываний от таймера TMRO.
    RTC++; // Инкремент счетчика реального времени.

    // Здесь можно разместить дополнительный код
    // для обработки прерываний от таймера TMRO.

    // Начало противодребезговой процедуры:
    if (!RBO) // Кнопка нажата.
        if (ButtonPress < 20)
            ButtonPress++;
        else;
    else // Кнопка отжата.
        if (ButtonPress != 0)
            ButtonPress--;
        else;

    // Конечный автомат для ЖКИ;
    switch(LCDState) { // В зависимости от текущего состояния:
        case 1: // Начать инициализацию ЖКИ.
            if (--LCDDlay == 0)
                LCDState++;
            break; // Ждать 20 мс.

        case 2: // Шаг 2.
            LCDNybble(0x003, 0);
            LCDDlay = 5;
            LCDState++;

        case 3: // Ждать выполнения команды.
            if (--LCDDlay == 0)
                LCDState++;
            break;

        case 4:
            LCDNybble(0x003, 0); // Шаг 3.
            LCDState++;
            break;

        case 5:
            LCDNybble(0x003, 0); // Шаг 4.
            LCDState++;
            break;

        case 6:
            LCDNybble(0x002, 0); // Шаг 5.
            LCDState++;
            break;

        case 7:
            LCDByte(0x028, 0); // Шаг 6.
            LCDState++;
            break;

        case 8:
            LCDByte(0x008, a); // Шаг 7.
            LCDState++;
            break;

        case 9:
            LCDByte(0x001, 0); // Шаг 8.
            LCDState++;
            LCDDlay = 5; // Ждать 5 мс.
    }
}

```

```

        break;
    case 10:          // Ждать выполнения команды.
        if (--LCDDlay == 0)
            LCDState++;
        break;
    case 11:
        LCDByte(0x006, 0);    // Шаг 9.
        LCDState++;
        break;
    case 12:
        LCDByte(0x00E, 0);    // Шаг 10.
        LCDState = 0;        // Все готово.
        break;

    case 100:         // Вывод сообщения.
        switch (temp = MessageOut[MessageOuti++]) {
            case "\0":     // Конец сообщения?
                LCDState = 0;
                MessageOuti = 0;
                break;
            case "\f":     // Очистка дисплея.
                LCDByte(0x001, 0);
                LCDState++;
                LCDDlay = 5;
                break;
            case 254:      // Команда?
                LCDState = 0;
            else {
                if (temp < 4) {
                    LCDState++;
                    LCDDlay = 5;
                }
                LCDByte(temp, 0);
            }
            break;
            default:       // Обычные символы.
                LCDByte(temp, 1);
        } // Конец внутреннего оператора switch.
        break;
    case 101:         // Message Delay
        if (--LCDDlay == 0)
            LCDState--;
        break;

    } // Конец внешнего оператора switch.
} // Конец обработчика прерываний от таймера.

// Здесь можно разместить другие обработчики прерываний.
} // Конец обработчика прерываний.

// Главная программа:
void main(void)
Ч

```

```

OPTION = 0x0D1;          // Предделитель работает с таймером TMR0,
                          // коэффициент деления равен 4.

TMR0 = 0;                // Начальный сброс таймера TMR0.
TOIE = 1;                // Разрешение прерываний от таймера TMR0.
GIE = 1;                 // Разрешение обработки прерываний.

LCDInit();              // Инициализация порта, к которому подключен ЖКИ.

// Здесь можно проинициализировать другие периферийные устройства.

while (1 == 1) {        // Бесконечный цикл.

    // Здесь можно разместить код биологического уровня.

    switch(ButtonPress) { // Каково состояние кнопки?

        case 0:          // Кнопка отжата.
            LCDOut(Message1);
            break;

        case 20:         // Кнопка нажата.
            LCDOut(Message3);
            break;

        default:         // Длится противодребезговая задержка.
            LCDOut(Message2);
            break;

    }

}

} // Конец главной программы.

```

Чтобы убедиться в правильности работы этой программы, желательно выполнить ее моделирование с помощью симулятора MPLAB и только потом подключать микроконтроллер к программатору.

Для имитации нажатий кнопки можно использовать асинхронные стимулы (порядок работы с ними уже обсуждался в главе 3).

После успешной компиляции проекта выполните команду **Debug ⇒ Asynchronous Stimulus** (Отладка ⇒ Асинхронные входные воздействия) - рис. 4.52. В появившемся диалоговом окне (рис. 4.53) надо щелкнуть правой кнопкой мыши по изображению любой кнопки, а затем указать режим работы **Toggle** (Переключатель) и имя вывода **RB0**, как показано на рис. 4.54.

Теперь состояние вывода **RB0** будет переключаться на противоположное каждый раз, когда во время отладки в этом диалоговом окне вы будете щелкать правой кнопкой мыши по изображению выбранной кнопки.

Как уже говорилось в главе 3, асинхронные стимулы очень удобны для моделирования внешних воздействий, которые могут произойти в любой момент. Но во многих случаях, когда требуется подавать входные сигналы с заданными задержками во времени, лучше прибегнуть к другим способам моделирования, имеющимся в MPLAB. Мы обязательно используем их в следующих проектах.

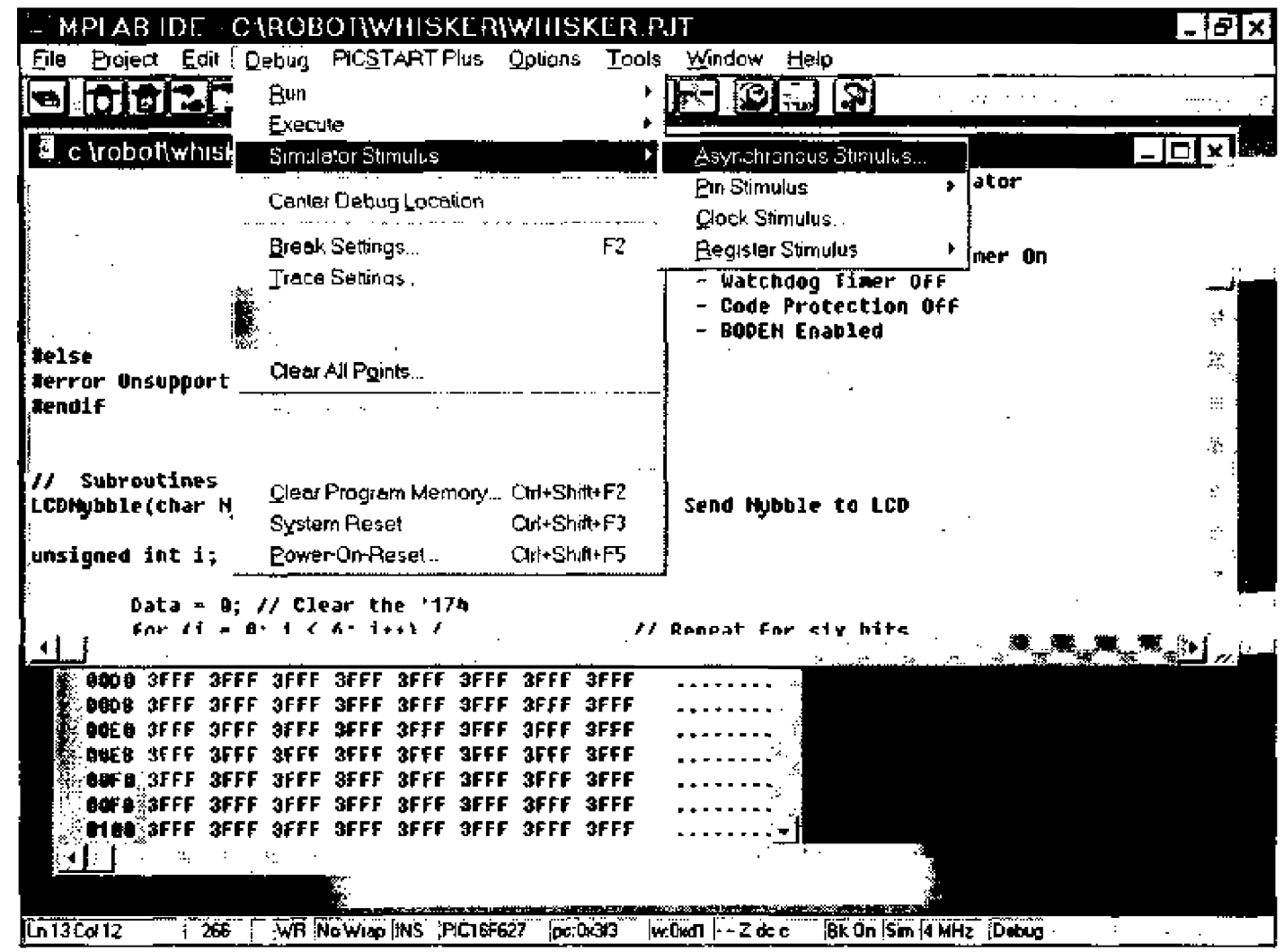


Рис. 4.52. Выбор асинхронных входных воздействий

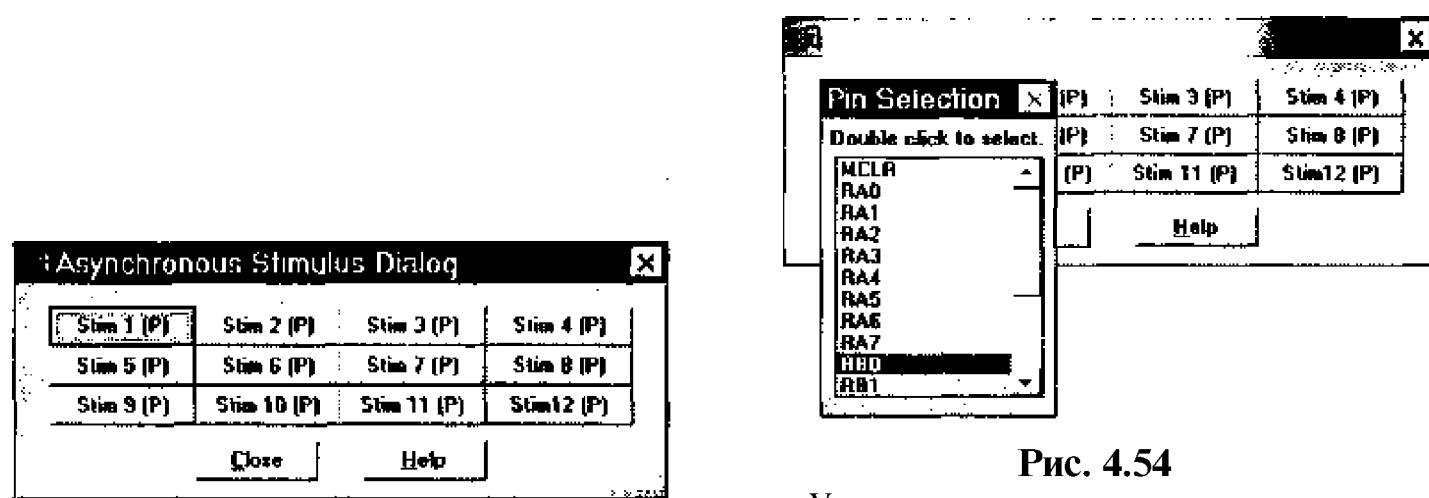


Рис. 4.53. Диалоговое окно выбора асинхронных входных воздействий

Рис. 4.54. Указание вывода микроконтроллера, на который будет подаваться входное воздействие

4.21. ИНФРАКРАСНЫЙ ДЕТЕКТОР СТОЛКНОВЕНИЙ

Разумеется, у вас дома найдется хотя бы один инфракрасный пульт дистанционного управления телевизором или каким-нибудь другим электронным прибором.

Когда я впервые приобрел такой пульт лет 20 назад, то испытал его работу в различных условиях, чтобы выяснить, с какого расстояния он начинает действовать, проходит ли его сигнал через стену и другие преграды, такие как картины, стекла, растения и т.д.

Кому-то такое поведение могло бы показаться странным, но в результате подобных испытаний родилась идея использовать инфракрасный излучатель и приемник отраженного сигнала в качестве детектора столкновений.

Принцип работы инфракрасного детектора поясняется на рис. 4.55. На корпусе робота устанавливаются инфракрасный излучатель и приемник. Если какой-либо объект, который отражает инфракрасные лучи, оказывается слишком близко от робота, то отраженный сигнал становится достаточно сильным. Он улавливается приемником и служит сигналом о скором столкновении. В качестве источника инфракрасных лучей обычно используется светодиод инфракрасногодиапазона.

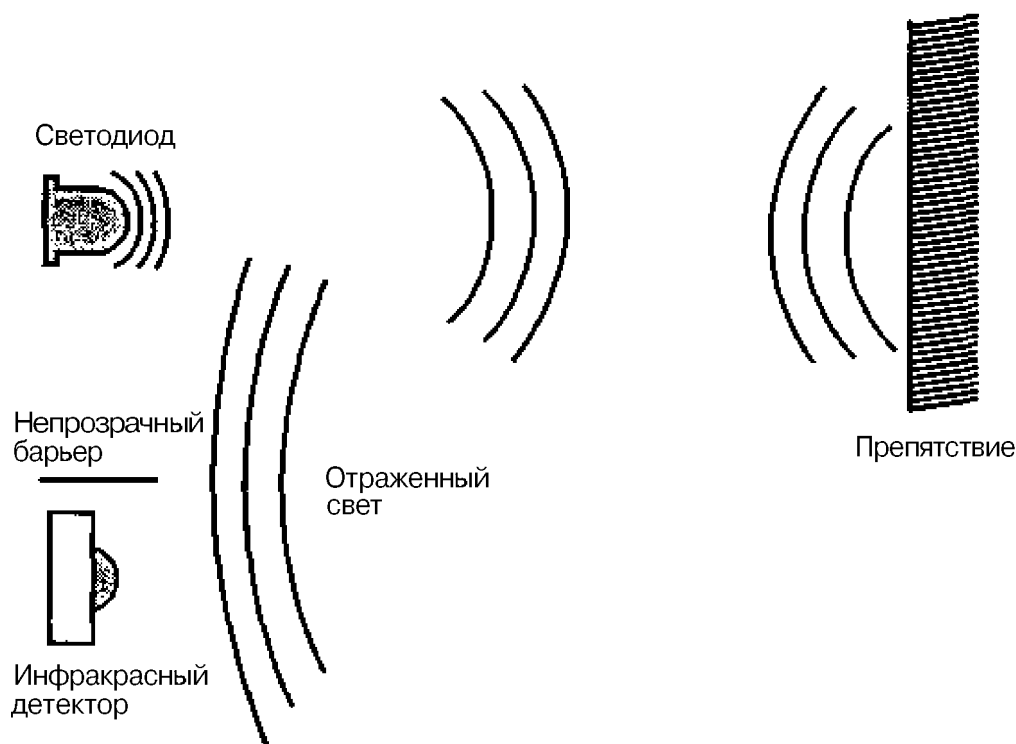


Рис. 4.55. Принцип работы инфракрасного детектора

Следует помнить, что существует огромное количество твердых и непрозрачных для обычного света материалов, которые могут оказаться прозрачными для инфракрасных лучей¹. Прекрасный пример материала, являющегося преградой для инфракрасного излучения, - это металлическая пластина, которую необходимо установить между излучателем и приемником, чтобы последний мог реагировать только на отраженный сигнал. Также можно использовать черную изоляционную ленту или резиновую трубку.

¹ Читатель может, например, убедиться, что используемый для монтажа электронных схем гетинакс прекрасно пропускает инфракрасное излучение. - Прим. перев.

Упомянутый бесконтактный метод обнаружения объектов (как и описанная ниже ультразвуковая дальнометрия) имеет несомненные преимущества: ведь в этом случае исключено механическое воздействие на детектор, которое может нарушить его работу. Кроме того, нет необходимости подавлять дребезг механических контактов.

Разумеется, за эти преимущества приходится расплачиваться усложнением аппаратных средств и программного обеспечения. Дело не только в стоимости элементов. Инфракрасный светодиод и фотоприемник не так уж дороги. Но применение данного метода обнаружения объектов накладывает определенные ограничения как на используемый микроконтроллер, так и на управляющую его работой программу. Описанный здесь инфракрасный детектор может иметь широкую сферу применений, но в некоторых ситуациях он может стать источником проблем.

Как известно, интенсивность излучения обратно пропорциональна квадрату расстояния до источника. Например, на расстоянии одного сантиметра от излучателя свет в четыре раза ярче, чем на расстоянии двух **сантиметров**, и в 16 раз ярче, чем на расстоянии четырех сантиметров.

Чем меньше сила света, падающего на фотоприемник, тем меньший ток течет через него. Чтобы можно было произвольно задать расстояние, на котором должен сработать наш детектор, нужно иметь возможность регулировать яркость излучателя. Существует по крайней мере три способа реализовать такую регулировку:

- использовать **подстроечный** резистор, включенный последовательно со светоизлучающим диодом (чем больше его сопротивление, тем меньше яркость светодиода);
- изменять частоту излучаемого сигнала (чем дальше она выходит за рабочий диапазон фотоприемника, тем слабее тот реагирует на излучаемый сигнал);
- использовать широтно-импульсную модуляцию излучаемого сигнала (этот способ легко реализовать, если микроконтроллер имеет встроенные средства для формирования **ШИМ-сигнала**).

Наш детектор должен воспринимать только тот сигнал, что **генерируется** излучателем, и не должен реагировать на инфракрасное излучение от любого другого источника (например, на солнце, заглянувшее в окно, или нагретую электроплитку). Поэтому приходится использовать узкополосный фильтр, настроенный на ту частоту, которую имеет сигнал, генерируемый излучателем; в нашем случае это частота 38 кГц.

- Для регулирования яркости излучателя пригодится встроенный в микроконтроллер модуль ШИМ - вот почему в одном из предыдущих проектов мы использовали программный метод генерации ШИМ-сигнала, оставив аппаратный модуль для других целей.

Если инфракрасный излучатель ориентирован преимущественно в прямом направлении, то детектор будет срабатывать на достаточном расстоянии от

объектов, расположенных спереди, но дальность срабатывания для объектов, расположенных сбоку, может оказаться существенно меньше. Это можно использовать, разместив один излучатель спереди и два фотоприемника по бокам робота.

Как уже говорилось, напряжение на излучатель подается в виде импульсов определенной частоты (чтобы можно было отфильтровывать паразитное действие других источников инфракрасного диапазона) и длительности (чтобы можно было регулировать яркость излучения). Обычно для срабатывания детектора достаточно нескольких импульсов - от двух до семи (рис. 4.56). Напомним, что светодиод горит при **низком** уровне напряжения на внешнем выводе микроконтроллера.

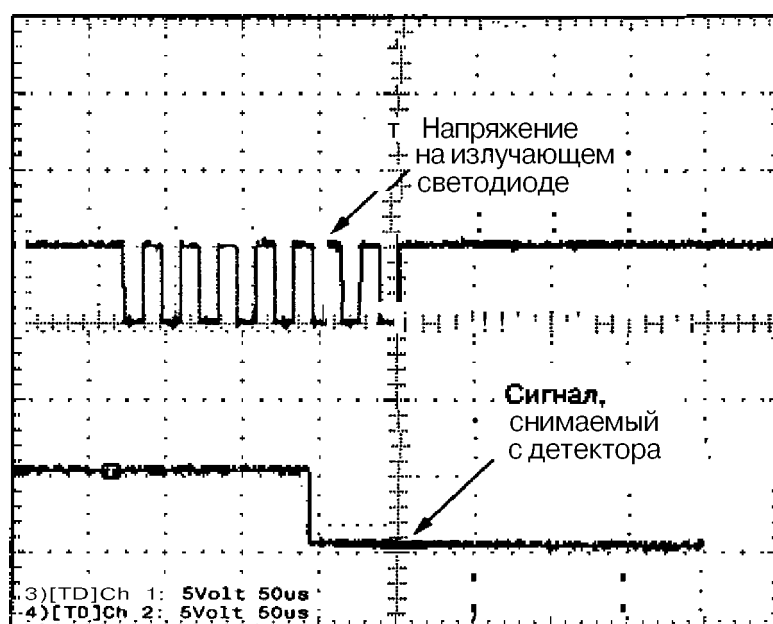


Рис. 4.56. Диаграммы сигналов **излучателя** и приемника

Кроме того, желательно совместить работу инфракрасного обнаружителя объектов и инфракрасного приемника сигналов пульта дистанционного управления. Для этого я обычно генерирую восемь световых импульсов с интервалом в миллисекунду. После восьмого импульса в течение некоторого времени фотоприемник используется для проверки того, не посылает ли пульт дистанционного управления какой-либо сигнал, а затем излучатель снова формирует восемь световых импульсов, необходимых для работы обнаружителя объектов.

4.22. ИНФРАКРАСНЫЙ ОБНАРУЖИТЕЛЬ ОБЪЕКТОВ

Хотя проект, рассматриваемый в этом разделе, даже проще, чем многие другие, представленные в книге, я должен признаться, что процесс написания программы оказался весьма нелегким. Только в результате продолжительных усилий удалось, наконец, добиться правильной работы приложения, и, как обычно, только после этого стала

очевидной его простота. К работающему проекту оказалось несложно подключить другие интерфейсы, такие как приемник сигналов дистанционного управления.

Как и в предыдущем случае, разработка нового устройства будет вестись на основе проекта с жидкокристаллическим индикатором. Это даст возможность в очередной раз продемонстрировать, как программные модули, работающие с различными периферийными устройствами, уживаются друг с другом в одном приложении. Как и прежде, наш обычный подход к разработке управляющих программ облегчит возможность совмещения различных функций.

Для формирования импульсов прямоугольной формы частотой 38 кГц, которые будут подаваться на светодиод инфракрасного диапазона, используется встроенный в микроконтроллер модуль ШИМ. Как уже говорилось, первоначально я разрабатывал свои проекты на основе микроконтроллера PIC16F84, а он не имеет аппаратных средств для генерации широтно-модулированного сигнала. Что касается других МК, то компилятор PICC Lite в то время не поддерживал возможность генерации машинного кода для какого бы то ни было микроконтроллера, имеющего встроенный модуль ШИМ. Поэтому в первом варианте этого проекта для генерации ШИМ-сигнала частотой 32 кГц мне пришлось прибегнуть к программированию на языке ассемблера.

После того как вышла новая версия компилятора PICC Lite, тексты программ были модифицированы для использования с микроконтроллером PIC16F627, который имеет встроенный модуль ШИМ. При тактовой частоте 4 МГц генерация ШИМ-сигнала частотой 38 кГц осуществляется довольно просто:

```
CCPR1L = 13;           // Длительность импульса 50%.
CCP1CON = 0b000001111; // Включение модуля ШИМ.
PR2 = 25;              // Частота 38 кГц соответствует
                       // периоду примерно 26 мкс.
TMR2 = 0;              // Сброс таймера TMR2.
T2CON = 0b000000100;   // Включение таймера TMR2,
                       // делитель 1:1.
TRISB3 = 0;            // Вывод RB3 (выходной ШИМ-сигнал)
                       // переводится в режим выходного.
```

На рис. 4.57 приведена принципиальная схема, а на рис. 4.58 показана схема размещения элементов на макетной плате.

По сравнению с предыдущими проектами способ прокладки соединительных проводов остался прежним, хотя пришлось внести некоторые изменения, чтобы освободить место для инфракрасного излучателя и подстроечного резистора.

Светодиод инфракрасного диапазона подключен ко входу RB3 микроконтроллера через подстроечный резистор, чтобы можно было не только программно, но и аппаратно регулировать яркость его свечения.

В качестве приемника отраженного сигнала используется детектор с тремя выводами, взятый со стандартного приемника дистанционного управления телевизором или музыкальным центром (рассчитанный на частоту сигнала 38 кГц). Для фильтрации сигнала, снимаемого с детектора, применен простой RC-фильтр, образованный резистором R5 и конденсатором C3. На всякий случай установлен

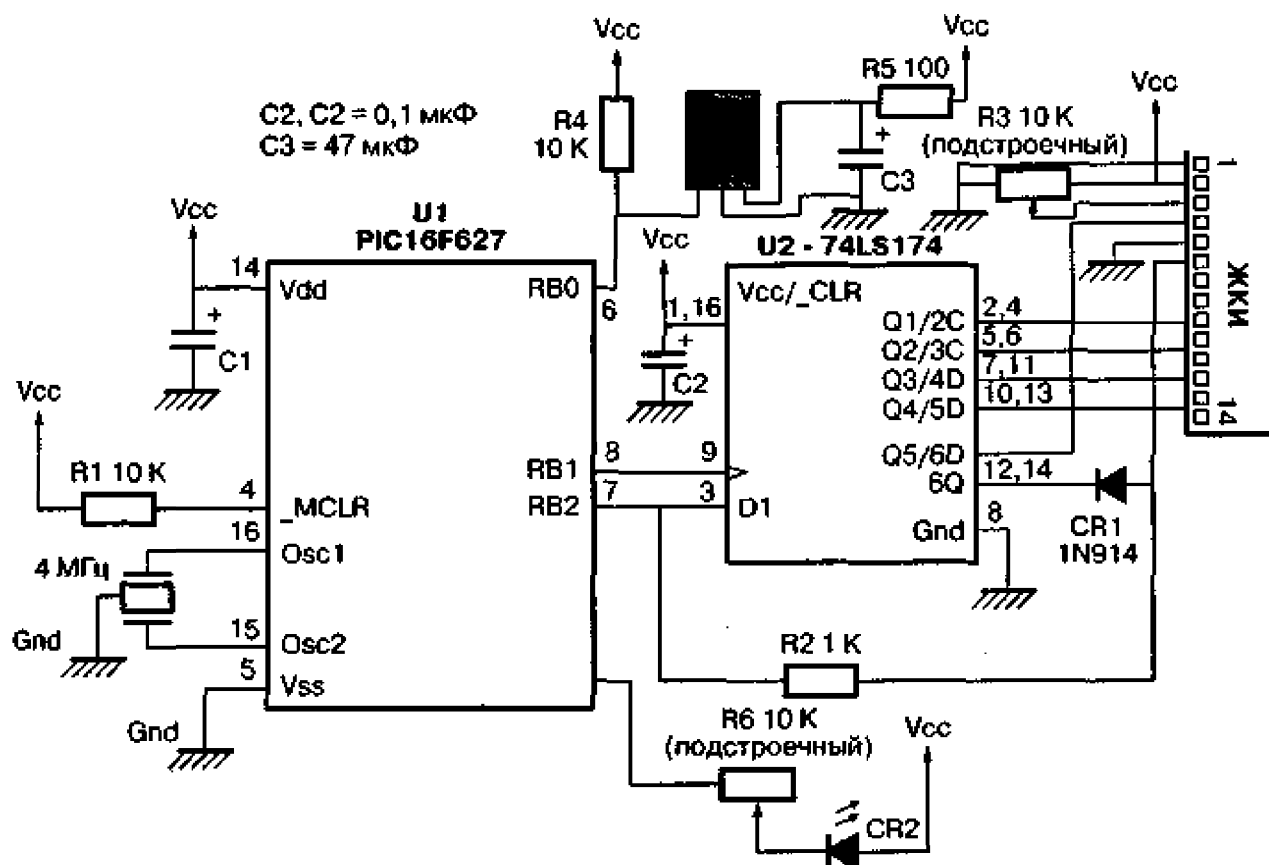


Рис. 4.57. Принципиальная схема обнаружителя объектов

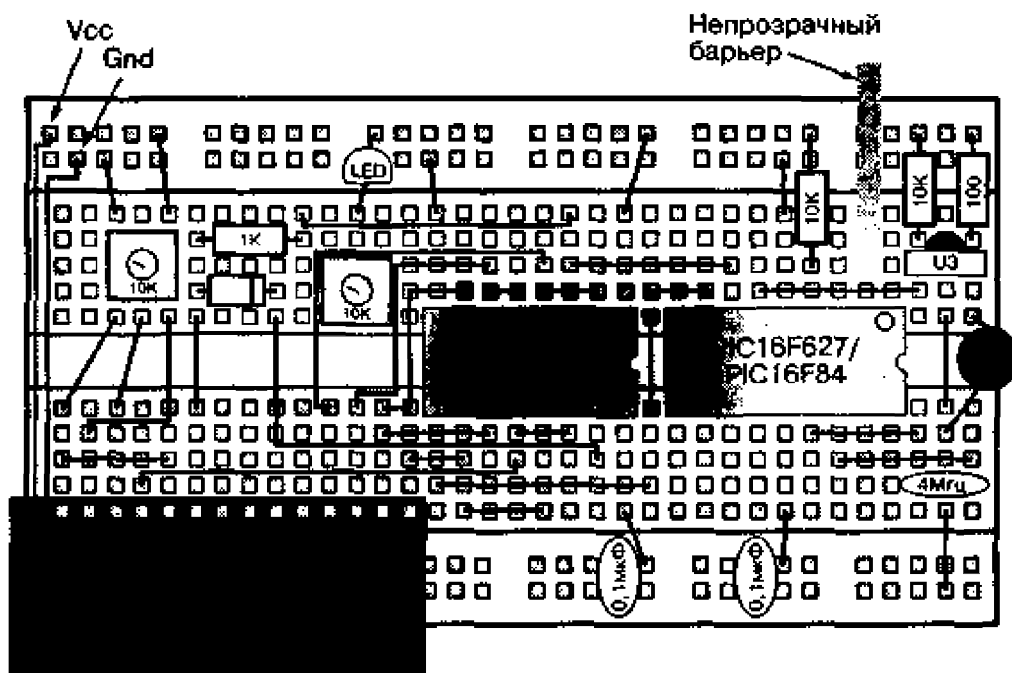


Рис. 4.58. Схема размещения элементов обнаружителя объектов на макетной плате

«подтягивающий» резистор R4, так как многие инфракрасные детекторы имеют выход с открытым коллектором.

В табл. 4.12 приведен список использованных в схеме элементов.

Здесь указаны светодиод и детектор инфракрасного диапазона фирмы Wait-gon только потому, что в Торонто они являются одними из самых доступных и дешевых, но вы можете взять практически любые детектор и светодиод.

Таблица 4.12. Список использованных элементов

Обозначение	Элемент	Примечание
U1	PIC16F627 или PIC16F84	Микроконтроллер
U2	74LS174(K555TM9)*	Сдвиговый регистр
U3	Детектор P/C- 1018SCL фирмы Waitronyили любой другой того же типа	Инфракрасный детектор с тремя внешними выводами
CR1	1N914(KD521)*	Любой кремниевый диод
CR2	Светодиод IE-0530HP или любой другой того же типа	Можно использовать любой светодиод инфракрасного диапазона
RJ, R4	10 кОм; 0,25 Вт	«Подтягивающие» резисторы
R2	1 кОм; 0,25 Вт	
R3, R6	10 кОм, подстроечный	Для регулирования контрастности дисплея и яркости светодиода
R5	100 Ом; 0,25 Вт	Фильтрация сигнала
C1, C2	0,1 мкФ	Для фильтрации напряжения питания микросхем
C3	47 мкФ, 16 В	Фильтрация сигнала
ЖКИ	Жидкокристаллический дисплей(14 выводов), совместимый с Hitachi 44780	
XTAL1	Керамический резонатор на 4 МГц, со встроенными конденсаторами	Для генератора тактовых импульсов микроконтроллера
	Материалы	Макетная плата, монтажные провода, источник питания +5 В

* В скобках указаны отечественные аналоги, добавленные при переводе. —Прим. перев.

Разумеется, детектор принимаемого сигнала должен быть ориентирован в ту же сторону, что и излучающий светодиод. Между детектором и излучателем необходимо установить непрозрачную для инфракрасных лучей перегородку.

Когда я собрал эту схему, она сначала не заработала (в отличие от всех предыдущих, которые начинали работать сразу же). Потребовалось несколько недель напряженных усилий, прежде чем удалось найти причину. Ошибку удалось обнаружить только после того, как я немного переделал схему, вернувшись к старой конструкции, которая уже доказала свою работоспособность во время создания одного из моих прошлых проектов около года назад.

Оказалось, что инфракрасный детектор рассчитан на работу с более-менее постоянным сигналом и имеет возможность подстраивать встроенный в него фильтр, адаптируя его к частоте принимаемого сигнала. Данная особенность не была указана в документации. В нашей конструкции, как уже говорилось, сигнал делится всего на восемь периодов, после чего на время прекращается (в одном из следующих проектов эта пауза будет использоваться для приема сигналов пульта дистанционного управления), поэтому фильтр внутри инфракрасного детектора не успевает подстроиться к сигналу. Проблема решилась с помощью небольшого фрагмента, добавленного к тексту обработчика прерываний от таймера:

```
TRISB3 = 0;           // Разрешить прохождение ШИМ-сигнала на выход
                        // (светодиод горит).
```

```

while (TMRO < 64); // Ждать, пока детектор настроится на сигнал.

if (!RBO)           // Если на входе низкий уровень,
    Collision = 1; // то выдача сигнала об обнаружении объекта,
else                // в противном случае никаких объектов нет.
    * Collision = 0;

TRISB3 = 1;         // Запрещение выхода ШИМ-сигнала
                    // (светодиод потушен).

```

Здесь ШИМ-сигнал пропускается на выход микроконтроллера только в течение 64 циклов работы счетчика таймера TMRO. При частоте тактирования 4 МГц и коэффициенте делителя 1:4 это соответствует промежутку времени длительностью 256 мкс. При частоте 38 кГц за это время формируются восемь или девять импульсов ШИМ-сигнала. В результате своих предыдущих экспериментов с инфракрасными детекторами я пришел к мысли, что, возможно, в детекторе выполняется синхронизация частоты настройки фильтра с частотой принимаемого сигнала.

Пытаясь найти ошибку, я сравнил фрагмент программы, управляющий работой модуля ШИМ, со своей прошлогодней разработкой, в которой использовалась та же самая электрическая схема. Используя приемник старого робота, удалось убедиться в том, что новая схема действительно генерирует сигнал. Что касается программы, то не было никаких сомнений в ее правильности. Не зная, что делать дальше, я попробовал вместо одиночных импульсов посылать пакеты из восьми-девяти импульсов, и тут новая схема неожиданно заработала!

Этот случай прекрасно доказывает, что во время отладки приложения надо проверить *все* возможности, так как любые наши предположения или наблюдения могут оказаться не слишком правильными - либо возникнет обстоятельство, обратить внимание на которое не приходило нам в голову.

Окончательную версию программы вы найдете в файле Code\Irdetect\irdetect.c. Здесь генерируется ШИМ-сигнал частотой 38 кГц. При наличии отраженного сигнала (то есть когда поблизости имеется какой-либо объект), на жидкокристаллическом дисплее высвечивается надпись COLLISION. Если отраженного сигнала нет, то на дисплей ничего не выводится.

Подстроечный резистор R6 должен быть откалиброван, чтобы можно было устанавливать его движок в нужное положение, соответствующее заданной дальности срабатывания детектора. Прежде чем начать калибровку, следует убедиться, что между светодиодным излучателем и детектором установлен экран, не пропускающий инфракрасное излучение. На рис. 4.58 показано, как следует расположить этот экран. Должен предупредить вас, что для такого экрана не столь уж просто найти подходящий материал. Вполне непрозрачные для видимого света предметы, в частности плотная бумага или ткань, могут оказаться прозрачными для инфракрасных лучей. Рекомендую выбрать фольгу или черную изоляционную ленту.

Для градуировки резистора расположите белую плотную бумагу, например визитную карточку, в четырех-пяти дюймах (то есть на расстоянии 10-12 см) от излучателя и приемника и осторожно перемещайте движок резистора до тех пор,

пока не произойдет срабатывание детектора — иными словами, пока на дисплее не высветится соответствующее сообщение. Теперь, когда при движении робота какой-либо объект окажется на этом расстоянии от детектора, на дисплее будет появляться надпись **COLLISION**.

Приведем текст программы `irdetect.c`:

```

#include <pic.h>

// 17 апреля 2002 - инфракрасный обнаружитель объектов.
//
// Для индикации используется жидкокристаллический дисплей,
// подключенный к микроконтроллеру через двухпроводной интерфейс
// на основе сдвигового регистра 74LS174.
//
// Используются прерывания от таймера TMR0.
//
// Замечания по аппаратным средствам:
// Микроконтроллер PIC16F627 работает на частоте 4 МГц.
// Используется внешний тактовый генератор.
//
// Подключение выводов микроконтроллера:
// RB0 - инфракрасный детектор,
// RB1 - линия Clock интерфейса жидкокристаллического дисплея;
// RB2 - линия Data интерфейса жидкокристаллического дисплея;
// RB3 - выход широтно-модулированного сигнала.

// Глобальные переменные и константы:
int RTC = 0; // Счетчик реального времени.
volatile char LCDDlay = 20; // Длительность задержки.
volatile char LCDState = 1; // Номер текущего состояния
// конечного автомата.

static volatile bit Clock @ (unsigned)&PORTB*8+1;
static volatile bit ClockTRIS @ (unsigned)&TRISB*8+1;
static volatile bit Data @ (unsigned)&PORTB*8+2;
static volatile bit DataTRIS @ (unsigned)&TRISB*8+2;

char * MessageOut; Указатель на строку выводимого сообщения.
volatile char MessageOuti = 0; // Смещение, указывающее позицию
// выводимого символа.
char Message1[2] = "\f"; // Очистка дисплея.
char Message2[11] = "\fCOLLISION";

volatile char Collision = 0; // Сначала никаких объектов поблизости нет.
volatile char OldCollision = 0;

// Слово конфигурации:
#if defined(_16F627)
#warning PIC16F627 with external oscillator selected
__CONFIG(0x03F61); // Для МК PIC16F627:
// внешний тактовый генератор XT,
// RA6/RA7 используются для ввода-вывода,
// внешний сигнал сброса,
// таймер PWRT включен,
// сторожевой таймер выключен,
// защита кода отключена,
// детектор BODEN включен.

```

```

«else
    Error Unsupported PICmicro MCU selected
#endif

// Служебные подпрограммы:
LCDNybble(char Nybble, char RS)          // Запись полубайта в ЖКИ.
...

LCDByte(char Byte, char RS)              // Запись байта в ЖКИ.
...

LCDInit()                                // Инициализация ЖКИ.
...

LCDOut(char * const LCDString)           // Вывод строки на ЖКИ.
...

// Обработчик прерываний:
void interrupt tmr0_int(void)
{
char temp;

    if (TOIF) {                          // Обработчик прерываний от таймера TMR0.
        TOIF = 0;                        // Сбросить флаг прерываний от таймера TMR0.
        RTC++;                           // Инкремент счетчика реального времени.

        // Здесь можно разместить дополнительный код
        // для обработки прерываний от таймера TMR0.

        // Вывести последовательность импульсов
        // для детектора столкновений:

        TRISB3 = 0;                      // Разрешить вывод ШИМ-импульсов.
        while (TMR0 < 64);               // В течение 64 «тиков» таймера.

        if (!RBO)                        // Если уровень низкий,
            Collision = 1;                // значит обнаружен объект;
        else                             // в противном случае
            Collision = 0;                // объектов нет.

        TRISB3 = 1;                      // Запретить вывод ШИМ-сигнала.

        // Конечный автомат для ЖКИ:
        switch(LCDState) { // В зависимости от текущего состояния:
            case 1:                    // Начать инициализацию ЖКИ.
                if (--LCDDlay == 0)
                    LCDState++;
                break;                // Ждать 20 мс.
            case 2:                    // Шаг 2.
                LCDNybble(0x003, 0);
                LCDDlay = 5;
                LCDState++;
            case 3:                    // Ждать выполнения команды.
                if (--LCDDlay == 0)
                    LCDState++;

```

```

        break;
case 4:
    LCDNybble(0x003, 0);  // Шаг 3.
    LCDState++;
    break;
case 5:
    LCDNybble(0x003, 0);  // Шаг 4.
    LCDState++;
    break;
case 6:
    LCDNybble(0x002, 0);  // Шаг 5.
    LCDState++;
    break;
case 7:
    LCDByte(0x028, 0);    // Шаг 6.
    LCDState++;
    break;
case 8:
    LCDByte(0x008, 0);    // Шаг 7.
    LCDState++;
    break;
case 9:
    LCDByte(0x001, 0);    // Шаг 8.
    LCDState++;
    LCDDlay = 5;          // Ждать 5 мс.
    break;
case 10:
    // Ждать выполнения команды.
    if (--LCDDlay == 0)
        LCDState++;
    break;
case 11:
    LCDByte(0x006, 0);    // Шаг 9.
    LCDState++;
    break;
case 12:
    LCDByte(0x00E, 0);    // Шаг 10.
    LCDState = 0;         // Все готово.
    break;

case 100:
    // Вывод сообщения.
    switch (temp = MessageOut[MessageOuti++]) {
        case "\0":        // Конец сообщения?
            LCDState = 0;
            MessageOuti = 0;
            break;
        case "\f":        // Очистка дисплея.
            LCDByte(0x001, 0);
            LCDState++;
            LCDDlay = 5;
            break;
        case 254:         // Команда?
            LCDState = 0;
            else {
                if (temp < 4) {
                    LCDState++;
                    LCDDlay = 5;
                }
            }
    }
}
if ((temp = MessageOut[MessageOuti++]) == 0)

```

```

        LCDByte(temp, 0);
    }
    break;
    default: // Обычные символы.
        LCDByte(temp, 1);
    } // Конец внутреннего оператора switch.
    break;
case 101: // Message Delay
    if (--LCDDlay == 0)
        LCDState--;
    break;

    } // Конец внешнего оператора switch.

} // Конец оператора if.

// Здесь можно разместить другие обработчики прерываний.
> // Конец обработчика прерываний.

// Главная программа:
void main(void)
{
    OPTION = 0x0D1; // Предделитель работает с таймером TMR0,
                   // коэффициент деления равен 4.
    TMR0 = 0; // Начальный сброс таймера TMR0.
    TOIE = 1; // Разрешение прерываний от таймера TMR0.
    GIE = 1; // Разрешение обработки прерываний.

    LCDInit(); // Инициализация псрта, к которому подключен ЖКИ.
    // Здесь можно проинициализировать другие периферийные устройства.

    CCP1L = 13; // Ширина импульсов - 50% периода.
    CCP1CON = 00000001111; // Включить модуль ШИМ.

    PR2 = 26; // При частоте 38 кГц период равен 26 мкс.
    TMR2 = 0; // Сброс таймера TMR2.
    T2CON = 0b0000000100; // Включение таймера TMR2,
                          // коэффициент предделителя 1:1.

    while (1 == 1) { // Бесконечный цикл.

        // Здесь можно разместить код биологического уровня.
        if (Collision != OldCollision) {
            OldCollision = Collision; // Сохранить текущее
                                     // состояние детектора.

            if (Collision)
                LCDOut(Message2); // Вывести сообщение.
            else
                LCDOut(Message1);
            while (LCDState); // Ждать готовности ЖКИ.
        }

    }

}
// Конец главной программы.

```

Когда вы закончите эксперименты с этим устройством, не разбирайте его, так как оно вам понадобится при разработке следующих двух проектов.

4.23. ДИСТАНЦИОННОЕ УПРАВЛЕНИЕ РОБОТОМ

Всегда желательно иметь какую-либо возможность управлять действиями и/или передвижением робота. Разумеется, для этих целей лучше всего подходит беспроводной метод, например радиоуправление или использование привычного пульта дистанционного управления. Для работы удобен стандартный телевизионный пульт, цифровые кнопки от 1 до 9 которого, расположенные в виде матрицы 3x3, хорошо подходят для выбора направления движения, а кнопки увеличения/уменьшения громкости - для регулирования скорости движения. Оставшиеся незанятыми кнопки (0 и включение питания) можно приспособить для подачи роботу каких-либо команд.

Я использовал телевизионный пульт дистанционного управления фирмы Sony. Стандарт Sony предписывает передачу команд импульсами света инфракрасного диапазона; при этом применяется код Manchester. В манчестерской кодировке информация представляется последовательностью импульсов, длительность каждого из которых может иметь одно из трех возможных значений: T , $2T$ или $4T$, где $T = 0,60$ мс - период тактовых импульсов кодера. Пример сигнала в манчестерской кодировке представлен на рис. 4.59, а на рис. 4.60 показан сигнал на входе приемника, полученный с помощью осциллографа.

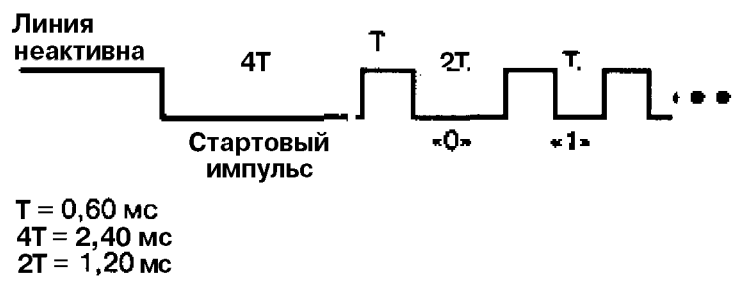


Рис. 4.59. Манчестерский код

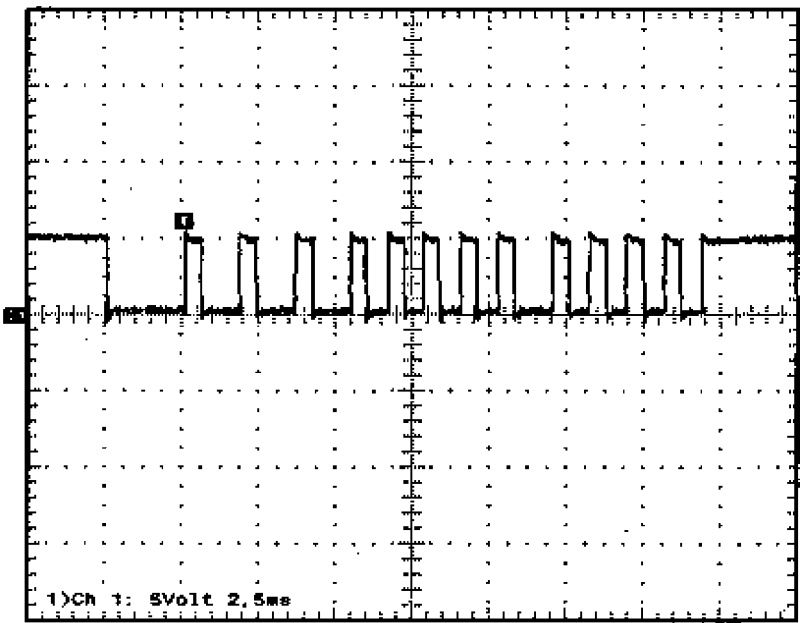


Рис. 4.60. Осциллограмма сигнала дистанционного управления

Информационный пакет состоит из 13 импульсов отрицательной полярности. Это значит, что сигнал высокого уровня представляет паузу между импульсами, а сами импульсы кодируются низким уровнем напряжения. Первый импульс низкого уровня, длительностью $4T$, является стартовым; за ним следуют информационные. Логический 0 представляется импульсом низкого уровня длительностью $2T$, а логическая 1 - длительностью T . Все информационные импульсы отделяются друг от друга синхрои́мпульсами высокого уровня длительностью T .

В табл. 4.13 показано, какие последовательности импульсов соответствуют различным кнопкам пульта дистанционного управления Sony.

Таблица 4.13. Команды инфракрасного пульта дистанционного управления фирмы Sony

Кнопка	Код
0	0b0011011101111
1	0b0011011101111
2	0b0011011101111
3	0b010111011101111
4	0b000111011101111
5	0b010111011101111
6	0b0010111101111
7	0b0100111011101111
8	0b0000111011101111
9	0b0110111011101111
Увеличить громкость	0b01011011101111
Уменьшить громкость	0b00011011101111
Канал +1	0b0111011101111
Канал -1	0b00111011101111
Предыдущий канал	0b0001000101111
Приглушить звук	0b01101011101111
Питание	0b00101011101111

Каждый информационный пакет повторяется примерно каждые 50 мс. Передатчик не имеет встроенной функции автоповтора, поэтому он должен обеспечиваться внутри приемника. При разработке дистанционно управляемых автоматических устройств это не является проблемой - ведь повторяющиеся команды удобны для того, чтобы поддерживать выполнение какой-либо функции в течение всего времени, пока нажата соответствующая клавиша. Во многих моих конструкциях двигатель робота, выполнив очередную команду дистанционного управления, остается включенным еще в течение 200 мс, ожидая следующего пакета. Такой защитный интервал предотвращает остановку робота в случае, если следующий пакет данных не будет получен вовремя или связь между передатчиком и приемником на короткое время нарушится.

Описанный здесь принцип работы пульта дистанционного управления довольно прост, но все же у разработчиков могут возникнуть проблемы с пониманием

того, как надлежащим образом декодировать поступающие команды. Наиболее очевидный способ заключается в том, чтобы принимать импульсы и на ходу определять их длительность с помощью следующей программы:

```
void main(void)    // Декодирование команд дистанционного управления.
{
    unsigned int i;
    unsigned int DataPacketStart; // Время получения начала импульса.
    unsigned int DataPacketEnd;   // Время окончания импульса.
    unsigned int DataPacket;      // Пакет данных.

    while (1 == 1) {            // Бесконечный цикл.
        while (IRData == High); // Ждать прихода импульса низкого уровня.

        DataPacketStart = TMRO; // Время начала импульса низкого уровня.
        while (IRData == low);  // Ждать конца текущего импульса.
        DataPacketEnd = TMRO - DataPacketStart; // Длительность импульса.

        if ( (DataPacketEnd > 2.2msecs) && (DataPacketEnd < 2.6msecs) ) <
            // Следующие 12 бит.
            DataPacket = i = 0; // Инициализация.
            DataPacketEnd = 1.8msecs;

            while ( (i < 12) && (DataPacketEnd != 0) ) {
                while (IRData == High); // Ждать начала импульса.
                DataPacketStart = TMRO; // Запомнить время
                // начала импульса.
                while (IRData == low); // Ждать конца импульса.

                DataPacketEnd = TMRO - DataPacketStart;
                // Длительность импульса.
                if ( (DataPacketEnd > 0.45msec) &&
                    (DataPacketEnd < 0.75msec) )
                    DataPacket = (DataPacket << 1) + 1;
                else
                    if ((DataPacketEnd > 0.95msec) &&
                        (DataPacketEnd < 1.35msec))
                        DataPacket = (DataPacket << 1) + 1;
                    else // Ошибка.
                        DataPacketEnd = 0; // Конец.
                        i++; // Указатель - на следующий бит.
            } // Конец внутреннего цикла while.
            if (DataPacketEnd != 0)
                printf ("Data Packet = %i\n", DataPacket);
        } // Конец оператора if.
    } // Конец внешнего оператора while.
} > // Конец программы.
```

Этот метод немного громоздкий, но хорошо показывает принцип работы приемника сигналов дистанционного управления. Проблема, однако, в том, что приведенная программа не может быть использована в наших конструкциях: во-первых, нарушены все наши правила о разделении функций различных уровней, а во-вторых, опрос инфракрасного детектора производится постоянно, и мы не сможем совместить прием сигналов дистанционного управления с работой инфракрасного обнаружителя объектов.

Разумней будет, если мы разрешим формирование запроса на прерывание по приходу каждого импульса на вход микроконтроллера, который подключен к инфракрасному детектору, а процедуру измерения длительности этих импульсов И-декодирования принимаемых команд поручим обработчику прерываний. В следующем разделе будет показано, как воплотить эту идею в жизнь.

4.24. ПРИЕМНИК СИГНАЛОВ ДИСТАНЦИОННОГО УПРАВЛЕНИЯ

Как уже упоминалось, многие радиолюбители избегают использования стандартных инфракрасных пультов дистанционного управления в своих устройствах, боясь сложностей, с которыми рассчитывают столкнуться при проектировании приемника и декодера команд. В этом разделе мы рассмотрим пример такого приемника и докажем, что эти опасения беспочвенны. Программная реализация нашего декодера не окажется слишком сложной, и он будет прекрасно работать совместно с другими интерфейсами, которые уже реализованы в вышеописанных проектах.

В одной из моих предыдущих книг, «Programming and Customizing PICmicro Microcontrollers» (Программирование и использование микроконтроллеров PICmicro), было рассмотрено несколько способов реализации приемника сигналов дистанционного управления на основе различных микроконтроллеров PICmicro. Как всегда, если какая-либо задача уже решена одним способом, то реализация других методов ее решения не вызывает особых проблем.

В самом начале процесса проектирования необходимо четко определить, **какие** команды должен принимать декодер, в каком виде они будут подаваться и какие возможности микроконтроллера будут использоваться при декодировании.

Если вы читали эту книгу с самого начала, то для вас уже должна быть очевидной формулировка, описывающая работу декодера: *декодер должен принимать сигналы пульта дистанционного управления фирмы Sony и передавать биологическому коду соответствующие команды.*

Здесь необходимо сделать два замечания. Во-первых, желательно выполнять отбраковку ошибочно принятых данных. Если длительность какого-либо импульса слишком отличается от трех разрешенных значений, то разумно будет проигнорировать весь 12-битный пакет. Во-вторых, для простоты можно не принимать в расчет стартовый бит в начале каждого пакета.

Мы уже обсуждали в главе 3 возможные источники прерываний в микроконтроллерах PICmicro. Поэтому без лишних комментариев выберем для нашего проекта вывод 6 микроконтроллера PIC16F84/PIC16F627 (RBO/INT), который может быть использован для формирования запроса на прерывание при каждом изменении уровня входного сигнала.

Необходимо, чтобы обработчик прерывания вызывался каждый раз, когда уровень принятого инфракрасным детектором сигнала изменяется с низкого на высокий, как показано на рис. 4.61. Тогда, измеряя время, прошедшее с момента последнего вызова, можно оценить ширину импульса и таким образом отличить 0 от 1.

Для отсчета времени, как обычно, можно использовать таймер TMRO, сохраняя его текущее значение в специально отведенной для этого переменной.

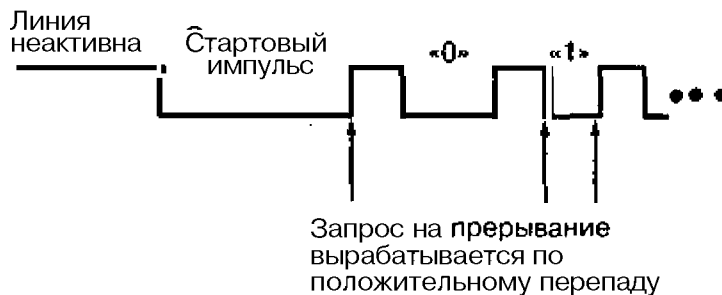


Рис. 4.61. Формирование запроса на прерывание по *положительному фронту* входного сигнала при приеме сигналов дистанционного управления

Приведенная в предыдущем разделе программа учитывала длительность стартового импульса, но мы упростим код, если вообще будем его игнорировать: ведь для определения длительности очередного импульса он совершенно не нужен.

При каждом вызове обработчика прерываний, осуществляемом по приходе положительного фронта очередного импульса, мы будем проверять, как давно был выполнен предыдущий вызов. Если прошло более 9 мс, то надо прекратить процесс декодирования до прихода следующего пакета. Как уже говорилось, пакеты повторяются каждый 50 мс. Поэтому, пропустив текущий пакет, мы всегда имеем шанс без ошибок принять следующий.

Для реализации нашего нового проекта не понадобится переделывать предыдущую схему, но, разумеется, придется изменить текст программы. Для отображения принимаемых команд будет использоваться все тот же жидкокристаллический дисплей. Текст программы вы найдете в файле `Code\Remote\remote.c`:

```
#include <pic.h>
```

```
// 17 апреля 2002 - адаптировано к новой версии компилятора.
// 15 февраля 2002 - прием и отображение команд инфракрасного пульта
// дистанционного управления фирмы Sony.
```

```
//
// Для индикации используется жидкокристаллический дисплей,
// подключенный к микроконтроллеру через двухпроводной интерфейс
// на основе сдвигового регистра 74LS174.
```

```
/// Используются прерывания от таймера TMRO.
```

```

//
//
// Формат пакета данных:
//
//          Стартовый бит          1          0
//  ---+          +-----+          +-----+          +-----+
//          I          I          I          I          I
//  +-----+          +-----+          +-----+          +-----+
//
//          |          2,4 мс          |540мкс|660мкс|540мкс|          1,2 мс          |

```

1,2 MC	1,76 MC

//				
///		300		440

```

//
// Замечания по аппаратным средствам:
// Микроконтроллер PIC16F84/PIC16F627 работает на частоте 4 МГц.
// Используется внешний тактовый генератор.
//
// Подключение выводов микроконтроллера:
// RB0 - детектор инфракрасного излучения
// (прерывания по положительному перепаду):
// RB1 - линия Clock интерфейса жидкокристаллического дисплея;
// RB2 - линия Data интерфейса жидкокристаллического дисплея.

// Глобальные переменные и константы:
int RTC = 0; // Счетчик реального времени.
volatile char LCDDlay = 20; // Длительность задержки.
volatile char LCDState = 1; // Номер текущего состояния
// конечного автомата.

static volatile bit Clock @ (unsigned)&PORTB*8+1;
static volatile bit ClockTRIS @ (unsigned)&TRISB*8+1;
static volatile bit Data @ (unsigned)&PORTB*8+2;
static volatile bit DataTRIS @ (unsigned)&TRISB*8+2;

char » MessageOut; // Указатель на строку выводимого сообщения.
volatile char MessageOuti = 0; // Смещение текущего символа в строке.

unsigned int DataIn; // Входной сигнал инфракрасного приемника.
unsigned char DataInCount = 0; // Число импульсов для приема.
unsigned char DataReady = 0; // Число принятых импульсов.
int SaveRTC; // Время последнего прерывания.
int CurrentRTC;
char Message[20] = "\\f-> 0b0xxxxxxxxxxxx"; // Шаблон сообщения.

// Слово конфигурации:
«if defined (_16F84)
    #warning PIC16F84 selected
    __CONFIG(0x03FF1); // Для МК PIC16F84:
                        // кварцевый тактовый генератор,
                        // таймер PWRT включен,
                        // сторожевой таймер выключен,
                        // защита кода отключена.
#elif defined(_16F627)
    #warning PIC16F627 with external oscillator selected
    __CONFIG(0x03F61); // Для МК PIC16F627:
                        // внешний тактовый генератор XT,
                        // RA6/RA7 используются для ввода-вывода,
                        // внешний сигнал сброса,
                        // таймер PWRT включен,
                        // сторожевой таймер выключен,
                        // защита кода отключена,
                        // детектор BODEN включен.
#else
    #error Unsupported PICmicro MCU selected
#endif

// Служебные подпрограммы:
LCDNybble(char Nybble, char RS) // Запись полубайта в ЖКИ.

```

```

LCDByte(char Byte, char RS)           // Запись байта в ЖКИ.

...

LCDInit()                             // Инициализация ЖКИ.

...

LCDOut(char • const LCDString)        // Вывод строки на ЖКИ.

...

// Обработчик прерываний:
void interrupt tmr0_int(void)
{
    char temp;

    if (TOIF) {                       // Обработчик прерываний от таймера TMR0.
        TOIF = 0;                     // Сбросить флаг прерываний от таймера TMR0.
        RTC++;                        // Инкремент счетчика реального времени.
        // Здесь можно разместить дополнительный код
        // для обработки прерываний от таймера TMR0.
        // Проверка времени последнего вызова обработчика
        // и пропуск текущего пакета при необходимости:
        CurrentRTC = (RTC & 0xOFF) - ((SaveRTC >> 8) & 0xOFF);
        if (CurrentRTC < 0)
            CurrentRTC = 0 - CurrentRTC;
        if ((DataInCount != 0) && (CurrentRTC > 9))
            DataInCount = 0;           // Пропустить текущий пакет.
        // Конечный автомат для ЖКИ:
        ...
    } // Конец обработчика прерываний от таймера.

    // Здесь можно разместить другие обработчики прерываний.

    if (INTF) {                       // Обработчик прерываний по положительному
        // фронту сигнала на выводе RBO/INT.
        if (DataReady)                // Предыдущий пакет уже принят?
            ;                          // Игнорировать импульсы до конца пакета.
        else if (DataInCount == 0) { // Начало нового пакета.
            DataInCount = 12;          // Надо принять 12 импульсов.
            SaveRTC = ((RTC & 0xOFF) << 8) + TMR0;
            // Запомнили текущее время.
            DataIn = 0;
        } else {                      // Обработка очередного бита.
            CurrentRTC = ((RTC & 0xOFF) << 8) + TMR0;
            // Узнали текущее время.
            if ((SaveRTC = CurrentRTC - SaveRTC) < 0)
                SaveRTC = 0 - SaveRTC;
            if ((SaveRTC > 250) && (SaveRTC < 350)) {
                DataIn = (DataIn << 1) + 1; // Получена 1.
                if (--DataInCount == 0)
                    DataReady = 1;
            } else if ((SaveRTC > 390) && (SaveRTC < 490)) {
                DataIn = DataIn << 1;      // Получен 0.
            }
        }
    }
}

```

```

        if (--DataInCount == 0)
            DataReady = 1;
    } else // Ошибка.
        DataInCount = 0;
    SaveRTC = CurrentRTC; // Запомнили текущее время.
}

INTF = 0; // Сброс флага прерывания.

} // Конец обработки прерываний по положительному фронту
// сигнала на входе RBO/INT.

} // Конец обработчика прерываний.

// Главная программа:

void main(void)
{
    int i;

    OPTION = 0x0D1; // Предделитель работает с таймером TMRO,
                   // коэффициент деления равен 4.

    TMRO = 0; // Начальный сброс таймера TMRO.
    TOIE = 1; // Разрешение прерываний от таймера TMRO.
    GIE = 1; // Разрешение обработки прерываний.

    // Здесь можно проинициализировать другие периферийные устройства.

    LCDInit(); // Инициализация порта, к которому подключен ЖКИ.

    INTEDG = 1; // Прерывания по положительному фронту сигнала
               // на входе RBO/INT.
    INTE = 1; // Разрешить прерывания по изменению сигнала
             // на выходе RBO/INT.

    while (1 == 1) { // Бесконечный цикл.

        // Здесь можно разместить код верхнего уровня.

        if (DataReady) { // Если данные готовы.
            while (LCDState != 0); // Ждать готовности ЖКИ.
            for (i = 0; i < 12; i++) {
                if (DataIn & (1 << 11))
                    Message[i + 7] = '1';
                else
                    Message[i + 7] = '0';
                DataIn = DataIn << 1;
            } // Конец цикла вывода кода принятой команды.
            LCDOut(Message); // Вывести сообщение.
            DataReady = 0; // Сбросить флаг
                          // готовности данных.

        } // Конец оператора if.
    } // Конец оператора while.

} // Конец главной программы.

```

Код этой программы может показаться достаточно сложным, но на самом деле в нем не так уж трудно разобраться. Когда на входе RBO/INT микроконтроллера, к которому подключен инфракрасный приемник, уровень сигнала изменяется с низкого на высокий, то устанавливается флаг INTF и генерируется запрос на

прерывание. В процедуре обработки прерываний проверяется состояние этого флага. Если он установлен, значит, источником прерывания послужил вход RBO/INT. В таком случае обработчик прерываний по значению счетчика DataInCount проверяет, первый ли это импульс. Если значение счетчика нулевое, оно свидетельствует о том, что пришел стартовый импульс нового пакета данных. Далее мы загружаем в счетчик число ожидаемых импульсов (то есть значение 12) и запоминаем текущее время, чтобы при следующем вызове определить длительность импульса. Так продолжается до тех пор, пока не будут декодированы все 12 импульсов пакета данных.

После приема двенадцатого импульса устанавливается флаг DataReady - таким образом главной программе сообщается, что очередная команда дистанционного управления декодирована. Обработчик будет игнорировать все остальные пакеты, пока главная программа, выполнив команду, не сбросит этот флаг.

Принятая последовательность нулей и единиц для контроля выводится на жидкокристаллический дисплей. Заметим, что перед загрузкой в выводимую строку текста нового сообщения мы проверяем, закончился ли вывод предыдущего, чтобы содержимое строки Message случайно не изменилось, пока еще продолжается ее вывод на дисплей. Если пренебречь этой проверкой, может оказаться, что на экране мы увидим начало предыдущей команды и конец следующей.

В данном случае вероятность такого события не слишком велика, поскольку вывод на дисплей занимает времени намного меньше, чем промежуток между пакетами данных, но мы всегда должны быть готовы к тому, что нам придется модифицировать свои приложения для более сложных задач.

4.25. СОВМЕЩЕНИЕ РАБОТЫ ДЕТЕКТОРА ОБЪЕКТОВ И ПРИЕМНИКА КОМАНД ДИСТАНЦИОННОГО УПРАВЛЕНИЯ

Итак, **наш** робот уже умеет обнаруживать объекты и принимать сигналы дистанционного управления. В обоих проектах был использован детектор инфракрасного излучения. Поэтому будет логично объединить эти две функции в одном устройстве.

Так как в одном устройстве будут объединены функции по обнаружению близких объектов и приему сигналов дистанционного управления, то придется совместить оба типа сообщений на одном жидкокристаллическом дисплее. Сообщения о состоянии детектора объектов и об очередной принятой команде должны будут выводиться в разных частях экрана, не затирая друг друга.

Сначала я полагал, что для реализации проекта достаточно лишь немного модифицировать код обработчика прерываний от таймера, срабатывающего каждую миллисекунду. При каждом его вызове можно было бы проверять состояние входа RBO/INT, и, если уровень сигнала на нем активный (низкий), начинать измерение длительности импульса пакета данных, пришедшего от пульта дистанционного управления. Если при входе в обработчик уровень сигнала высокий, можно было бы сгенерировать **очередной** пакет импульсов частотой 38 кГц для инфракрасного излучателя. Идея казалась достаточно простой, но после нескольких часов безуспешных попыток заставить все это работать я был вынужден еще раз пересмотреть свои требования к системе, чтобы, наконец, понять, что же я хочу получить.

При первоначальной реализации проекта оказались допущены две ошибки. Во-первых, я забыл о своем решении игнорировать стартовый импульс пакета данных и использовать положительные (задние) фронты остальных двенадцати информационных импульсов для измерения длительности каждого из них. Совершенно справедливо посчитав, что раз ширина стартового импульса больше двух миллисекунд, то очередное прерывание от таймера, срабатывающего каждую миллисекунду, обязательно зафиксирует начало пакета данных, я забыл, что все еще использую задний фронт стартового импульса для вычисления длительности первого бита в пакете.

Поэтому в обработчике прерываний, обнаружив низкий уровень сигнала на входе RBO/INT, я разрешал прерывания по положительному перепаду напряжения на этом входе, а затем, как и в предыдущей программе, инициализировал счетчик `DataInCount` значением 12, хотя на самом деле надо было ожидать 13 срабатываний обработчика. Устройство начало работать только после того, как я изменил указанное значение на 13 и обеспечил пропуск интервала между двумя первыми срабатываниями (с того момента, когда был замечен низкий уровень сигнала на входе RBO/INT и было разрешено формирование запросов на прерывание по этому входу, до момента прихода первого такого запроса, соответствующего концу стартового импульса).

Вторая ошибка была обнаружена при испытании правильности приема команд. Иногда в момент нажатия какой-либо кнопки пульта дистанционного управления на экране появлялось сообщение `COLLISION`, то есть вместо приемника команд срабатывал детектор столкновений. Виновником ошибки оказался стартовый импульс пакета данных, в момент прихода которого инфракрасный излучатель обнаружителя объектов еще не успевал выключиться. Детектор воспринимал начало команды дистанционного управления как сигнал, отраженный от объекта.

Пришлось сделать так, чтобы детектор реагировал на принятый сигнал только в том случае, если он действует более 3 мс. Так как длительность стартового импульса равна 2,4 мс, задержка срабатывания гарантирует, что стартовый импульс будет проигнорирован детектором. Разумеется, уже с самого начала было непростительной ошибкой рапортовать о столкновении только на основании первого же срабатывания детектора. Я никогда бы не допустил подобной ошибки, если бы имел дело с механическим детектором столкновений, но, испытывая свои многочисленные конструкции, основанные на инфракрасном обнаружителе объектов, я уже привык к его высокой надежности, поскольку еще ни разу не видел, чтобы он допускал ложные срабатывания.

После всех описанных изменений устройство стало работать почти без ошибок. Однако когда вблизи детектора находился какой-либо объект и в то же время я нажимал любую кнопку пульта дистанционного управления, каждый десятый раз при приеме команды происходила ошибка (единичный бит пакета данных принимался декодером за нулевой).

Теоретически такая ошибка могла объясняться тем, что во время выполнения фрагмента обработчика прерываний, реализующего вывод сообщений на экран жидкокристаллического дисплея, устройство не успевало правильно измерить ширину очередного импульса. Эта идея подтверждалась тем, что ошибка пропадала

при отключении вывода сообщений на экран (или строки COLLISION, или кода принятой команды).

Проблему можно решить, запретив работу приемника команд дистанционного управления на то время, пока подаются импульсы на светодиод обнаружителя объектов. Это не оказало бы влияния на работу приемника команд, так как пакеты от передатчика команд дистанционного управления повторяются каждые 50 мс, и, пропустив один из них, мы всегда имеем возможность принять и декодировать следующий.

Окончательный вариант программы содержится в файле Code\Combine\combine.c:

```
#include <pic.h>

// 19 апреля 2002 - объединение функций детектора столкновений
// и приемника команд дистанционного управления.
//
// 15 февраля 2002 - прием команд инфракрасного пульта
// дистанционного управления фирмы Sony и отображение их
// на жидкокристаллическом индикаторе.
//
// Для работы инфракрасного детектора столкновений используется
// ШИМ-сигнал частотой 38 кГц.
//
// Для индикации используется жидкокристаллический дисплей,
// подключенный к микроконтроллеру через двухпроводной интерфейс
// на основе сдвигового регистра 74LS174.
//
// Формат пакета данных:
//
//      Стартовый бит      1      0
//      ---+-----+-----+-----+-----+
//      |         |         |         |         |
//      +-----+-----+-----+-----+
//
//      |  2,4 мс  | 540мкс|660мкс|540мкс|  1,2 мс  |
//
//      |         | 1,2 мс  |  1,76 мс  |
//
//      |   300   |   440   |
//
//
// Замечания по аппаратным средствам:
// Микроконтроллер PIC16F627 работает на частоте 4 МГц.
// Используется внешний тактовый генератор.
//
// Подключение выводов микроконтроллера:
// RB0 - детектор инфракрасного излучения
// (прерывания по положительному перепаду);
// RB1 - линия Clock интерфейса жидкокристаллического дисплея;
// RB2 - линия Data интерфейса жидкокристаллического дисплея.
//
// Глобальные переменные и константы:
int RTC = 0;           // Счетчик реального времени.
```

```

volatile char LCDDlay = 20;           // Длительность задержки.
volatile char LCDState = 1;           // Номер текущего состояния
                                       // конечного автомата.

static volatile bit Clock              @ (unsigned)&PORTB*8+1;
static volatile bit ClockTRIS          @ (unsigned)&TRISB*8+1;
static volatile bit Data               @ (unsigned)&PORTB*8+2;
static volatile bit DataTRIS           @ (unsigned)&TRISB*8+2;

char * MessageOut; // Указатель на строку выводимого сообщения.
volatile char MessageOuti = 0; // Смещение текущего символа в строке.

unsigned int DataIn; // Входной сигнал инфракрасного приемника.
unsigned char DataInCount = 0; // Число импульсов для приема.
unsigned char DataReady = 0; // Число принятых импульсов.
int SaveRTC; // Время последнего прерывания.
int CurrentRTC;

char Message[22] = "\376\0022-> 0b0xxxxxxxxxxxxx!";
                                       // Шаблон для вывода декодированной команды.
char Message1[12] = "\376\300 "; // Очистка экрана.
char Message2[12] = "\376\300COLLISION";

volatile char Collision = 0; // Сначала обнаруженных объектов нет.
volatile char OldCollision = 0;

// Слово конфигурации:
#if defined(_16F627)
    _CONFIG(0x03F61); // Для МК PIC16F627:
                       // внешний тактовый генератор ХТ,
                       // RA6/RA7 используются для ввода-вывода,
                       // внешний сигнал сброса,
                       // таймер PWRT включен,
                       // сторожевой таймер выключен,
                       // защита кода отключена,
                       // детектор BODEN включен.
#else
    (error Unsupported PICmicro MCU selected)
#endif

// Служебные подпрограммы:
LCDNybble(char Nybble, char RS) // Запись полубайта в ЖКИ.

...

LCDByte(char Byte, char RS) // Запись байта в ЖКИ.

...

LCDInit() // Инициализация ЖКИ.

...

LCDOut(char * const LCDString) // Вывод строки на ЖКИ.

...

// Обработчик прерываний:
void interrupt tmr0_int(void)
{

char temp;

```

```

if (TOIF) {           // Обработчик прерываний от таймера TMR0.

    TOIF = 0;         // Сбросить флаг прерываний от таймера TMR0.

    RTC++;           // Инкремент счетчика реального времени.

    // Здесь можно разместить дополнительный код
    // для обработки прерываний от таймера TMR0.

    // Проверка времени последнего вызова обработчика
    // и пропуск текущего пакета при необходимости:
    if (DataInCount != 0) <
        if ((CurrentRTC = (RTC & 0xOFF) -
            ((SaveRTC >> 8) & 0xOFF)) < 0)
            CurrentRTC = 0 - CurrentRTC;
            // Сколько прошло времени?
        if (CurrentRTC > 9) { // Слишком много.
            DataInCount = 0; // Пропустить этот пакет.
            INTE = 0;
            // Запретить прерывания по приходе данных.
        }
    } else if ((!RBO) && (!DataReady)) { // Пришел пакет?
        DataInCount = 13;
            // Ждем конца стартового импульса
            // и еще 12 информационных импульсов.
        SaveRTC = ((RTC & 0xOFF) << 8) + TMR0;
            // Запомнили текущее время.
        INTF = 0; // Сбросить флаг прерывания
        INTE = 1; // и разрешить прерывания по приходе
            // сигнала на вход RBO/INT.

    } else {           // Детектор столкновений.

        TRISB3 = 0;    // Разрешить вывод ШИМ-сигнала.

        while (TMR0 < 64); // В течение 64 "тиков"
            // работы таймера TMR0.

        if (!RBO)       // Если уровень низкий, то сработал
            // детектор столкновений.
            Collision++; // Ждем трех срабатываний подряд.
        else
            Collision = 0;

        TRISB3 = 1;    // Запретить вывод ШИМ-сигнала.

    } // Конец кода для управления детектором столкновений.

    // Конечный автомат для ЖКИ:

    ...

} // Конец обработчика прерываний от таймера.

// Здесь можно разместить другие обработчики прерываний.

if (INTF) {           // Обработчик прерываний по положительному
    // фронту сигнала на выводе RBO/INT.
    CurrentRTC = ((RTC & 0xOFF) << 8) + TMR0;
        // Запомнили текущее время.
    if ((SaveRTC = CurrentRTC - SaveRTC) < 0)

```

```

    SaveRTC = 0 - SaveRTC;
if ((SaveRTC > 250) && (SaveRTC < 350)) {
    DataIn = (DataIn << 1) + 1; // Получена 1.
    if (--DataInCount == 0) {
        DataReady = 1;
        INTE = 0; // Конец, запретить прерывания,
    } // endif
} else if ((SaveRTC > 390) && (SaveRTC < 490)) {
    DataIn = DataIn << 1; // Получен 0.
    if (--DataInCount == 0) {
        DataReady = 1;
        INTE = 0; // Конец, запретить прерывания.
    } // endif
} else if (--DataInCount != 12) { // Ошибка.
    DataInCount = 0;
    INTE = 0; // Конец, запретить прерывания.
} // endif
SaveRTC = CurrentRTC; // Запомнить текущее время.
INTF = 0; // Сбросить флаг прерывания.
} // Конец обработки прерываний по положительному фронту
// сигнала на входе RBO/INT.

```

```

} // Конец обработчика прерываний.

```

```

// Главная программа:

```

```

void main(void)
{
    int i;

```

```

    OPTION = 0x0D1; // Предделитель работает с таймером TMRO,
                    // коэффициент деления равен 4.

```

```

    TMRO = 0; // Начальный сброс таймера TMRO.
    TOIE = 1; // Разрешение прерываний от таймера TMRO.
    GIE = 1; // Разрешение обработки прерываний.

```

```

// Здесь можно проинициализировать другие периферийные устройства.

```

```

LCDInit(); // Инициализация порта, к которому подключен ЖКИ.

```

```

INTEDG = 1; // Прерывания по положительному фронту сигнала
            // на входе RBO/INT.

```

```

CCPR1L = 13; // Ширина импульсов ШИМ составляет 50% периода.
CCPR1CON = 0b0000001111; // Включить модуль ШИМ.

```

```

PR2 = 26; // При частоте ШИМ-сигнала 38 кГц
           // период должен быть равен 26 мкс.

```

```

TMR2 = 0; // Сброс таймера TMR2.
T2CON = 0b0000000100; // Включение таймера TMR2,
                       // коэффициент предделителя 1:1.

```

```

while (1 == 1) { // Бесконечный цикл.

```

```

    // Здесь можно разместить код верхнего уровня.

```

```

    if (DataReady) { // Если данные готовы,
        while (LCDState != 0); // ждать готовности ЖКИ.
        for (i = 0; i < 12; i++) {

```

```

        if. (DataIn & (1 << 11))
            Message[i + 8] = '1';
        else
            Message[i + 8] = '0';
        DataIn = DataIn << 1;
    } // Конец цикла вывода кода принятой команды.
    LCDOut(Message); // Pass String to Output
    DataReady = 0; // Reset the Data Flag

} // Конец оператора if.

if ((Collision == 0) && (OldCollision == 3)) {
    OldCollision = Collision; // Запомнить текущее состояние
                             // детектора столкновений.
    LCDOut(Message1);
}
if ((Collision == 3) && (OldCollision == 0)) {
    OldCollision = Collision; // Запомнить текущее состояние
                             // детектора столкновений.
    LCDOut(Message2);
}

} // Конец оператора while.

} // Конец главной программы.

```

4.26. УЛЬТРАЗВУКОВОЙ ДАЛЬНОМЕР

Мы уже разобрали два метода реализации детектора столкновений: механический контактный и на основе приемника отраженного инфракрасного излучения. В этом разделе будет рассмотрен третий метод обнаружения объектов; он также будет бесконтактным, только на этот раз мы будем использовать ультразвук. Примерно таким же образом летучие мыши ориентируются в пространстве: они излучают вперед направленный пучок звуковых колебаний и ловят отраженный сигнал. Звуковые волны распространяются в воздухе с определенной скоростью, поэтому по задержке прихода отраженного сигнала можно с достаточной степенью точности судить, на каком расстоянии находится тот предмет, который отразил звук.

Оба бесконтактных способа обнаружения объектов (инфракрасный и ультразвуковой) прекрасно дополняют друг друга: предметы, которые не отражают инфракрасные лучи, например пластмассовые детали черного цвета, могут обнаруживаться ультразвуковым детектором, поскольку отражают звуковые волны, и наоборот, материалы, прозрачные для ультразвуковых колебаний (вроде тонкой ткани), могут быть замечены с помощью инфракрасного детектора.

Ультразвуковые дальномеры (*гидролокаторы*, или *сонары*), используемые на морских кораблях и подводных лодках, работают несколько иначе, чем описанный здесь дальномер. Излучатель гидролокатора не имеет узкой диаграммы направленности - он излучает ультразвуковой сигнал во всех направлениях, а для определения направления до ближайшего объекта используется узконаправленный микрофон. При этом расстояние до объекта вычисляется не по задержке прихода отраженного сигнала, а с помощью триангуляции по нескольким порядком сделанным измерениям при различных координатах движущегося корабля.

Дело в том, что скорость звука в воде сильно зависит от ее температуры, плотности, количества находящейся в ней соли и многих других факторов, поэтому не может служить основой для точных измерений.

Для ультразвукового обнаружителя объектов удобно использовать какой-нибудь готовый промышленный прибор, например дальномер фотоаппарата Polaroid 6500. Он обеспечивает довольно небольшой обзор, игнорируя все, что находится вне угла $\pm 5^\circ$. Этот недостаток можно компенсировать, разместив дальномер на управляемом приводе, который мог бы периодически изменять его ориентацию, сканируя пространство необходимой ширины.

Значительно более серьезная проблема состоит в том, что дальномер потребляет много энергии, в результате чего резко снижается срок службы батарей автономного робота. Например, дальномер Polaroid 6500 во время генерации ультразвукового сигнала имеет ток потребления около 1 А. Разумеется, сигнал можно генерировать с перерывами, запасая во время паузы необходимую энергию в конденсаторах, но это только частично решает проблему.

При своей работе ультразвуковой дальномер посылает узконаправленный звуковой импульс и измеряет, с какой задержкой придет отраженный сигнал. В воздухе скорость звука составляет около 1087,4 футов в секунду (то есть примерно 331,4 м/с); значит, дистанцию длиной в один дюйм (2,54 см) туда и обратно звук преодолеет за 153,3 нс. Если, например, объект находится на расстоянии одного сантиметра от ультразвукового детектора, то отраженный сигнал будет обнаружен спустя 60,4 мкс после того, как сработает излучатель. Большинство промышленных дальномеров (в том числе и Polaroid 6500) игнорирует отраженный сигнал, пришедший с задержкой, меньшей 2,76 мс, то есть исходящий от предметов, расположенных на расстоянии менее 18 дюймов (46 см).

Ультразвуковой дальномер Polaroid 6500 выпускается с 1970-х годов. Сначала он был очень дорогим, но сейчас широко доступен и весьма надежен в работе. Поэтому его часто используют радиолюбители в своих проектах.

Дальномер Polaroid 6500 состоит из двух частей: из небольшой печатной платы и круглого черного блока, в котором расположен излучатель. Плата соединена с детектором с помощью провода длиной около 15 дюймов (38 см). На плате расположен 9-контактный разъем, с помощью которого она может сопрягаться с различными устройствами. Назначение контактов этого разъема объясняется в табл. 4.14.

Таблица 4.14. Назначение контактов разъема дальномера Polaroid 6500

Вывод	Имя	Вход/выход	Комментарий
1	Gnd	—	
2	BLNK	Вход	Когда BLNK - 1, любой отраженный сигнал игнорируется
3	—		
4	INIT	Вход	Когда INIT = 1, изучается инфракрасный импульс
5	—		
6	OSC	Выход	Импульсы частотой 49,4 кГц

Таблица 4.14. Назначение контактов разъема дальномера Polaroid 6500 (окончание)

Вывод	Имя	Вход/выход	Комментарий
7	ECHO	Выход	Принятый отраженный сигнал (выход с открытым коллектором; необходимо использовать «подтягивающий» резистор сопротивлением 4,7 кОм)
8	BINH	Вход	Когда BINH = 1 (blank inhibit), выключается игнорирование сигнала, отраженного от близких объектов
9	Vcc	—	

Обычно при конструировании своих устройств я выпаиваю этот разъем и использую многожильный провод для подключения платы дальномера к основной схеме. Во время работы дальномера на линиях **BLNK** и **BINH** поддерживается сигнал низкого уровня. При этом на линии **INIT** вырабатываются прямоугольные импульсы. Во время высокого уровня напряжения сигнала **INIT** излучатель дальномера вырабатывает ультразвуковой сигнал. Отраженный сигнал, принятый приемником, формируется на линии **ECHO**. На рис. 4.62 показаны осциллограммы обоих сигналов. На рисунке этого не видно, но после приема отраженного сигнала на линии **INIT** устанавливается низкий уровень напряжения.

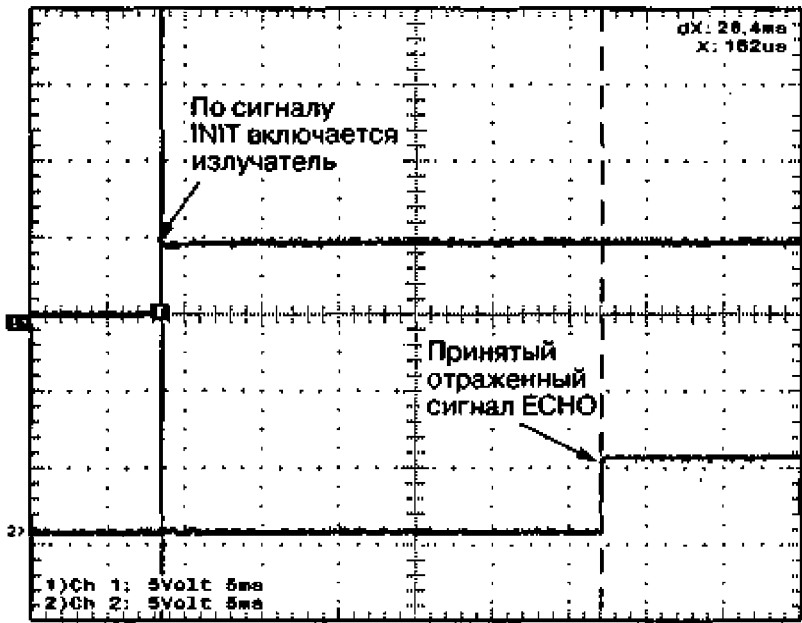


Рис. 4.62. Осциллограммы **излучаемого** и принимаемого сигналов **ультразвукового** дальномера

В документации рекомендовано включать между шинами питания конденсатор емкостью 1000 мкФ для снижения скачков напряжения во время работы излучателя. Следует помнить, что для работы ультразвукового излучателя в схеме вырабатывается достаточно большое напряжение, и если держать модуль излучателя

в руках, то можно получить серьезный удар током. Ни в коем случае не допускайте того, чтобы сразу обе ваши руки контактировали с излучателем или соединительными проводами. По собственному опыту я могу утверждать, что **даже** касание одной рукой не проходит безболезненно.

Максимальное расстояние, на котором дальномер Polaroid 6500 может обнаруживать предметы, равно 35 футам (примерно 10,7 м). Соответствующая задержка прихода отраженного сигнала составляет около 64,4 мс. Это значит, что если отраженный сигнал не обнаружен спустя 65 мс, то можно считать, что в зоне действия ультразвукового детектора нет посторонних объектов.

4.27. ПОДКЛЮЧЕНИЕ УЛЬТРАЗВУКОВОГО ДАЛЬНОМЕРА К МИКРОКОНТРОЛЛЕРУ

В одном из вышеописанных проектов для декодирования команд дистанционного управления мы осуществляли программное измерение длительности импульсов, подаваемых на вход RBO/INT микроконтроллера. Этот же принцип будет использоваться для измерения задержки прихода отраженного сигнала ультразвукового дальномера. Время, прошедшее с момента срабатывания ультразвукового излучателя до регистрации сигнала приемником, будет пересчитываться микроконтроллером в футы и дюймы и выводиться на экран жидкокристаллического дисплея.

Принципиальная схема рассматриваемого устройства показана на рис. 4.63, схема размещения элементов на макетной плате - на рис. 4.64, а список использованных элементов приведен в табл. 4.15.

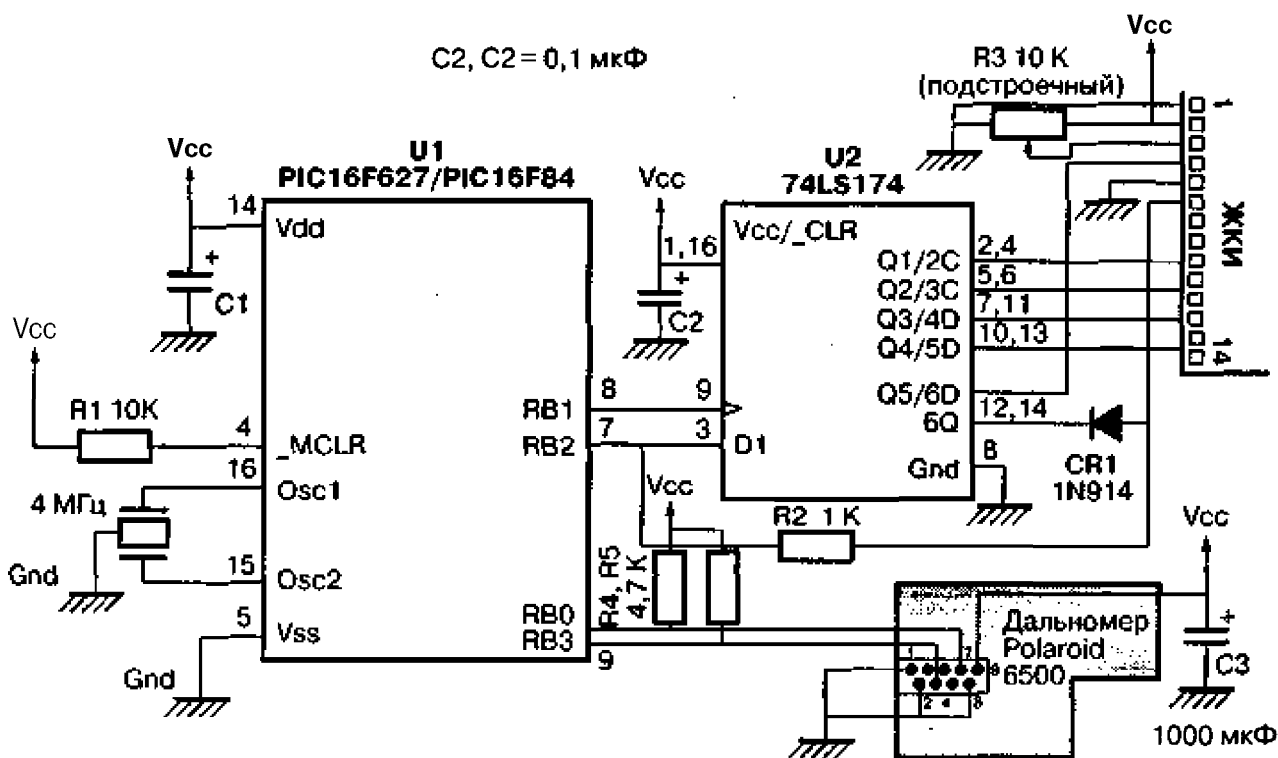


Рис. 4.63. Принципиальная схема ультразвукового дальномера

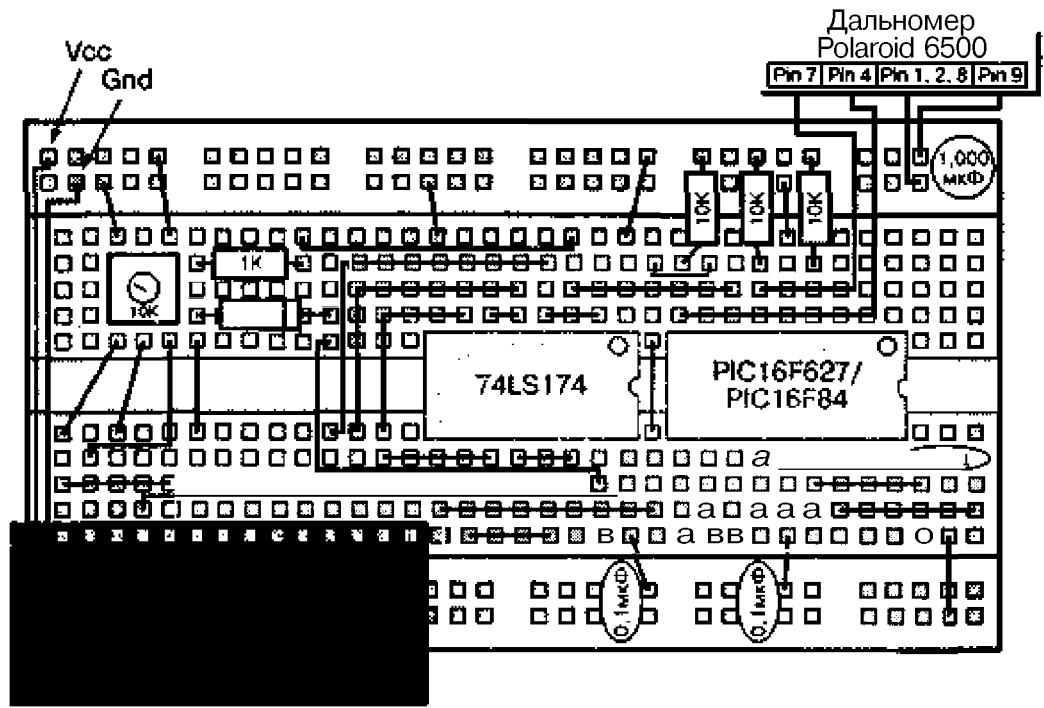


Рис. 4.64. Схема размещения элементов **ультразвукового** дальномера на макетной плате

Таблица 4.15. Список использованных элементов

Обозначение	Элемент	Примечание
U1	PIC16F627 или PIC16F84	Микроконтроллер
U2	74LS174 (K555TM9)*	Сдвиговый регистр
CR1	1N914 (KD521)*	Любой кремниевый диод
R1	10 кОм; 0,25 Вт	Для «подтягивания» напряжения на выводе <u>MCLR</u> до напряжения положительной шины питания
R2	1 кОм; 0,25 Вт	
R3	10 кОм, подстроечный	Для регулирования контрастности
R4, R5	4,7 Ом; 0,25 Вт	
C1, C2	0,1 мкФ	Для фильтрации напряжения питания микросхем
C3	1000 мкФ, 16 В	Оксидный (электролитический) конденсатор
ЖКИ	Жидкокристаллический дисплей /14 выводов), совместимый с Hitachi 44780	
XTAL1	Керамический резонатор на 4 МГц, со встроенными конденсаторами	Для генератора тактовых импульсов микроконтроллера
Polaroid 6500	Ультразвуковой дальномер	
	Материалы	Макетная плата, монтажные провода, источник питания +5 В

* В скобках указаны отечественные аналоги, добавленные при переводе. — Прим. перев.

Так как линии INIT и ECHO дальномера имеют выход с открытым коллектором, то приходится использовать «подтягивающие» резисторы R4 и R5 сопротивлением 4,7 кОм. Для фильтрации бросков напряжения на шинах питания, которые

возникают из-за большого тока потребления ультразвукового излучателя и могут негативно повлиять на работу всех остальных элементов схемы, используется конденсатор СЗ емкостью 1000 мкФ.

В качестве источника питания для этого устройства лучше взять что-нибудь помощнее. Если для всех конструкций, рассмотренных выше, можно было использовать одну 9-вольтовую щелочную батарейку (или набор батареек АД соединенных последовательно), то наша новая схема с таким источником питания, скорее всего, не заработает.

Текст программы можно найти в файле Code\Ultra\ultrac:

```
#include <pic.h>

// 19 апреля 2002 - ультразвуковой дальномер Polaroid 6500.
// Индикация измеренного расстояния на жидкокристаллическом дисплее.
//
// Для индикации используется жидкокристаллический дисплей,
// подключенный к микроконтроллеру через двухпроводной интерфейс
// на основе сдвигового регистра 74LS174.
//
// Используются прерывания от таймера каждую миллисекунду.
//
// Замечания по аппаратным средствам:
// Микроконтроллер PIC16F84/PIC16F627 работает на частоте 4 МГц.
// Используется внешний тактовый генератор.
//
// Подключение выводов микроконтроллера:
// RBO - сигнал ECHO ультразвукового дальномера Polaroid 6500
//      (с "подтягивающим резистором");
// RB1 - линия Clock интерфейса жидкокристаллического дисплея;
// RB2 - линия Data интерфейса жидкокристаллического дисплея;
// RB3 - сигнал INIT ультразвукового дальномера Polaroid 6500.

// Глобальные переменные и константы:
int RTC = 0; // Счетчик реального времени.
volatile char LCDDay = 20; // Длительность задержки.
volatile char LCDState = 1; // Номер текущего состояния.

static volatile bit Clock @ (unsigned)&PORTB*8+1;
static volatile bit ClockTRIS @ (unsigned)&TRISB*8+1;
static volatile bit Data @ (unsigned)&PORTB*8+2;
static volatile bit DataTRIS @ (unsigned)&TRISB*8+2;

char * MessageOut; // Указатель на выводимое сообщение.
volatile char MessageOuti = 0; // Смещение текущего символа в строке.
char Message[9] = "\fxx\` xx\`"; // Формат вывода измеренной дальности.
char Message2[9] = "\fInvalid"; // Сообщение об ошибке.

unsigned int CheckDay = 50; // Пауза 500 мс между моментами
// срабатывания ультразвукового излучателя.
unsigned int PulseStartRTC; // Когда начался излучаемый импульс.
unsigned int PulseEndRTC; // Когда закончился излучаемый импульс.
unsigned char PulseEndTMR0; // Значение счетчика таймера TMR0
// в момент окончания излучаемого импульса.
```

```

unsigned char PulseState = 0;
    // 0 - ожидание перед новым измерением.
    // 1 - измерение задержки прихода отраженного сигнала.
    // 2 - расстояние измерено.
    // 3 - ошибка измерения.

unsigned long PulseTime;        // Задержка в миллисекундах.
unsigned long PulseInches;      // Количество дюймов.
unsigned long PulseFeet;        // Количество футов.

// Слово конфигурации:
#ifdef _16F84
    #warning PIC16F84 selected
    __CONFIG(0x03FF1);          // Для МК PIC16F84:
                                // кварцевый тактовый генератор,
                                // таймер PWRT включен,
                                // сторожевой таймер выключен,
                                // защита кода отключена.
#elif defined(_16F627)
    #warning PIC16F627 with external XT oscillator selected
    __CONFIG(0x03F61);          // Для МК PIC16F627:
                                // внешний тактовый генератор XT,
                                // RA6/RA7 используются для ввода-вывода,
                                // внешний сигнал сброса,
                                // таймер PWRT включен,
                                // сторожевой таймер выключен,
                                // защита кода отключена,
                                // детектор BODEN включен.
#else
    #error Unsupported PICmicro MCU selected
#endif

// Служебные подпрограммы:
LCDNybble(char Nybble, char RS)    // Запись полубайта в ЖКИ.
...

LCDByte(char Byte, char RS)         // Запись байта в ЖКИ.
...

LCDInit()                          // Инициализация ЖКИ.
...

LCDOut(char * const LCDString)      // Вывод строки на ЖКИ.
...

// Обработчик прерываний:
void interrupt tmr0_int(void)
{
    char temp;

    if (TOIF) {                    // Обработчик прерываний от таймера TMR0.
        TOIF = 0;                 // Сбросить флаг прерываний от таймера TMR0.
        RTC++;                    // Инкремент счетчика реального времени.
    }
}

```

```
// Здесь можно разместить дополнительный код
// для обработки прерываний от таймера TMRO.

// Работа с ультразвуковым дальномером:

if (--CheckDlay == 0) {           // Пора выдать импульс
                                   // на ультразвуковой излучатель.
    PulseStartRTC = RTC;
    PulseState = 1;               // Измерение расстояния.
    RB3 = 1;                      // Подать сигнал ШТ.
    INTF = 0;                    // Сбросить флаг прерывания.
    INTE = 1;                    // Разрешить прерывания
                                   // по приходе сигнала на вход RBO/INT.
    CheckDlay = 500;              // Ждать 500 мс.
    if (RTC > 64900)              // Увеличить задержку
        CheckDlay += 200;        // на 200 мс.
}

if ((PulseState == 1) && ((PulseStartRTC + 60) < RTC)) {
    RB3 = 0;                      // Неправильная задержка,
    INTF = 0;                    // выключить дальномер.
    INTE = 0;
    PulseState = 3;
}

// Конец кода для работы с дальномером.

// Конечный автомат для ЖКИ:

...

Конец обработчика прерываний от таймера.

ь можно разместить другие обработчики прерываний.

F) {                               // Обработчик прерываний
    // по фронту сигнала на выводе RBO/INT.
    PulseEndTMRO = TMRO; // Запомнили момент окончания импульса.
    PulseEndRTC = RTC;
    RB3 = 0;                      // Сброс сигнала INIT дальномера.
    PulseState = 2;
    INTF = 0;                    // Сбросили флаг прерывания.
    INTE = 0;                    // Запретили прерывания по входу RBO/INT.

    // Конец обработки прерываний по фронту сигнала
    // на входе RBO/INT.

д обработчика прерываний.

программа:
void)

nt temp; // Служебная переменная.

= 0x0D1; // Предделитель работает с таймером TMRO,
         // коэффициент деления равен 4.

0; // Начальный сброс таймера TMRO.
1; // Разрешение прерываний от таймера TMRO.
; // Разрешение обработки прерываний.
```

```

LCDInit();      // Инициализация порта, к которому подключен ЖКИ.

// Здесь можно проинициализировать другие периферийные устройства.

RB3 = 0;        // Вывод RB3 перевели в режим выходного
TRISB3 = 0;     // и вывели сигнал низкого уровня.

while (1 == 1) {    // Бесконечный цикл.

    // Здесь можно разместить код верхнего уровня.

    switch(PulseState) { // В зависимости от состояния.
        case 2:         // Вывод результата измерения.
            PulseTime = (((long) PulseEndRTC * 256) +
                          (long) PulseEndTMRO) * 40;
            PulseTime -= ((long) PulseStartRTC *
                          (256 * 40));

            PulseInches = PulseTime / 1533;
            PulseFeet = PulseInches / 12;
            PulseInches = PulseInches % 12;
            if ((temp = PulseFeet / 10) == 0)
                Message[1] = ' ';
                // Расстояние меньше 10 футов,
                // отображаем пробел
                // вместо первой цифры.
            else
                Message[1] = temp + '0';
            Message[2] = (PulseFeet % 10) + '0';
            if ((temp = PulseInches / 10) == 0)
                Message[5] = ' ';
            else
                Message[5] = temp + '0';
            Message[6] = (PulseInches % 10) + '0';
            LCDOut(Message);
            PulseState = 0; // Ждать следующего импульса.
            break;

        case 3:         // Ошибка.
            LCDOut(Message2);
            PulseState = 0; // Ждать следующего импульса.
            break;

    } // Конец оператора switch.

} // Конец оператора while.

} // Конец главной программы.

```

Здесь после включения питания и небольшой задержки (около 50 мс) начинается процесс измерения расстояния до ближайшего объекта в поле зрения детектора. Если объект не найден, то есть находится слишком близко (на расстоянии менее 46 см) или слишком далеко (дальше 11 м), то на жидкокристаллическом дисплее отображается сообщение INVALID (ошибка). Если расстояние до объекта лежит в пределах рабочего диапазона дальномера, то на дисплей выводится измеренная дальность в футах и дюймах.

Для перевода длительности задержки отраженного сигнала в единицы длины полученное количество микросекунд умножается на 10 и делится на 1533. Это эквивалентно делению на $153,3^1$, но не требует использования вещественной

арифметики. После того как вычислено расстояние в дюймах, полученное значение делится на 12^2 ; целая часть результата дает необходимое количество футов. Результат вычислений высвечивается на жидкокристаллическом дисплее.

Перед загрузкой этой программы в микроконтроллер желательно выполнить моделирование с помощью симулятора MPLAB. Для имитации входных воздействий я подготовил файл Code\Ultra\ultrasti:

```
! Моделирование сигнала, принимаемого ультразвуковым датчиком.
!
! Предполагаем, что излучатель посылает импульс
! через 51200 мкс после включения питания.
! Через 23148,3 мкс после этого приходит отраженный сигнал,
! что соответствует расстоянию 12' 7" (12 футов 7 дюймов).
!
! Момент прихода отраженного сигнала: 51200 + 23148 = 74348.
!
! Приемник датчика подключен к выводу RB0 микроконтроллера.
!
! Майк Предко
! 19 апреля 2002 года
!
```

Step	RB0	RB4	RB5	RB6	RB7	
1	0	0	0	0	0	! Сначала все ! уровни низкие.
15000	0	0	0	0	1	
30000	0	0	0	1	1	
45000	0	0	1	1	1	
60000	0	1	1	1	1	
74348	1	1	1	1	1	! Пришел передний фронт ! отраженного импульса.
75000	0	1	1	1	0	! Задний фронт ! отраженного импульса.

Здесь мы моделируем расстояние до объекта, равное 12 футам 7 дюймам (3,84 м). В файле стимулов указываются только входные сигналы; здесь не определяется значение выхода RB3, к которому подключена линия INIT датчика. Состояние этой линии определяется программой, а не файлом стимулов. Мы предполагаем, что программа устанавливает на линии INIT сигнал высокого уровня через 51200 мкс после начала моделирования. Спустя 23148 мкс после этого приходит отраженный сигнал, то есть мы подаем единичный уровень на вход RB0 микроконтроллера.

Внимательный читатель заметит, что входы RB4, RB5, RB6 и RB7 в схеме не используются, поэтому их включение в файл стимулов может вызвать недоумение. Дело в том, что я (как, вероятно, и кто-нибудь из вас) часто терял много времени, выполняя пошаговый прогон программы, прежде чем обнаружить, что забыл в самом начале подключить файл стимулов. Тогда все приходится начинать заново.

¹ То есть на количество микросекунд, которое требуется звуку для преодоления расстояния длиной один дюйм (туда и обратно). - Прим. перев.

² В одном футе двенадцать дюймов. - Прим. перев.

Если указать в файле стимулов неиспользуемые входы и изменять их состояние с самых первых шагов, то уже в начале моделирования можно убедиться в том, что входные воздействия подаются правильно.

Разумеется, вы можете не использовать данный прием - в этом случае ваш файл стимулов для отладки описанной программы будет состоять всего из двух столбцов (номера шага и соответствующих значений на входе RBO).

Напомню, что для подключения файла стимулов к проекту надо выполнить команду главного меню **Debug ⇒ Simulator Stimulus ⇒ Pin Stimulus ⇒ Enable** (Отладка ⇒ Входные воздействия ⇒ Файл стимулов ⇒ Подключить). В папке **Code\Ultra** следует выбрать файл **ultra.sti** и нажать кнопку **ОК**. После этого можно загрузить конфигурацию окна просмотра **Watch**, указав файл **P16F627.wat**.

Установите точку останова в первой строке ветви **case 2** оператора выбора:

```
PulseTime = (((long) PulseEndATC » 256) + (long) PulseEndTMR0) • 40;
```

Нажмите клавишу **F6**, чтобы подать сигнал сброса микроконтроллера, и запустите программу нажатием клавиши **F9**.

После того как сработает точка останова, выполните команду меню **Window ⇒ File Registers** (Окно ⇒ Регистровый файл), чтобы вывести на Рабочий стол **MPLAB** окно, в котором отображается содержимое регистров микроконтроллера. Чтобы легче было найти строку, где хранится текст выводимого на дисплей сообщения, надо переключить это окно в режим **ASCII**. Для этого щелкните по пиктограмме в левом верхнем углу окна **File Register Window** и выберите пункт **ASCII Display** (Текстовый формат).

Теперь выполните в пошаговом режиме (клавиша **F7**) еще несколько строк программы, пока выполнение не дойдет до строки

```
LCDOut(Message);
```

Постепенно вы сможете видеть, как будет заполняться строка сообщения. Обратите внимание, как медленно продвигается процесс моделирования. Каждая строка программы, содержащая оператор умножения или деления, выполняется несколько миллисекунд, не раз приостанавливаясь из-за процедуры обработки прерываний.

Вот почему вся обработка информации, достаточно сложная, должна проводиться в главной программе, а не в процедуре обработки прерываний. Если бы мы выполняли вычисления в обработчике прерываний, то многие запросы на прерывание были бы пропущены, так как во время работы обработчика другие прерывания запрещены.

Кроме того, при пошаговой отладке вы должны заметить, что расстояние вычисляется неправильно. Вместо ожидаемой нами строки

```
12' 7"
```

в окне **File Register Window** мы увидим

```
12' 1"
```

Причиной ошибки послужила небрежность, допущенная мною при составлении файла стимулов.

Выполните сброс (клавиша F6) и поставьте точку останова в строке

```
RB3 = 1;
```

обработчика прерываний. Если теперь запустить выполнение программы клавишей F9, то можно убедиться, что включение излучателя дальномера произойдет на 52037-м командном цикле, а не на 51200-м, как ожидалось. Лишние 837 циклов задержки соответствуют ошибке в 5,46 дюймов (13,87 см) при измерении дальности. При вычитании из 7 числа 5,46, округляя до целых, как раз получим значение 1, которое выдает наша программа.

Если мы теперь исправим файл стимулов так, чтобы отраженный сигнал появлялся не на 73248-м, а на 75185-м шаге моделирования, то увидим правильное значение 12' 7".

Советую вам вернуться к предыдущим проектам и создать похожий файл стимулов для моделирования приложений whisker.c и remote.c.

Возможно, кому-то процесс моделирования покажется скучным, но уверяю вас, что прогон программы в симуляторе сэкономит многие часы, которые вам пришлось бы провести, безуспешно пытаясь найти ошибку в неправильно работающем устройстве.

4.28. СВЕТОВЫЕ ДАТЧИКИ

Не так уж сложно наделить нашего робота элементарной способностью реагировать на свет. Разумеется, пока речь не идет о распознавании изображений, но мы без труда можем добавить к описанным выше конструкциям несколько световых датчиков и так изменить управляющую программу, чтобы робот двигался в сторону источника света, подобно тому как это делают насекомые.

Обычно для этого в передней части корпуса под углом 45° к продольной оси робота устанавливают два фотодиода, каждый из которых будет иметь свое поле обзора, как показано на рис. 4.65. Такое расположение световых датчиков позволит легко определить направление до источника света.

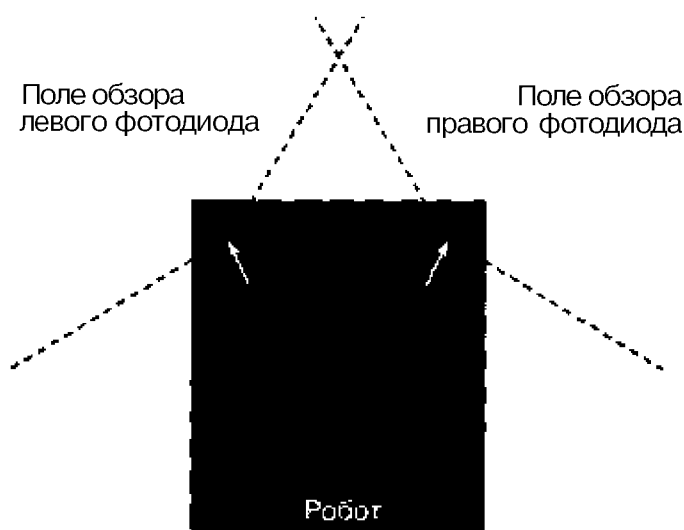


Рис. 4.65. Границы поля обзора левого и правого световых датчиков

Разумеется, можно не ограничиваться двумя фотодиодами спереди, а расположить их по всему периметру, чтобы робот мог реагировать и на те источники света, которые расположены сзади.

В качестве датчиков света подойдут не только фотодиоды, но также фототранзисторы или фоторезисторы из сульфида кадмия.

Принцип работы датчиков света основан на том, что сопротивление фоторезистора или фотодиода в обратном включении (либо коллектор-эмиттерное сопротивление фототранзистора) изменяется под действием света: оно уменьшается при увеличении освещенности. Что касается фотодиода, он может использоваться и в качестве фотоэлемента, то есть служить источником фототока, сила которого тем больше, чем ярче освещен фотодиод. Сила фототока очень мала - она должна быть увеличена с помощью специальной электронной схемы. При работе с фототранзисторами часто используют включение по схеме Дарлингтона, которое позволяет усилить сигнал.

Фоторезисторы обычно весьма инерционны (имеют время реакции порядка десятков и даже сотен миллисекунд), поэтому они не могут быть использованы для регистрации быстрых изменений освещенности, например для приема сигналов дистанционного управления. Даже, возможно, придется уменьшить скорость движения робота до минимума, чтобы он успевал сканировать все окружающее пространство в поисках источника света.

Тем не менее фоторезисторы из сульфида кадмия весьма популярны в радиолюбительских конструкциях; так, датчики на их основе обладают хорошей чувствительностью, то есть имеют максимальное отношение изменения тока через резистор к соответствующему изменению освещенности.

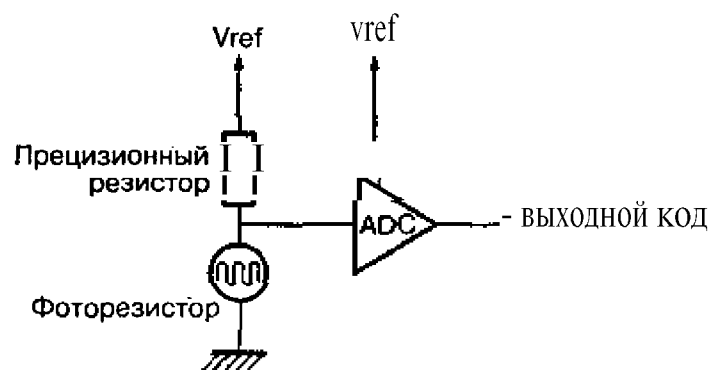


Рис. 4.66. Измерение сопротивления фоторезистора с помощью АЦП

Все три описанных типа световых датчиков формируют аналоговый сигнал. Но микроконтроллер работает только с сигналами, представленными в цифровой форме. Для того чтобы ввести в микроконтроллер сигнал, снимаемый с аналогового светового датчика, можно использовать *аналого-цифровой преобразователь* (АЦП). Схема подключения показана на рис. 4.66. Здесь фоторезистор является нижним плечом делителя напряжения источника питания, а в качестве верхнего плеча используется прецизионный резистор. Напряжение в средней точке делителя преобразуется с помощью АЦП в цифровой код и подается на вход микроконтроллера. Для работы АЦП обычно требуется высокостабильный источник опорного напряжения Vref. Сопротивление прецизионного резистора должно быть равно наибольшему возможному сопротивлению фоторезистора.

Если необходимо, чтобы выходной код АЦП увеличивался при возрастании освещенности фоторезистора, надо поменять местами плечи делителя, то есть переставить фоторезистор и прецизионный резистор.

Другой способ измерения уровня яркости не требует использования АЦП (рис. 4.67). Здесь вывод микроконтроллера подключен к средней точке делителя напряжения, нижнее плечо которого, как и в предыдущей схеме, образовано сопротивлением фоторезистора, а в верхнем плече установлен конденсатор. Сначала вывод микроконтроллера переводится в режим выходного и на него подается импульс напряжения; в результате происходит разряд конденсатора. После этого вывод микроконтроллера переключается на ввод сигнала, а конденсатор начинает снова заряжаться, напряжение на нем увеличивается. Как следствие, напряжение в средней точке делителя уменьшается. Когда оно достигнет порога переключения, на входе микроконтроллера будет зафиксирован сигнал низкого уровня. Время разряда конденсатора тем больше, чем меньше сопротивление фоторезистора, то есть чем меньше он освещен.

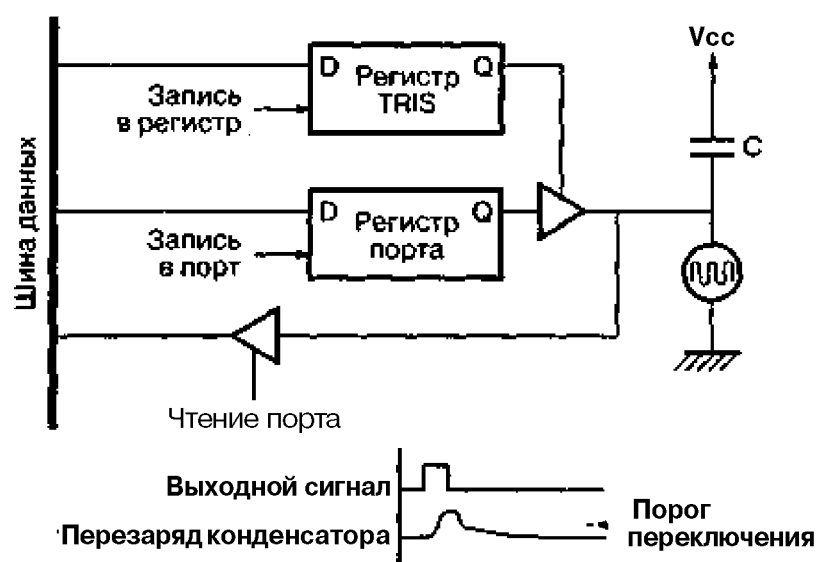


Рис. 4.67. Измерение сопротивления фоторезистора с помощью RC-цепочки

Для вычисления времени, в течение которого напряжение на входе микроконтроллера падает от напряжения питания делителя V_{cc} до напряжения переключения, можно использовать приближенную формулу

$$T = 2,2 \times R \times C.$$

Эта формула позволяет лишь оценить ожидаемое время разряда конденсатора по порядку величины - реальное время зависит не только от емкости конденсатора, но и от напряжения питания и характеристик входных цепей микроконтроллера, поэтому для более точных измерений приходится проводить индивидуальную градуировку.

Наконец, третий метод ввода аналогового сигнала основан на использовании встроенного в микроконтроллер аналогового компаратора (рис. 4.68). Здесь оба плеча делителя напряжения образованы сопротивлениями фоторезисторов, а сигнал с общей точки их соединения подается на вход микроконтроллера, где сравнивается компаратором с опорным значением V_{ref} . Если напряжение, снимаемое с делителя, больше опорного (это говорит о том, что верхний фоторезистор освещен

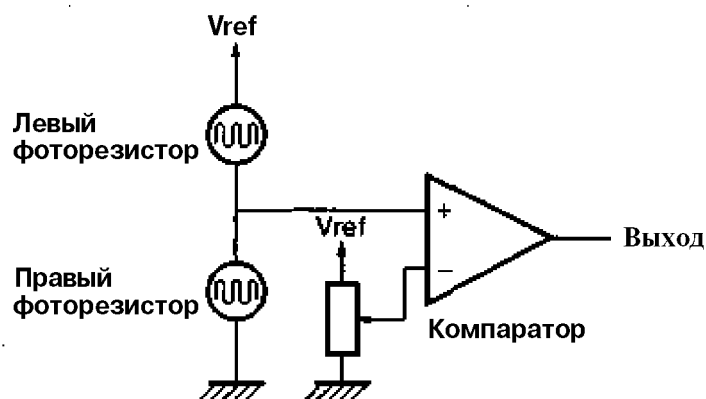


Рис. 4.68. Измерение сопротивления фоторезисторов с помощью компаратора

меньше, чем нижний), то на выходе аналогового компаратора будет сигнал высокого уровня, а в противном случае - сигнал низкого уровня.

Последний метод очень прост, но менее информативен, поскольку позволяет только определить, какой световой датчик освещен больше, а какой меньше, в то время как предыдущие два метода действительно обеспечивали измерение уровня освещенности.

Следует помнить, что все три метода ввода аналоговых сигналов имеют низкую помехоустойчивость. Может даже потребоваться выключать двигатели на то время, пока длится опрос световых датчиков.

Если вы хотите, чтобы ваш робот с высокой достоверностью определял направление до источника света в любой обстановке, то, вероятно, придется использовать небольшую телекамеру. Ее видеосигнал можно будет подвергнуть несложной обработке, чтобы найти наиболее освещенный участок комнаты.

4.29. ПОДКЛЮЧЕНИЕ СВЕТОВЫХ ДАТЧИКОВ К МИКРОКОНТРОЛЛЕРУ

В предыдущем разделе мы познакомились с тремя основными методами подключения световых датчиков к микроконтроллеру. Каждый из них имеет свои преимущества и недостатки; мы еще обсудим их, но сначала рассмотрим несколько конкретных примеров.

Обычно используется дифференциальная схема на двух фоторезисторах, включенных по схеме делителя напряжения (как на рис. 4.68). Когда оба фоторезистора освещены одинаково, то их сопротивления примерно равны, и в средней точке делителя напряжение равно половине напряжения питания. Если верхний фоторезистор освещен больше, чем нижний, то его сопротивление меньше; в этом случае напряжение, снимаемое с делителя, больше половины напряжения питания. Наоборот, если больше освещенность нижнего резистора, то напряжение в средней точке делителя меньше половины напряжения питания.

Чтобы узнать, какой из двух фоторезисторов освещен больше, достаточно использовать аналоговый компаратор, один вход которого должен быть подключен к выходу делителя, а на второй следует подать напряжение, равное половине

напряжения питания делителя. Как мы уже говорили в главе 3, микроконтроллер PIC16F627 имеет два встроенных аналоговых компаратора. Поэтому для нашего первого примера выберем именно этот способ подключения датчиков.

Принципиальная схема показана на рис. 4.69. Здесь мы будем формировать опорное напряжение для компаратора с помощью средств, встроенных в микроконтроллер. Мы должны запрограммировать внутренний источник, чтобы его напряжение равнялось половине напряжения питания микроконтроллера. Для проверки это напряжение подается на вывод RA2.

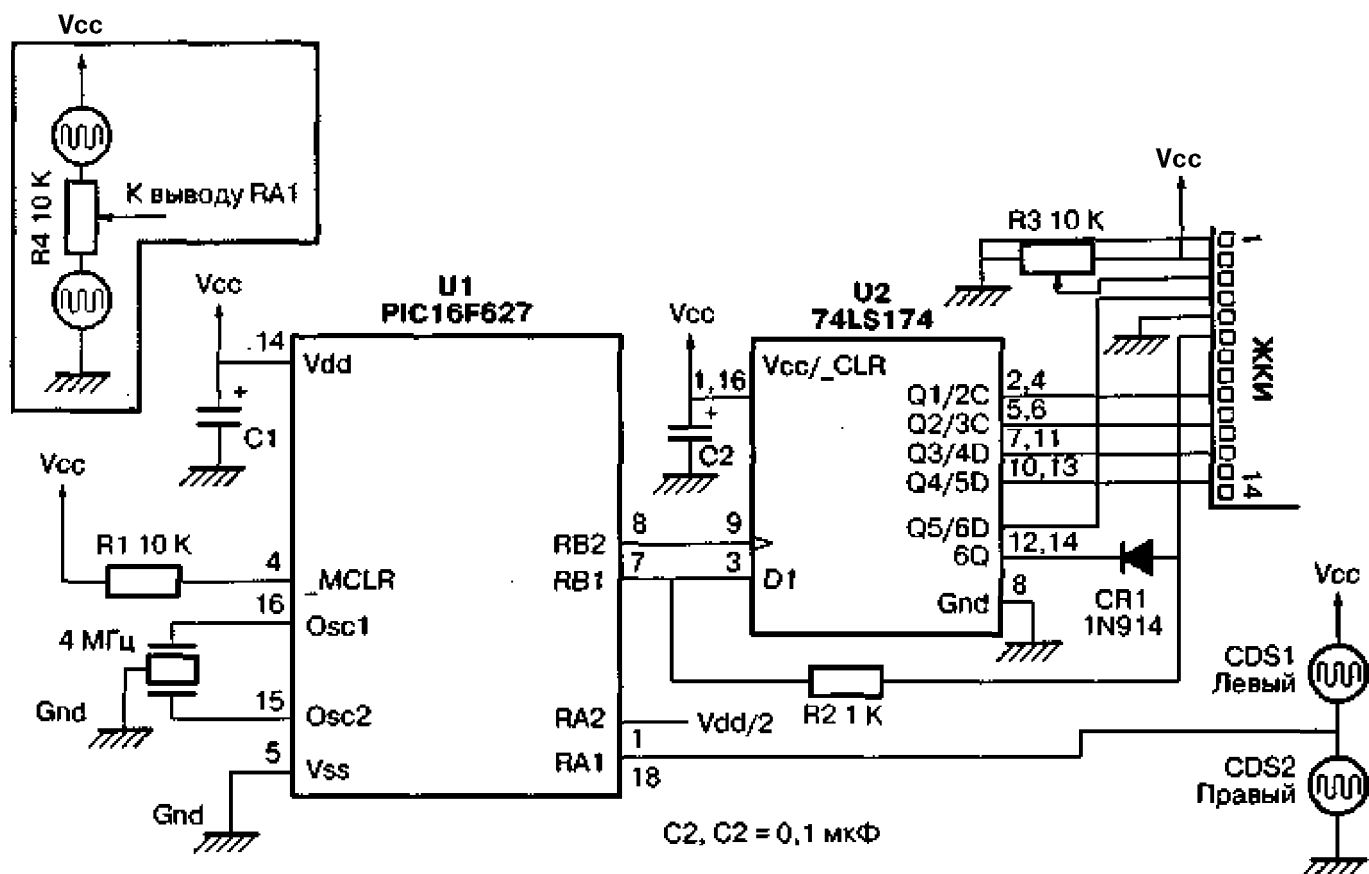


Рис. 4.69. Схема подключения фоторезисторов к микроконтроллеру

На рис. 4.70 показана схема размещения элементов на макетной плате. По сравнению с предыдущими конструкциями почти ничего не изменилось, только добавлены два фоторезистора. На рис. 4.70 не показаны длинные провода, используемые для подключения фоторезисторов. Возможно, для проверки работы нашего устройства вы разместите их прямо на макетной плате, но в работающем устройстве они должны быть вынесены на переднюю панель корпуса робота, как можно дальше друг от друга.

Чтобы в средней точке делителя, образованного сопротивлениями этих фоторезисторов, напряжение в точности равнялось половине напряжения питания, желательно использовать еще подстроечный резистор R4, с движка которого можно снимать выходное напряжение делителя (как показано на врезке к рис. 4.69, в левом верхнем углу).

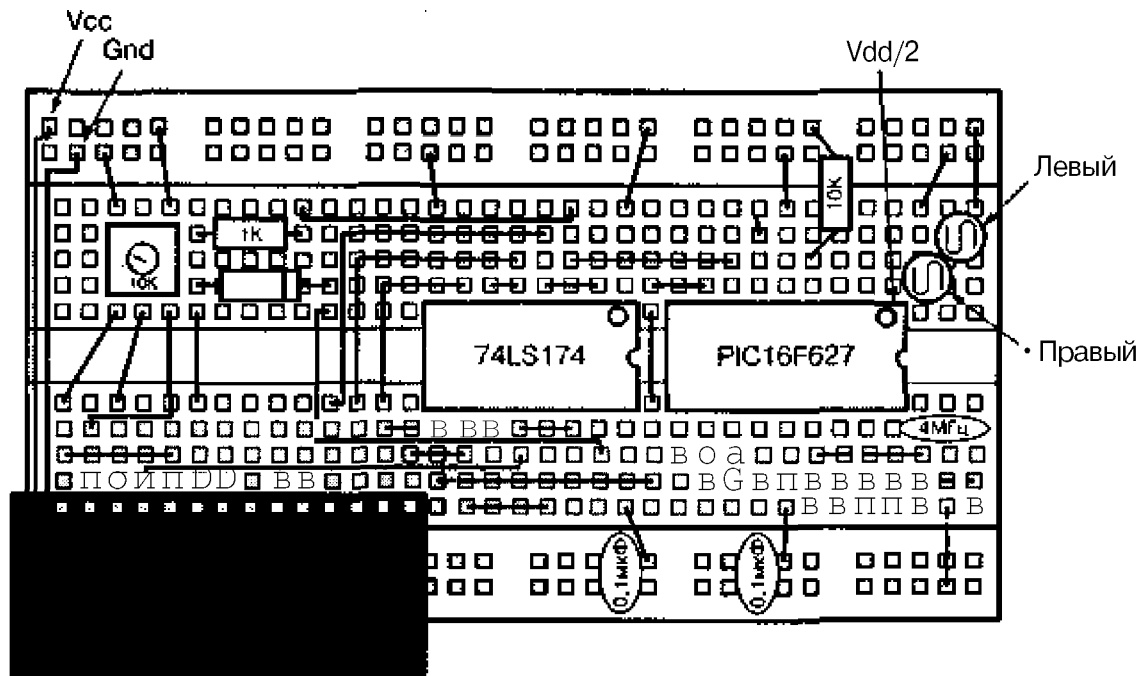


Рис. 4.70. Схема размещения элементов на макетной плате

Перечень использованных в схеме элементов приведен в табл. 4.16. Некоторые указанные там элементы потребуются нам только для второго примера.

Таблица 4.16. Перечень использованных элементов

Обозначение	Элемент	Примечание
U1	PIC16F627	Микроконтроллер
U2	74LS174 (K555TM9)*	Сдвиговый регистр
CR1	1N914 (КД521)*	Любой кремниевый диод
R1	10 кОм; 0,25 Вт	«Подтягивающий» резистор
R2	1 кОм; 0,25 Вт	
R3	10 кОм, подстроечный	Для регулирования контрастности дисплея и яркости светодиода
R4	10 кОм, подстроечный	Можно исключить
R5, R6	470 Ом; 0,25 Вт	Используются только во второй схеме
CDS1, CDS2	Фоторезистор, 10 кОм при нулевом освещении	
C1, C2	0,1 мкФ	Для фильтрации напряжения питания микросхем
C3, C4	0,01 мкФ	Используются только во второй схеме
ЖКИ	Жидкокристаллический дисплей (14 выводов), совместимый с Hitachi 44780	
XTAL1	Керамический резонатор на 4 МГц, со встроенными конденсаторами	Для генератора тактовых импульсов микроконтроллера
Материалы		Макетная плата, монтажные провода, источник питания +5 В

* В скобках указаны отечественные аналоги, добавленные при переводе. - Прим. перев.

Текст программы можно найти в файле `Code\Light\light1.c`:

```
#include <pic.h>

// 24 апреля 2002 - два фоторезистора, дифференциальная схема включения.
//
// Для индикации используется жидкокристаллический дисплей,
// подключенный к микроконтроллеру через двухпроводной интерфейс
// на основе сдвигового регистра 74LS174.
//
// Используются прерывания от таймера каждую миллисекунду.
//
// Замечания по аппаратным средствам:
// Микроконтроллер PIC16F627 работает на частоте 4 МГц.
// Используется внешний тактовый генератор.
//
// Подключение выводов микроконтроллера
// (в электронной версии программы здесь перепутаны
// обозначения выводов. - Прим. перев.):
// RA1 - инверсный вход компаратора 2 - подключен к выходу делителя
//       напряжения, вернее плечо которого образовано левым
//       фоторезистором, а нижнее - правым;
// RA0 - инверсный вход компаратора 1 - подключен к общему проводу Vss;
// RA2 - выход, на котором можно контролировать правильность
//       формирования напряжения  $V_{dd} / 2$ ;
// RB1 - линия Clock интерфейса жидкокристаллического дисплея;
// RB2 - линия Data интерфейса жидкокристаллического дисплея.

// Глобальные переменные и константы:
int RTC = 0; // Счетчик реального времени.
volatile char LCDD1ay = 20; // Длительность задержки.
volatile char LCDState = 1; // Номер текущего состояния.

static volatile bit Clock @ (unsigned)&PORTB*8+1;
static volatile bit ClockTRIS @ (unsigned)&TRISB*8+1;
static volatile bit Data @ (unsigned)&PORTB*8+2;
static volatile bit DataTRIS @ (unsigned)&TRISB*8+2;

char * MessageOut; // Указатель на выводимое сообщение.
volatile char MessageOuti = 0; // Смещение текущего символа в строке.
char Message1[3] = "\f*"; // Звездочка выводится слева.
char Message2[18] = "\f *"; // Звездочка выводится справа.
// 1 2345678901234567

// Слово конфигурации:
#if defined(_16F627)
#warning PIC16F627 with external XT oscillator selected
__CONFIG(0x03F61); // Для МК PIC16F627:
// внешний тактовый генератор XT,
// RA6/RA7 используются для ввода-вывода,
// внешний сигнал сброса,
// таймер PWRT включен,
```

Электронное приложение к этой книге содержит тексты программ в том виде, в котором они представлены в оригинале книги. Обратите внимание на то, что электронная версия данной программы несколько отличается от печатного варианта. Обе версии работоспособны, хотя в оригинале имеется ошибка в комментариях. - *Прим. перев.*


```

TOIE = 1;           // Разрешение прерываний от таймера TMRO.
GIE = 1;           // Разрешение обработки прерываний.

LCDInit();         // Инициализация порта, к которому подключен ЖКИ.

// Здесь можно проинициализировать другие периферийные устройства.

CMCON = 0x002;      // Разрешение работы компараторов.
                   // Выходы не инвертируются.
                   // Бит переключения входов CIS = 0.
                   // Режим 2 (с внутренним источником
                   // опорного напряжения).

TRISA = 0x003;      // Входными являются только выводы RA0 и RA1.
                   // Обращаем внимание читателя на еще одну ошибку в электронном
                   // варианте программы; она приводит к тому, что невозможно
                   // проконтролировать величину опорного напряжения
                   // на выводе RA2. - Прим. перев.

VRCON = 0x0EC;      // Разрешение работы внутреннего источника
                   // опорного напряжения Vref;
                   // выход Vref подключен к выводу RA2;
                   // верхний поддиапазон;
                   // биты задания величины напряжения; 12.

while (1 == 1) {    // Бесконечный цикл.

    // Здесь можно разместить код верхнего уровня.

    if (!ADCPoll()) // Каково состояние бита 7
                   // регистра CMCON?
        LCDOut(Message1); // Источник света
                           // расположен слева.
    else
        LCDOut(Message2); // Источник света
                           // расположен справа.

} // Конец оператора while.

} // Конец главной программы.

```

Здесь после инициализации жидкокристаллического дисплея в бесконечном цикле производится опрос состояния компаратора. Если левый (верхний по схеме на рис. 4.69) фоторезистор освещен больше правого, то напряжение на выходе делителя больше половины напряжения питания. Так как выход делителя подключен к инвертирующему входу компаратора, а инвертирование выхода отключено, то на выходе C2OUT при этом будет напряжение низкого уровня (логический 0), а следовательно, на дисплей будет выводиться сообщение Message1 (символ звездочки слева). Если источник света, наоборот, больше освещает правый фоторезистор, то будет выведено сообщение Message2 (символ звездочки справа).

Аналоговый компаратор работает непрерывно, для опроса его состояния нет надобности выполнять какую-либо последовательность действий, как часто бывает в цифровых схемах. Поэтому программа получилась достаточно простой.

Вывод курсора на экран не разрешается нигде. По большому счету, необходимость в этом возникает только в тех приложениях, которые требуют, чтобы

пользователь вводил какую-либо информацию. В случае, если дисплей нужен только для вывода данных, мигающий знак подчеркивания обычно только раздражает.

Недостаток схемы заключается в том, что робот получает только один бит информации: какой из двух фоторезисторов освещен больше. Мы не можем измерить, насколько больше, и выяснить, каков общий уровень освещенности. В ситуации, когда один фоторезистор залеплен непрозрачным материалом, а на другой направлен свет яркой лампы, робот будет реагировать так же, как если бы использовались два источника света, один из которых светит чуть ярче другого. Поведение робота не изменится, если в комнате будет полумрак и только в одном ее углу будет чуть мерцать свеча.

Другой недостаток, характерный для всех **двухпозиционных** автоматических устройств, заключается в том, что наш робот на самом деле не будет двигаться в сторону источника света по прямой траектории. Он будет вилять, то резко забирая вправо, то снова возвращаясь влево. Амплитуда колебаний может оказаться настолько **велика**¹, что робот так и не сможет приблизиться к источнику света из-за того, что после очередного резкого поворота этот источник окажется вне поля обзора обоих фоторезисторов.

Кроме того, может оказаться, что из-за разброса параметров фоторезисторов при равной их освещенности выходное напряжение делителя не равно половине напряжения питания. В результате колебательная траектория движения робота вместо того, чтобы стремиться точно к источнику света, будет направлена к некоторой точке в стороне от него. Но эта проблема, в отличие от предыдущих, легко решается путем добавления подстроечного резистора. Точной его регулировкой можно добиться, чтобы на дисплее мигали обе звездочки, если робот ориентирован в направлении источника света.

Если мы хотим, чтобы устройство действительно измеряло уровень освещенности, придется усложнить схему и управляющую программу. Микроконтроллер PIC16F627 не имеет встроенного АЦП, так что мы будем измерять сопротивление фоторезистора с помощью RC-цепочки, время разряда которой зависит от сопротивления R (см. рис. 4.67). Чем больше освещен фоторезистор, тем меньше его сопротивление и тем быстрее разряжается конденсатор.

Принципиальная схема второго устройства приведена на рис. 4.71. По сравнению с предыдущей здесь добавлены два резистора ($R5$ и $R6$) и конденсатора ($C3$ и $C4$). На рис. 4.72 показана схема размещения элементов на макетной плате. Перечень использованных элементов уже приводился в табл. 4.16.

Конечно, было бы интересно разобрать еще одну схему на основе АЦП, но на нашей макетной плате уже не осталось места для аналого-цифрового преобразователя. К тому же программа для работы с АЦП не слишком-то отличается от той, что мы неоднократно использовали для вывода информации на экран жидкокристаллического дисплея. Хотя я потратил немного времени на эксперименты

¹ Даже если читатель не изучал теорию автоматического управления и **методы** исследования систем на устойчивость, он может снизить скорость вращения двигателя с целью уменьшения амплитуды колебаний. - *Прим. перев.*

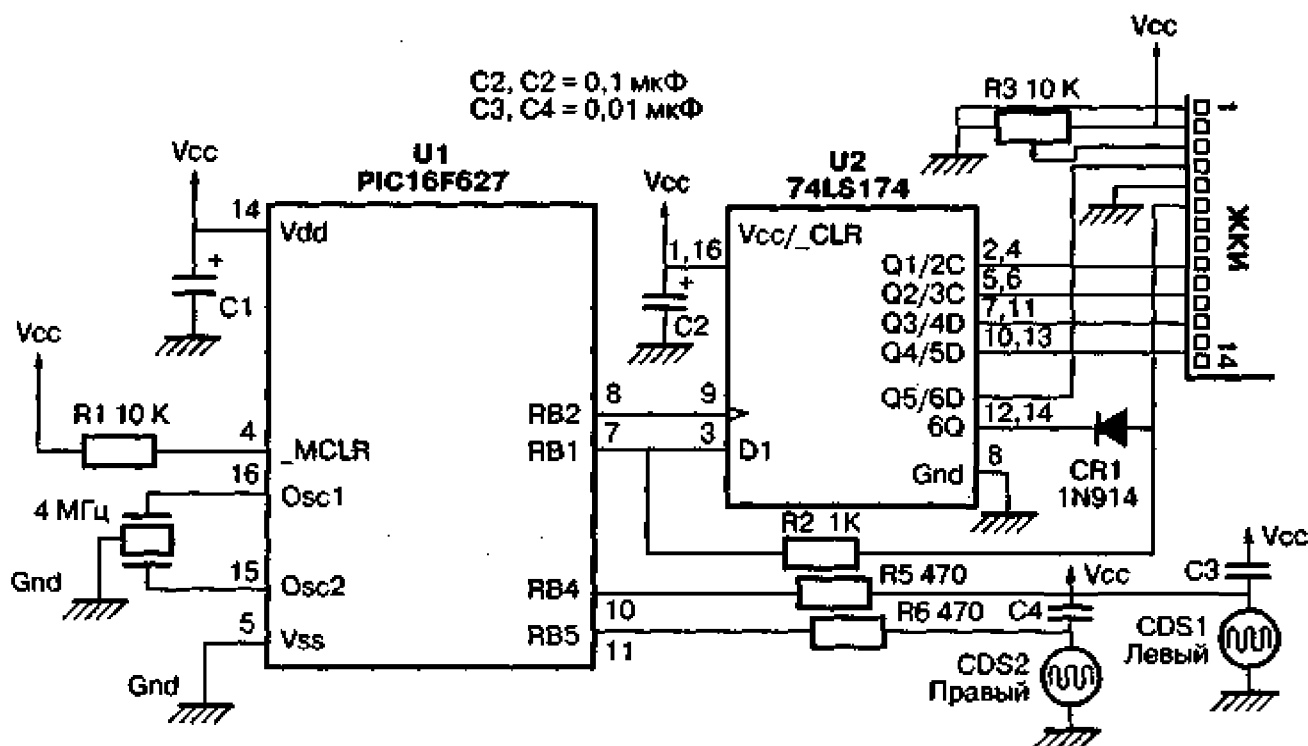


Рис. 4.71. Схема, позволяющая измерять сопротивление фоторезисторов

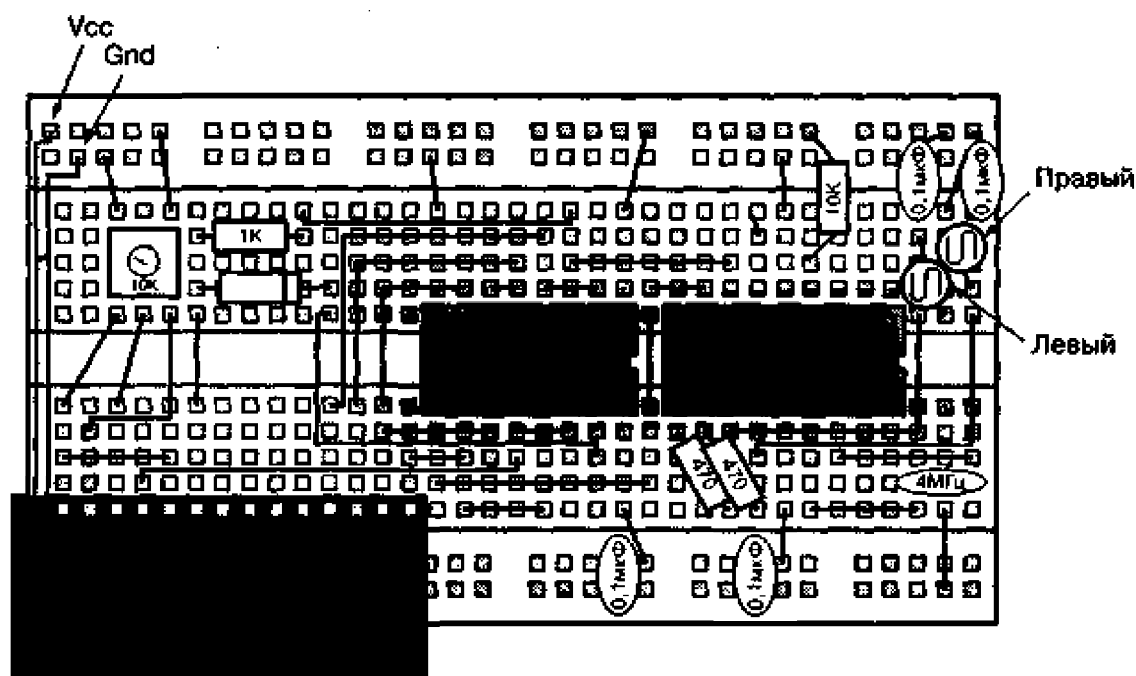


Рис. 4.72. Схема размещения элементов на макетной плате

с методами подключения внешней микросхемы АЦП к микроконтроллеру, устройство получилось несколько сложным, и реализовать его не так-то легко.

Поэтому вернемся к нашему проекту. Исходный текст программы можно найти в файле `Code\Light\light2.c`:

```
#include <pic.h>
```

```
// 24 апреля 2002 - два фоторезистора, сопротивление которых
// измеряется с помощью RC-цепочек.
```

```
//
```

```

// Для индикации используется жидкокристаллический дисплей,
// подключенный к микроконтроллеру через двухпроводной интерфейс
// на основе сдвигового регистра 74LS174.
//
// Используются прерывания от таймера каждую миллисекунду.
//
// Замечания по аппаратным средствам:
// Микроконтроллер PIC16F627 работает на частоте 4 МГц.
// Используется внешний тактовый генератор.
//
// Подключение выводов микроконтроллера:
// RB1 - линия Clock интерфейса жидкокристаллического дисплея;
// RB2 - линия Data интерфейса жидкокристаллического дисплея;
// RB4 - левый фоторезистор;
// RB5 - правый фоторезистор.

// Глобальные переменные и константы:
int RTC = 0; // Счетчик реального времени.
volatile char LCDDlay = 20; // Длительность задержки.
volatile char LCDState = 1; // Номер текущего состояния.

static volatile bit Clock @ (unsigned)&PORTB*8+1;
static volatile bit ClockTRIS @ (unsigned)&TRISB*8+1;
static volatile bit Data @ (unsigned)&PORTB*8+2;
static volatile bit DataTRIS @ (unsigned)&TRISB*8+2;

char * MessageOut; // Указатель на выводимое сообщение.
volatile char MessageOuti = 0; // Смещение текущего символа в строке.
char MessageL[19] = "\376\200 *"; // Перемещение курсора
// в первую строку.

// 1 2 34567890123456789
char MessageR[19] = "\376\300 *"; // Перемещение курсора
// во вторую строку.

volatile char ADCState = 0; // Текущее состояние конечного автомата
// для работы с фоторезисторами.
volatile char ADCDlay = 1; // Задержка 1 мс между операциями
// с фоторезисторами.
volatile char LeftADC = 0; // Сопротивление левого фоторезистора.
volatile char RightADC = 0; // Сопротивление правого фоторезистора.

// Слово конфигурации:
#if defined(_16F627)
    #warning PIC16F627 with external XT oscillator selected
    __CONFIG(0x03F61); // Для МК PIC16F627:
                        // внешний тактовый генератор XT,
                        // RA6/RA7 используются для ввода-вывода,
                        // внешний сигнал сброса,
                        // таймер PWRT включен,
                        // сторожевой таймер выключен,
                        // защита кода отключена,
                        // детектор BODEN включен.
#else
    #error Unsupported PICmicro MCU selected
#endif

// Служебные подпрограммы:
LCDNybble(char Nybble, char RS) // Запись полубайта в ЖКИ.

```

```

...

LCDByte(char Byte, char RS)           // Запись байта в ЖКИ.

...

LCDInit()                             // Инициализация ЖКИ.

...

LCDOut(char * const LCDString)        // Вывод строки на ЖКИ.
{
    while (LCDState);                 // Ждать готовности ЖКИ.

    MessageOut = LCDString;           // Загрузка выводимой строки.
    LCDState = 100;                   // Послать строку на ЖКИ.
}

// Обработчик прерываний:
void interrupt tmr0_int(void)
{
    char temp;

    if (TOIF) {                       // Обработчик прерываний от таймера TMRO.
        TOIF = 0;                     // Сбросить флаг прерываний от таймера TMRO.
        RTC++;                         // Инкремент счетчика реального времени.

        // Здесь можно разместить дополнительный код
        // для обработки прерываний от таймера TMRO.

        // Конечный автомат для ЖКИ:
        ...

        // Конечный автомат для работы с фоторезисторами:
        if (!LCDState) {              // Если нет работы с ЖКИ.
            switch (ADCState) {
                case 0:                // Начальная задержка.
                    if (--ADCDlay == 0)
                        ADCState++;
                    break;
                case 1:                // Начинаем подачу импульса
                                     // на обе RC-цепочки (1 мс).
                    TRISB4 = 0;
                    TRISB5 = 0;
                    RB4 = 1;
                    RB5 = 1;
                    ADCState++;        // Переход к следующему
                                     // шагу через 1 мс.
                    break;
                case 2:
                    TRISB4 = 1;        // Закончить подачу
                                     // импульса на левую RC-цепочку.
                    temp = PORTB;      // Разрешить прерывания
                                     // по изменении сигнала на входе.
                    RBIF = 0;
                    RBIE = 1;
            }
        }
    }
}

```

```

        ADCState++;
        break;
    case 3:
        LeftADC = 0xOFF;
        temp = PORTB;
        RBIF = 0;          // Запретить прерывания
        RBIE = 0;          // по изменении входного
                           // сигнала.
        ADCState++;
        break;
    case 4:
        TRISB5 = 1;        // Закончить подачу
                           // импульса на правую RC-цепочку.
        temp = PORTB;
        RBIF = 0;          // Разрешить прерывания по
        RBIE = 1;          // изменении входного сигнала.
        ADCState++;
        break;
    case 5:
        RightADC = 0xOFF;
        temp = PORTB;
        RBIF = 0;          // Запретить прерывания
        RBIE = 0;          // по изменении входного
                           // сигнала.
        ADCDlay = 10;
        ADCState = 0;
        break;
    } // Конец оператора switch.
} // Конец оператора if.

} // Конец обработчика прерываний от таймера.

//• Здесь можно разместить другие обработчики прерываний.

if (RBIF) { // Обработка прерываний по изменении входного сигнала.
    switch(ADCState) {
        case 3: // Работаем с левым фоторезистором.
            LeftADC = TMRO; // Счетчик времени.
            ADCState++;
            break;
        case 5: // Работаем с правым фоторезистором.
            RightADC = TMRO; // Счетчик времени.
            ADCDlay = 10;
            ADCState = 0;
            break;
    } // endswitch
    temp = PORTB;
    RBIF = 0; // Очищаем флаг и запрещаем прерывания
    RBIE = 0; // по изменении входного сигнала.

} // Конец обработки прерываний по изменении входного сигнала.

}, // Конец обработчика прерываний.

// Главная программа:
void main(void)
{

```

```

unsigned int i, j;
unsigned int tempLeft, tempRight;

OPTION = 0x0D1;      // Предделитель работает с таймером TMRO,
                     // коэффициент деления равен 4.

THRO = 0;            // Начальный оброс таймера TMRO.
TOIE = 1;            // Разрешение прерываний от таймера TMRO.
GIE = 1;             // Разрешение обработки прерываний.

LCDInit();          // Инициализация порта, к которому подключен ЖКИ.

// Здесь можно проинициализировать другие периферийные устройства.

while (1 == 1) {    // Бесконечный цикл.

    // Здесь можно разместить код верхнего уровня.

    while (ADCState != 2);      // Ожидаем начала работы
                                // с фоторезисторами.
    while (ADCState);           // Ожидаем окончания работы
                                // с фоторезисторами.
    tempLeft = LeftADC;  tempRight = RightADC;
                            // Сохраняем измеренные значения
                            // сопротивлений фоторезисторов.

    j = (tempLeft / 8) + 1;     // Количество звездочек
                                // для индикации состояния левого фоторезистора.
    for (i = 2; i < 18; i++)
        if ((i - 2) <= j)      // Выводим звездочку на дисплей.
            MessageL[i] = '*';
        else                    // Выводим пробел.
            MessageL[i] = ' ';

    j = (tempRight / 8) + 1;    // Количество звездочек
                                // для индикации состояния правого фоторезистора.
    for (i = 2; i < 18; i++)
        if ((i - 2) <= j)      // Выводим звездочку на дисплей.
            MessageR[i] = '*';
        else                    // Выводим пробел.
            MessageR[i] = ' ';

    LCDOut(MessageL);          // Состояние левого датчика выводится
                                // в первой строке дисплея.
    LCDOut(MessageR);          // Состояние правого датчика выводится
                                // во второй строке дисплея.

} // Конец цикла while.

} // Конец главной программы.

```

Для определения времени заряда конденсатора здесь используются прерывания, формируемые при изменении уровня сигнала на входных линиях порта PORTB. Многие эксперты не рекомендуют использовать эту возможность микроконтроллеров PICmicro, но я придерживаюсь противоположного мнения. Многочисленные эксперименты с работающими устройствами показывают, что прерывания по

изменении входного сигнала от внешнего источника достаточно надежны и их использование не вызывает особых трудностей. Единственное, что при этом требуется учитывать, - некоторые особенности программирования:

- прерывания по изменению уровня сигнала на линиях порта PORTB формируются, только если эти линии работают в режиме входных;
- во время использования любой линии PORTB в качестве источника прерываний нельзя обращаться ни к одному разряду регистра этого порта (обращение к любому биту требует использования команды, работающей со всеми восемью разрядами регистра). При необходимости всегда можно использовать служебную переменную для хранения текущего значения данного регистра;
- для очистки флага прерывания надо использовать следующий код:

```
Variable = PORTB;      // Запомнили текущее состояние регистра порта
                        // в служебной переменной.
RBIF = 0;               // Очистили флаг прерывания.
RBIE = 0;               // Запретили (если необходимо) прерывания
                        // по изменении входного сигнала.
```

- следует убедиться, что изменение уровня входного сигнала не произойдет раньше, чем программа окажется готова обработать запрос на прерывание. Только из-за того, чтобы удовлетворить этому требованию, мне пришлось ввести в схему два ограничительных резистора R5 и R6, благодаря которым удалось снизить скорость перезаряда конденсаторов.

После включения питания микроконтроллера программа начинает цикл работы с фоторезисторами. На обе RC-цепочки подается прямоугольный импульс, затем засекается время и разрешаются прерывания по изменению уровня сигнала на входах RB4 и RB5. В момент возникновения прерывания определяется время перезаряда конденсатора и вычисляется значение сопротивления фоторезистора.

Конечный автомат, реализующий интерфейс с фоторезисторами, начинает свою работу, только если не активен конечный автомат, обслуживающий жидкокристаллический дисплей. Это сделано для того, чтобы одновременное выполнение обоих автоматов не привело к нарушению временных промежутков между отдельными операциями. Для тех же целей в главной программе перед началом вывода на дисплей ожидается завершение работы с фоторезисторами.

Хотя в нашей новой схеме микроконтроллер получает намного больше информации об уровне освещенности обоих фоторезисторов, на выполнение всех необходимых для этого операций требуется много времени, и робот может не успеть среагировать на то или иное изменение окружающей обстановки, особенно если скорость его движения достаточно велика.

В нашей программе один цикл опроса состояния двух фоторезисторов занимает 6 мс, что позволяет делать около 330 измерений в секунду. Но большинство современных АЦП работает намного быстрее.

4.30. ЗВУКОВЫЕ ДАТЧИКИ

Использование звукового датчика в электронной схеме не должно представлять особой сложности. У многих роботов часто имеется какое-либо устройство ввода звуковой информации, которое предоставляет пользователю возможность управлять ими, хлопая в ладоши или громко произнося команды. В некоторых автоматических устройствах звуковые сенсоры работают в качестве датчиков столкновений (прислушайтесь к звукам, которые раздаются, когда робот сталкивается с предметами). Подключить к микроконтроллеру микрофон, который бы мог распознавать громкие звуки, не так уж сложно, и мы не упустим эту возможность в наших экспериментах.

Многим сложнее всего понять, что аналоговые сигналы (в том числе и звуковые) могут быть представлены в виде суммы множества колебаний разной частоты. Обычно для такого разложения используются синусоидальный и косинусоидальный сигналы (рис. 4.73).

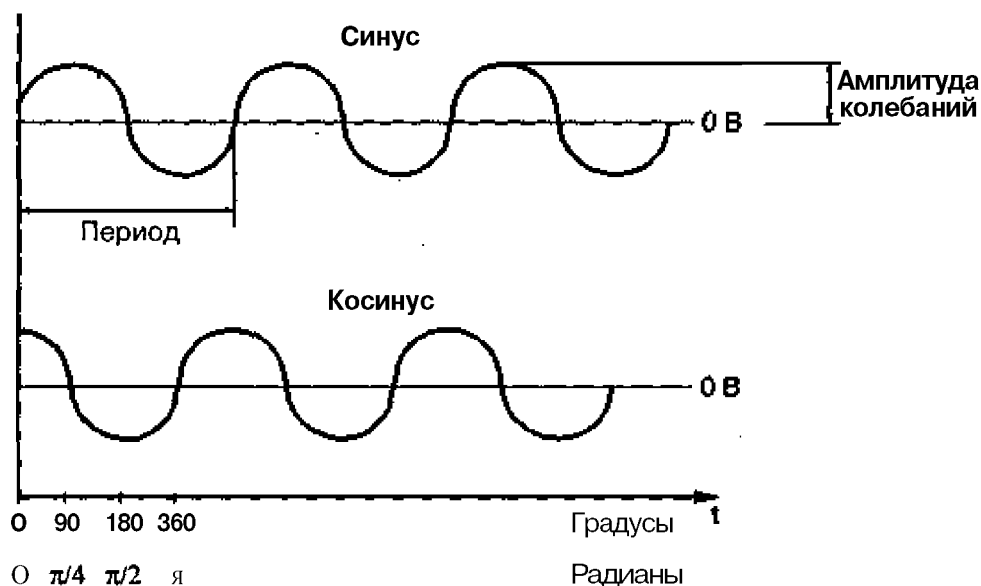


Рис. 4.73. Синусоидальное и косинусоидальное колебание

В виде суммы нескольких таких колебаний (их называют *гармониками*), имеющих разную частоту и амплитуду, можно представить любой аналоговый сигнал. Напомним, что синусоидальное колебание описывается формулой вида

$$x(t) = A \sin(2\pi t / T) = A \sin(2\pi f t),$$

где A - амплитуда, T - период, $f = 1/T$ - частота колебаний, t - время. Именно по этому закону изменяется координата точки, лежащей на ободе колеса радиусом A при его вращении. Когда колесо совершит полный оборот (360°), все будет повторяться с самого начала. Говорят, что начался новый период колебания.

Косинусоидальная волна сдвинута относительно синусоидальной по фазе на 90° (то есть на $\pi/2$):

$$x(t) = A \sin(2\pi f t) = A \cos(2\pi f t - \pi/2).$$

В электронике фазовый угол колебаний $j = 2\pi ft$ обычно измеряется в радианах. Напомним, что для перевода в радианы величины θ , выраженной в градусах, можно воспользоваться формулой

$$j = \pi\theta / 180,$$

где число $\pi \approx 3,14159265359$. Например, 45° - это $\pi/4 = 0,7854$ рад, а полный оборот (360°) - это $2\pi = 6,2832$ рад.

Тактовые импульсы, которые используются в микропроцессорной технике, обычно имеют прямоугольную форму. Последовательность прямоугольных импульсов называют *меандром*. Сигнал такого вида не слишком подходит для того, чтобы его представляли в виде суммы синусоид и косинусоид. На рис. 4.74 показано, что получится, если сложить три первые гармоники¹ в разложении прямоугольного сигнала, ширина импульса которого равна половине периода. Как можно видеть, суммарное колебание не слишком-то совпадает с прямоугольными импульсами. Для более точного представления следует учесть большее количество гармоник.

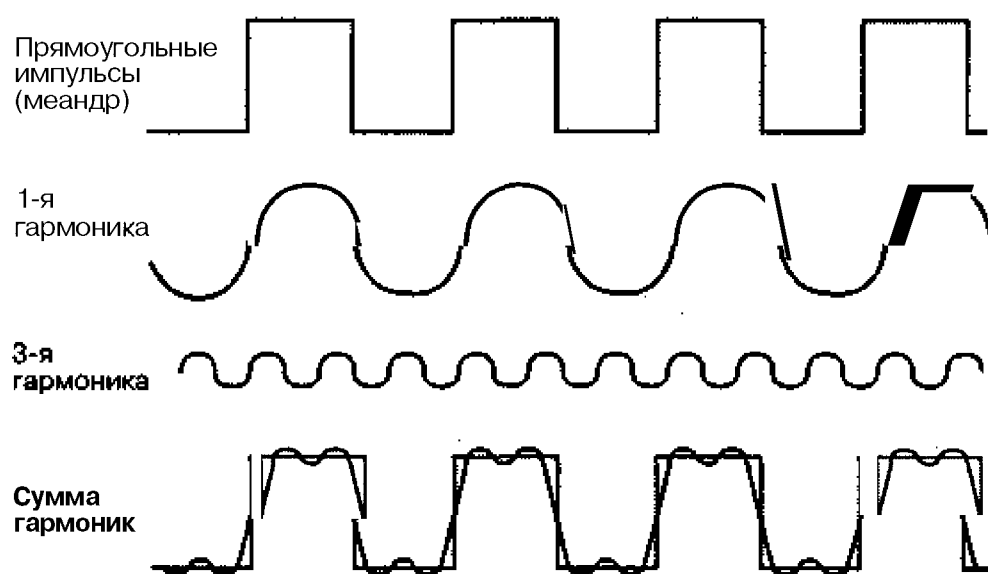


Рис. 4.74. Гармонический анализ прямоугольных импульсов

Таким образом, складывая несколько синусоидальных и косинусоидальных колебаний, можно с достаточной точностью воспроизвести любое аналоговое колебание. При этом частоты суммируемых гармоник всегда кратны частоте самого сигнала. Частота первой гармоники совпадает с частотой сигнала, частота второй

¹ Если сигнал нечетный, то есть его график антисимметричен относительно оси ординат (в данном примере этого можно добиться, расположив начало отсчета в начале какого-нибудь импульса), то все **косинусоидальные** гармоники будут иметь нулевую амплитуду. Поэтому на рис. 4.74 суммируются только синусоидальные гармоники: нулевая (постоянная составляющая, на рисунке не показана), первая, основная, гармоника (ее частота совпадает с частотой самого сигнала), вторая (также не показана на рисунке, так как в данном случае ее амплитуда равна 0) и третья гармоника, имеющая частоту, в три раза превышающую частоту сигнала. Следует отметить, что колебания, изображенные на рис. 4.73 и 4.74, которые буквально воспроизводятся здесь с английского оригинала, только отдаленно напоминают по форме синусоидальные. - Прим. перев.

гармоники в два раза больше частоты сигнала и т.д. Чтобы найти амплитуды гармоник, используют формулы

$$a_i = \frac{2}{T} \int_T x(t) \cos \frac{2\pi i t}{T} dt, \quad b_i = -\frac{2}{T} \int_T x(t) \sin \frac{2\pi i t}{T} dt,$$

где a_i - амплитуда косинусоид, а b_i - амплитуда синусоид в разложении сигнала; интегрирование ведется по любому отрезку, длина которого равна периоду сигнала. В этом случае само колебание может быть записано в виде ряда Фурье:

$$x(t) = \sum_{i=-\infty}^{\infty} \left\{ a_i \cos \frac{2\pi i t}{T} + b_i \sin \frac{2\pi i t}{T} \right\}.$$

Например, рассмотренная выше последовательность прямоугольных импульсов, длительность каждого из которых равна половине периода, будет представлена в виде ряда

$$x(t) = A \sum_{i=0}^{\infty} \frac{1}{2i+1} \cdot \sin \frac{(2i+1)\pi t}{T}.$$

При синтезе сигнала мы должны рано или поздно оборвать бесконечный ряд. Чем позже мы это сделаем, тем больше гармоник учтем, и тем точнее будет полученная сумма представлять искомое колебание.

Но реальна и обратная ситуация: исходный прямоугольный сигнал можно пропустить через специальный **фильтр**, чтобы отсеять одни гармоники и учесть другие.

Фильтр, который пропускает только несколько первых гармоник, называется **фильтром нижних частот (ФНЧ)**. Его *амплитудно-частотная характеристика (АЧХ)* представлена на рис. 4.75. Она показывает, во сколько раз ослабляется сигнал определенной частоты.

В электронике коэффициент усиления или ослабления часто измеряют в *децибелах*. Если речь идет о напряжении или токе сигнала, то ослабление в два раза соответствует 6 дБ. Полосу частот, которые пропускает фильтр, оценивают по уровню 3 дБ, что соответствует ослаблению в $\sqrt{2} = 1,414$ раз. Другими словами, граница полосы пропускания ФНЧ (так называемая *частота среза*) - это частота, на которой коэффициент пропускания фильтра падает до $1/1,414 = 0,707$ своей максимальной величины.

Часто для фильтрации сигналов используют так называемые **активные фильтры**, выполненные на основе операционных усилителей. Один такой пример - **фильтр Баттерворта** - показан на рис. 4.76.

Здесь использованы два резистора с одинаковым сопротивлением R и два конденсатора с одинаковой емкостью C . Другие два резистора R_1 и R_2 задают коэффициент усиления схемы. Для дополнительного усиления я часто использую второй каскад на операционном усилителе.

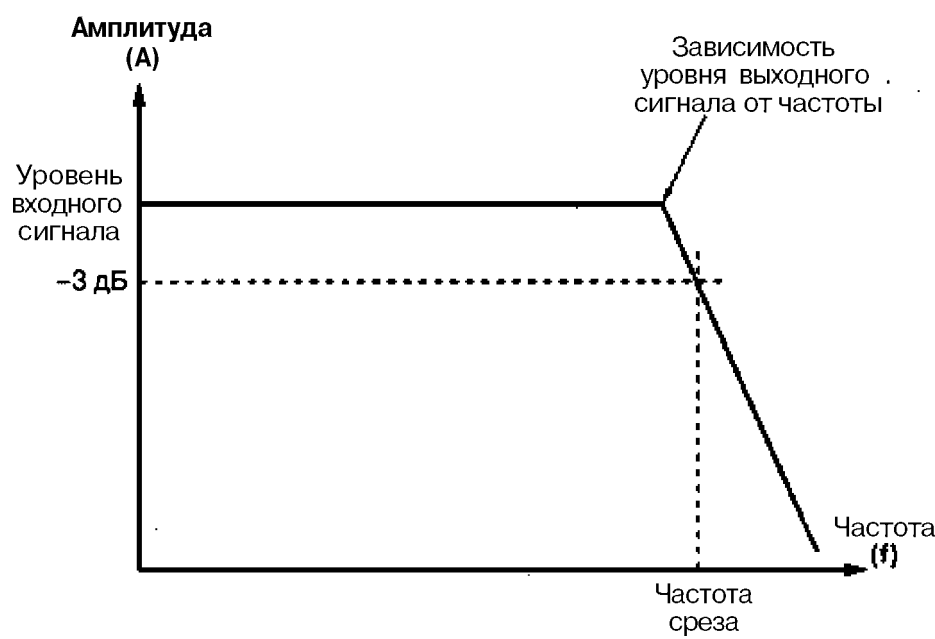


Рис. 4.75. Амплитудно-частотная характеристика фильтра нижних частот

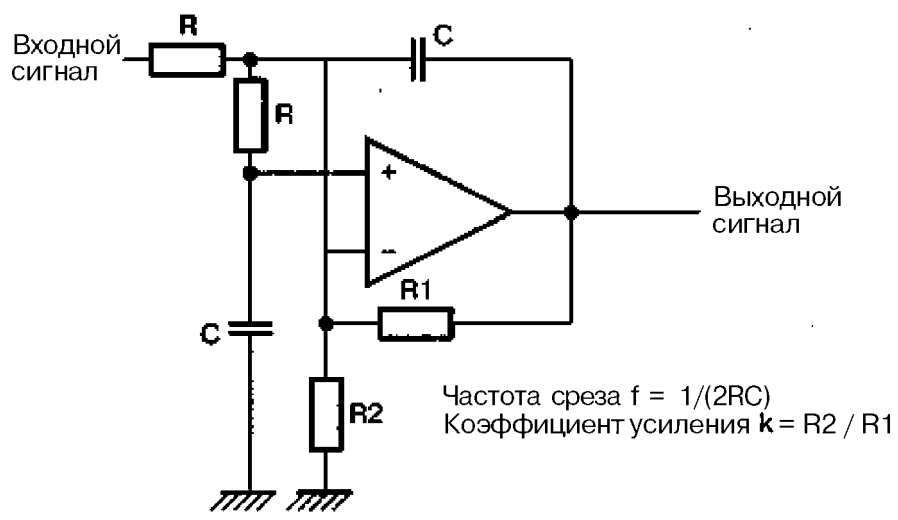


Рис. 4.76. Фильтр Баттерворта

Если коэффициент усиления достаточно велик, то сигнал на входе фильтра превращается в хороший меандр на выходе. Возможно, вы пришли в недоумение: мы начали с того, что раскладывали прямоугольные импульсы на гармоники, затем пропускали их через фильтр, чтобы отсечь высокочастотные составляющие, а закончили тем, что снова получили прямоугольные импульсы! Но в этом есть смысл. Ведь наша задача состоит не в том, чтобы получить меандр из него самого, а в том, чтобы преобразовать произвольное звуковое колебание в низкочастотную синусоиду. При этом, если на вход фильтра вместе со звуковой командой подается какое-либо более высокочастотное колебание (например, шум двигателей), то фильтр его ослабит, насколько возможно.

Существует большое количество разнообразных фильтров. Кроме ФНЧ, используются также *фильтры верхних частот* (ФВЧ), *полосовые* и *режекторные* фильтры. ФВЧ ослабляют сигналы низкой частоты, а пропускают только старшие гармоники (то есть по сравнению с ФНЧ ведут себя противоположным образом).

Полосовые фильтры пропускают колебания только в определенном диапазоне частот, а режекторные фильтры, наоборот, пропускают все частоты, кроме тех, что лежат в некотором диапазоне. Для фильтрации помех в робототехнике обычно используются полосовые фильтры и ФНЧ.

Итак, чтобы робот имел возможность реагировать на звуковые команды или мог обнаруживать по звуку момент столкновения с окружающими предметами, разработчик должен определить, в каком частотном диапазоне лежит звуковое колебание, на которое должен реагировать робот. Этот диапазон частот необходимо выделить с помощью соответствующего фильтра, чтобы отсеять все остальные звуки.

Проще сказать это, чем сделать, поэтому многие радиолюбители избегают использования звуковых датчиков в своих конструкциях. Часто бывает трудно найти звуковой диапазон, который отличает команду (или звук при столкновении) от посторонних шумов, которые действуют внутри и снаружи автоматического устройства.

Однако я считаю, что в большинстве практических случаев фильтр Баттерворта позволяет получить хорошие результаты.

Чтобы робот как можно меньше реагировал на собственные звуки, надо постараться изолировать датчики от его корпуса. Неплохим решением будет и использование картонного или пластикового конуса, который повысит направленность микрофона.

Современные операционные усилители слабочувствительны к электрическим помехам. Но работа с аналоговыми сигналами всегда требует осторожности. Выход активного фильтра лучше подключать не к стандартному порту ввода-вывода микроконтроллера, а к входу какого-нибудь счетчика (например, к счетчику таймера TMR1). В этом случае проще организовать защиту от дребезга и других паразитных колебаний.

4.31. РАСПОЗНАВАНИЕ ЗВУКОВЫХ КОМАНД

Вас не должно вводить в заблуждение громкое название этого раздела. На самом деле я немного преувеличил возможности описываемого здесь интерфейса. Наш робот сможет реагировать только на громкие звуки, такие как хлопки в ладоши или крики «Стоп!».

Принципиальная схема устройства показана на рис. 4.77. Здесь представлен двухкаскадный активный фильтр нижних частот, выход которого следует подключить ко входу таймера TMR1 (вывод RB6) микроконтроллера PIC16F627.

Здесь мы не приводим схему включения самого микроконтроллера и устройств вывода информации (которые, несомненно, необходимы хотя бы для того, чтобы вы имели возможность убедиться, что робот распознал вашу команду), – эти вопросы здесь не принципиальны. Вы можете использовать любую из схем, описанных выше.

В табл. 4.17 представлен перечень использованных элементов.

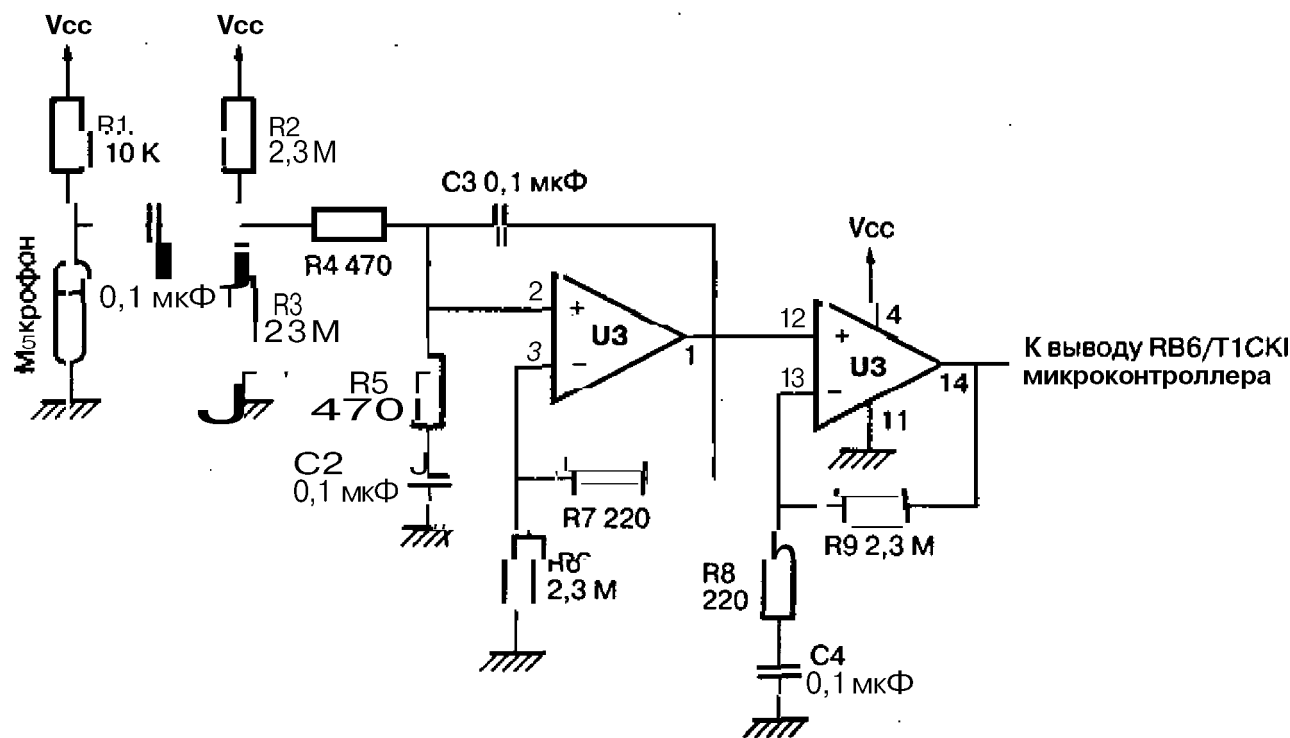


Рис. 4.77. Схема ФНЧ для ввода аудиоконанд в микроконтроллер

Таблица 4.17. Список элементов схемы, представленной на рис. 4.77

Обозначение	Элемент	Примечание
U3	LM342 (K140УД2А или Б)*	Операционный усилитель
Микрофон		Любой угольный или электретный
R1	10 кОм; 0,25 Вт	
R2, R3, R6, R9	2,3 МОм; 0,25 Вт	
R4, R5	470 Ом, 0,25 Вт	
R7, R8	220 Ом, 0,25 Вт	
C1 - C4	0,1 мкФ	Конденсаторы любого типа
	Материалы	Макетная плата с размещенным на ней микроконтроллером PIC16F627 и жидкокристаллическим дисплеем, монтажные провода, источник питания +5 В

* 6 скобках указаны отечественные аналоги, добавленные при переводе. — Прим. перев.

Операционный усилитель LM324 в схеме ФНЧ применяется только потому, что он оказался у меня под рукой. Вы можете взять практически любой малосигнальный сдвоенный операционный усилитель (например, 17741). Я часто использую приведенную схему при конструировании самых различных автоматических устройств, так как она обладает хорошей устойчивостью к помехам.

На рис. 4.78 показаны сигналы на входе и выходе ФНЧ (я громко крикнул «D’oh!»). Схема отфильтровала высокочастотные составляющие, и на вход микроконтроллера прошло только несколько импульсов, которые можно проанализировать в программе.

Мы будем опрашивать текущее состояние счетчика таймера TMR1 каждые 20 мс, чтобы узнать, не был ли обнаружен какой-либо громкий звук. Если был, то

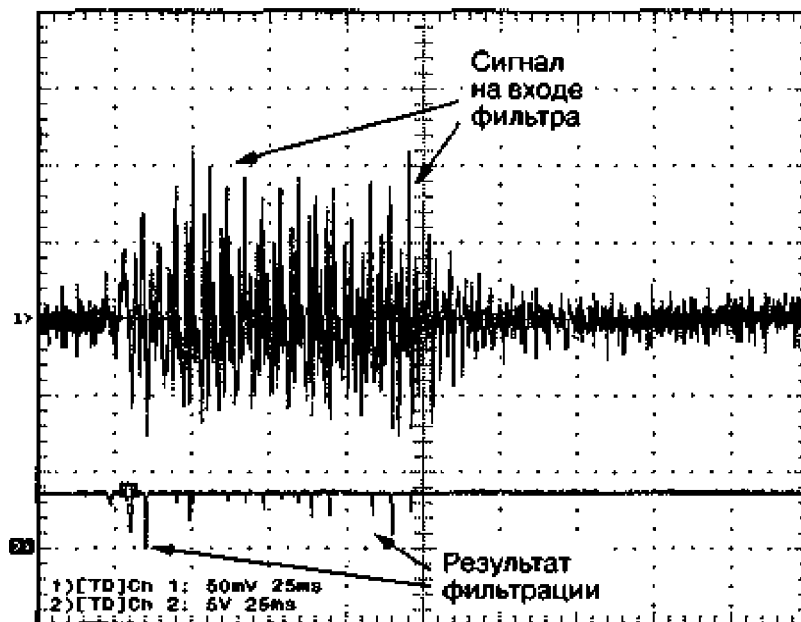


Рис. 4.78. Действие фильтра нижних частот

сообщение об этом отобразится вместе с текущим значением счетчика на жидкокристаллическом дисплее в течение одной секунды. В данном проекте можно использовать только микроконтроллер PIC16F627, так как у PIC16F84 нет второго таймера.

Текст программы вы найдете в файле Code\Sound\sound.c:

```
#include <pic.h>

// 27 января 2002 - сообщение о получении звуковой команды
// выводится на ЖКИ.
//
// Для фильтрации сигнала, снимаемого с микрофона, используется
// активный фильтр нижних частот на двух операционных усилителях.
// Выходной сигнал фильтра подается на вход счетчика таймера TMR1
// (вывод RB6/T1CKI микроконтроллера).
// Таймер опрашивается каждые 20 мс, и его текущее значение сравнивается
// с прошлым. Если оно увеличилось на 5 или более, то регистрируется
// наличие сигнала.
//
// Для индикации используется жидкокристаллический дисплей,
// подключенный к микроконтроллеру через двухпроводной интерфейс
// на основе сдвигового регистра 74LS174.
//
// Используются прерывания от таймера каждую миллисекунду.
//
// Замечания по аппаратным средствам:
// Микроконтроллер PIC16F627 работает на частоте 4 МГц.
// Используется внешний тактовый генератор.
//
// Подключение выводов микроконтроллера:
// RB1 - линия Clock интерфейса жидкокристаллического дисплея;
// RB2 - линия Data интерфейса жидкокристаллического дисплея.
//
// Глобальные переменные и константы:
int RTC = 0; // Счетчик реального времени.
```

```

volatile char LCDDlay = 20;      // Длительность задержки.
volatile char LCDState = 1;      // Номер текущего состояния.

static volatile bit Clock        9 (unsigned)&PORTB*8+1;
static volatile bit ClockTRIS    9 (unsigned)&TRISB*8+1;
static volatile bit Data         @ (unsigned)&PORTB*8+2;
static volatile bit DataTRIS     9 (unsigned)&TRISB*8+2;

char * MessageOut; // Указатель на выводимое сообщение.
volatile char MessageOuti = 0; // Смещение текущего символа в строке.

char Message1[18] = "\376\200sound - 0x0xxxx";
                        // Сообщение о полученной звуковой команде.
//                        1      2      3456789012345678
char Message2[18] = "\376\200                ";
                        // Выводится, когда нет команды.

char soundCounter = 20;          // Задержка 20 мс.

int OldTMR1 = 0;                 // Простое значение таймера.
int CurrentTMR1;                // Текущее значение таймера.

volatile char soundState = 0;    // Есть звук?
                                // 0 - нет;
                                // 1 - есть, ждем дальше;
                                // 2 - звук окончен.

// Слово конфигурации:
#if defined(_16F627)
    <warning PIC16F627 with external XT oscillator selected
    __CONFIG(0x03F61); // Для МК PIC16F627:
                        // внешний тактовый генератор XT,
                        // RA6/RA7 используются для ввода-вывода,
                        // внешний сигнал сброса,
                        // таймер PWRT включен,
                        // сторожевой таймер выключен,
                        // защита кода отключена,
                        // детектор BODEN включен.
#else
    #error Unsupported PICmicro MCU selected
#endif

// Служебные подпрограммы:

char GetHex(char Value)          // Перевод числа Value
{                                // в шестнадцатеричное представление.

char returnValue;

    if (Value > 9)
        returnValue = Value + 'A' - 10;
    else
        returnValue = Value + '0';

    return returnValue;

} // Конец функции GetHex

Dlay(int msecs)                // Задержка на msecs миллисекунд.
{

```



```

volatile int variableDlay;

    variableDlay = RTC + msec + 1;    // Конечное время.
    while (variableDlay != RTC);

} // Конец функции Dlay

LCDNybble(char Nybble, char RS)        // Запись полубайта в ЖКИ.

...

LCDByte(char Byte, char RS)            // Запись байта в ЖКИ.

...

LCDInit()                             // Инициализация ЖКИ.

...

LCDOut(char « const LCDString)        // Вывод строки на ЖКИ.

...

// Обработчик прерываний:
void interrupt tmr0_int(void)
{
    char temp;
    int  TMR1Value;

    TOIF = 0;    // Сбросить флаг прерываний от таймера TMR0.

    RTC++;    // Инкремент счетчика реального времени.

    // Здесь можно разместить дополнительный код
    // для обработки прерываний от таймера TMR0.

    // Работа с звуковым датчиком:

    if (soundState == 0) // Проверка текущего состояния.
        if (--soundCounter == 0) { // Пора?
            soundCounter = 20;    // Сброс.
            TMR1Value = (TMR1H * 0x0100) + TMR1L;
            if (TMR1Value >= (OldTMR1 + 2))
                soundState = 1;    // Есть звук.
            OldTMR1 = TMR1Value;
            // Сохранили состояние счетчика таймера.
        } else;
    else if (soundState == 1) // Звук продолжается?
        if (--soundCounter == 0) { // Пора?
            soundCounter = 20;    // Сброс.
            TMR1Value = (TMR1H * 0x0100) + TMR1L;
            if (TMR1Value < (OldTMR1 + 2))
                soundState = 2;
            // Звук прекратился.
            OldTMR1 = TMR1Value;
            // Сохранили состояние счетчика таймера.
        } else;

```

```

        // Конечный автомат для ЖКИ:

        ...

    } // Конец обработки прерываний от таймера.

    // Здесь можно разместить код для обработки других прерываний.

} // Конец обработчика прерываний.

// Главная программа:
void main(void)
{

    OPTION = 0x0D1;          // Предделитель работает с таймером TMRO,
                             // коэффициент деления равен 4.

    TMRO = 0;                // Начальный сброс таймера TMRO.
    TOIE = 1;                // Разрешение прерываний от таймера TMRO.
    GIE = 1;                 // Разрешение обработки прерываний.

    LCDInit();               // Инициализация порта, к которому подключен ЖКИ.

    // Здесь можно проинициализировать другие периферийные устройства.

    TMR1H = TMR1L = 0;       // Инициализация таймера TMR1.
    T1CON = 0x003;           // Коэффициент деления 1:1;
                             // T10SCEN (внутренний генератор) выключен;
                             // _T1SYNC активен (синхронизация
                             // с командными циклами включена);
                             // TMR1CS установлен - внешний источник;
                             // TMR1ON установлен - таймер TMR1 включен.

    while (1 == 1) {         // Бесконечный цикл.

        // Здесь можно разместить код верхнего уровня.

        while (soundState != 2); // Ждать появления звука.

        CurrentTMR1 = (TMR1H * 0x0100) + THR1L;

        Message1[13] = GetHex((CurrentTMR1 & 0x0F000) >> 12);
        Message1[14] = GetHex((CurrentTMR1 & 0x0F00) >> 8);
        Message1[15] = GetHex((CurrentTMR1 & 0x0F0) >> 4);
        Message1[16] = GetHex(CurrentTMR1 & 0x0F);

        LCDOut(Message1);     // Вывести сообщение "Sound"
                             // (звук обнаружен).

        Delay(1000);          // Ждать 1 с.

        LCDOut(Message2);     // Стереть сообщение.

        soundState = 0;        // Сброс. Ждать следующего звука.

    } // Конец цикла while.

} // Конец главной программы.

```

Испытание этого приложения не должно вызвать у вас проблем. Возможно, придется подобрать сопротивление резистора R1, чтобы амплитуда входного сигнала, снимаемого с микрофона, находилась в диапазоне 150-200 мВ, как показано на рис. 4.78.

4.32. УПРАВЛЕНИЕ ДВИГАТЕЛЕМ

Чтобы робот был достаточно мобильным, необходимо иметь какой-либо простой и надежный способ для включения и выключения его двигателей, быстрого изменения направления и, возможно, скорости движения. В больших устройствах для этого используется специальный привод (трансмиссия), но для маленьких механизмов более практичны электрические методы, например **Н-образный мост**, образованный четырьмя выключателями, в качестве которых обычно используются транзисторы.

Электродвигатели могут управляться с помощью тех же аппаратных и программных средств, которые описывались в предыдущих разделах. Но все сигналы, которые формировал микроконтроллер с помощью рассмотренных выше программ, были однополярными. Это значит, что даже если мы будем регулировать скорость вращения двигателя с помощью программного или аппаратного широтно-импульсного модулятора, двигатель сможет вращаться только в одном направлении. Чтобы изменить направление вращения (и тем самым дать роботу возможность двигаться не только вперед, но и назад), используем Н-образную мостовую схему, показанную на рис. 4.79.

В этой схеме если все выключатели разомкнуты, то обмотки электродвигателя обесточены. Когда замыкаются выключатели 1 и 4, то двигатель начинает вращаться в одном направлении. Если, разомкнув контакты 1 и 4, замкнуть выключатели 2 и 3, то через обмотки двигателя ток потечет в противоположном направлении; значит, направление вращения якоря тоже изменится на противоположное. Нельзя допустить замыкания обоих контактов, находящихся с одной стороны: при этом возникнет короткое замыкание и выйдет из строя источник питания и/или сами электронные ключи.

Если мы хотим изменять не только направление, но и скорость вращения, то управлять состоянием электронных ключей Н-образного моста должен широтно-импульсный модулятор (ШИМ). Чем большую часть периода длится импульс (см. рис. 4.33), тем больше средняя мощность, потребляемая двигателем, и тем быстрее он вращается.

Собрать схему, показанную на рис. 4.79, можно на обычных транзисторах. Но для этой цели выпускаются также специальные микросборки, состоящие из четырех электронных ключей. Обычно я применяю микросхему 293D (рис. 4.80). Она может быть использована для управления двумя двигателями, которые должны подключаться к выводам 3, 6, 11 и 14. Напряжение питания подается на выводы 2, 7, 10 и 15. К выводам 1 и 9 подводится напряжение, управляющее электронными ключами (это может быть ШИМ-сигнал).

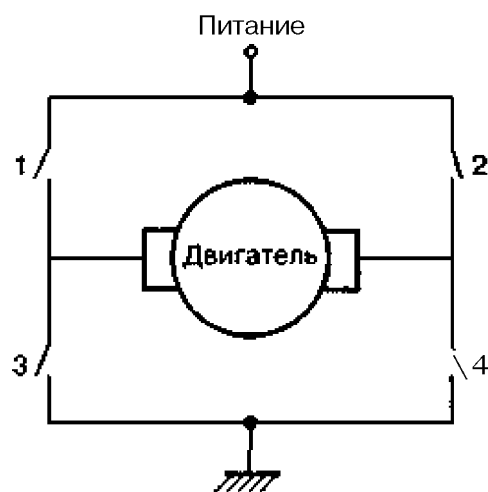


Рис. 4.79. Переключение направления вращения электродвигателя постоянного тока

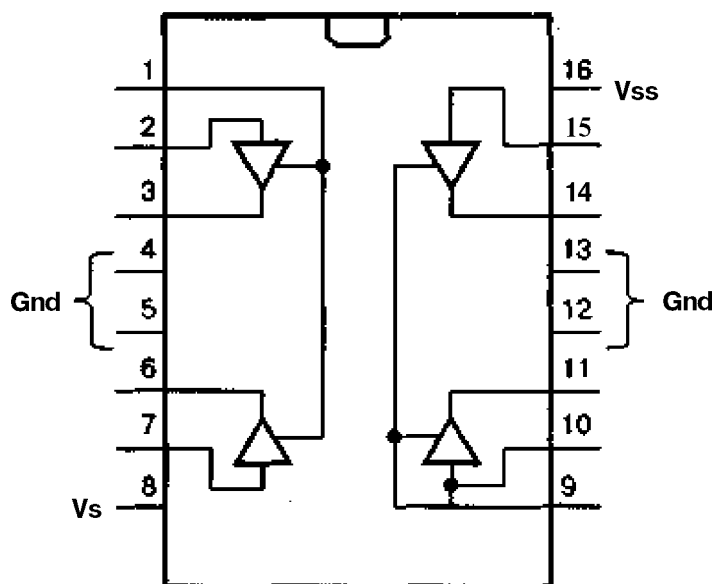


Рис. 4.80. Микросхема 293D
для управления направлением
вращения электродвигателя

К выводу Vs подводится напряжение питания +5 В для самой микросхемы, а к выводу Vss - напряжение питания для электродвигателей; оно должно находиться в диапазоне от 4,5 до 36 В. Максимальный ток, который могут потреблять подключаемые к микросхеме двигатели, не должен превышать 500 мА. Внутри микросхемы 293D уже имеются шунтирующие диоды, которые обычно подключаются параллельно обмоткам электродвигателей (а также к любым другим нагрузкам, имеющим преимущественно индуктивное сопротивление), поэтому при использовании 293D не требуются внешние шунтирующие диоды.

Схема подключения источника ШИМ-сигнала управления и одного электродвигателя к микросхеме 293D приведена на рис. 4.81.

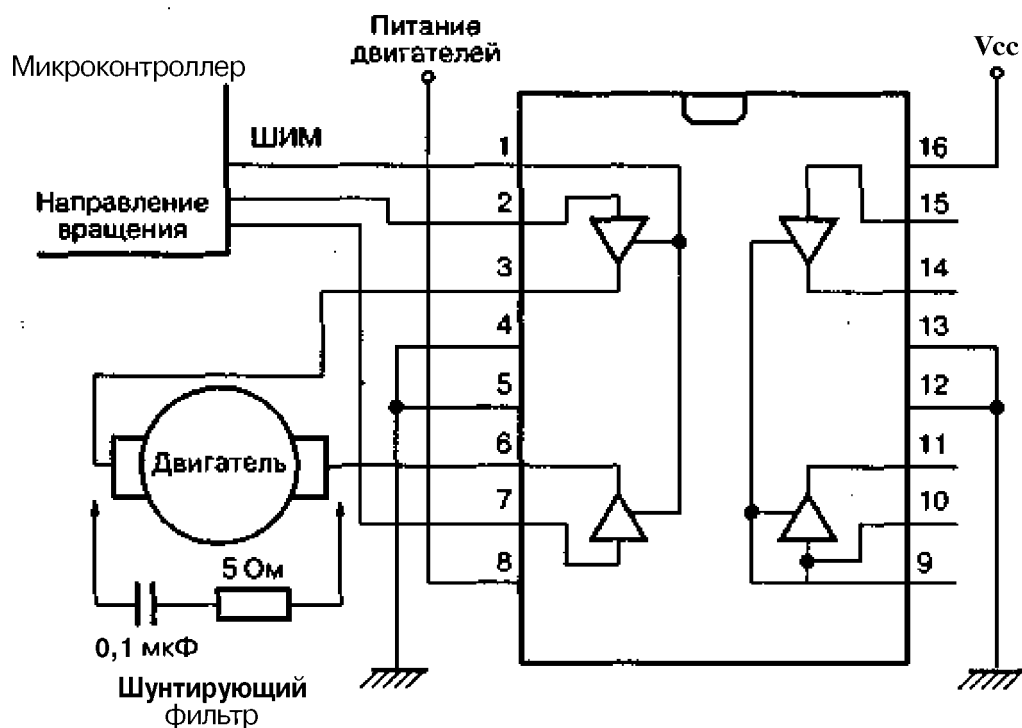


Рис. 4.81. Управление электродвигателем

Параллельно щеткам электродвигателя здесь подключена цепочка из последовательно соединенных резистора и конденсатора - она предназначена для снижения импульсных помех, возникающих при его работе. Без такой защиты при включенном двигателе могут возникать перебои в работе микроконтроллера. Емкость конденсатора следует выбрать около 0,1 мкФ, а резистор должен иметь сопротивление примерно 5 Ом и быть рассчитан на мощность не менее 2 Вт.

Следует помнить, что в качестве электронных ключей в микросхеме 293D используются биполярные транзисторы, поэтому падение напряжения на каждом из них (в открытом состоянии) составляет около 0,7 В. Так как в Н-образной схеме последовательно соединены два ключа, то двигателю «достаётся» напряжение, которое на 1,4–1,5 В меньше, чем подаваемое на вход V_{ss} . Умножив это напряжение на ток, потребляемый двигателем, можно оценить, какую тепловую мощность будет рассеивать микросхема. Хотя максимальный суммарный выходной ток микросхемы 293D может достигать 1 А, для нормальной работы желательно, чтобы его значение было меньше.

В литературе по робототехнике вы найдете более подробное изложение темы управления электродвигателями и встретите описание других электронных ключей, которые могут для этого использоваться, например КМОП транзисторы.

Вопрос об управлении электроприводом достаточно сложен — ему следует уделить большое внимание, так как его решение в немалой степени повлияет на эффективность и функциональность автоматического устройства. Но для управления небольшим роботом вполне подойдет вышеупомянутая микросхема 293D.

4.33. МОДЕЛИ ФИРМЫ ТАМИЯ В КАЧЕСТВЕ ПРОТОТИПА РАДИОЛЮБИТЕЛЬСКИХ КОНСТРУКЦИЙ

Не всегда просто найти для робота хороший прототип. Иногда радиолюбители используют электромеханические игрушки, однако зачастую у них достаточно узкое назначение, поэтому с ними тяжело работать. Можно, конечно, приобрести какой-нибудь из специальных наборов, но их стоимость достаточно высока. Прекрасный пример «золотой середины» между двумя названными крайностями — модели фирмы Tamiya¹.

В этом разделе мы рассмотрим, как простая модель (фактически игрушка) может быть использована в экспериментах по робототехнике. Описанная ниже конструкция недостаточно оптимальна и надежна, чтобы стать законченным промышленным устройством, но она послужит хорошей базой для дальнейших опытов.

На рис. 4.82 показано простое устройство на основе одной из разработок Tamiya, которую я несколько лет назад всего за несколько минут превратил из игрушки в самого настоящего робота. Здесь использована модель мыши (скорее, грызуна, чем одноименного компьютерного манипулятора); в каталоге она значится под именем *Wall-Hugging Mouse* или под номером 70068. В ее корпусе прекрасно удалось разместить дополнительную электронную схему для управления движением (на основе микроконтроллера, конечно).

Рис. 4.82. Внешний вид робота, выполненного на основе модели мыши фирмы Tamiya

¹ С продукцией этой японской фирмы вы можете познакомиться, заглянув на русскоязычный сайт www.tamiya.ru. — Прим перев.

Модель мыши имеет два колеса, каждым из которых управляет отдельный электродвигатель постоянного тока. Направление движения задается с помощью контактного детектора на основе микропереключателя. Один двигатель включается при замыкании микропереключателя, второй - при его размыкании. В результате мышь движется вдоль стенки по зигзагообразной траектории и может даже выбраться из несложного лабиринта¹.

Прежде чем начать экспериментировать с этой простой игрушкой, соберите набор и посмотрите, как он работает. При сборке лучше припаять провода, так как скрученные контакты быстро растреплются, пока мышка будет бегать по комнате. Проводить эксперименты с данной моделью крайне увлекательно, но они быстро надоедают, и скоро вы будете готовы использовать игрушку в качестве основы для каких-то более сложных конструкций.

Если вы уже готовы, то начнем:

1. Удалите батарейку и ее держатель.
2. Припаяйте двухконтактный разъем для 9-вольтовой батарейки.
3. Можно (хотя и необязательно) удалить микропереключатель, управляющий двигателями.
4. Просверлите отверстия для крепления печатной платы, на которой будет размещаться электронное устройство управления.

• В качестве детектора касания в нашей конструкции можно использовать все тот же микропереключатель, который уже имеется в комплекте с мышкой. Но для обнаружения объектов с обеих сторон нужны два контактных усика и два переключателя. Я взял второй из другого такого же набора.

Разумеется, еще лучше использовать инфракрасный обнаружитель объектов, вроде того, который уже описан в этой главе.

Принципиальная схема блока управления представлена на рис. 4.83. Внешний вид печатной платы показан на рис. 4.84. На то, чтобы собрать эту схему, лично я потратил два часа. В табл. 4.18 приведен список использованных элементов и материалов.

В устройстве используется 9-вольтовая щелочная батарея и переключатель (на днище мышки) для включения питания. Не забывайте, что переключатель нужно располагать до цепей питания (то есть диода CR3 и фильтрующих конденсаторов C5 и C7), иначе эти конденсаторы будут заряжены даже при выключенном питании - тогда их токи утечки будут разряжать батарейку, а кроме того, в момент включения питания могут возникнуть нежелательные импульсы.

Возможно, вас удивят большая емкость конденсатора C7 (1000 мкФ) и наличие блокирующего диода CR3 (см. рис. 4.83). Как будет показано ниже в этом разделе, применение одной 9-вольтовой батарейки для питания двух 1,5-вольтовых электродвигателей, микросхемы L293D и микроконтроллера - далеко не самое оптимальное, но вполне допустимое решение.

¹ Для этого не понадобится особый интеллект. Как известно, простой алгоритм, позволяющий найти путь из лабиринта (не обязательно самый короткий), заключается в том, чтобы все время держаться какой-либо стены - правой или левой. - Прим перев.

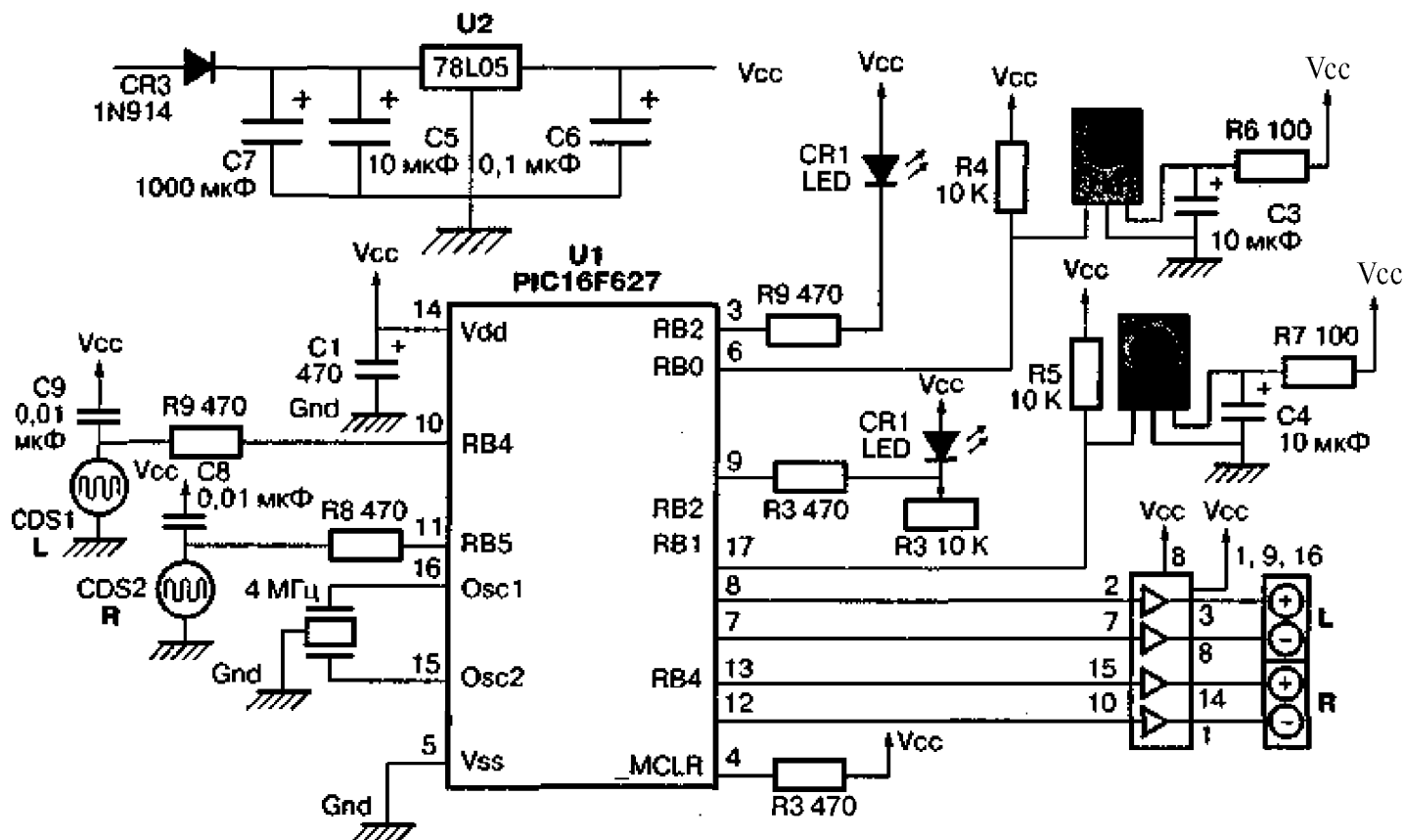


Рис. 4.83. Схема для управления моделью мыши

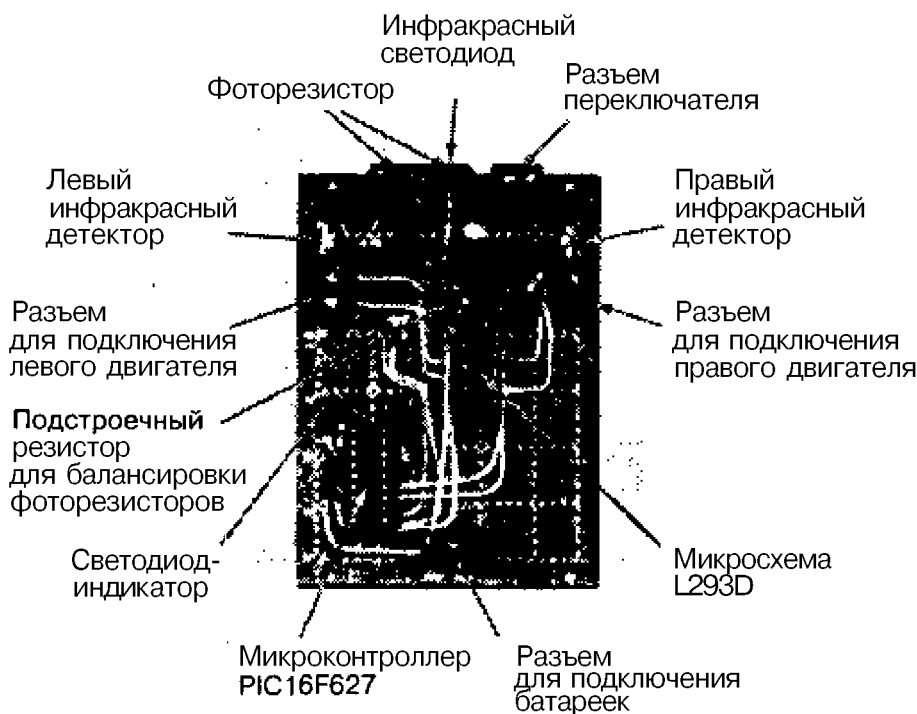


Рис. 4.84. Внешний вид печатной платы блока управления

Таблица 4.18. Список использованных элементов

Обозначение	Элемент	Примечание
U1	PIC16F627	Микроконтроллер
U2	78L05 (КР142ЕН5А)*	Микросхема стабилизатора напряжения

Таблица 4.18. Список использованных элементов (окончание)

Обозначение	Элемент	Примечание
U3, U4	Инфракрасный детектор	P/C- 1018SCL или другой совместимый (три вывода)
U5	L293D	
CR1	Светодиод	Любой видимого диапазона
CR2	Светодиод	Любой инфракрасный
CR3	1N914	Любой кремниевый диод, рассчитанный на ток 200 мА
R1, R4, R5	10 кОм; 0,25 Вт	
R2, R3, R8, R9	470 Ом; 0,25 Вт	
R6, R7	100 Ом; 0,25 Вт	
CDS1, CDS2	Фоторезисторы, 10 кОм при нулевом освещении	При освещении должны уменьшать свое сопротивление
C1, C6	0,1 мкФ	Конденсаторы любого типа
C3 – C5	10 мкФ; 16 В	Оксидные конденсаторы
C7	1000 мкФ; 16 В	Оксидный конденсатор
C8, C9	0,01 мкФ	Конденсаторы любого типа
XTAL1	Керамический резонатор на 4 МГц, со встроенными конденсаторами	Для генератора тактовых импульсов микроконтроллера
	Материалы	Модель мыши фирмы Tamiya, макетная плата, монтажные провода, батарейка 9 В
* В скобках указаны отечественные аналоги, добавленные при переводе. -Прим. перев.		

Во время работы двигателей потребляемый ток увеличивается; как следствие, на внутреннем сопротивлении батарейки падает достаточно большое напряжение. Чтобы свести эти потери к минимуму, надо использовать достаточно качественную щелочную батарейку с низким внутренним сопротивлением. Чтобы обеспечить микроконтроллер стабильным питанием, используем фильтрующий конденсатор большой емкости. Схема источника питания в данном случае далека от идеальной, но практика показала, что со своей задачей она вполне справляется. И контактные датчики, и дистанционное управление работают без заметных перебоев.

Когда будете воспроизводить эту конструкцию, расположите панель для микроконтроллера в таком месте, откуда его можно было бы без труда извлечь, например в левой задней части робота. Поверьте, нет ничего более обидного, когда прекрасно работающую схему приходится разбирать только из-за того, что иначе невозможно вытащить микроконтроллер для перепрограммирования!

При монтаже клемм на двигатель необходимо следить за тем, чтобы клемма со знаком «плюс» располагалась со стороны передней части робота, а положительный вывод двигателя соединялся с ней красным проводом. Отрицательный провод должен быть черным: в противном случае легко перепутать провода при наладке схемы, а это может привести самое малое к тому, что робот поедет назад (если вообще сможет куда-либо ездить). Также имеет смысл делать провода как

можно более короткими, чтобы уменьшить риск их наматывания на колеса и другие движущиеся части.

Если вы читали главу без пропусков, у вас не должен вызвать затруднений монтаж фоторезисторов, а также инфракрасного датчика системы дистанционного управления и обнаружителя объектов. На схеме с рис. 4.83 легко узнать проекты, описанные выше в этой главе.

Текст управляющей программы вы найдете в файле `Code\Wallhug\wallhug.c`:

```
#include <pic.h>

// 28 апреля 2002 - модификация модели мыши фирмы Tamiya.
//
// Инфракрасный обнаружитель объектов (ШИМ-сигнал частотой 38 кГц).
//
// Дистанционное управление на основе инфракрасного детектора
// фирмы Sony.
//
// Для индикации используется светодиод.
//
// Формат пакета данных:
//
//      Стартовый бит      1      0
//      ---+      +-----+      +-----+      +-----+
//      |      |      |      |      |      |
//      +-----+      +-----+      +-----+
//
//      |      2,4 мс      |540мкс|660мкс|540мкс|      1,2 мс      |
//
//      |      1,2 мс      |      1,76 мс      |
//
//      |      300      |      440      |
//
//
// Замечания по аппаратным средствам:
// Микроконтроллер PIC16F627 работает на частоте 4 МГц.
// Используется внешний тактовый генератор.
//
// Подключение выводов микроконтроллера:
// RA0 - правый инфракрасный детектор для обнаружения объектов
// (опрашивается программно);
// RA4 - светодиод для индикации принятой команды дистанционного
// управления;
// RB0 - левый инфракрасный детектор для приема команд дистанционного
// управления фирмы Sony и для обнаружения объектов
// (прерывания по положительному перепаду);
// RB1 - левый двигатель (вывод "-");
// RB2 - левый двигатель (вывод "+");
// RB3 - светодиод инфракрасного обнаружителя объектов;
// RB4 - левый фоторезистор;
// RB5 - правый фоторезистор;
// RB6 - правый двигатель (вывод "-");
// RB7 - правый двигатель (вывод "+");
//
```

```

// Глобальные переменные и константы:
int RTC = 0; // Счётчик реального времени.

#define remote0      0x06EF // Определения констант для команд
#define remote1      0x0FEF // дистанционного управления.
#define remote2      0x07EF
#define remote3      0x0BEF
#define remote4      0x03EF
#define remote5      0x0DEF
#define remote6      0x05EF
#define remote7      0x09EF
#define remote8      0x01EF
#define remote9      0x0EEF
#define remoteVolUp  0x0B6F
#define remoteVolDown 0x036F
#define remoteChanUp  0x0F6F
#define remoteChanDown 0x076F
#define remotePrevChan 0x022F
#define remoteMute    0x0D6F
#define remotePower   0x056F

unsigned int  DataIn; // Входные данные.
unsigned int  DataTime = 0;
unsigned char DataInCount = 0; // Количество входных символов.
unsigned int  CurrentRTC;

volatile char LeftCollision = 0; // Есть объект слева?
volatile char RightCollision = 0; // Есть объект справа?

volatile char ADCState = 0; // Текущее состояние конечного
                           // автомата опроса фоторезисторов.
volatile char ADCDlay = 1; // Задержка 50 мс между срабатываниями
                           // конечного автомата.
volatile unsigned char LeftADC = 0xFF; // Состояние левого фоторезистора.
volatile unsigned char RightADC = 0xFF; // Состояние правого фоторезистора.

#define goStop      0x030 // Стоп двигателя!
#define goForward   0x072 // Вперед!
#define goReverse   0x0B4 // Реверс!
#define turnLeft    0x074 // Левый двигатель, реверс!
                           // (поворот налево).
#define turnRight   0x0B2 // Правый двигатель, реверс!
                           // (поворот направо).
                           // Замечание: на выводах
                           // RB4/RB5 - высокий уровень.
#define LeftForward  1 // Левый двигатель, вперед!
#define LeftStop     0 // Левый двигатель, стоп!
#define LeftReverse  -1 // Левый двигатель, реверс!
#define RightForward 1 // Правый двигатель, вперед!
#define RightStop    0 // Правый двигатель, стоп!
#define RightReverse -1 // Правый двигатель, реверс!

unsigned char motorState = goStop; // Состояние двигателей.
unsigned char PWMDuty = 15;
unsigned char PWMCycle = 0;
unsigned char OpnCount = 1;

char ExecuteFlag = 0; // Флаг для индикации состояния.

```



```

        LeftCollision++;
    else;
else
    RightCollision = 0;

    TRISB3 = 1;
}

// Программная генерация ШИМ-сигнала для управления
// скоростью вращения двигателей.

if (PWMDuty == 0)           // Нулевая ширина импульсов?
    ;
else if (PWMDuty == 29) // Ширина импульса равна периоду?
    PORTB = temp | motorState;
else                        // Нечто среднее?
    if (PWMCycle <= PWMDuty)
        PORTB = temp | motorState;
    else;

if (++PWMCycle == 30)
    PWMCycle = 0;

// Индикация:
if ((!ExecuteFlag) && (--OpnCount == 0)) {
    // Опрос фоторезисторов.
    OpnCount = 1;
    PORTB = (PORTB & 0x039) | (motorState = goStop);
    if (LeftADC < RightADC)
        RA4 = 0;           // Зажечь светодиод -
                            // источник света справа.
    else
        RA4 = 1;           // Потушить светодиод -
                            // источник света слева.
}

// Прием команд дистанционного управления:
if (OataTime) {
    if (DataInCount == 0) {           // Есть команда.
        DataInCount = 12;           // Ждать 12 импульсов.
        DataIn = 0;
    } else {                           // Прием очередного бита.
        DataIn = DataIn << 1;
        if ((DateTime > 225) && (DateTime < 375)) {
            // Получена 1.
            DataIn++;
            if (--DataInCount == 0) {
                OpnCount = 200;           // Конец.
                if (DataIn == remote2) {
                    motorState = goForward;
                } else if (DataIn == remote4) {
                    motorState = turnLeft;
                } else if (DataIn == remote6) {
                    motorState = turnRight;
                } else if (DataIn == remote8) {
                    motorState = goReverse;
                }
            }
        }
    }
}

```

```

    } else
    {
        if (DataIn == remoteVolUp) {
            if (PWMDuty < 29)
                PWMDuty++;
            else;
        } else
        if (DataIn == remoteVolDown) {
            if (PWMDuty != 0)
                PWMDuty--;
            else;
        } else
        if (DataIn == remotePower) {
            ExecuteFlag = 1;
            motorState = goStop;
            OpnCount = 1; // Стоп.
        } else {
            ExecuteFlag = 0;
            motorState = goStop;
            OpnCount = 1; // Стоп.
        }
    }
}

} else if ((DataTime > 375) &&
(DataTime < 500)) { // Получен 0.
if (--DataInCount == 0) {
    OpnCount = 200; // Конец.
    if (DataIn == remote2) {
        motorState = goForward;
    } else if (DataIn == remote4) {
        motorState = turnLeft;
    } else if (DataIn == remote6) {
        motorState = turnRight;
    } else if (DataIn == remote8) {
        motorState = goReverse;
    } else
    if (DataIn == remoteVolUp) {
        if (PWMDuty < 29)
            PWMDuty++;
        else;
    } else
    if (DataIn == remoteVolDown) {
        if (PWMDuty != 0)
            PWMDuty--;
        else;
    } else
    if (DataIn == remotePower) {
        ExecuteFlag = 1;
        motorState = goStop;
        OpnCount = 1; // Стоп.
    } else {
        ExecuteFlag = 0;
        motorState = goStop;
        OpnCount = 1; // Стоп.
    }
}

} else if (--DataInCount != 12) // Ошибка.
    DataInCount = 0;

```

```

    }
    DateTime = 0;           // Все сначала.
}

// Задержка:
CurrentRTC += 0x0100;

// Проверка на потерянный импульс:
if ((DataInCount) && (CurrentRTC > 0x00900))
    DataInCount = 0;      // Сброс. Ждать следующей команды.

// Конечный автомат для работы с фоторезисторами:
if (!DataInCount) {      // Если не принимается команда.
    if (ADCState == 0) { // Начальная задержка.
        if (--ADCDelay == 0)
            ADCState++;
    } else if (ADCState == 1) {
        // Подать импульс на обе RC-цепочки (1 мс).
        PORTB = PORTB | 0x030;
        TRISB4 = 0;
        TRISB5 = 0;
        ADCState++;      // Переход к следующему
                          // состоянию через 1 мс.
    } else if (ADCState == 2) {
        TRISB4 = 1;      // Прекратить левый импульс.
        temp = PORTB;
        RBIF = 0;        // Разрешение прерываний
        RBIE = 1;        // по изменении сигнала.
        ADCState++;
    } else if (ADCState == 3) {
        LeftADC = 0xOFF;
        temp = PORTB;
        RBIF = 0;        // Запрещение прерываний
        RBIE = 0;        // по изменении сигнала.
        ADCState++;
    } else if (ADCState == 4) {
        TRISB5 = 1;      // Прекратить правый импульс.
        temp = PORTB;
        RBIF = 0;        // Разрешение прерываний
        RBIE = 1;        // по изменении сигнала.
        ADCState++;
    } else if (ADCState == 5) {
        RightADC = 0xOFF;
        temp = PORTB;
        RBIF = 0;        // Запрещение прерываний
        RBIE = 0;        // по изменении сигнала.
        ADCDelay = 50;
        ADCState = 0;
    }
}

}

// Конец обработчика прерываний от таймера.

// Здесь можно разместить другие обработчики прерываний.

if (INTF) {              // Обработка прерываний по изменении
                          // сигнала на входе RBO/INT.
    DateTime = CurrentRTC + TMR0; // Время для текущего бита.
}

```

```

    CurrentRTC = 0x0FFFF - TMR0; // Время для следующего бита.
    INTF = 0;                      // Сброс флага прерываний.
} // Конец обработчика прерываний по изменении сигнала на RBO/INT.

if (RBIF) {                       // Обработка прерываний по изменении
    // сигнала на входных линиях порта PORTB.
    if (ADCState == 3) {
        LeftADC = TMR0;          // Для левого фоторезистора.
        ADCState++;
    } else {
        RightADC = TMR0;         // Для правого фоторезистора.
        ADCDelay = 50;           // Повторять каждые 50 мс.
        ADCState = 0;
    }
    temp = PORTB;
    RBIF = 0;                    // Запретить прерывания по изменении
    RBIE = 0;                    // сигнала на входах PORTB.
} // Конец обработки прерываний по изменении
// сигнала на входных линиях порта PORTB.

} // Конец обработчика прерываний.

// Служебные подпрограммы:
void Delay(int msecs)             // Задержка на msecs миллисекунд.
{
    int valueDelay;

    valueDelay = RTC + msecs + 1; // Время окончания задержки.
    while (valueDelay != RTC);     // Ждать.
} // Конец функции Delay.

void LEDOutput(int state)         // Установка состояния светодиода.
{
    if (state)
        RA4 = 0;                 // Включить светодиод.
    else
        RA4 = 1;                 // Выключить светодиод.
} // Конец функции LEDOutput.

int GetLeftLight() // Опрос состояния левого фоторезистора.
{
    return LeftADC;
}

>

int GetRightLight() // Опрос состояния правого фоторезистора.
{
    return RightADC;
}

int GetLeftWhisker() // Опрос состояния левого контактора.
{
    if (LeftCollision == 3)
        return 1;                // Есть касание.
    else

```

```

        return 0;                // Нет касания.
    }

    int GetRightWhisker()        // Опрос состояния правого контактора.
    <
        if (RightCollision == 3)
            return 1;            // Есть касание.
        else
            return 0;            // Нет касания.
    }

    void LeftMotor(int Movement, int Speed) // Управление левым двигателем.
    {
        switch (Movement) {
            case 1:                // Вперед.
                motorState = (motorState & 0x0F9) + 2;
                break;
            case 0:                // Стоп.
                motorState = motorState & 0x0F9;
                break;
            case -1:               // Назад.
                motorState = (motorState & 0x0F9) + 4;
                break;
        }

        PWMDuty = Speed;
    }

    void RightMotor(int Movement, int Speed) // Управление правым двигателем.
    {
        switch (Movement) {
            case 1:                // Вперед.
                motorState = (motorState & 0x03F) + 0x040;
                break;
            case 0:                // Стоп.
                motorState = motorState & 0x03F;
                break;
            case -1:               // Назад.
                motorState = (motorState & 0x03F) + 0x080;
                break;
        }

        PWMDuty = Speed;
    }

    // Главная программа:
    void main(void)
    {
        OPTION = 0x001;           // Предделитель работает с таймером TMRO,
                                   // коэффициент деления 1:4.
        TMRO = 0;                 // Начальный сброс таймера TMRO.
        TOIE = 1;                 // Разрешение прерываний от таймера TMRO.
        GIE = 1;                  // Разрешение обработки прерываний.
    }

```



```

// Здесь можно разместить код верхнего уровня.

INTEDG = 1;           // Прерывания вырабатываются по положительному
                      // перепаду сигнала на выходе RBO/INT.

CCPR1L = 13;          // Ширина ШИМ-импульсов равна 50% периода.
CCP1CON = 0b000001111; // Разрешить работу модуля ШИМ.

PR2 = 26;             // При частоте 38 кГц длительность
                      // периода равна 26 мкс.

TMR2 = 0;             // Сброс таймера TMR2.
T2CON = 0b000000100;  // Включить таймер TMR2,
                      // коэффициент делителя 1:1.

CHCON = 0x007;        // Выключить компараторы порта PORTA.

PORTA = 0x010;        // RA4 - выход
                      // (светодиод пока не горит).

TRISA = 0x0EF;

PORTB = 0x000;
TRISB = 0x039;        // RB7/RB6 и RB2/RB1 - выходы.

while (1 == 1) {      // Бесконечный цикл.
    if (ExecuteFlag) {
        // Здесь можно разместить код для управления
        // действиями робота.
    } else {
        // Здесь можно разместить код для сброса
        // робота в начальное состояние.
    }
}

} // Конец цикла while.
} // Конец главной программы.

```

Используя эту программу в качестве основы, можно поэкспериментировать со следующим набором функций:

```

int GetLeftWhisker(); // Опрос состояния левого
                      // детектора столкновений.
int GetRightWhisker(); // Опрос состояния правого
                      // детектора столкновений.
// Детекторы могут быть контактными или инфракрасными.

int GetLeftLight();   // Опрос состояния левого фоторезистора.
int GetRightLight();  // Опрос состояния правого фоторезистора.
// Фоторезисторы могут быть включены по схеме делителя напряжения
// или в виде RC-цепочки.

void LeftMotor(int Movement [ , int Speed ] ); // Управление
                                                // левым двигателем.
void RightMotor(int Movement [ , int Speed ] ); // Управление
                                                // правым двигателем.
// Второй параметр (Speed) доступен при использовании инфракрасного
// пульта дистанционного управления, но недоступен для радиуправления.

void LED(iut State); // Включить/выключить светодиод.

```

```
void LCDOutput(char * Message); // Вывод сообщения на ЖКИ.  
void Delaynt(msecs);           // Функция задержки.
```

При использовании инфракрасного пульта дистанционного управления я обычно назначаю для кнопки 2 команду «Вперед», 4 и 6 соответствуют поворотам влево и вправо, 8 - «Назад» (реверс обоих двигателей), 5 - «Стоп» (остановка обоих двигателей).

С помощью кнопки, которая в обычных применениях управляет громкостью звучания телевизора, можно изменять скорость движения робота. Ширина импульса **ШИМ-сигнала**, генерируемого в нашей программе для управления обоими двигателями, по умолчанию установлена равной половине периода (50%). Если довести это значение до 100%, то робот начнет довольно быстро носиться по комнате. К сожалению, в таком режиме батарейки хватит ненадолго.

В программе предполагается, что для начала выполнения кода биологического уровня надо нажать кнопку включения питания на пульте дистанционного управления. Для отключения кода следует подать сигнал «Стоп» (кнопка 5) или какую-либо другую прямую команду. Обеспечение или отключение в произвольное время «самостоятельности» робота может потребоваться для того, чтобы сначала с помощью дистанционного управления подвести его к какому-либо месту, затем запустить для проверки биологический код, а если робот попадет в трудную ситуацию, снова взять управление на себя.

Следует заметить, что функция, выполнение которой наиболее трудно обеспечить, - это именно дистанционное управление. Если вы достаточно придирчиво испытаете робота в действии, то обнаружите, что при нажатии на кнопки пульта он может начать двигаться рывками и вести себя так, будто вместо одной команды вы подали сразу несколько.

Дело в том, что при работе двигателей присутствует большое число помех, оказывающих влияние на функционирование всех остальных подсистем, в том числе и приемника команд дистанционного управления. В следующей главе мы обсудим некоторые возможные пути решения подобных проблем.

Для проверки работы механических и световых датчиков, а также инфракрасного обнаружителя объектов надо сначала остановить оба двигателя, после чего подождать как минимум 60 мс, чтобы дать возможность управляющей программе опросить состояние всех датчиков. Это время необходимо для наиболее медленных входных устройств - **РС-цепочек** с фоторезисторами. Для проверки датчиков обнаружителя объектов достаточно и 10 мс.

А еще лучше выключать двигатели на 6 мс каждый раз, пока продолжается очередной цикл работы с фоторезисторами и датчиком обнаружителя объектов.

Электродвигатели, входящие в комплект рассматриваемой модели, рассчитаны на напряжение 1,5 В, поэтому для работы с нашей 9-вольтовой батареей они имеют слишком маленькое сопротивление и в результате потребляют слишком большой ток. Вы можете убедиться в том, насколько велики потери напряжения на электронных ключах микросхемы L293D и как сильно она греется при работе.

Если использовать электродвигатели, рассчитанные на более высокое напряжение, то батарейка прослужит намного дольше, а нагрев микросхемы L293D резко уменьшится.

4.34. ОДОМЕТРИЯ

Одометр - это устройство, связанное с колесом робота, которое по количеству сделанных оборотов позволяет измерять пройденное им расстояние. Эти данные можно использовать в программе для вычисления текущих координат робота (навигация) или для измерения скорости его движения (спидометр). В этом разделе мы рассмотрим основные методы одометрии, которые можно применять в радиолюбительских конструкциях мобильных автоматических устройств.

Чтобы не разочаровывать читателя, должен сразу признаться, что здесь вы не найдете конкретных примеров реализации подобных функций. Во-первых, они были бы слишком громоздки на фоне остальных представленных в этой книге простых примеров, а во-вторых, я всерьез сомневаюсь в возможности реализации описанных в данном разделе алгоритмов с помощью микроконтроллера PIC16F627.

Сам по себе принцип измерения расстояния по количеству оборотов двигателя или колеса довольно прост. Не слишком сложна и реализация этого принципа. Именно так ваш компьютер узнает о перемещении манипулятора типа «мышь».

Если вы разберете компьютерную мышку (заметьте, я сказал *«если»!*), то обнаружите, что вращающийся внутри нее шарик соединен с двумя колесиками, имеющими по краям множество отверстий. По одну сторону от каждого колесика расположены два оптических излучателя, а по другую напротив каждого излучателя имеется оптический детектор (рис. 4.85). Пара «излучатель-детектор» вместе с прорезями на ободе колесика образуют так называемый *оптический прерыватель*. Во время вращения колесика принимаемый детектором сигнал периодически прерывается, так как между детектором и излучателем прозрачные и непрозрачные участки сменяют друг друга. Число принятых импульсов соответствует количеству отверстий; таким образом, обычный цифровой счетчик, подключенный к прерывателю, позволяет измерить угол, на который повернулось колесико. Два излучателя и детектора нужны для того, чтобы определять направление вращения.

Если мы хотим использовать тот же принцип в наших автоматических устройствах, то должны соединить оптический прерыватель непосредственно с валом

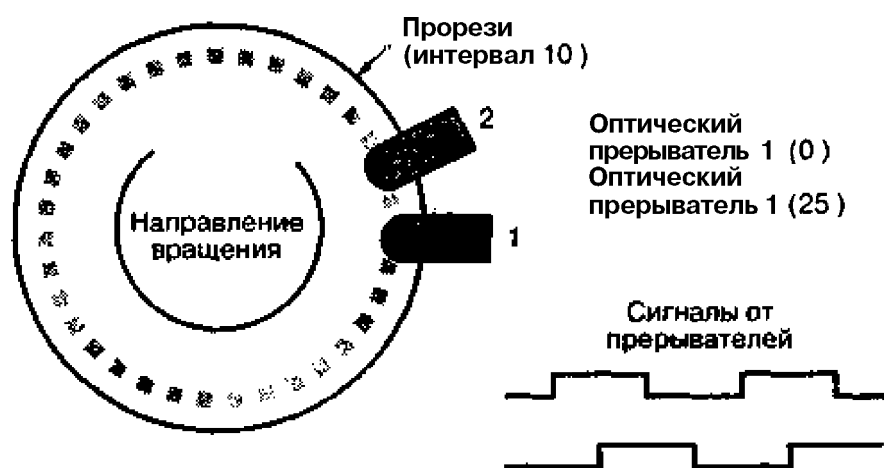


Рис. 4.85. Оптический детектор вращения колеса

двигателя. Прим этом в большинстве случаев достаточно одного излучателя и детектора, так как направление вращения известно управляющей программе.

Зная угол, на который повернулся вал в каждый момент времени, несложно вычислить скорость движения робота V : для этого можно воспользоваться формулой

$$V = 2\pi r N / (MT),$$

где r - радиус колеса робота; M - количество отверстий в колесике прерывателя; N - число импульсов, принятых счетчиком за время T .

Например, если колесо радиусом 3 см используется вместе с прерывателем, в котором имеется 30 отверстий, то 10 импульсов в секунду соответствуют скорости движения робота

$$V = 2 \times 3,14 \times 3 \text{ см} \times 10 / (30 \times 1 \text{ с}) = 6,28 \text{ см/с}.$$

Разумеется, это не слишком высокая скорость. Среднее значение обычно составляет около 15 см/с, а скорость, достигающая 1 м/с, по-настоящему впечатляет, особенно если робот при этом ни во что не врежется.

Используя один датчик и предполагая, что оба колеса вращаются с одинаковой скоростью, можно рассчитать длину пути, пройденного роботом за данный период времени.

Если ваше устройство основано на модели мыши фирмы Tamiya, вы вряд ли заставите его двигаться прямолинейно. Когда каждое колесо управляется отдельным двигателем и не принимаются специальные меры по точной подстройке скорости их вращения (а эти меры требуют использования двух прерывателей - по одному на каждое колесо), то разброс параметров двигателей и различные силы трения, действующие в левом и правом приводе, всегда приведут к тому, что робот будет стремиться повернуть в какую-либо сторону.

Впрочем, даже если скорости вращения левого и правого колеса немного отличаются друг от друга, разница эта настолько мала, что ею можно пренебречь при вычислении длины пройденного пути. Но, разумеется, этого же нельзя утверждать применительно к направлению движения: даже самый небольшой дисбаланс приводит к заметным искривлениям траектории.

Что касается методов стабилизации направления движения, основанных на регулировании скорости вращения левого и правого двигателей, то их рассмотрение и тем более программирование относятся к области высшего пилотажа в теории автоматического управления, а потому выходят за рамки данной книги (хотя это интересные темы и, быть может, я вернусь к ним в какой-нибудь следующей книге). Чтобы вы получили возможность оценить круг возникающих при этом проблем, приведем два основных момента, которые приходится учитывать.

Во-первых, надо знать инерцию двигателей (и самого робота) при изменении управляющих сигналов. Очевидно, что двигатели не могут сразу начать вращаться с той скоростью, которую мы для них хотим задать. Это очень важный вопрос, потому что два двигателя, которые по-разному ускоряют или замедляют свое вращение, даже хуже, чем два двигателя, вращающиеся с разной скоростью.

Во-вторых, надо рассчитать стабильный режим работы. А он зависит от того, насколько правильно двигатель «держит» заданную скорость.

Наиболее распространенный метод управления двигателями основан на использовании так называемого пропорционально-интегрально-дифференциального регулятора (ПИД). Входным сигналом регулятора обычно является отклонение измеряемой величины (в нашем случае это скорость вращения) от ее желаемого значения. Как следует из названия, ПИД-регулятор состоит из трех частей: пропорциональная часть вырабатывает управляющий сигнал, величина которого пропорциональна входному сигналу (то есть текущее отклонение скорости вращения двигателя от желаемого умножается на какой-то коэффициент); выходной сигнал интегральной части, как нетрудно догадаться, вычисляется с помощью интегрирования входной величины за некоторый уже прошедший промежуток времени (другими словами, на какой-то другой коэффициент умножается лишнее число оборотов, которые двигатель совершил за данное время); наконец, дифференциальная часть вырабатывает управляющий сигнал по величине скорости изменения входного сигнала (то есть на некий третий коэффициент умножается ошибка ускорения вращения двигателя). Все это проще выразить формулой

$$u = k_v v + k_n n + k_a a,$$

где v , n и a - это соответственно ошибка скорости, ошибка числа оборотов и ошибка ускорения вращения двигателя, а k_v , k_n и k_a - три коэффициента, значения которых необходимо подобрать так, чтобы достичь оптимальных характеристик проектируемой системы.

Существуют специальные методы (и программы) для расчета этих коэффициентов, но их можно подобрать и экспериментально, если предусмотреть какой-либо способ их оперативного изменения, например с помощью пульта дистанционного управления.

В первую очередь надо выбрать оптимальное значение коэффициента k_v пропорциональной части. Это лучше делать, испытывая работу двигателей и регулятора на неизменных режимах. Затем можно заняться подстройкой остальных двух коэффициентов, обращая внимание на моменты включения и выключения двигателей. Необходимо убедиться, что при смене режима работы не происходит слишком резких рывков или даже изменения направления вращения двигателей.

Вполне возможно реализовать с помощью микроконтроллера PIC16F627 все необходимые для ПИД-регулятора вычисления в промежутке между двумя вызовами обработчика прерываний от таймера TMRO, происходящих каждую миллисекунду. В принципе, эти вычисления лучше было оформить в виде биологического кода верхнего уровня, но на самом деле соответствующий уровень не совсем биологический. Он скорее механический, и его выполнение в главной программе может помешать работе действительно биологического кода. В некоторых случаях целесообразно рассмотреть вопрос о возможности использования отдельного микроконтроллера, который взял бы на себя функции ПИД-регулятора.

С помощью информации, получаемой от двух оптических прерывателей, можно с той или иной точностью вычислить текущие координаты робота, подобно тому как это делается в самолетах с помощью системы INS (Inertial Navigation System). В ней компьютеры с помощью нескольких гироскопов определяют, какое ускорение действует на воздушный лайнер, и тем самым вычисляют его текущие координаты.

Для обсуждения хотя бы самых основ этих методов нам потребуется высшая математика. Если вы не помните, что такое тригонометрия, вам стоит освежить свои знания, заглянув в учебник по математике для старших классов.

Как бы то ни было, сама по себе идея управления роботом на основе навигационной информации не так уж и сложна. Если известны начальная и конечная позиции робота, то можно вычислить направление движения и расстояние между ними.

Например, робот сейчас находится в точке плоскости (x, y) с координатами $(x_1, y_1) = (0, 0)$, причем его ориентация совпадает с осью абсцисс, а мы хотим переместить его в точку с координатами $(x_2, y_2) = (3, 4)$. Сначала надо рассчитать величину угла, на который должен повернуться робот. Тангенс этого угла определяется по формуле

$$\operatorname{tg} \alpha = \Delta y / \Delta x = (y_2 - y_1) / (x_2 - x_1).$$

Для нашего примера получаем

$$\operatorname{tg} \alpha = \Delta y / \Delta x = 4 / 3 = 1,333...$$

Для вычисления самого угла приходится использовать функцию *арктангенс*:

$$\alpha = \operatorname{arctg} \Delta y / \Delta x.$$

Для нашего случая $\operatorname{arctg} (4/3) = 53,13^\circ = 0,927$ рад.

Разумеется, вы помните теорему Пифагора, поэтому без труда вычислите, что после поворота на этот угол робот должен преодолеть

$$s = \sqrt{\Delta x^2 + \Delta y^2} = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$$

единиц расстояния, чтобы попасть в нужную точку.

Проблема состоит в том, что идеального робота не существует. Трудно добиться поворота на точно заданный угол, и нелегко правильно измерить пройденное расстояние, особенно если учесть, что робот перемещается не строго по прямой линии. На рис. 4.86 показана траектория, по которой движется робот, если разница в скорости вращения двух его колес остается строго постоянной. Эта траектория близка к окружности. Если робот будет двигаться в таком режиме достаточно долго, а угол поворота будет выдерживаться весьма точно, то, совершив полный оборот вокруг некоторой точки, робот вернется в то место, откуда начал свой путь.

Для определенности будем называть текущей позицией робота ту его точку, которая лежит на оси строго посередине между двумя его колесами. Рассмотрим для примера рис. 4.87. Робот медленно поворачивается направо; при этом его правое колесо проходит путь, длина которого чуть меньше той, что приходится

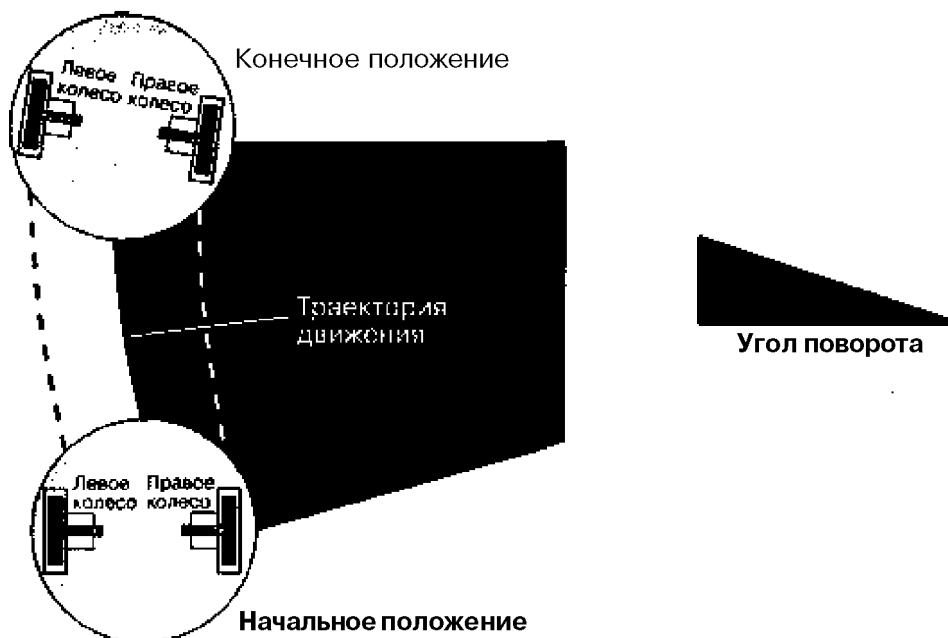


Рис. 4.86. Траектория движения робота



Рис. 4.87. Вычисление координат по данным одометрии

преодолевать левому колесу. Обозначим буквой X длину пути, пройденную внутренним (при повороте направо - правым, при повороте налево - левым) колесом, а длину пути, пройденного за то же время вторым колесом, - буквой Y . Треугольник на рис. 4.87 при небольших углах поворота похож на прямоугольный. Тогда, согласно формулам тригонометрии, для внутреннего треугольника

$$X = R \sin \alpha; \quad X / R = \sin \alpha.$$

Здесь R - радиус поворота (то есть радиус окружности, которую образует траектория движения робота); α - угол поворота.

Для наружного треугольника

$$Y = (R + Z) \sin \alpha; \quad Y / (R + Z) = \sin \alpha,$$

где Z - расстояние между колесами.

Приравняв второе равенство, записанное для одного колеса, и второе равенство, записанное для другого колеса, получим

$$X / R = Y / (R + Z).$$

Отсюда можно найти радиус окружности, которую описывает внутреннее колесо:

$$R = XZ / (Y - X).$$

При известных значениях X , Y и Z можно без труда определить R , что, в свою очередь, позволяет вычислить угол поворота:

$$a = \arcsin(X / R).$$

Если, например, $X = 2$ см, $Y = 2,1$ см, а расстояние между колесами $Z = 4$ см, то

$$R = XZ / (Y - X) = 2 \times 4 / (2,1 - 2) = 8 / 0,1 = 80 \text{ см};$$

$$a = \arcsin(X / R) \approx \arcsin(2 / 80) = \arcsin 0,025 = 0,025 \text{ рад} = 1,43^\circ.$$

Заметим, что арксинус малого угла всегда примерно равен величине самого угла, выраженной в радианах.

Позицию робота мы договорились определять по координатам его средней точки, которая на $Z/2$ отстоит от внутреннего колеса. Поэтому радиус окружности, которую описывает эта точка, равен $R + Z/2 = 82$ см. Следовательно, конечные координаты робота

$$y_2 = (R + Z / 2) \sin \alpha = 82 \times \sin 0,025 = 2,05 \text{ см};$$

$$x_2 = (R + Z / 2) (\cos \alpha - 1) \approx 0.$$

Итак, конечные координаты робота в нашем примере (0; 2,05).

Глядя на эти формулы, вы можете решить, что теперь легко сумеете направить робота напрямик к любой точке комнаты. К сожалению, это не совсем так. Применяя формулы на практике, следует учитывать, что с увеличением расстояний, которые используются при вычислениях, растет и погрешность результата, поскольку перестают выполняться допущения, которые были сделаны при выводе формул. Кроме того, в программе придется использовать вычисления с плавающей точкой, причем все выкладки надо делать быстро, иначе вы не будете успевать остановить робота в нужной точке.

А если еще учесть неточность определения начальной позиции и весьма вероятные проскальзывания колес, особенно заметные, когда робот будет преодолевать неровные участки своего пути, наши выкладки окажутся еще менее надежными, а задача навигации на основе одометрии еще более усложнится.

4.35. РАДИОУПРАВЛЯЕМЫЙ СЕРВОПРИВОД

Для передачи команд дистанционного управления различными игрушками часто используются радиопередатчик и сервопривод с радиоприемником. Поэтому при конструировании робота мы можем использовать серводвигатели, установленные на радиоуправляемых моделях самолетов, автомобилей или лодок. Радиоуправляемый сервопривод (R/C servo) обычно имеет на выходе колесо, которое может

поворачиваться на угол в диапазоне от 0° до 90° (некоторые устройства - от 0° до 180°). Усилие (вращающий момент), развиваемое серводвигателем, обычно невелико, хотя существуют специальные устройства, в которых оно может достигать больших значений. Напряжение питания в большинстве случаев составляет +5 В. Никаких сложных подключений сервоприводы не требуют - чаще всего используются всего три внешних вывода: напряжение питания, общий вывод и входной сигнал управления.

Серводвигатель является аналоговым устройством, управляемым с помощью широтно-модулированного сигнала постоянной амплитуды. Обычно это импульсы длительностью от 1 до 2 мс, подаваемые каждые 20 мс (рис. 4.88).

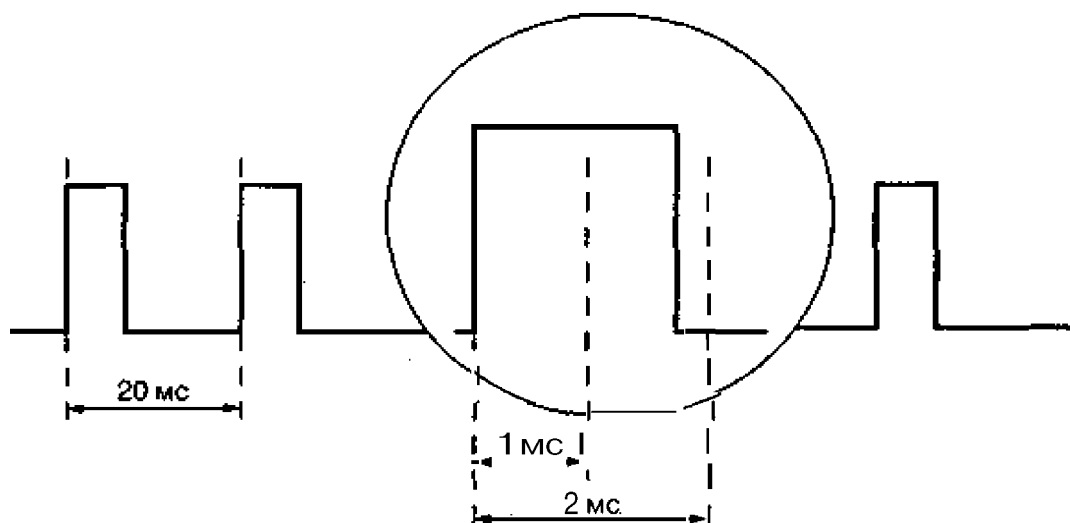


Рис. 4.88. Временная диаграмма ШИМ-сигнала управления сервоприводом

Большая длительность импульсов соответствует большему углу поворота серводвигателя. Для формирования управляющего ШИМ-сигнала можно использовать следующий код, который понадобится поместить в процедуру обработки прерываний от таймера:

```
Interrupt() {                                     // Процедура обработки прерываний.
int i = 0;
  BitOutput( Servo, 1 )                          // Выходной сигнал.
  for ( i = 0, i < (1 msec + ServoDelay); i++ );
  BitOutput( Servo, 2 );
  for ( ; i < 2 msec; i++ );                      // Задержка на 2 мс.
}                                                  // Конец обработчика прерываний.
```

Эту программу несложно модифицировать для работы с несколькими серводвигателями (добавив новые выходные переменные Servo и счетчики для формирования задержек ServoDelay). Основное преимущество продемонстрированного подхода заключается в том, что значение переменной ServoDelay можно изменять, не оказывая никакого влияния на выполнение обработчика прерываний.

Приведенный код прекрасно работает с автономным радиоуправляемым устройством, но показывает не слишком хорошие результаты, если мы захотим

использовать его в качестве одного из многих интерфейсов, реализованных в виде процедуры обработки прерываний от таймера, срабатывающего каждую миллисекунду. В этом случае миллисекундная задержка оказывается слишком большой. Надо раза в четыре уменьшить период срабатываний таймера, доведя его значение хотя бы до 256 мкс. Обычно радиоуправляемые сервоприводы могут распознавать около пяти команд; возможные варианты и соответствующие длительности импульсов приведены в табл. 4.19.

Таблица 4.19. Команды радиоуправляемого сервопривода

Номер команды	Задержка во времени, мс	Команда
1	1,00	Реверс на полной скорости
2	1,25	Реверс на половинной скорости
3	1,50	Стоп
4	1,75	Вперед на половинной скорости
5	2,00	Вперед на полной скорости

Серводвигатель можно заставить вращаться постоянно, заменив в его схеме потенциометр, который входит в цепь обратной связи, управляющей положением привода, подстроечным резистором, а также убрав пластмассовый выступ, который ограничивает угол поворота привода, не давая ему повернуться на угол, больший 90°. Положение движка подстроечного резистора следует подобрать экспериментально. Мы еще вернемся к этому вопросу в следующем разделе.

4.36. ПРОСТОЕ РАДИОУПРАВЛЯЕМОЕ УСТРОЙСТВО

Здесь мы рассмотрим простую радиоуправляемую конструкцию на основе стандартного серводвигателя. Нам потребуется два таких двигателя, каждый из которых будет управлять соответствующим колесом робота. Радиоуправляемые серводвигатели не слишком дороги и достаточно неприхотливы с точки зрения как требуемых для их подключения аппаратных средств, так и управляющего программного обеспечения. Но их придется немного модифицировать, чтобы они могли крутиться постоянно.

На постройку описанной здесь конструкции я затратил не более получаса, включая то время, которое потребовалось на модификацию обоих серводвигателей.

Я использовал лист толстой фанеры размером 8х3,75 дюймов (203х95 мм). Перед размещением элементов поверхность фанеры желательно покрыть грунтовкой (для этого удобно использовать пульверизатор с краской), чтобы элементы приклеились к основанию. В крайнем случае можно обмотать уже собранное устройство изоляционной лентой.

Внешний вид приемника показан на рис. 4.89. Принципиальная схема приведена на рис. 4.90, а использованные элементы описаны в табл. 4.20.



Рис. 4,89. Приемник команд радиуправления

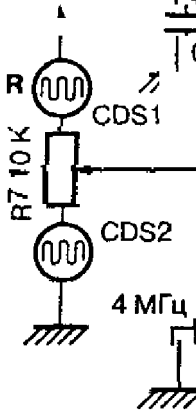


Рис. 4.90. Принципиальная схема приемника команд радиуправления

Таблица 4.20. Перечень использованных элементов

Обозначение	Элемент	Примечание
U1	PIC16F627	Микроконтроллер
U3	Инфракрасный детектор	PIC-1018SCL фирмы Waitronyили другой совместимый /три вывода)
CR1	Светодиод	Любой видимого диапазона
R1, K3, R5, R6, R8	10 кОм; 0,25 Вт	

Таблица 4.20. Перечень использованных элементов (окончание)

Обозначение	Элемент	Примечание
R2	470 Ом; 0,25 Вт	
R4	100 Ом, 0,25 Вт	
R7	10 кОм (подстроечный)	
CDS1, CDS2	Фоторезистор, 10 кОм при нулевом освещении	При освещении должен уменьшать свое сопротивление
C1	0,1 мкФ	Конденсатор любого типа
C2	47 мкФ; 16 В	Оксидный конденсатор
C3 – C5	10 мкФ; 16 В	Оксидный конденсатор
SW1	Выключатель любого типа	Выключатель питания
SW2, SW3	Микропереключатель	Для детектора столкновений
XTAL1	Керамический резонатор на 4 МГц, со встроенными конденсаторами	Для генератора тактовых импульсов микроконтроллера
	Материалы	Два модифицированных серводвигателя, два колесика от модели самолета, фанера, макетная плата, монтажные провода, четыре батарейки 1,2 В типа AA

Сборку устройства лучше производить в следующем порядке:

1. Модифицируйте оба сервопривода для постоянного вращения двигателей, как описано выше.
2. Подсоедините серводвигатели к колесам робота.
3. С помощью эпоксидной смолы закрепите на днище робота в передней его части светодиод для индикации срабатывания детектора столкновений (одновременно он может играть роль третьей точки опоры для всей конструкции).
4. Прикрепите к корпусу серводвигатели, батареи питания и контактные усики обоих детекторов столкновений.
5. Выполните необходимые соединения между компонентами конструкции.
6. Запрограммируйте микроконтроллер и проверьте, как работает дистанционное управление.

Все шесть этапов достаточно просты, но надо соблюдать внимательность, чтобы ничего не упустить.

Первый этап (модификация сервопривода) не должен вызвать затруднений, но прежде чем приобретать ту или иную конкретную модель, следует убедиться, что вы сможете найти для нее нужную документацию (возможно, в сети Internet), где будет подробно объяснено, как модифицировать привод для того, чтобы двигатели могли совершать полный оборот вокруг своей оси. В некоторых случаях такая операция оказывается довольно затруднительной. Лучше всего приобрести сервопривод попроще и подешевле.

Как уже говорилось, надо будет удалить встроенный потенциометр (или два резистора, которые его заменяют) и поставить вместо него подстроечный резистор,

который можно просто приклеить к корпусу двигателя. Резистор следует расположить так, чтобы вы могли изменить положение движка, не меняя положения робота, пока он стоит на месте. Разумеется, эту регулировку можно было бы реализовать программно, но управляющая программа от этого неоправданно усложнилась бы. Гораздо проще подстроить сервопривод с помощью отвертки.

Используемые в конструкции колеса должны подходить для серводвигателей. Самое простое — взять колесики от модели самолета (диаметром 2 или 3 дюйма, то есть 5-7,5 см) и просверлить в центре каждого из них небольшое отверстие, в которое мог бы пройти болт, — с его помощью колесики будут крепиться к серводвигателям. Затем надо наклеить сверху (на ступицу) декоративную крышку, продающуюся в комплекте с колесиком, используя, к примеру, эпоксидную смолу (рис. 4.91).

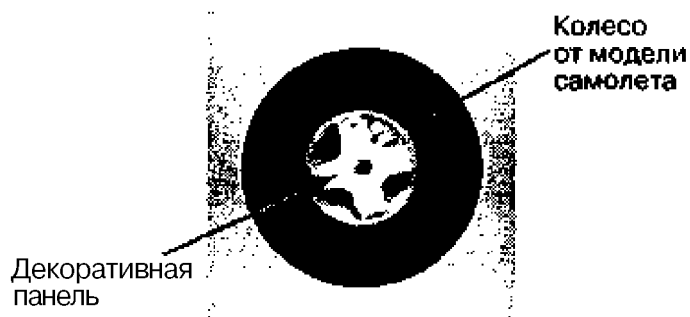


Рис. 4.91. Одно из **двух** колес для **радиоуправляемого** робота

После того как все части окажутся скреплены друг с другом, настанет время поместить их в корпус робота. Для крепления батареек, сервопривода и электронного блока управления можно обмотать их липкой лентой.

Микропереключатели детектора столкновений также можно приклеить к корпусу робота.

Работая с клеем или эпоксидной смолой, не забывайте следовать инструкциям завода-изготовителя и хорошо проветривайте помещение. Выделяемые при этом пары не слишком ядовиты, но могут вызывать неприятные ощущения.

На рис. 4.89 видно, что сбоку от батареек с одной стороны мы оставили место для выключателя питания, а с другой — для проводов, идущих к серводвигателям. Выключатель можно приклеить, а для крепления проводки следует просверлить отверстия в фанерном корпусе.

Если вы не новичок и умеете планировать свои действия на несколько шагов вперед, то можете приклеить оба контактных детектора, светодиод, переключатель, подстроечные резисторы серводвигателей и декоративные крышки колесиков за один раз, но если это первая ваша конструкция, лучше не пытаться все сделать единым махом.

В отличие от прошлых проектов, здесь не применяется стабилизатор напряжения. Как видно на рис. 4.89, в качестве источника питания используются четыре батарейки типа АА (они могут быть никель-кадмиевые или металлгидридные). Каждая из них дает напряжение 1,2 В, поэтому при последовательном соединении батареек получается источник питания напряжением 4,8 В, чего вполне достаточно для микроконтроллера и серводвигателей.

Подсоединение остальных элементов не должно вызвать у вас затруднений. Последовательно со светодиодом надо включить резистор 470 Ом. Светодиод будет сигнализировать о срабатывании левого или правого детектора столкновений.

Такая обратная связь очень пригодится при настройке контактных усиков. Кроме того, надо подключить «подтягивающие» резисторы сопротивлением 10 кОм к микропереключателем детекторов столкновений.

Хотя мы взяли сервопривод от радиоуправляемой модели, радиоприемник и передатчик нам не потребуются. В этой конструкции опять же использованы инфракрасный пульт дистанционного управления и инфракрасный детектор фирмы Sony. Все это работает точно так же, как и в предыдущем проекте, выполненном на основе модели мыши фирмы Tamiya.

А датчики света для простоты взяты из нашей первой конструкции такого типа. Если вы были внимательны, то уже заметили подстроечный резистор (элемент R7 на рис. 4.90), который предназначен для балансировки плеч делителя напряжения, образованного двумя фоторезисторами.

Когда все будет готово, можно приступить к программированию микроконтроллера PIC16F627. Исходный текст управляющей программы содержится в файле Code\Servo\servo2.c:

```

#include <pic.h>

// 28 апреля 2002 - добавлено дистанционное управление.
// 23 февраля 2002 - разработано Майком Предко.
//
// Используются прерывания от таймера TMRO,
// вырабатываемые каждые 512 мкс.
//
// Замечания по аппаратным средствам:
// Микроконтроллер PIC16F627 работает на частоте 4 МГц.
// Используется внешний тактовый генератор.
//
// Подключение выводов микроконтроллера:
// RA1 - вход компаратора 1 (сигнал с делителя напряжения
// на фоторезисторах);
// RA2 - контрольный выход внутреннего источника опорного напряжения
// для компараторов (половина напряжения питания);
//
// RB0 - инфракрасный детектор для приема команд дистанционного
// управления фирмы Sony (прерывания по положительному перепаду);
static volatile bit IRDetect @ (unsigned)&PORTB*8+0;

// RB1 - правый контактор;
static volatile bit RightWhiskerPin @ (unsigned)&PORTB*8+1;

// RB2 - левый контактор;
static volatile bit LeftWhiskerPin @ (unsigned)&PORTB*8+2;

// RB3 - выход индикаторного светодиода;
static volatile bit LED @ (unsigned)&PORTB*8+3;
static volatile bit LEDTRIS @ (unsigned)&TRISB*8+3;

// RB4 - левый серводвигатель;
static volatile bit leftServo @ (unsigned)&PORTB*8+4;
static volatile bit leftServoTRIS @ (unsigned)&TRISB*8+4;

// RB5 - правый серводвигатель.
static volatile bit rightServo @ (unsigned)&PORTB*8+5;

```

```
static volatile bit rightServoTRIS @ (unsigned)&TRISB*8+5;

// Глобальные переменные и константы:
unsigned int RTC = 0; // Счетчик реального времени.
unsigned char RTCOlav = 2; // Задержка 1 мс (период срабатывания таймера равен 512 мс).

// Константы для серводвигателей:
#define LeftForward 38 // Для левого двигателя.
#define LeftStop 37
#define LeftReverse 36
#define RightForward 36 // Для правого двигателя.
#define RightStop 37
#define RightReverse 38
char ServoDlay = 1; // Выход управления сервоприводом (каждые 20 мс).

char leftSpeed = LeftStop; // Скорость вращения левого двигателя.
char rightSpeed = RightStop; // Скорость вращения правого двигателя.

// Константы для декодера команд дистанционного управления:
#define remote0 0x06EF // Цифры 0-9.
#define remote1 0x0FEF
#define remote2 0x07EF
#define remote3 0x0BEF
#define remote4 0x03EF
#define remote5 0x0DEF
#define remote6 0x05EF
#define remote7 0x09EF
#define remote8 0x01EF
#define remote9 0x0EEF
#define remoteVolUp 0x0B6F // Кнопка "Громче".
#define remoteVolDown 0x036F // Кнопка "Тише".
#define remoteChanUp 0x0F6F // Кнопка "Канал +1".
#define remoteChanDown 0x076F // Кнопка "Канал -1".
#define remotePrevChan 0x022F // Кнопка "Предыдущий канал".
#define remoteMute 0x0D6F // Кнопка "Приглушить звук".
#define remotePower 0x056F // Кнопка "Питание".

unsigned int DataIn; // Входные данные детектора команд управления.
unsigned char DataInCount = 0; // Число символов для приема.
unsigned int DateTime = 0;
unsigned int CurrentRTC = 0;
int OpnCount = 1; // Если двигатели не работают, можно проверить компаратор.

volatile char LeftWhisker = 0; // Счетчик для левого контактора.
volatile char RightWhisker = 0; // Счетчик для правого контактора.

char ExecuteFlag = 0; // Флаг для выполнения биологического кода.

// Слово конфигурации:
#ifdef _16F627
#warning PIC16F627 with external XT oscillator selected
_CONFIG(0x03F21); // Для МК PIC16F627:
// внешний тактовый генератор XT,
// RA6/RA7 используются для ввода-вывода,
// внешний сигнал сброса,
// таймер PWRT включен,
```

```

        // сторожевой таймер выключен,
        // защита кода отключена,
        // детектор BODEN включен.
#else
    (terror Unsupported PICmicro MCU selected
#endif

// Обработчик прерываний:
void interrupt tmr0_int(void)
{
    if (TOIF) {          // Обработчик прерываний от таймера TMR0.
        TOIF = 0;        // Сбросить флаг прерываний от таймера TMR0.
        if (--ServoDelay == 0) {

            ServoDelay = 40;      // Задержка 20 мс.

            leftServo = 1;        // Высокий уровень сигнала
            rightServo = 1;       // управления для обоих
                                   // серводвигателей.

        } else {

            if (leftSpeed >= ServoDelay)
                leftServo = 0; // Закончить подачу импульса
                               // для левого серводвигателя.

            if (rightSpeed >= ServoDelay)
                rightServo = 0; // Закончить подачу импульса
                               // для правого серводвигателя.

        }

        // Декодирование команд дистанционного управления
        // (если оно установлено в системе).

        if (DataTime) {
            if (DataInCount == 0) {      // Новый пакет.
                DataInCount = 12;        // Ждать 12 бит.
                DataIn = 0;
            }
            > else {                     // Очередной бит.
                DataIn = DataIn << 1;
                if ((DataTime > 450) && (DataTime < 750)) {
                    // Получена 1.
                    DataIn++;
                    if (--DataInCount == 0) {
                        OpnCount = 400; // Конец.
                        leftSpeed = LeftForward;
                        rightSpeed = RightForward;
                        if (DataIn == remote2)
                            ;
                        else if (DataIn == remote4)
                            leftSpeed = LeftReverse;
                        else if (DataIn == remote6)
                            rightSpeed = RightReverse;
                        else if (DataIn == remote8) {
                            leftSpeed = LeftReverse;
                            rightSpeed = RightReverse;
                        } else

```



```

        if (DataIn == remotePower) {
            ExecuteFlag = 1;
            leftSpeed = LeftStop;
            rightSpeed = RightStop;
            OpnCount = 1; // Конец.
        } else { // Стоп или ничего.
            ExecuteFlag = 0;
            leftSpeed = LeftStop;
            rightSpeed = RightStop;
            OpnCount = 1; // Конец.
        }
    }
} else if ((DataTime > 750) &&
           (DataTime < 1000)) {
    // Получен 0.
    if (-DataInCount == 0) {
        OpnCount = 400; // Конец.
        leftSpeed = LeftForward;
        rightSpeed = RightForward;
        if (DataIn == remote2)
            ;
        else if (DataIn == remote4)
            leftSpeed = LeftReverse;
        else if (DataIn == remote6)
            rightSpeed = RightReverse;
        else if (DataIn == remote8) {
            leftSpeed = LeftReverse;
            rightSpeed = RightReverse;
        } else
            if (DataIn == remotePower) {
                ExecuteFlag = 1;
                leftSpeed = LeftStop;
                rightSpeed = RightStop;
                OpnCount = 1; // Конец.
            } else { // Стоп или ничего.
                ExecuteFlag = 0;
                leftSpeed = LeftStop;
                rightSpeed = RightStop;
                OpnCount = 1; // Конец.
            }
    }
} else // Ошибка.
    DataInCount = 0;
}
DataTime = 0;
}

CurrentRTC += 0x0100; // Задержка для приема одного бита.

// Проверка на пропущенный импульс:
if ((DataInCount) && (CurrentRTC > 0x01200))
    DataInCount = 0;
    // Сброс и ожидание следующего пакета.

// Код для обработки каждого второго прерывания
// от таймера (работает через 1 мс):
if (--RTCDlay == 0) {

```

```

    RTCDlay = 2;    // Следующий раз - через один вызов.
    RTC++;    // Инкремент счетчика реального времени.

    // Код для левого контактора:
    if (!LeftWhiskerPin)    // Левый контактор сработал.
        if (LeftWhisker < 20)    // Противодребезговая
            LeftWhisker++;    // задержка.
        else;    // Нажат уже 20 мс.
    else    // Контакттор разомкнут.
        if (LeftWhisker != 0)    // Противодребезговая
            LeftWhisker--;    // задержка.
        else;    // Разомкнут уже 20 мс.

    // Код для правого контактора:
    if (!RightWhiskerPin)    // Правый контактор сработал.
        if (RightWhisker < 20)    // Противодребезговая
            RightWhisker++;    // задержка.
        else;    // Нажат уже 20 мс.
    else    // Контакттор разомкнут.
        if (RightWhisker != 0)    // Противодребезговая
            RightWhisker--;    // задержка.
        else;    // Разомкнут уже 20 мс.

    // Работа с фоторезисторами и светодиодом:
    if ((!ExecuteFlag) && (--OpnCount == 0)) {
        OpnCount = 1;
        leftSpeed = LeftStop;
        rightSpeed = RightStop;
        if (C2OUT)
            LED = 0;    // Светодиод включить.
        else
            LED = 1;    // Светодиод выключить.
    }

}

}    // Конец обработки прерываний от таймера.

// Здесь можно разместить другие обработчики прерываний.

if (INTF) {    // Обработка прерываний по изменении
               // сигнала на входе RBO/INT.
    DateTime = CurrentRTC + TMRO;    // Время приема текущего бита.
    CurrentRTC = 0x0FFFF - THRO;    // Время для следующего бита.
    INTF = 0;    // Сброс флага прерывания.
}    // Конец обработки прерываний по изменении входного сигнала.

}    // Конец обработчика прерываний.

// Служебные подпрограммы:
void Dlay(int msecs)    // Задержка на msecs миллисекунд.
{
    int valueDlay;

    valueDlay = RTC + msecs + 1;    // Время окончания задержки.
    while (valueDlay != RTC);    // Ждать.

}    // Конец функции Dlay.

void LEDOutput(int state)    // Управление состоянием светодиода.

```

```
{
    if (state)
        LED = 0;                // Светодиод включить.
    else
        LED = 1;                // Светодиод выключить.
} // Конец функции LEDOutput.

int GetLeftLight()              // Опрос состояния фоторезисторов.
{
    if (C2OUT)
        return 0x080;          // Источник света слева.
    else
        return 0x0FF;
} // Конец функции GetLeftLight.

int GetRightLight()
<
    if (!C2OUT)
        return 0x080;          // Источник света справа.
    else
        return 0x0FF;
} // Конец функции GetRightLight.

int GetLeftWhisker()            // Опрос состояния левого контактора.
<
    if (LeftWhisker == 20)
        return 1;              // Левый контактор сработал.
    else
        return 0;
} // Конец функции GetLeftWhisker.

int GetRightWhisker()           // Опрос состояния правого контактора.
{
    if (RightWhisker == 20)
        return 1;              // Правый контактор сработал.
    else
        return 0;
} // Конец функции GetRightWhisker.

void LeftMotor(int Movement)    // Управление левым двигателем.
{
    leftSpeed = Movement;
} // Конец функции LeftMotor.

void RightMotor(int Movement)   // Управление правым двигателем.
{
    rightSpeed = Movement;
} // Конец функции rightMotor.

// Главная программа;
```

```

void main(void)
{
    OPTION = 0x000;    // Предделитель работает с таймером TMR0,
                        // коэффициент деления 1:2.
    TMR0 = 0;          // Начальный сброс таймера TMR0.
    TOIE = 1;          // Разрешение прерываний от таймера TMR0.
    GIE = 1;           // Разрешение обработки прерываний.
    // Здесь можно разместить код верхнего уровня.

    INTEDG = 1;        // Прерывания вырабатываются по положительному
                        // перепаду сигнала на выходе RBO/INT.
    INTF = 0;          // Сброс флага прерываний.
    INTE = 1;          // Разрешение прерываний по изменении
                        // сигнала на входе RBO/INT.

    CMCON = 0x002;     // Разрешение работы компараторов:
                        // выходы компараторов не инвертируются;
                        // бит CIS = 0;
                        // режим 2 (внутренний источник
                        // опорного напряжения).
    TRISA = 0x007;     // Входными являются только RA0 и RA1.
    VRCON = 0x0EC;     // Разрешение работы внутреннего источника
                        // опорного напряжения:
                        // для контроля его напряжение выводится
                        // на выход RA2,
                        // верхний поддиапазон,
                        // установочное значение равно 12.

    LED = 1;           // Разрешить работу выхода, к которому
    LEDTRIS = 0;        // подключен светодиод.

    leftServo = rightServo = 0; // Начальные значения
                                // для сигналов управления двигателями.
    leftServoTRIS = rightServoTRIS = 0;

    // Биологический код:
    while (1 == 1) {    // Бесконечный цикл.

        if (ExecuteFlag) {
            // Здесь надо разместить код, определяющий поведение робота.
        } else {
            // Код для инициализации состояния робота.
        }

    } // Конец оператора while.
} // Конец главной программы.

```

Теперь необходимо провести две операции по калибровке систем. Во-первых, следует проградуировать подстроечные резисторы серводвигателей. Это не должно вызвать у вас затруднений, особенно если вы уже испытали в работе предыдущую конструкцию. Надо просто включить питание робота и с помощью отвертки медленно изменять положение движка резистора, пока двигатель не остановится.

Во-вторых, необходимо настроить подстроечный резистор R7, который предназначен для балансировки плеч делителя напряжения на фоторезисторах. Ваша

задача - добиться такого положения его движка, при котором индикаторный светодиод мигает, если робот незначительно отклоняется от направления на источник света. В идеале все настройки надо проводить, создав те же условия (используя то же помещение, ту же освещенность), при которых планируется испытывать робота.

После всех проверок вы, возможно, захотите усовершенствовать конструкцию робота, добавив к ней более сложные интерфейсные модули, которые уже описывались в предыдущих проектах.

ГЛАВА 5

ВДОХНИТЕ В РОБОТА ЖИЗНЬ

В предыдущей главе мы рассмотрели достаточно много примеров реализации функций двух нижних (механического и электронного) уровней в виде обработчика прерываний от таймера, формируемых каждую миллисекунду. Что касается функций верхнего (биологического) уровня, то представленный код выполнял пока лишь только тестовые функции, позволяя проверить работу интерфейсов нижнего уровня. Одновременно мы обсудили методы совмещения нескольких интерфейсов в одной процедуре обработки прерываний.

В этой главе, наконец, будет показано, как реализуется выполнение функций **верхнего** уровня. Другими словами, мы обсудим *методы принятия решений*, которые и определяют функциональность робота. Для этого применяются самые различные подходы, и здесь будут рассмотрены только самые популярные из них. Большинство подходов базируется на методе *нелинейного программирования*, с помощью которого удастся реализовывать достаточно сложные функции и который в большинстве случаев не требует больших объемов памяти или трудоемких вычислений. Именно эти достоинства становятся определяющими, когда речь идет о программировании микроконтроллеров, поскольку, в отличие от персональных компьютеров или, тем более, больших ЭВМ, здесь очень жестко задаются ограничения на время реакции программы и объем аппаратных ресурсов.

Еще одно преимущество описываемого здесь подхода состоит в том, что он позволяет свести достаточно сложный процесс проектирования программного обеспечения к более простой задаче, которая решается оптимальным подбором нескольких параметров. Это не означает, что мы больше не будем программировать (будем, насколько вы сможете выдержать), но после написания программы, которое теперь уже не осуществляется столь хаотично и неформально, понадобится выполнить настройку значений нескольких параметров, пока мы не добьемся нужного поведения робота. При этом настройка параметров в идеале не должна потребовать внесения изменений в текст программы.

При создании биологического кода необходимо помнить, что поведение робота кажется осмысленным только в том случае, если оно является *детерминированным*. Это значит, что любая реакция управляющей программы на то или иное входное воздействие не должна быть случайной. Только строго запланированные и предсказуемые (то есть повторяющиеся при одних и тех же условиях) реакции на изменения внешней обстановки можно считать проявлениями искусственного интеллекта.

5.1. ОПЕРАЦИОННЫЕ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

Операционной системой реального времени (ОСРВ, Real Time Operating System - RTOS) называется программная среда, которая обеспечивает поддержку выполнения *прикладных программ* (tasks - задач), выступая, таким образом, в качестве своеобразного интерфейса между программным обеспечением и аппаратными средствами. ОСРВ предоставляет прикладным программам необходимый набор стандартных *сервисов* (**интерфейсов**, функций), позволяющий программам наиболее простым способом взаимодействовать друг с другом и с аппаратурой. Благодаря тому что часто используемые функции реализуются с помощью ОСРВ, программы становятся менее объемными и сложными, а сам процесс программирования - менее трудоемким. От обычных *операционных систем* (ОС) операционные системы реального времени отличаются тем, что обеспечивают поддержку выполнения задач, в которых большую роль играет соблюдение заданного времени реакции на то или иное внешнее событие.

Многие микроконтроллеры имеют встроенную поддержку какой-либо операционной системы. К сожалению, этого нельзя сказать о микроконтроллерах PICmicro среднего семейства. Поэтому в следующем разделе при обсуждении методов работы с ОСРВ мы будем вынуждены иметь дело больше с теоретическими концепциями, чем с конкретными примерами.

Лучшая формулировка, которую я **могу** предложить для определения ОСРВ, такова: *операционная система реального времени - это фоновая программа, которая управляет выполнением множества прикладных задач и облегчает их взаимодействие*.

Напомним, что операционная система, способная поддерживать выполнение сразу нескольких прикладных задач, называется *многозадачной*. При этом для выполнения каждой задачи обычно отводится малый промежуток (квант) процессорного времени, по истечении которого управление передается следующей задаче; данный процесс повторяется по кругу, создавая иллюзию одновременного выполнения многих задач.

Если вы знакомы с системным программным обеспечением персональных компьютеров, то, возможно, слышали что-то о процессах и потоках, а также различных методах планирования задач (в том числе и реального времени), используемых в современных операционных системах. Здесь мы не будем вдаваться во все эти тонкости, так как большинство из них оказывается совершенно не нужно, когда речь заходит о программировании микроконтроллеров. Даже различия между ОС и ОСРВ не будут нас волновать, так как ядро (kernel) тех и других устроено одинаково. Разумеется, при более пристальном рассмотрении могут быть выделены некоторые отличия, в частности касающиеся начальной загрузки системы. При запуске ОС персональных компьютеров вместе с ядром загружаются: 1) *базовая система ввода-вывода* (Basic Input/Output System - BIOS), осуществляющая поддержку ввода информации с клавиатуры и вывода ее на экран, а также работу с файлами на диске; 2) интерпретатор команд, которые вводит пользователь.

В ОСРВ вместе с ядром загружаются наиболее приоритетные задачи, выполнение которых жизненно необходимо для поддержания работоспособности всей системы.

Важная особенность ОСРВ, используемых в робототехнике, заключается в том, что они должны требовать минимально возможных аппаратных ресурсов, то есть экономно расходовать память программ и данных, а также процессорное время. Объем памяти микроконтроллеров сильно ограничен, и весьма накладно было бы хранить в ней, как это обычно бывает в ОС, код для поддержки всего набора возможных функций (в том числе и сетевых), даже если многие из них в данный момент не требуются.

Все задачи в ОСРВ (как и в ОС) имеют свой приоритет. В первую очередь процессорное время отводится самым приоритетным задачам. Планировщик использует так называемые *очереди*, в которых задачи ожидают своего выполнения. Обычно для **задач**, имеющих один и тот же уровень приоритета, используется отдельная очередь.

В каждый момент времени может выполняться только одна задача. Она будет приостановлена, когда истечет отведенный для нее квант времени, или произойдет какое-либо особое внешнее событие, или эта задача завершится сама по себе либо перейдет в режим ожидания некоторого события (например, запроса на прерывание от внешнего устройства). После этого из наиболее приоритетной очереди будет выбрана для выполнения следующая задача, и так будет повторяться снова и снова. Длительность кванта времени должна быть не слишком велика, чтобы все задачи могли получать процессорное время достаточно часто и тем самым имели возможность своевременно реагировать на все внешние события. В то же время длительность кванта не должна быть слишком маленькой, иначе большую часть времени микроконтроллер или микропроцессор будет занят переключением задач, а не их выполнением.

В каждом конкретном случае при разработке того или иного автоматического устройства в процессе его испытания может потребоваться индивидуальная настройка длительности кванта времени. Все зависит от тех функций, которые должна выполнять система.

Уровни приоритета назначаются задачам в зависимости от того, как быстро они должны уметь реагировать на запрос, поступивший от внешнего устройства. Вас может удивить тот факт, что вроде бы наиболее важный по значимости код верхнего (биологического) уровня обычно имеет самый низкий уровень приоритета, в то время как менее важным механическим и электронным интерфейсам, обслуживающим различные периферийные устройства, зачастую присваивается наивысший уровень. Все дело в том, что робот должен быстро реагировать на запросы, поступающие от внешних устройств. Если какое-либо внешнее устройство формирует запрос на обслуживание, необходимо как можно быстрее выполнить его. Если никаких запросов нет, то может выполняться наименее приоритетный биологический код, определяющий поведение робота. Для успешного выполнения биологического кода должны вовремя выполняться все обеспечивающие его работу функции, а это в первую очередь функции нижних уровней.

При распределении приоритетов важно понимать, как протекают все процессы в системе и как они взаимодействуют. Очень просто попасть в ловушку, когда какая-либо важная задача ожидает результата выполнения менее приоритетной, которая, в свою очередь, не может завершиться из-за того, что ей мешает эта более приоритетная задача. Также возможны ситуации, когда какая-либо задача с низким приоритетом (но в данный момент необходимая) вообще не может начать свое выполнение, потому что ей все время мешает более приоритетная задача.

Очень часто в системах реального времени некоторые задачи не могут начать свое выполнение, потому что ждут определенного внешнего события (обычно запроса на прерывание от какого-то известного устройства). В ожидании внешнего события такие задачи размещаются в специальной очереди. ОСРВ должна уметь при наступлении события сообщить о нем всем ожидающим его задачам и переместить их в очередь задач, готовых к выполнению.

При этом ОСРВ должна распознать источник запроса на прерывание и передать управление соответствующему обработчику, который выполнит все необходимые действия по обработке этого запроса, в том числе не забудет сбросить контроллер прерываний и оповестит внешнее устройство, что его запрос уже обрабатывается.

Следует помнить, что в больших и сложных системах во многих случаях невозможно обеспечить немедленную обработку запроса на прерывание, и может пройти не одна сотня микросекунд, пока запрос действительно будет обработан (даже если процедура его обработки имеет достаточно высокий приоритет).

Чтобы минимизировать эту задержку, необходимо весьма тщательно проектировать код обработчика прерываний, стараясь выполнить все необходимые действия как можно быстрее. Иногда хороший результат может быть достигнут только ценой усложнения аппаратных средств, берущих на себя те функции, которые программное обеспечение не может выполнить достаточно быстро.

Хороший пример такого рода - **широотно-импульсный** модулятор, который мы рассматривали в прошлых главах. Он может быть реализован программно, но более эффективна его аппаратная реализация.

Разумеется, в любой ОСРВ должен быть какой-то механизм, который позволяет через определенные промежутки времени выполнять те или иные действия, подобно тому как это было у нас при разработке интерфейсов, основанных на прерывании от таймера TMRO.

Часто требуется, чтобы задачи, выполняемые под управлением ОСРВ, взаимодействовали, обмениваясь необходимыми данными. Такое взаимодействие может быть организовано напрямую, с помощью глобальных (доступных всем задачам) переменных, но в большинстве случаев лучше использовать поддерживаемый операционной системой *механизм передачи сообщений*. Тогда не требуется принимать специальных мер по синхронизации доступа к глобальным переменным, следя за тем, чтобы значение любой из них, записанное одной задачей, не изменилось раньше времени другой.

Процесс передачи сообщений состоит из двух фаз: сначала одна задача посылает сообщение другой, а затем помещается в специальную очередь, где ожидает от второй задачи подтверждения того, что сообщение получено.

С другой стороны, вторая задача посылает первой запрос на пересылку очередного блока данных и ожидает его прихода. Это может быть сделано или до того, как первая задача пошлет свое сообщение, или после. После получения сообщения вторая задача высылает первой подтверждение, а сама переводится в очередь задач, готовых к выполнению, ожидая, когда планировщик задач выделит ей квант рабочего времени. После прихода подтверждения первая задача также переводится в эту очередь.

Для разграничения доступа задач к общим системным ресурсам используются так называемые *семафоры* - специальные флажки. Каждому ресурсу (переменной или порту ввода-вывода) соответствует свой флажок. Когда какая-либо задача хочет использовать общий ресурс, она проверяет состояние его флажка. Если он сброшен, значит, ресурс свободен. Задача занимает этот ресурс (то есть читает или изменяет значение переменной либо регистра порта ввода-вывода), но предварительно устанавливает флажок, таким образом сообщая остальным задачам, что данный ресурс занят. Все другие задачи, которые захотят работать с тем же ресурсом, запросив состояние флажка, будут помещены в очередь ожидания. По выполнении операции задача освобождает ресурс, снова сбрасывая его флажок и давая другим задачам возможность использовать его.

Например, если одна задача начинает вывод текстового сообщения на экран жидкокристаллического дисплея, то до окончания этого процесса никакая другая задача не может получить доступ к этому дисплею — в противном случае мы увидели бы на экране начало другой строки вместо продолжения предыдущей. Поэтому вывод информации на дисплей должен регулироваться с помощью семафора, и все задачи, прежде чем начать запись, должны проверять его состояние.

Если программист по окончании работы с общим ресурсом забудет освободить семафор, то заблокирует выполнение всех других задач, которым требуется тот же ресурс, и даже задача, вызвавшая ошибку при необходимости повторного обращения к этому ресурсу, не сможет получить к нему доступ.

В табл. 5.1 перечислены основные типы запросов к ОСРВ; там же указаны параметры этих запросов и формат ответа. В зависимости от используемых инструментальных средств, типа операционной системы и процессора для обращения к ОСРВ и посылки запросов могут использоваться макросы или подпрограммы.

Для идентификации задач, выполняемых в системе, используются номера или специальные метки — адресные ссылки. С их помощью операционная система узнает, кому надо послать то или иное сообщение. Номера занимают меньше места, чем адреса, поэтому последние реже используются в микроконтроллерах.

Заметим, что в табл. 5.1 не указаны обращения к процедурам обработки прерываний. Они весьма специфичны, и мнения разных разработчиков по поводу того, как должен быть реализован обработчик прерываний в ОСРВ, иногда диаметрально противоположны. Зато имеется функция задержки на заданное число миллисекунд, формируемой с помощью аппаратного таймера (в многозадачной среде реализовать задержку программно без использования таймера и системы прерываний не удастся, если только не заблокировать все остальные задачи).

Таблица 5. 1. Основные сервисы ОСРВ

Сервис	Формат вызова	Возвращаемые параметры	Комментарии
Запуск задачи	StartTask (Address, Priority [, TaskName])	Дескриптор (номер) задачи или ошибка	Запустить указанную задачу начиная с заданного адреса
Завершение задачи	End ()	—	Удалить выполнявшуюся задачу из памяти
Переход к следующей задаче	Next ()	~	Перейти к выполнению следующей задачи, не дожидаясь окончания или приостановления текущей
Посылка сообщения	Send (Task, Message)	''	Послать заданное сообщение указанной задаче и ждать подтверждения его получения
Проверка сообщений	Check ()	Номер сообщения (или ошибка)	Проверить очередь задач, ожидающих подтверждения получения посланных сообщений
Ожидание сообщения	Wait ()	Номер сообщения	Поместить задачу в очередь ожидания, где она будет находиться до тех пор, пока не получит сообщение от другой задачи
Чтение полученного сообщения	Read (Message#)	Текст сообщения (или ошибка)	Получить данные, посланные другой задачей
Подтверждение получения сообщения	Ack (Message#)	Нет (или ошибка)	Послать уведомление о получении сообщения с указанным номером
Захват ресурса	Get (Semaphore#)	—	Если ресурс свободен, то занять его; в противном случае ждать освобождения
Проверка ресурса	Poll (Semaphore#)	Нет (или ошибка, если ресурс уже занят)	Проверить состояние ресурса по его семафору
Освобождение ресурса	Free (Semaphore#)	Нет (или ошибка, если этот семафор не был занят данной задачей)	Освободить ресурс, сбросив его семафор
Задержка	Dlay (msecs)	—	Выполнить задержку на указанное количество миллисекунд

Работа операционной системы реального времени осуществляется следующим образом. После включения питания и загрузки ОСРВ обычно запускается специальная задача - назовем ее AllTask. Она выполняет двойную функцию. Во-первых, с ее помощью осуществляется запуск других программ (задач). Для выполнения этой функции используется системный вызов StartTask (см. табл. 5.1):

```
void AllTask()           // Постоянно выполняемая задача.
{
    StartTask(Task1, 1);  // Запускается задача Task1.
    while(1 == 1);        // Выполняется бесконечный цикл.
}
```

В приведенном фрагменте кода происходит запуск задачи Task1, причем ей назначается уровень приоритета 1.

Заметим, что приоритет фоновой задачи AllTask равен 0, поэтому тело бесконечного цикла выполняется, только если нет какой-либо другой задачи, которая в любом случае будет более приоритетной.

При разработке программного обеспечения для управления роботами надо учитывать следующие правила:

- существует три типа задач: *управляемые оператором* (вручную), *биологические* (код верхнего уровня по нашей классификации) и *периферийные* (обслуживающие внешние устройства). Задачи первого типа используют различные средства ввода информации — кнопки на боковой панели робота, интерфейс с компьютером, приемник команд дистанционного управления, — которые позволяют человеку вмешиваться в процесс выполнения программы. Задачи второго типа (биологический код) обычно заняты опросом датчиков, обработкой поступающих с них данных и выработкой команд для исполнительных устройств. Как правило, в ОСРВ должна выполняться только одна задача данного уровня. Каждая задача третьего типа предназначена для работы с каким-то одним (и только одним) внешним устройством. Эти задачи не способны инициировать обмен данными с внешними устройствами — они только могут обслужить запросы, сформированные биологическим кодом;
- если задача биологического уровня в настоящий момент не активна, то производится вызов функции задержки на некоторое количество миллисекунд. Пока длится задержка, могут выполняться другие задачи;
- в некоторых ОСРВ не требуется в явном виде использовать оператор цикла: дойдя до конца, задача сама продолжает свое выполнение с самого начала. В других системах все может оказаться совершенно наоборот: завершенная задача удаляется из памяти. Поэтому лучше использовать явный цикл для продолжения выполнения и явный системный вызов End () для завершения задачи.

5.2. ПРИМЕР ПРИЛОЖЕНИЯ, РАБОТАЮЩЕГО ПОД УПРАВЛЕНИЕМ ОСРВ

В предыдущем разделе при обсуждении операционных систем реального времени мы так и не показали, как они позволяют упростить разработку приложений. Здесь мы рассмотрим этот вопрос на простом примере, чтобы вы воочию убедились, как облегчается при использовании ОСРВ процесс добавления новых функций к работающему приложению.

Хотя примеры, приведенные выше, разрабатывались именно таким образом, чтобы облегчить модернизацию программ и обеспечить безболезненное добавление новых интерфейсов, наиболее простой и эффективный способ модернизации, не требующий изменения уже работающих программ, основан на использовании многозадачной ОСРВ.

Назовем наше приложение «Таракан», потому что функция робота будет заключаться в том, чтобы убежать подальше от источника света, скрывшись в самом

темном углу комнаты. Для этого потребуются четыре задачи: первая будет выполнять биологический код, вторая - осуществлять опрос датчиков столкновений (контактных или инфракрасных), третья - опрашивать состояние световых датчиков, а четвертая - управлять работой двух двигателей, каждый из которых связан со своим (левым и правым) колесом. Биологический код должен будет по состоянию датчиков принимать решение о направлении движения, изменяя его, чтобы робот огибал возникшее на пути препятствие, но в любом случае пытался оказаться подальше от источника света.

Чтобы представить связи между задачами, вы можете нарисовать их на листе бумаги, изобразив сами задачи в виде вершин графа скругленной формы, а семафоры, используемые для управления доступом к ресурсам, - в виде прямоугольных блоков.

Обратившись к табл. 5.1, вы не сможете найти в ней какой-либо функции, позволяющей одной задаче узнать состояние другой. Для этого используется стандартный механизм пересылки сообщений. Одна задача всегда может послать другой сообщение, запросив нужные данные, а та, в свою очередь, пошлет первой задаче сообщение, где будут содержаться значения необходимых переменных.

Биологический код для нашего примера выглядит следующим образом:

```
void biologic() // Биологический код для приложения "Таракан" (Roach)
{
    int left, right; // Состояние световых датчиков.
    int msgnum; // Номер сообщения.
    msg message; // Текст сообщения.
    while (1 == 1) < // Бесконечный цикл.
        if (Poll(control_sem) == free) { // Если нет активной задачи.
            message = biologic_task;
            Send("bump", message); // bump - имя задачи,
            // обслуживающей детекторы столкновений.

            msgnum = Wait();
            message = Read (msgnum);
            Ack(msgnum); // Узнали состояние детекторов.
            if (message != no_collision) { // Нет столкновений.
                message = motor_stop; // Остановить двигатели.
                Send("motor", message);
            } else < // Двигаться от источника света.
                message = biologic_task;
                Send("light", message); // Узнать состояние
                message = get_left_light_sensor; // левого
                Send("light", message); // светового датчика.
                msgnum = Wait();
                left = Read(msgnum);
                Ack(msgnum);

                message = biologic_task;
                Send("light", message); // Узнать состояние
                message = get_right_light_sensor; // правого
                Send("light", message); // светового датчика.
                msgnum = Wait();
                right = Read(msgnum);
                Ack(msgnum);
```

```

        // "Отвернуться" от света:
        if (right >= left)
            message = motor_left;
        else
            message = motor_right;
        Send("motor", message);
    }
    >
    delay(100);           // Задержка на 100 мс
                        // перед новым опросом датчиков.
}                        // Конец цикла while.
}                        // Конец биологического кода.

```

Здесь мы предполагали, что значение, получаемое от светового датчика, тем больше, чем сильнее он освещен. Чтобы удалиться от источника света, наш «таракан» включает двигатель, противоположный тому датчику, который освещен ярче. Например, если **источник** света расположен слева (выполняется ветвь «иначе» условного оператора **if**), то будет включен правый двигатель.

Для работы с командами дистанционного управления перед входом в программу мы опрашиваем состояние семафора **control_sem** - этот вопрос будет обсуждаться чуть позже.

Для опроса датчиков столкновений предназначена следующая программа:

```

void bump()                // Опрос датчиков столкновений.
{
    int left, right;        // Счетчики для левого и правого датчиков.
    int msgnum;             // Номер сообщения.
    msg message;           // Текст сообщения.
    msg Sender;            // Задача - отправитель сообщения.

    left = 0; right = 0;    // Инициализация счетчиков.
    while (1 == 1) {       // Бесконечный цикл.

        if (leftsensor == collision) // Сработал левый контактор.
            if (left < 5)
                left++; // Противоребрезговая задержка.
            else ;
        else // Нет объектов слева.
            if (left > 0)
                left--; // Противоребрезговая задержка.

        if (rightsensor == collision) // Сработал правый контактор.
            if (right < 5)
                right++; // Противоребрезговая задержка.
            else ;
        else // Нет объектов справа.
            if (right > 0)
                right--; // Противоребрезговая задержка.

        msgnum = Check(); // Было сообщение?

        if (msgnum != no_message) { // Да, было.
            Sender = Read(msgnum); // Прочитать его
            Ack(msgnum);           // и подтвердить получение.
            if (Left == 5)
                if (Right == 5)

```

```

        message = "Both";        // Оба контактора.
    else
        message = "Left";        // Левый контактор.
    else if (Right == 5)
        message = "Right"        // Правый контактор.
    else if ((Right != 0) | (Left != 0))
        message = "Indeterminate" // Неопределенность.
    else // Оба счетчика имеют нулевое значение.
        message = "no_collision"; // Нет контакта.
    Send(Sender, message);        // Послать сообщение.
}
Delay(10); // Теперь пусть выполняются и другие задачи.
} // Конец цикла while.
}

```

Здесь в бесконечном цикле происходит опрос состояния обоих датчиков (в данном случае неважно, какого типа эти датчики - они могут быть и дистанционными, и механическими; в комментариях к программе используется термин «контактор»). С помощью счетчиков реализуется необходимая задержка, устраняющая возможныйдребезг контактов, если датчики являются механическими. Получив сообщение от другой задачи, программа посылает в ответ строку, описывающую состояние датчиков; при этом используется системный вызов Send. Обратите внимание на небольшую задержку в конце программы: она нужна, чтобы планировщик мог передать управление какой-либо другой задаче.

Если контакторы к текущему времени не находились в определенном состоянии хотя бы 50 мс, то их состояние считается неопределенным (посылается строка "Indeterminate").

Предварительный запрос от другой задачи нужен для того, чтобы программа знала, кому посылать сообщение о состоянии датчиков. При описании основных системных вызовов мы не конкретизировали, каким способом задача-получатель сообщения может узнать идентификатор задачи-отправителя. В большинстве ОСРВ сообщение на самом деле - не простая строка, а структура, состоящая из нескольких полей, в одном из которых хранится номер или имя отправителя.

Следующая программа предназначена для управления двигателями. Она включает нужный двигатель на 100 мс и ожидает прихода сообщения, которое будет содержать указания для дальнейших действий. Перед включением двигателя проверяется состояние семафора `motor_running` и, если он свободен (сброшен), устанавливаем его, чтобы другая задача, ответственная за опрос световых датчиков, могла узнать, что в данный момент ведется работа с двигателями (напомним, что измерение сопротивления фоторезисторов в прошлой главе мы договорились производить только при выключенных двигателях, когда уровень помех минимальный). Вот текст программы:

```

void motor() // Работа с двигателями.
{
    int 'i;
    int msgnum; // Номер сообщения.
    msg message; // Текст сообщения.

    i = 0; // Сброс счетчика.

```

```

while (1 == 1) {
    if (--i == 0) { // Если прошло 100 мс, остановить двигатель.
        motors = stop;
        i = 1;
        Free(motor_sem) ;
    }

    msgnum = Check(); // Есть сообщение?
    if (msgnum != no_message) { // Да.
        message = Read(msgnum) ; // Прочитать его
        Ack(msgnum); // и послать подтверждение.
        motors = message;
        if (message == stop) {
            i = 1; // Было сообщение об остановке двигателей.
            Free(motor_sem); // Освободить семафор.
        } else { // Двигатель работает.
            i = 10;
            Get(motor_sem); // Занять семафор.
        }
    }
}

Delay(10); // Дать возможность выполняться другим задачам.
} // Конец цикла while.
}

```

Четвертая задача предназначена для работы со световыми датчиками. Она ожидает запроса от другой задачи, а получив его, отправляет в ответ сообщение о состоянии датчиков. Текст программы:

```

void light() // Опрос световых датчиков.
{
    int msgnum; // Номер сообщения.
    msg message; // Текст сообщения.
    msg Sender; // Идентификатор задачи - отправителя сообщения.
    while (1 == 1) { // Бесконечный цикл.
        msgnum = Wait() // Ожидаем сообщение.

        Sender = Read(msgnum); // Запомнили идентификатор отправителя.
        Ack(msgnum); // Подтвердили получение сообщения.
        msgnum = Wait() // Ждем запрос.

        message = Read(msgnum); // Прочитали текст запроса.
        Ack(msgnum); // Подтвердили получение запроса.
        if (Poll(motor_sem) != Free) {

            // Выключить двигатели на время измерения:
            message = motor_stop;
            Send("motor", message);
        }
        if (message == get_left_light_sensor)
            message = LeftLightLevel;
        else
            message = RightLightLevel;
        Send(Sender, message) ;
    }
}

```



```

}          // Конец цикла while.
}

```

Здесь мы не вызываем в явном виде функцию задержки; ее роль играет функция ожидания прихода сообщения от другой задачи.

Для того чтобы обеспечить одновременное выполнение всех четырех описанных задач, создадим еще одну задачу под названием **Task1**:

```

void Task1()          // Загрузка задач.
{

    Start(biologic, 2, "biologic");    // Запуск биологического кода.
    Start(motor, 3, "motor");          // Задача для работы с двигателями.
    Start(bump, 3, "bump");            // Задача для работы с контакторами.
    Start(light, 3, "light");          // Задача для работы
                                        // со световыми датчиками.

    End();          // Завершение задач и освобождение ресурсов.
}

```

Однако всего перечисленного еще недостаточно. Наше приложение работает правильно, но пока только один раз. «Убежав» от источника света, робот остается в темном месте комнаты до тех пор, пока не изменяется положение источника или пока вы сами не переносите «таракана» на новое, более освещенное место. Будучи достаточно ленивым, я решил добавить к приложению еще одну задачу для приема и декодирования команд, подаваемых с помощью инфракрасного пульта дистанционного управления.

Мы будем использовать шесть кнопок: две - для старта и останова, еще две - для ускорения и замедления движения, и последние две - для поворота налево и направо. Ниже приведен текст программы:

```

void control()          // Дистанционное управление.
{
    Get(control_sem);    // Занять семафор (ждать прихода команды)

    msg message;          // Текст сообщения.
    while (1 == 1) {      // Бесконечный цикл.
        if (RemoteControl == ButtonPress) {    // Есть команда.
            switch(Button) {    // Какая кнопка нажата?
                case Forward:    // Команда "Вперед"
                                    // (ускорить движение).
                    if (Poll(control_sem) == Free)
                        Get(control_sem);
                    Send(motor, Forward);
                    break;
                case Reverse:    // Команда "Назад"
                                    // (замедлить движение).
                    if (Poll(control_sem) == Free)
                        Get(control_sem);
                    Send(motor, Reverse);
                    break;
                case TurnLeft:    // Команда "Налево".
                    if (Poll(control_sem) == Free)
                        Get(control_sem);
                    Send (motor, TurnLeft);

```

```

        break;
    case TurnRight:      // Команда "Направо".
        if (Poll(control_sem) == Free)
            Get(control_sem);
        Send(motor, TurnRight);
        break;
    Case Stop:           // Команда "Стоп".
        if (Poll(control_sem) == Free)
            Get(control_sem);
        Break;
    Case Start:          // Команда "Старт".
        Free(control_sem);
        break;
    } // Конец оператора witch.
} // Конец оператора if.

Delay(10);              // Дать возможность выполняться другим задачам.

}                        // Конец цикла while.
}

```

Ниже приведена программа, которая обеспечивает запуск всех пяти задач:

```

void Task1()            // Загрузчик.
{
    Start(control, 1, "control");
    Start(biologic, 2, "biologic");
    Start(motor, 3, "motor");
    Start(bump, 3, "bump");
    Start(light, 3, "light");
    End();
}

```

Как вы видите, при добавлении новой задачи не пришлось изменять текст уже написанных программ: только добавилась новая строка в программу-загрузчик. Именно в этом и заключается важное преимущество операционных систем реального времени.

Еще одно преимущество ОСРВ состоит в том, что при их использовании заметно уменьшается размер памяти, требуемый для хранения данных.

На разработку всех пяти вышеописанных программ я затратил несколько часов. Если бы программа разрабатывалась как монолитное приложение теми методами, которые были рассмотрены в предыдущей главе, то есть без поддержки ОСРВ, мне пришлось бы потратить несколько недель. Даже если не учитывать всех трудностей, которые при этом пришлось бы преодолеть, выигрыш при использовании ОСРВ очевиден.

5.3. КОНЕЧНЫЕ АВТОМАТЫ

Для проектирования кода верхнего уровня также можно использовать модель конечного автомата (КА), которую мы уже упоминали в предыдущей главе. Этот подход позволяет отказаться от вложенных условных операторов и запутанных

проверок состояния различных флагов, которые в противном случае потребовалось бы использовать в программе для принятия тех или иных решений.

Первоначально конечные автоматы возникли как специальные цифровые устройства (рис. 5.1), которые могли выполнять заданную последовательность действий в определенном порядке.

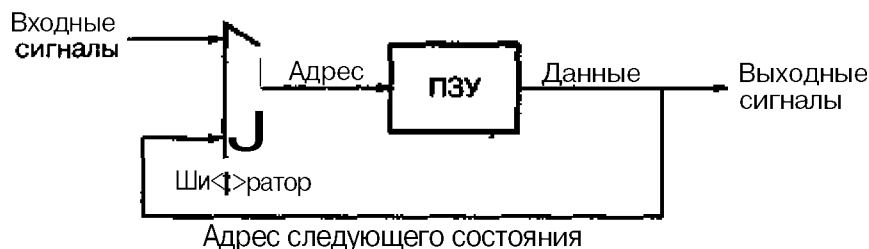


Рис. 5.1. Аппаратная реализация конечного автомата

По каждому фронту тактового генератора (на рис. 5.1 тактовый вход не показан) конечный автомат переходит в новое состояние - оно вычисляется исходя из текущего состояния автомата и значения входных сигналов. Состояние КА задается значением адреса Addr на входе постоянного запоминающего устройства (ПЗУ). Хранящаяся по этому адресу информация определяет, какими должны быть выходные сигналы КА и какой адрес будет на входе ПЗУ в следующем такте. Заметим, что заранее задана только часть адреса (некоторое число его разрядов); остальные биты адреса зависят от входных сигналов КА.

Для программной реализации КА хорошо подходит оператор выбора `switch`, который мы уже использовали при разработке конечных автоматов для работы с периферийными устройствами.

Приведем пример программной реализации КА для управления роботом. Допустим, мы хотим, чтобы наш робот двигался случайным образом:

```
main()                                // Случайные блуждания.
{
    int State = 1;                     // Начальное состояние.
    while(1 == 1)                      // Бесконечный цикл.
        switch (State) {               // Конечный автомат: .
            case 1:                     // Двигаться вперед в течение 2 с.
                Move(Forward);
                Delay(2000);             // Задержка на 2 с (используются
                                         // прерывания от таймера).
                if (Collision == "No") // Нет столкновения.
                    State = 5;          // Номер следующего состояния.
                else // Есть столкновение, надо объехать препятствие.
                    State = 10;         // Задний ход.
                break;
            case 5:                     // Поворот налево (0,75 с).
                Move(TurnLeft);
                Delay(750);
                if (Collision == "No") // Нет столкновения.
                    State = 1;          // Номер следующего состояния.
                else // Есть столкновение, надо объехать препятствие.
```

```

        State = 10;    // Номер следующего состояния.
        break;
    case 10:           // Было столкновение, дать задний ход (1 с).
        Move(Reverse) ;
        Delay(1000) ;
        State = 15;    // Номер следующего состояния.
        break;
    case 15:           // Поворот направо (0,5 с).
        Move(TurnRight);
        Delay(500);
        State = 1;     // Номер следующего состояния.
        break;
} // Конец оператора switch.
} // Конец цикла while.
} // Конец примера.

```

Здесь перед переходом в очередное состояние выполняется задержка на определенное время, то есть функция `Delay` выполняет роль тактовых импульсов, вызывающих срабатывание аппаратного конечного автомата.

При этом возникает новая проблема: во время действия задержки любые изменения входных сигналов будут проигнорированы, так как новое состояние КА уже определено (переменная `State` уже получила нужное значение).

Конечно, желательно, чтобы световые датчики и детекторы столкновений опрашивались постоянно. В принципе это не так уж сложно осуществить, но тогда надо добиться, чтобы подпрограмма задержки работала вне цикла **конечного** автомата, причем длительность задержки должна изменяться в зависимости от текущего состояния.

В следующем примере каждое состояние конечного автомата разбито на два. Первое состояние определяет характер движения робота и число циклов длительностью 50 мс каждый, в течение которых будет длиться задержка, а второе состояние соответствует периоду ожидания (в течение которого с каждым срабатыванием таймера уменьшается значение счетчика задержки), но в это же время происходит непрерывный опрос датчиков столкновений. Если какой-то датчик сработал, то выполняются необходимые действия.

```

main()                                // Случайные блуждания, вариант 2.
{
    int State = 1;                     // Начальное состояние.
    int DelayCount = 0;                // Счетчик задержки.
    while (1 == 1) {                  // Бесконечный цикл.
        switch(State) {
            case 1:                    // Движение вперед (2 с)
                Move(Forward);
                DelayCount = 40;        // 40 раз выполнить
                                         // задержку на 50 мс.
                State = 2;              // Выполнение задержки
                                         // (и опрос датчиков).
                break;
            case 2:                    // Движение вперед (2 с).
                if (Collision == "No") // Нет столкновений.

```

```

        if (--DlayCount)
            State = 2; // Если не 0, то оставаться
                        // в текущем состоянии.
        else
            State = 5;
    else // Было столкновение.
        State = 10;
    break;
case 5: // Поворот налево (0,75 с).
    Move(TurnLeft);
    DlayCount = 15; // Соответствует задержке 750 мс.
    State = 6; // Выполнение задержки.
    break;
case 6: // Поворот налево (0,75 с).
    if (Collision == "No")
        if (--DiayCount)
            State = 6;
        else
            State = 1;
    else
        State = 10;
    break;
case 10: // Было столкновение (ехать назад в течение 1,с).
    Move(Reverse);
    Dlay(1000) ;
    State = 15;
    break;
case 15: // Поворот направо (0,5 с).
    Move(TurnRight);
    Dlay(500);
    State = 1;
    break;
} // Конец оператора switch.

Dlay(50); // Общая задержка 50 мс.

} // Конец цикла while.

} // Конец примера.

```

Здесь состояния с номерами 10 и 15 не имеют задержек, потому что определяют действия робота при срабатывании датчиков столкновений. Предполагается, что, отъезжая чуть назад, робот не столкнется с каким-либо другим предметом. Вообще-то это предположение не совсем корректно для нашего беспорядочно двигающегося робота.

Лучше было бы разместить дополнительные датчики в задней части корпуса робота и опрашивать их состояние, пока он движется назад. При необходимости вы сможете самостоятельно внести соответствующие изменения в программу.

Достоинства конечно-автоматного подхода к процессу разработки управляющей программы - его простота и легкость, с которой могут быть добавлены новые состояния. Так как программный код, соответствующий каждому отдельному состоянию, достаточно прост, он не отнимает много процессорного времени, давая возможность выполняться функциям нижних уровней.

Однако при усложнении логики работы программы код КА также становится достаточно сложным для понимания и отладки. Тем не менее конечно-автоматный подход в большинстве случаев достаточно перспективен при проектировании программного обеспечения для управления роботами.

5.4. ДИСТАНЦИОННОЕ УПРАВЛЕНИЕ РОБОТОМ, СОВЕРШАЮЩИМ СЛУЧАЙНЫЕ БЛУЖДЕНИЯ

В этом разделе мы рассмотрим несложный пример использования программного конечного автомата, который позволяет роботу, в свободном состоянии совершающему случайные блуждания, принимать и выполнять команды дистанционного управления.

Допустим, робот должен двигаться вперед в течение двух секунд, на следующей секунде совершать поворот направо (затратив на это 0,75 с), после чего все повторяется заново. Натолкнувшись на какой-либо предмет, робот сначала отъезжает назад (затратив на это полсекунды), а затем в течение следующих 0,5 с поворачивает в сторону, противоположную той, с которой зафиксировано срабатывание контактора. После объезда препятствия робот снова возвращается к своему обычному циклу «вперед-направо».

При использовании традиционного метода программирования (с помощью условных операторов) биологический код будет выглядеть следующим образом:

```
while (1 == 1) {

    CollisionFlag = 0;           // Сначала столкновений нет.

    LeftMotor(LeftForward);     // Левый двигатель включен.
    RightMotor(RightForward);    // Правый двигатель включен.
    Dlay = RTC + 2000;          // В течение 2 с.

    while (Dlay != RTC)
        if (GetLeftWhisker())    // Сработал левый контактор?
            CollisionFlag = 1;
        else if (GetRightWhisker()) // Сработал правый контактор?
            CollisionFlag = 2;

    if (!CollisionFlag) {

        // Код, выполняемый, когда столкновений нет.
        LeftMotor(LeftForward); // Поворот направо.
        RightMotor(RightReverse); // Реверс правого двигателя.
        Dlay = RTC + 750;       // Задержка 0,75 с.
        while (Dlay != RTC)
            if (GetLeftWhisker0)
                // Сработал левый контактор?
                CollisionFlag = 1;
            else if (GetRightWhisker0)
                // Сработал правый контактор?
                CollisionFlag = 2;
    } // Конец условного оператора if.

    if (CollisionFlag == 1) { // Столкновение зафиксировано слева.
```

```

    LeftMotor(LeftReverse);    // Обезд.
    RightMotor(RightReverse);  // Реверс правого двигателя.
    Dlay = RTC + 500;          // В течение 0,5 с.
    while (Dlay != RTC);       // Задержка.
    LeftMotor(LeftForward);    // Поворачиваем направо.
    RightMotor(RightReverse);  // Реверс правого двигателя.
    Dlay = RTC + 500;          // Задержка 0,5 с.
    while (Dlay != RTC);
} // Конец условного оператора if.

if (CollisionFlag == 2){ // Столкновение зафиксировано справа.
    LeftMotor(LeftReverse);    // Обезд.
    RightMotor(RightReverse);  // Реверс правого двигателя.
    Dlay = RTC + 500;          // 0,5 с.
    while (Dlay != RTC);
    LeftMotor(LeftReverse);    // Реверс левого двигателя.
    RightMotor(RightForward);  // Поворачиваем налево.
    Dlay = RTC + 500;          // 0,5 с.
    while (Dlay != RTC);
} // Конец условного оператора if.
} // Конец цикла while.

```

Эта программа вполне работоспособна, но выглядит довольно неуклюже. Вряд ли вы смогли бы быстро разобраться, что делает робот, если бы не дополнительные пояснения. Кроме того, весьма проблематично добавить к данному коду новые функции. Поэтому целесообразно выбрать какой-либо другой способ программирования.

Все намного проще, если мы используем конечно-автоматный подход:

```

while (1 == 1) {                // Бескойечный цикл.
    switch (State) {
        case 1:                  // Двигаться вперед в течение 2 с.
            State++;              // Начать с состояния 2.
            StateDlay = RTC + 2000; // Задержка 2 с.
            LeftMotor(LeftForward); // Левый двигатель работает.
            RightMotor(RightForward); // Правый двигатель работает.
            break;

        case 2:                  // Ждать 2 с.
            if (GetLeftWhisker()) // Сработал левый контактор.
                State = 10;
            else
                if (GetRightWhisker()) // Сработал правый контактор.
                    State = 20;
            else if (StateDlay == RTC)
                State++;          // Нет столкновений.
            break;

        case 3:                  // Поворот направо (0,75 с).
            State++;
            StateDlay = RTC + 750;
            LeftMotor(LeftForward); // Левый двигатель работает.
            RightMotor(RightReverse); // Реверс правого двигателя.
            break;
    }
}

```

```
case 4:                // Ждать 0,75 с.
    if (GetLeftWhisker())
        State = 10;
    else if (GetRightWhisker())
        State = 20;
    else if (StateDlay == RTC)
        State = 1;
    break;

case 10:               // Препятствие слева.
    State++;
    StateDlay = RTC + 500;
    LeftMotor(LeftReverse);
    RightMotor(RightReverse);
    break;

case 11:              // Ждать 0,5 с.
    if (StateDlay == RTC)
        State++;
    break;

case 12:              // Объехать препятствие (0,5 с).
    State++;
    StateDlay = RTC + 500;
    LeftMotor(LeftForward);
    RightMotor(RightReverse);
    break;

case 13:              // Ждать 0,5 с.
    if (StateDlay == RTC)
        State = 1;
    break;

case 20:              // Препятствие справа.
    State++;
    StateDlay = RTC + 500;
    LeftMotor(LeftReverse);
    RightMotor(RightReverse);
    break;

case 21:              // Ждать 0,5 с.
    if (StateDlay == RTC)
        State++;
    break;

case 22:              // Объехать препятствие (0,5 с).
    State++;
    StateDlay = RTC + 500;
    LeftMotor(LeftReverse);
    RightMotor(RightForward);
    break;

case 23:              // Ждать 0,5 с.
    if (StateDlay == RTC)
        State = 1;
    break;

} // Конец оператора switch.

} // Конец цикла while.
```


Не так уж сложно добавить сюда какие-либо дополнительные функции. При этом просто увеличится число состояний. Заметим, что размер программы растет по линейному закону, в отличие от обычного метода программирования, когда сложность и длина программы при наращивании числа функций возрастают в геометрической прогрессии.

Здесь приведен только пример биологического кода. В электронном приложении к этой книге в файле `Code\Statemc\wallhug.c` приводится полный исходный текст программы для дистанционно управляемой модели мыши фирмы *Tamiya*, а в файле `Code\Statemc\servo.c` - аналогичный текст программы для управления роботом с двумя серводвигателями и световыми датчиками.

Если вы сравните эти файлы, то с удивлением обнаружите, что биологический код в обоих случаях совершенно одинаков. Единственное отличие касается функций `LeftMotor` и `RightMotor`. Скорость движения модели мыши может изменяться с помощью команд дистанционного управления (для этого приходится использовать широтно-модулированный сигнал), а у второго робота серводвигатели вращаются с постоянной скоростью.

Такой результат достигнут не только благодаря достоинствам конечно-автоматного подхода. Следует не забывать и о нашей концепции «трех уровней», согласно которой необходимо отделять реализацию механических и электронных интерфейсов от высокоуровневого биологического кода. Как вы видите, сочетание этих двух подходов дает прекрасные результаты.

5.5. ПОВЕДЕНЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Если задуматься, можно прийти к выводу, что поведение робота - это просто набор его реакций на те или иные внешние воздействия. Например, робот, имитирующий поведение маленького ребенка, может действовать следующим образом:

```
GetToyBehavior()          // Вижу игрушку, хочу играть...
{
    printf("Мама, папа! Можно мне вон ту игрушку?!\n");
    Response = Input.Data;  // Узнать реакцию родителей.

    while. ((Response != "Да") && (Response != "Хорошо, мы купим ее")) {
        // Реагировать на отказ родителей купить игрушку.
        Position(LieDown);    // Лечь на пол, стучать ногами и руками
        Crying(Start);        // и кричать.
        printf("Ну, почему-у-у\n");

        if (Input.Pending) {
            // Если нет ответа, то ждать:
            Response = Input.Data;          // Узнать реакцию родителей.

            if (Response != "Перестань немедленно, а не то выключу!") {
                // Перестать капризничать.
                printf("Я больше не буду!")
                Response = "Да";
            }
        }
    }
}
```

```

        } // Конец оператора if.

    } // Конец оператора if.

} // Конец цикла while.

Position(Stand);           // Подняться с пола.
Crying(Stop);              // Перестать кричать.

} // Конец.

```

Этот шуточный пример довольно точно иллюстрирует поведение робота, который опрашивает состояние своих входных датчиков, чтобы узнать об окружающей его ситуации, а затем согласно заложенному в него алгоритму выбирает, как реагировать на то или иное изменение обстановки. Даже на этом простом примере нам удалось проиллюстрировать множественность целевых установок, характерную для реальных программ: цель ребенка — покупка понравившейся ему игрушки, но если дело заходит слишком далеко (к сожалению или к счастью, настоящего ребенка нельзя «выключить», но можно придумать более педагогичный выход из создавшегося положения), то перед угрозой наказания ребенок перестает капризничать.

Отличие большинства реальных случаев от нашего примера заключается в том, что робот в каждый момент времени не обязательно выполняет только одну задачу. Возможно, вы подумали, что очень сложно реализовать выполнение сразу многих функций в одной программе, но на самом деле это вполне возможно и часто не требует какого-то особо искусного программирования.

Единственное, что действительно представляет трудность, — выбор подходящего для данной ситуации механизма арбитража, который комбинировал бы решения, принимаемые по нескольким алгоритмам, чтобы выработать то единственное, что должно подаваться на двигатели **робота**.

Рассмотрим характерный для многих практических случаев пример, когда помимо основного кода верхнего уровня, принимающего решения и контролирующего поведение робота, имеется некоторая фоновая задача, которая при выполнении определенных условий включается в работу и перехватывает управление роботом.

Допустим, мы хотим, чтобы, увидев человека, робот тут же развернулся в его сторону и начал движение либо к нему, либо от него — в зависимости от того расстояния, на котором находится человек.

Для нашего проекта, который на этот раз так и останется только на бумаге, выберем робота с двумя сервоприводами (по одному для каждого колеса) и ультразвуковым дальномером. Допустим, для управления положением колес робота в выходной порт записываются целые 8-разрядные числа: **-128** соответствует команде «полный реверс», **0** — «стоп», **+127** — «полный вперед». Пусть имеются две функции: `distance()` — для определения расстояния до человека и `vector()` — для определения угла его расположения относительно робота. Обе функции возвращают целые числа со знаком (типа `int`). Данные о режиме работы обоих двигателей будем хранить в структуре

```

struct motor {
    int left;           // Левый двигатель.
    int right;          // Правый двигатель.
}

```

Для простоты будем полагать, что робот находится на ровной площадке бесконечных размеров, так что не требуется отвлекаться на опрос датчиков столкновений. Единственное, чем будет занят робот, - движение по своему маршруту (согласно заложенному в него алгоритму) и измерение расстояния до человека. На практике, конечно, придется побеспокоиться о том, чтобы различать нескольких людей, да и вообще как-то отличать человека от обычных предметов. Хотя здесь мы не обсуждаем все эти задачи, но при используемом подходе добавление новых функций к биологическому коду не составляет никакой проблемы.

Сначала давайте выясним, как сделать, чтобы робот поворачивался в сторону человека:

```

motor directionBehavior(motor Data)
{
    // Управление двигателями (поворот в сторону человека).

    int dir; // Направление.

    Data.left = Data.right = 0;
    dir = direction(); // Узнать направление.
    if (dir > 0) // Надо повернуться налево.
        if (dir < tuneAngle) { // Медленный поворот.
            Data.left = -slowTurn;
            Data.right = slowTurn;
        } else if (dir < turnAngle) {
            Data.left = -mediumTurn;
            Data.right = mediumTurn;
        } else { // Быстрый поворот.
            Data.left = -fastTurn;
            Data.right = fastTurn;
        }

    else if (dir < 0) // Надо повернуться направо.
        if (dir > -tuneAngle) { // Медленный поворот.
            Data.left = slowTurn;
            Data.right = -slowTurn;
        } else if (dir > -turnAngle) {
            Data.left = mediumTurn;
            Data.right = -mediumTurn;
        } else { // Быстрый поворот.
            Data.left = fastTurn;
            Data.right = -fastTurn;
        }

    return Data; // Вернуть два числа - команду двигателям.
} // Конец.

```

Здесь предполагается, что все углы и скорости объявлены как глобальные переменные, то есть в главной программе. Если угол поворота, который необходимо

выполнить роботу, слишком велик, то поворот происходит быстро. После того как угол достигнет заданного предела, скорость замедляется. Когда робот развернулся в сторону человека, оба двигателя выключаются.

Если расстояние до человека больше заданного порога, то робот должен двигаться в его сторону:

```
motor closeBehavior(motor Data)
{
    // Управление двигателями (движение в сторону человека).

    int dist;                // Расстояние до человека.

    dist = distance();       // Узнать расстояние до человека.
    Data.left = Data.right = 0;
    if (dist > closeClosest)
        if (dist > closeFurthest)
            Data.left = Data.right = forwardFull;
        else
            Data.left = Data.right =
                ( forwardFull * (dist - closeClosest) ) /
                (closeFurthest - closeClosest);

    return Data;
} // Конец.
```

Чем дальше человек, тем быстрее робот стремится к нему подъехать. Когда расстояние сокращается до некоторого предела, команды перестают вырабатываться.

Обратите внимание, что для вычисления скорости мы сначала выполняем операцию умножения, а только потом - деления. Дело в том, что при делении целочисленных величин, если делитель больше делимого, результат будет нулевым. Поэтому сначала полная скорость умножается на разность текущего и минимального расстояний, и только затем результат делится на весь диапазон расстояний (от максимального до минимального). В результате скорость сближения убывает по линейному закону. Напомним, что использование вещественной арифметики существенно увеличило бы размер программы, так как все операции над числами в формате с плавающей точкой в микроконтроллерах PICmicro реализованы только программно.

Наконец, если расстояние до человека слишком мало, то робот должен двигаться назад:

```
motor awayBehavior(motor Data)
{
    // Управление двигателями (пятимся назад).

    int dist;

    dist = distance0;       // Узнать расстояние до человека.
    Data.left = Data.right = 0;
    if (dist < awayFurthest)
        if (dist < awayClosest)
            Data.left = Data.right = reverseFull; // Полный назад.
        else
            Data.left = Data.right =
                (reverseFull * (awayFurthest - dist)) /
                (awayFurthest - awayClosest);

    return Data;
} // Конец.
```

Для выполнения всех трех задач составим главную программу:

```
void main(void)                // Фоновое приложение.
{
    motor directionData;        // Команда двигателям.
    motor closeData;
    motor awayData;
    motor CombinedData;         // Комбинация всех трех команд.

    while (1 == 1) {           // Бесконечный цикл.
        pause(RunDelay);       // Сначала робот подчиняется
                                // заложенному в него алгоритму.

        directionData = directionBehavior(directionData);
        closeData = closeBehavior(closeData);
        awayData = awayBehavior(awayData);

        // Сформировали три команды, теперь комбинируем из них одну:
        CombinedData.left = directionData.left +
                               closeData.left + awayData.left;

        CombinedData.right = directionData.right +
                               closeData.right + awayData.right;

        WheelCommand(CombinedData); // Послать команду для выполнения.

        pause(LoopDelay);       // Дать роботу время на выполнение
                                // команды и новый цикл измерений.
    }                            // Конец оператора while.
}                                // Конец программы.
```

В табл. 5.2 перечислены все используемые здесь параметры и приведены их значения по умолчанию.

Таблица 5.2. Параметры фоновой задачи управления роботом

Параметр	Значение по умолчанию	Описание
<i>tuneAngle</i>	5°	Максимальный угол, при котором еще происходит медленный поворот
<i>turnAngle</i>	20°	Угол для средней скорости поворота
<i>slowTurn</i>	5	Медленная скорость поворота
<i>mediumTurn</i>	/5	Стандартная скорость поворота
<i>fastTurn</i>	40	Скорость быстрого поворота
<i>closeClosest</i>	36 дюймов	Минимальное расстояние до человека
<i>closeFurthest</i>	48 дюймов	Предельное расстояние, на котором скорость приближения к человеку должна быть максимальной
<i>forwardFull</i>	50	Максимальная скорость приближения к человеку
<i>awayClosest</i>	36 дюймов	Минимальное расстояние до человека, на котором робот должен начать удаляться от него с максимальной скоростью
<i>awayFurthest</i>	48 дюймов	Максимальное расстояние до человека, на котором робот должен начать удаляться от него с минимальной скоростью
<i>reverseFull</i>	-50	Максимальная скорость при движении от человека

Значения этих параметров можно несколько изменить в ту или иную сторону, чтобы добиться наиболее естественного поведения робота.

Разберем простейшую ситуацию. Допустим, робот находится от человека на расстоянии 47 дюймов (119 см) и ориентирован так, что человек оказывается на 15° левее продольной оси робота. Наши три программы сформируют в этом случае три команды, представленные в табл. 5.3.

Таблица 5.3. Команды, сформированные программами

	<i>directionBehavior</i>	<i>closeBehavior</i>	<i>awayBehavior</i>	<i>Combined</i>
<i>Left</i>	<u>-15</u>	<u>45</u>	<u>-4</u>	<u>26</u>
<i>Right</i>	<u>15</u>	<u>45</u>	<u>-4</u>	<u>56</u>

Так как левый двигатель (соответствующий параметр - *Left*) будет вращаться медленнее правого (соответствующий параметр - *Right*), то робот будет поворачиваться налево, в сторону человека. При этом он будет приближаться к нему, пока не окажется слишком близко, после чего либо остановится, либо начнет отъезжать назад. Поведение робота было бы аналогичным, если бы он управлялся с помощью ПИД-регулятора или fuzzy-контроллера¹. Но приведенная здесь программа заметно проще той, что пришлось написать для реализации ПИД-регулятора.

Еще раз обращаю ваше внимание на тот факт, что все параметры в программе целочисленные. Везде, где это возможно, я старался избегать использования арифметики с плавающей точкой.

5.6. НЕЙРОННЫЕ СЕТИ И ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

Рассказывая о роботах, было бы невозможно обойти молчанием такое понятие, как *искусственный интеллект*. Многие полагают, что роботы (и компьютеры), наделенные искусственным интеллектом, должны быть столь же разумны, как и человек. В предыдущей главе этой книги мы рассматривали множество различных интерфейсов, позволяющих роботу воспринимать те или иные условия и проявлять Свою реакцию на их изменение, а в этой главе обсудили, как можно программно реализовать алгоритмы, заставляющие робота демонстрировать окружающим нечто похожее на разумное поведение. Но это были лишь попытки имитировать настоящий интеллект; наши конструкции не только не являлись разумными формами жизни, но даже не могли претендовать на «звание» искусственного интеллекта.

Очень приблизительное определение искусственного интеллекта выглядит следующим образом: *это умение машины выполнять поставленные задачи, пытаясь преодолеть противодействие окружающей обстановки и обучаясь, то есть*

¹ Fuzzy-контроллеры (англ. fuzzy - нечеткий) отличаются от обычных тем, что оперируют близкими человеку качественными понятиями «большой», «средний», «малый» и т.п. - Прим. перев.

извлекая из собственного опыта знания, которые затем можно будет использовать для того, чтобы **выполнять** свои задачи еще более эффективно.

В этом определении не упоминаются обработка визуальной информации, умение вести разговор или разрушить укрепления противника и другие аналогичные способности, которые, может быть, ассоциируются у многих с термином «искусственный интеллект». Легко впасть в заблуждение, если трактовать это понятие слишком широко.

Но даже реализация интеллекта в представленном здесь узком смысле - не такая уж простая задача. Дело осложняется тем, что мы не можем точно определить, что же такое «знание» или «обучение».

Здесь уместно вспомнить известный эксперимент с червяком в лабиринте. Червяка помещали в лабиринт, имеющий форму буквы «Г». Двигаясь от основания лабиринта вперед, червяк должен был выбрать, в какую сторону повернуть: налево или направо. В одном коридоре его ждал удар электрическим током, а в противоположном - корм. После нескольких неудачных попыток червяк в конце концов запомнил, куда следует идти, и всегда выбирал коридор с едой.

Этот эксперимент получил широкую известность благодаря не столько тому, что была доказана способность червей к получению простейших навыков, сколько другому замечательному факту. После того как достаточное количество особей было обучено поворачивать в определенном (одном и том же для всех) направлении, они были скормлены своим сородичам, которые еще не прошли такого курса обучения. Спустя некоторое время с насытившимися «новичками» провели аналогичный эксперимент. Оказалось, что при первом же прохождении лабиринта вместо ожидаемых 50% «попаданий» они выбирали правильное направление в 56% случаев¹.

Оставляя в стороне вопрос об обучении за обеденным столом, обсудим, каким способом можно заставить программу вести себя так же, как и не слишком интеллектуальный червяк. Иными словами, после нескольких тренировочных «забегов» в лабиринте робот, управляемый нашей программой, наконец должен научиться выбирать правильное направление. Если оперировать теми же понятиями, что и в опыте с червями, мы должны как-то запрограммировать боль от удара электрическим током и удовольствие, получаемое при приеме пищи (а возможно, и чувство радости от того, что электроды оказались в другом коридоре). Ведь даже заставить программу сделать случайный выбор в самом первом опыте - и то задача не из тривиальных.

¹ Эксперименты над плоскими червями (**планариями**) в лабиринте впервые были проведены американским исследователем **Йерксом** в 1912 году. Они показали, что черви способны обучаться и переобучаться, но полученный навык держался недолго (10-15 ч). Затем подобные эксперименты в различных модификациях (в том числе и опыты с кормлением необученных особей обученными) были повторены многими учеными. Справедливости ради следует заметить, что многие исследователи оспаривают существование эффекта передачи опыта при кормлении, так как при повторении экспериментов не получили аналогичного результата. Те же, кому удалось зафиксировать передачу навыков столь безжалостным образом, расходятся друг с другом в объяснении данного феномена. Диапазон предположений довольно широк - от передачи информации на уровне РНК до телепатии. - *Прим. перев.*

Когда речь идет о тренировке животных (или человека), все эти вопросы интуитивно понятны и близки нам, но как перевести их на язык программирования? Как заставить робота или компьютер чувствовать голод, жажду, боль, удовольствие?

При рождении мы получаем некоторый стандартный набор рефлексов, заложенных на генетическом уровне, и с их помощью постепенно обучаемся жить в мире, руководствуясь соответствующими концепциями и обобщая свой опыт каким-то неосознаваемым способом.

В детстве я однажды довольно сильно обжег руку утюгом. До сих пор, когда в моем сознании совмещаются образы боли, одежды, утюга и руки, я всегда вспоминаю о том случае, и мое тело автоматически делает шаг назад, а рука отдергивается, дабы уберечься от ожога. После этого я много раз обжигался в разных местах, но ни один из этих инцидентов (хотя некоторые из них тоже были весьма болезненными) не оставил в моей памяти такого заметного следа, и я не пытаюсь инстинктивно защитить другую руку или ногу. Уверен, вы тоже сможете припомнить какой-нибудь случай из своей жизни, когда некоторая комбинация внешних воздействий глубоко врезалась в вашу память и с тех пор при возникновении похожих ситуаций вы инстинктивно действуете так, чтобы оградить себя от повторения неприятных событий¹.

По большому счету, обучение людей или животных не должно отличаться от обучения компьютеров. Вспомните приведенное выше определение искусственного интеллекта - ведь оно похоже на описание действий любого разумного существа, хотя, к сожалению, не дает нам возможности понять, как именно мы обобщаем наблюдаемые факты, чтобы выбрать наиболее эффективный способ действий.

Я верю в то, что в ближайшем будущем появятся компьютерные программы, которые смогут и сдать экзамен, и поддержать разговор, но их, скорее всего, нельзя будет сравнить с настоящим (пусть даже и искусственным) интеллектом, если они не будут обладать возможностью обучаться, подобно тому как это делают животные и человек. Вероятно, пройдет немало лет, прежде чем ученые действительно смогут понять, как можно научить программы воспринимать и обобщать информацию, приходящую из окружающего мира.

Современное состояние науки об искусственном интеллекте позволяет предположить, что первые шаги к пониманию принципов работы нашего сознания мы можем преодолеть, изучая работу основных клеток, из которых состоит мозг животных и человека, - *нейронов*.

Эксперименты с искусственными моделями нервных клеток ученые начали проводить в 40-х годах XX века, пытаясь имитировать работу мозга. Большое

¹ Если читатель увлекается не только программированием, но и психологией, то, вероятно, слышал о нейролингвистическом программировании (НЛП), в котором среди прочих используется техника так называемых «якорей», когда за той или иной комбинацией внешних воздействий психотерапевт закрепляет в подсознании человека нужную реакцию. - *Прим. перев.*

количество нейронов соединялось, образуя так называемые искусственные *нейронные сети*.

Если вы знакомы с цифровой электроникой, то можете представить нейрон в виде некоторого многовходового логического элемента (AND, OR, XOR и т.п.). Но на самом деле это аналоговый элемент, который воспринимает несколько входных сигналов, причем имеет различную чувствительность к тем сигналам, которые поступают на разные его входы. Суммарное воздействие всех входных сигналов определяет *активность* нейрона - степень его возбуждения. Эта величина подвергается обработке и в виде результирующего сигнала подается на единственный выход нейрона. Выходной его сигнал, в свою очередь, является входным для многих других нервных клеток, которые соединены с рассматриваемым нейроном.

Можно придумать несколько хороших определений нейронной сети, но наилучшее из них выглядит следующим образом: нейронная сеть - это один или множество связанных друг с другом простых процессорных элементов, работающих параллельно, функционирование которых зависит от структуры сети, степени связи между элементами и алгоритма их *работы*².

Проще говоря, нейронная сеть - это набор связанных элементов, каждый из которых воспринимает свои входные сигналы и вырабатывает по некоторому правилу выходной сигнал. На рис. 5.2 показан пример простой нейронной сети, которая по состоянию двух световых датчиков и двух детекторов столкновений формирует команды для двух двигателей робота.

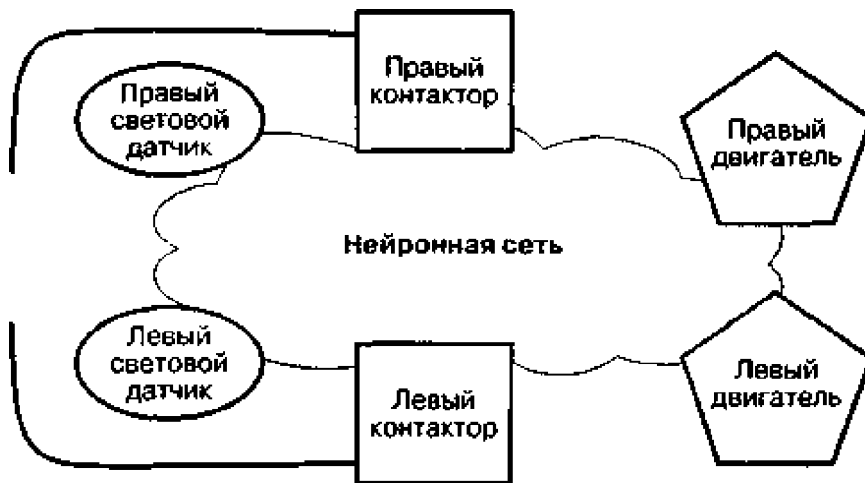


Рис. 5.2. Нейронная сеть управляет двигателями робота

На языке математики все это можно описать простой формулой:

Выходы = f (Входы),

где f - некоторое правило, согласно которому набор значений входных сигналов преобразуется в соответствующий набор значений выходных сигналов.

² DARPA Neural Network Study, 1988. - AFCEA International Press. - P. 60.

Алгоритм, определяющий это правило, может быть реализован с помощью логических функций, конечно-автоматных моделей или просто в виде некоторой последовательности операторов любого языка программирования, но важно понимать, что данный алгоритм не предопределяет реакцию сети на то или иное входное воздействие, а только описывает способ, который используется нейронной сетью для ее обучения.

Такое обучение строится на примерах, что позволяет провести аналогию с тренировкой животных. На вход сети подается тот или иной набор внешних воздействий, и выходные сигналы сравниваются с теми, которые мы желали бы иметь в данных условиях. Если ошибка сети (разница между желаемым и действительным значениями выходного сигнала) велика, то чувствительность входов каждого нейрона по определенным правилам изменяется таким образом, чтобы снизить эту ошибку. Так повторяется много раз для различных наборов входных воздействий и соответствующих наборов желаемых реакций, пока сеть не научится правильно реагировать на предлагаемые воздействия.

Очень важно правильно выбрать структуру нейронной сети, то есть количество составляющих ее нейронных элементов и связи между ними. Если нейронов будет слишком мало или связи между ними окажутся неэффективными, то сеть не сможет научиться тем вещам, которым мы хотим ее обучить. Если нейронов, наоборот, будет слишком много, то процесс обучения грозит затянуться на неопределенное время.

После начальной тренировки сеть может быть использована для управления роботом, но при этом ее обучение продолжится, если она сама будет способна каким-то образом оценивать качество своего поведения. Например, показанная на рис. 5.2 нейронная сеть может следить за временем, которое потребовалось ей на то, чтобы в определенной ситуации найти источник света, минимальное количество раз столкнувшись с окружающими предметами.

Кроме имитации поведения бабочки, летящей на свет, такой механизм обучения может быть использован и для более полезных применений, например для управления захватом робота-манипулятора, который должен найти и подобрать определенный предмет. Человек делает это совершенно не задумываясь, но программирование такой задачи традиционными методами довольно затруднительно.

Способ обучения нейронной сети, основанный на примерах, может быть реализован различными способами. По существу, такое обучение сродни тренировке собак: дрессировщик дает команду, после чего поощряет правильное ее выполнение и наказывает за неправильное. Это многим покажется удивительным, но существуют также методы обучения без учителя, когда на входы нейронной сети подаются различные наборы воздействий, но никто не говорит ей, какими в каждом из этих случаев должны быть ее выходные сигналы.

Для выбора правильного набора параметров нейронной сети можно использовать так называемые *генетические алгоритмы*. Они, как и сами нейронные сети, заимствованы исследователями у живой природы, поскольку основаны на имита-

ции происходящих в ней процессов размножения, изменчивости и естественного отбора. Генетический алгоритм позволяет набору (популяции) нескольких нейронных сетей, предназначенных для решения некоторой задачи, эволюционировать таким образом, чтобы в результате сформировать сеть, которая смогла бы решить эту задачу самым эффективным образом.

Обладает ли нейронная сеть искусственным интеллектом? Многие исследователи склонны ответить на этот вопрос положительно, ведь способ формирования реакции нейронной сети на входные воздействия очень напоминает процесс принятия решений живыми организмами. Но, на мой взгляд, это было бы правильнее назвать «искусственным рефлексом». Пока что нейронная сеть скорее моделирует рефлекторное движение руки человека, стукнувшего молотком по своему пальцу, чем проявление настоящего интеллекта.

ГЛАВА 6

ПРОЕКТИРОВАНИЕ

АВТОМАТИЧЕСКИХ УСТРОЙСТВ

Тот факт, что эта глава находится в конце книги, может вызвать некоторое удивление. Но нам нужно было обсудить много важных вещей, прежде чем приступить к проектированию настоящих роботов.

В предыдущих главах мы рассмотрели, как реализуются различные интерфейсы и как программируется процесс принятия решений на верхнем уровне. Теперь пришла пора выяснить, как можно заставить все эти отдельные части работать совместно.

Но перед тем, как начать разработку автоматического устройства, надо спросить себя, для чего конкретно оно будет предназначено. Мало сказать: «Я хочу сконструировать робота, чтобы разобраться, как он работает». Надо сформулировать свои намерения более точно, например: «Я хочу сконструировать двухколесного робота, каждое колесо которого управляется отдельным двигателем и который двигается по прямой линии, пока не встретит на своем пути препятствие, после чего объезжает его и снова продолжает движение в прежнем направлении».

После того как вы четко определите, что же хотите получить, можно приступить к проектированию. По мере своего продвижения вперед вы, вероятно, захотите кое-что изменить в исходной формулировке, но главное, чтобы у вас была отправная точка. Иначе вам даже не удастся толком описать, что необходимо добавить к первоначальному замыслу.

Важно, чтобы ваш первоначальный замысел был не слишком фантастичен. Многие радиолюбители поначалу хотят достичь результатов, в погоне за которыми даже целые коллективы профессиональных разработчиков безрезультатно потратили много лет и миллионы долларов. Классическим примером такого безумного проекта может служить робот, который по приказу своего хозяина умел бы приносить пиво из холодильника. Здесь сначала надо придумать, как робот откроет дверцу холодильника, как отыщет там бутылку с пивом, возьмет ее, закроет дверцу и, наконец, как найдет своего хозяина, чтобы отдать ему пиво. А если хозяин и холодильник находятся на разных этажах? И это только некоторые из самых очевидных вопросов. Уверен, немного поразмыслив, вы придумаете еще кучу других, не менее заковыристых. Но на главные вопросы надо найти ответ прежде, чем начинать работу над проектом. Иначе лучше и не начинать.

Я несколько не преувеличиваю, говоря, что ученые потратили кучу времени и денег на подобные разработки. В самом деле, если робот должен уметь

при необходимости подняться вверх по лестнице, значит, это умение не должно зависеть от ширины и высоты ступенек, а также материала, из которого они сделаны.

В главе 4 мы рассмотрели множество различных интерфейсов; они были разработаны таким образом, чтобы в максимальной степени облегчить их интеграцию друг с другом в пределах одного большого проекта. Тем не менее очень часто при проектировании сложных автоматических устройств на основе микроконтроллеров бывает весьма нелегко достичь устойчивой работы всех подсистем. То не хватает объема памяти для размещения всех подпрограмм, то слишком долго выполняется биологический код, вследствие чего робот не успевает вовремя среагировать на все изменения окружающей обстановки, то один интерфейс мешает работе других... Эти проблемы по возможности лучше решать с самого начала, так как намного сложнее вносить изменения в почти готовое приложение.

Здесь стоит упомянуть такое явление, как «журнальная зависть». Часто, прочитав интересную статью в радиолобительском журнале, понимаешь, что именно эту идею надо использовать в своей новой разработке, но проблема в том, что многим советам, которые дает автор статьи, нельзя последовать в точности. Некоторые комплектующие могут оказаться слишком дорогими или вовсе недоступными, поэтому придется искать им замену, и не всегда равноценную. Описание каких-либо особенностей конструкции в статье может быть опущено. В результате, даже повторяя чужую разработку, вы получаете нечто отличное от нее — и не обязательно с тем же результатом. Разумеется, в любом случае вы только выиграете, если постараетесь перенять опыт других радиолобителей и использовать понравившиеся вам находки в своих проектах.

В конце концов, сконструировать робота довольно просто. Сложнее заставить его делать в точности то, что вы задумали. Поэтому в данной главе мы рассмотрим несколько полезных приемов отладки приложений, которые помогут вам понять, что на самом деле происходит с роботом в тех или иных случаях. Я подчеркиваю, что любые решения разработчик должен принимать, опираясь на данные экспериментов, а не на свои предположения по поводу того, что обуславливает неправильное поведение робота.

6.1. ТЕХНИЧЕСКОЕ ЗАДАНИЕ

Хочу дать вам хороший совет: записывайте все! Заведите привычку носить с собой блокнот, и как только у вас появится стоящая идея, запишите ее. На самом деле человеческая память не так надежна, как многие склонны полагать. Должен признаться, не раз я часами — и безуспешно! — пытался вспомнить гениальную идею, которая осеняла меня во время завтрака.

Всегда старайтесь начинать с малого. Если вы задумали сконструировать робота-слугу, который всюду следует за вами и с готовностью выполняет любые ваши поручения, то разумнее начать с создания небольшой тележки (но с большим запасом места для добавления новых электронных блоков), которая только и умеет, что передвигаться по горизонтальной поверхности.

Планируя свою работу, всегда прогнозируйте самый худший случай (поверьте, вы с ним довольно скоро столкнетесь). Никогда не предполагайте, что первый же пришедший вам в голову способ реализации той или иной функции окажется удачным. Как правило, для достижения небольшого успеха требуется перепробовать несколько возможностей.

Возьмем для примера одну из программ, представленных в этой книге, - управление светодиодным индикатором. Пытаясь добиться того, чтобы микроконтроллер включал и выключал светодиод с заданной периодичностью, я был вынужден рассмотреть следующие возможности:

- программная реализация задержек;
- программный опрос таймера TMR0;
- часы реального времени на основе прерываний от таймера TMR0 (именно этот метод и был использован);
- широтно-импульсная модуляция сигнала с применением таймера TMR2.

Какой из этих методов является наилучшим? Какие дополнительные возможности обеспечивает тот или иной способ реализации заданной функции? Какое решение наиболее простое и не затрудняет дальнейшую модернизацию устройства? Помните о том, что метод, прекрасно показавший себя на лабораторном столе, может оказаться совершенно ненадежным, работая в реальных условиях, поэтому вы всегда должны иметь пространство для маневра.

К сожалению, многие светлые мысли приходят слишком поздно, когда работа над проектом уже близится к концу. Поэтому иногда разумнее зафиксировать новую идею на бумаге, чтобы опробовать ее в будущем, вместо того чтобы без конца переделывать уже почти готовое устройство. Лучше руководствоваться правилом: «Если не сломано - не следует исправлять».

И последнее: не бойтесь заимствовать чужие идеи. Разумеется, я не подстрекаю вас к плагиату или похищению чужих чертежей, но свежие мысли и решения, почерпнутые вами у других разработчиков, иногда оказываются единственным выходом из тупика, в который вы зашли. К тому же чужие идеи всегда можно применить совершенно новым способом.

В сети Internet вы найдете огромное количество информации по микроконтроллерам и методам конструирования автоматических устройств, готовые схемы, идеи, документацию по различным комплектующим и многие другие сведения, которые будут для вас чрезвычайно полезными или, напротив, никогда не пригодятся. Используйте эту информацию. Но должен предостеречь: не все, что вы найдете в Сети, имеет практическую ценность. Многие данные могут быть не вполне корректными, а некоторые и вовсе неверными. Разумеется, это касается любой области, не только робототехники. И дело здесь не только в том, что разместить страничку в сети Internet может любой желающий. Нередко я встречал на разных сайтах так называемые «бесплатные» программы и проекты, которые были доступны для свободного скачивания, но на проверку оказывались неработоспособными, а лишь играли роль приманки, вынуждая вас приобрести более «продвинутый» проект или обратиться к его автору за консультацией - разумеется, тоже не бесплатной.

Итак, вы хотите приступить к проектированию нового автоматического устройства. Начать следует с *технического задания*. Звучит несколько официально, но имеется в виду всего лишь список требований, которым должно удовлетворять разрабатываемое устройство. Начните с простых утверждений, поясняющих, что и как должен делать робот.

Далее пункт «как» надо конкретизировать. Приведем примерный список основных вопросов:

Какую аппаратную платформу следует использовать?

Каковы должны быть размеры и вес устройства?

Какая необходима скорость передвижения?

Какие двигатели будут использоваться?

Что послужит источником питания?

Какие меры безопасности надо предусмотреть?

Какие контроллеры будут использованы?

На каком языке программирования будет разрабатываться управляющее программное обеспечение?

Какие датчики будут снабжать робота информацией об окружающей обстановке?

Какие будут применены исполнительные (выходные) устройства?

Какие методы ручного управления необходимы?

Каким должен быть интерфейс пользователя?

Какова будет стоимость всего проекта?

На такие вопросы нельзя формально ответить «нет» или «без ограничений». Вы можете считать робота совершенно безопасным (потому что он маленький), но если в нем используются свинцовые, кислотные или **никель-кадмиевые** батареи, то они вредны для окружающей среды. А если робот весит несколько десятков килограммов, то трудно представить, что он может натворить, если выйдет из-под контроля из-за неправильной работы управляющего устройства.

Кроме списка аппаратных требований надо подготовить список требований к программному обеспечению. Следует определить:

- язык программирования;
- средства отладки приложения (эмулятор, симулятор);
- доступность средств загрузки программы в микроконтроллер (наличие специального программатора);
- используемые в программе векторы прерываний;
- программные интерфейсы, встроенные в систему;
- используемые в программе структуры данных.

Следует отметить, что оба списка требований взаимосвязаны. Выбор того или иного микроконтроллера ограничивает ваши возможности по выбору средств программирования и отладки.

Заметьте, что здесь пока не учтены такие важные требования, как, например, предположительный размер программы и данных. Знание этих параметров может существенно повлиять на выбор контроллера и методов программирования, но на начальных этапах проектирования очень трудно оценить, сколько памяти потребуется для

всего приложения. Я уже много лет занимаюсь такими вещами, но до сих пор часто выходит, что реальный объём готового приложения оказывается раза в два больше того, который был запланирован исходно.

Ничто не оказывает такого впечатления на зрителей, как быстрота передвижения робота. Как ни странно, ваша конструкция произведет впечатление более сложной и «умной», если будет двигаться быстро. Однако за это приходится расплачиваться, используя более мощные двигатели и, соответственно, источники питания. Кроме того, двигаясь с большой скоростью, робот должен уметь быстро реагировать на окружающую обстановку. Поэтому в погоне за высокими скоростями разработчик рискует приобрести себе кучу неприятностей в будущем, когда захочет модернизировать свой проект, так как зачастую ни малейшего запаса прочности уже не остается.

Чуть ниже мы обсудим, как составлять список требований к будущему проекту. Если вы радиолюбитель, а не профессиональный разработчик, то, возможно, все это покажется вам пустой тратой времени, но уверяю вас, что дело обстоит иначе. Хорошо составленное техническое задание существенно облегчит дальнейшую работу по проектированию автоматических устройств.

По мере того как вы будете продвигаться вперед, может возникнуть необходимость внесения тех или иных изменений в уже составленное техническое задание (например, если окажется, что выбранный вами микроконтроллер не поддерживает какую-либо возможность, которую вы предусмотрели в самом начале, или вам так и не удастся достать нужный элемент, а возможные варианты его замены не позволяют обойтись без некоторых переделок). Ничего страшного - смело вносите изменения в первоначальный список требований!

6.2. ВЫБОР ПЕРИФЕРИЙНЫХ УСТРОЙСТВ

После того как будут определены и перечислены все функции, которые должен выполнять робот, можно приступить к выбору периферийных устройств. Если выбор окажется не оптимальным, вы рискуете потратить кучу времени на безуспешные попытки найти ошибку в программе, в то время как истинным виновником неправильной работы является какое-нибудь устройство ввода-вывода.

Например, может оказаться, что некоторое входное устройство слишком медленно или требует сложного программирования; в результате процедура обработки прерываний будет отнимать слишком много командных циклов работы процессора и не оставит достаточно времени для выполнения биологического кода. Или некоторые интерфейсы в борьбе за общий аппаратный ресурс проявят плохую совместимость друг с другом, и приложение будет пропускать некоторые внешние события (скажем, команды дистанционного управления или моменты столкновений с окружающими предметами).

Продемонстрированный в данной книге подход, основанный на разделении функций механического, электронного и биологического уровней и реализации процедур первых двух уровней в виде обработчика прерываний от таймера, позволяет свести к минимуму вероятность подобных проблем, однако следует понимать, что

такое решение не является панацеей - вполне возможны случаи, когда разработчику придется проявить изобретательность, пытаясь увязать друг с другом различные интерфейсы. В любом случае при проектировании следует стремиться к тому, чтобы отдельные функции механического и электронного уровней как можно меньше взаимодействовали друг с другом.

Кроме того, не следует забывать, что для правильной работы биологического кода необходимо оставить достаточно процессорного времени на его выполнение. Поэтому надо стремиться к наиболее простой реализации механических и электронных интерфейсов, избегая использования в обработчике прерываний сложных вычислений, чисел в формате с плавающей точкой, операций в цикле и т.п. Все трудоемкие вычисления следует производить в главной программе (то есть с помощью биологического кода). Только тогда будет гарантия, что во время их выполнения не нарушится работа какого-либо периферийного устройства.

Таким образом, можно сформулировать следующие основные рекомендации по разработке функций механического и электронного уровней:

- используйте конечные автоматы для реализации механических и электронных интерфейсов. Это может несколько замедлить их работу, но зато не мешает работе других интерфейсов;
- не используйте переменные с плавающей точкой и массивы;
- старайтесь избегать сложных математических операций (умножение, деление, вычисление логарифмов и тригонометрических функций);
- используйте микроконтроллер с наиболее широким набором встроенных функций.

6.3. ВЫБОР ЭЛЕКТРОННЫХ КОМПОНЕНТОВ И МЕТОДОВ ПРОГРАММИРОВАНИЯ

При разработке технического задания пригодится следующий список вопросов, ответы на которые определяют выбор электронных компонентов, требуемых при создании робота.

Какой микроконтроллер желательно использовать?

Какого типа будет тактовый генератор?

Каким должен быть источник питания?

Какие потребуются датчики и другие устройства ввода информации?

Какие потребуются двигатели и другие устройства вывода?

Нужно ли использовать запоминающие устройства (кроме памяти, встроенной в микроконтроллер)?

Насколько доступны необходимые электронные компоненты?

Последний вопрос особенно важен, если вы хотите опубликовать схему своего устройства и надеетесь, что другие радиолюбители смогут ее собрать. Например, в самом начале работы над программатором El Cheapo я использовал МОП транзистор с каналом р-типа, считая, что он широко распространен и вряд ли у кого-то возникнут трудности с его приобретением или заменой достаточно близким

аналогом. К сожалению, мое предположение не подтвердилось: многие радиолюбители стали жаловаться на то, что не могут достать полевой транзистор, так что его пришлось заменить распространенным биполярным р-п-р транзистором 2N3906.

Следует помнить, что в радиолюбительских конструкциях в первую очередь важна доступность элементов — зачастую она более значима, чем стоимость. Поэтому перед тем, как опубликовать свою схему в сети Internet или в радиолюбительском журнале, стоит просмотреть каталоги ведущих производителей радиоэлементов.

Рано или поздно те элементы, которые вы на протяжении многих лет использовали в своих конструкциях, устаревают. Это большая проблема, но часто только психологическая: во многих случаях среди новых элементов можно подыскать достаточно близкий аналог. Ну, а если нет, то приходится вносить изменения в устройство и в управляющую программу.

Чтобы обезопасить себя от возможных проблем, желательно уже в самом начале работы над проектом стараться использовать только самые распространенные и стандартные электронные компоненты. Для примера рекомендую:

- диоды 1N4001, 1N4192 (1N915);
- транзисторы 2N3904/2N3906, BC547/BC557;
- полевые транзисторы IRF510/IRF9510;
- керамические резонаторы ECS/Panasonic;
- инфракрасные светоизлучающие диоды и детекторы фирмы Watrony;
- логические элементы ТТЛ серии 74LS;
- интегральные стабилизаторы напряжения 78(L)xx;
- таймер (7)555.

Каких-либо проблем с резисторами и конденсаторами, как правило, не возникает. Иногда трудно найти танталовые конденсаторы, используемые для фильтрации напряжения питания логических микросхем, но их можно заменить керамическими.

Ключевой вопрос - выбор микроконтроллера. Обычно разработчики основывают свое решение не столько на объективном сравнении требований технического задания и характеристик того или иного микроконтроллера, сколько на своих пристрастиях и опыте предыдущих проектов. Не последними доводами служат доступность микроконтроллера и разнообразие программного обеспечения для него. Разумеется, не следует останавливать свой выбор на микроконтроллере, для работы которого требуются еще какие-то дополнительные элементы кроме источника питания 5 В, генератора тактовых импульсов и «подтягивающего» резистора для входа сброса.

Что касается быстродействия микроконтроллера (его максимальной тактовой частоты), этот параметр обычно не является существенным. Наоборот, для уменьшения мощности, потребляемой от источника питания, а также для снижения уровня помех я рекомендую вам выбирать для своих конструкций не слишком высокую частоту работы тактового генератора. Гораздо более важная характеристика контроллера - его устойчивость к помехам и работоспособность при помехах от периферийных устройств, входящих в конструкцию робота.

Наиболее оптимальным решением при конструировании электронных управляющих устройств мобильных роботов является использование печатного монтажа.

Разрабатывая печатную плату электронного блока, следует иметь в виду, что при работе автоматических устройств всегда присутствуют следующие мешающие факторы:

- электрические помехи;
- электромагнитные наводки;
- механические вибрации;
- акустические шумы;
- колебания температуры и влажности окружающей среды.

Кроме того, не следует забывать, что у мобильных автоматических устройств есть подвижные части, на которые могут наматываться провода, поэтому все соединения надо производить очень аккуратно.

Я советую использовать монтажные провода минимальной длины и паяные соединения; всегда заботьтесь о качестве заземления и надежности крепления механических узлов.

При размещении элементов схемы на печатной плате учтите, что в дальнейшем может возникнуть необходимость модернизации схемы или добавления новых блоков. Не старайтесь компоновать все элементы слишком плотно - оставляйте место для дополнительных элементов.

Следует подчеркнуть, что макетные платы, которые используются в проектах, описанных в главе 4, являются наихудшим решением с точки зрения надежности конструкции и возможности модернизации схемы. Они облегчают проверку несложных схем, но вы можете на собственном опыте убедиться, как нелегко оказывается разобраться в переплетении проводов, чтобы найти неисправность или подключить новый блок.

Для размещения микроконтроллера всегда необходимо использовать специальный разъем, который **облегчит** возможность его перепрограммирования или замены.

6.4. ИСПЫТАНИЯ РОБОТА

Разработчику следует определить список испытаний, которые должен пройти робот, прежде чем можно будет сказать, что работа закончена и готова к тому, чтобы вынести ее на суд зрителей. Типичный список испытаний выглядит следующим образом:

- проверка стабильности работы во всех режимах (в том числе при пуске-останове двигателей) при крайних значениях напряжения питания (4,75 и 5,25 В);
- проверка начальной инициализации при включении питания и готовности выполнения команд пользователя;
- проверка правильности выполнения команд пользователя;
- проверка правильности работы датчиков.

Кроме того, возможно, придется позаботиться о разработке необходимой документации, расчете надежности работы всех узлов и даже о тестировании разработанной конструкции на соответствие санитарным и прочим требованиям и нормам.

Хотя большинство микроконтроллеров сохраняет свою работоспособность при изменении напряжения питания в более широких пределах, чем указано выше, не стоит забывать и о других электронных компонентах, многие из которых выдерживают только небольшие колебания напряжения питания ($\pm 5\%$).

Помните о соблюдении мер безопасности. Разумеется, маленький робот не сможет причинить вреда окружающим, но он может разбиться сам, упав со стола, если вы вдруг потеряете над ним контроль.

По этой причине я советую всегда предусматривать какой-либо метод блокировки колес, который предохранит робота от поломки, если вы вдруг забудете выключить напряжение питания, так как даже полностью отлаженное устройство может неожиданно убежать от вас, случайно получив сигнал от светового датчика.

Во многих случаях при тестировании схемы недостаточно одного логического пробника. Идеальный вариант - использование осциллографа и логического анализатора, позволяющее проверить правильность всех выходных сигналов при всех возможных значениях входных.

По роду занятий я почти всю жизнь проверял правильность работы тех или иных устройств, поэтому вопрос о тестировании мне наиболее близок. К сожалению, даже самое добросовестное тестирование не дает полной гарантии работоспособности устройства во всех обстоятельствах: всегда существует огромное количество вариантов возможных неисправностей.

При тестировании в первую очередь надо убедиться, что на предусмотренные значения входных сигналов ваше устройство реагирует правильно. Следует контролировать значение этих входных сигналов, так как всегда может оказаться, что странности в поведении робота обусловлены не ошибкой в программе, а случайным отблеском, попавшим на световой датчик.

Если значения входных сигналов или служебных переменных управляющей программы вышли за допустимые пределы, то робот должен остановиться и каким-либо образом сигнализировать о возникшей проблеме. В наших проектах мы не использовали сторожевой таймер, но в реальных конструкциях он бывает весьма полезным, позволяя сформировать сигнал сброса микроконтроллера, если управляющая программа зависла и команда сброса сторожевого таймера не выполнялась вовремя.

Ваш проект должен быть подробно документирован. В противном случае вряд ли кто-нибудь захочет его реализовать. Разумеется, вся документация должна быть доступна в электронном виде. Самым простым и переносимым для электронных документов является обычный текстовый формат ASCII. Но если необходимо включение в текст рисунков, графиков, схем и/или формул, то лучшим решением будут доступные на всех компьютерных платформах гипертекстовый формат HTML и формат PDF фирмы Adobe.

6.5. ПОИСК ОШИБОК

Вместо термина *отладка* (debug), который традиционно ассоциируется с несколько небрежным и не слишком вдумчивым процессом устранения ошибок в программах, я предпочитаю использовать более солидное выражение *анализ ошибок* (failure analysis). Интересно, как простая замена технического термина сразу смещает акценты: не поиск «жучков», а структурированный анализ проблемы, позволяющий локализовать и устранить источник неправильной работы устройства!

Вместо того чтобы вслепую заменять все элементы подряд, надеясь, что после очередной замены устройство вдруг начнет работать правильно, примите к сведению следующий алгоритм:

- четко сформулируйте, в чем заключается проблема;
- рассмотрите возможные варианты причин неправильной работы;
- разработайте план **выявления** истинной причины и ее устранения.

Разработчик должен уметь провести подобный анализ в разнообразных ситуациях, в том числе и в самых крайних случаях, когда кажется, что устройство совсем не работает или работает «практически правильно».

Когда еще нет никаких идей по поводу того, что именно вызывает проблему, следует проверить логические уровни на всех элементах схемы. На данном этапе я обычно использую простой логический пробник. Если такое исследование не прояснило ситуацию, то, возможно, придется взять в руки авометр и проверить уровни напряжений на аналоговых и цифровых входах и выходах. Следует записывать все результаты измерений.

Проверяя работу встроенного тактового генератора, надо помнить, что дополнительная емкость, вносимая логическим пробником, может нарушить процесс генерации и вызовет зависание программы. Чтобы убедиться в том, что программа все еще выполняется, следует инициализировать выполнение какой-нибудь простой процедуры с четко видимым внешним эффектом (например, миганием светодиода).

От вас требуется предельное внимание. Мне стыдно признаться, сколько раз я готов был воскликнуть «Эврика!» и тут же обнаруживал, что проверяю не ту ножку микросхемы. После того как все замеры сделаны, отложите устройство в сторону и проанализируйте, значения каких сигналов оказались вне ожидаемого диапазона.

Любое несоответствие объясняется одой из двух причин: либо внешнее устройство подает на данный вывод неправильный сигнал, либо управляющая программа формирует на нем неправильное значение. Чтобы выяснить, с каким именно вариантом вы имеете дело, можно использовать омметр, проверяя с его помощью сопротивление между данным выводом и массой до и после его отключения от остальной схемы. К сожалению, для отключения вывода может потребоваться паяльник.

Следует соблюдать максимальную объективность, стараясь не делать слишком поспешных предположений относительно того, что привело к возникновению

ошибки. Неверно определив причину, люди начинают искать только подтверждения своей догадки, пропуская все остальные возможные варианты.

После проверки схемы можно приступать к тестированию программного обеспечения. Прежде всего я рекомендую вынуть микроконтроллер из схемы и проверить прошивку РПЗУ с помощью программатора. Обратите внимание на правильность слова конфигурации, проверив, корректно ли установлен тип тактового генератора, не разрешена ли случайно работа сторожевого таймера и т.п.

Если все в порядке, переходите к самой программе. При этом не следует забывать о необходимости подробно записывать все действия - как ни удивительно, это экономит много времени.

Например, если проблемы возникли с последовательным интерфейсом передачи данных, выясните, связана ли проблема с временными задержками, значениями отдельных битов или их количеством. После того как ситуация стала более определенной, можно выполнить моделирование работы программы с помощью симулятора, чтобы в еще большей степени локализовать источник неправильного поведения робота.

Теперь требуется выяснить, что стало причиной ошибки. Для этого может потребоваться мысленный эксперимент, в ходе которого вы должны представить себе во всех подробностях, как и что происходит в управляющей программе и самой схеме в тот или иной момент.

Следует понимать, что причин неисправности может быть сразу несколько. Работая над своими конструкциями, я не раз сталкивался со случаями, когда насчитывалось до пяти различных источников проблем, которые мешали правильной работе устройства.

Поэтому не следует быть слишком ленивым. Поверьте, во многих случаях вы только сэкономите время, если сделаете шаг немного в сторону и начнете с отладки более простого приложения, где должна будет проявиться одна из имеющихся проблем. Тогда можно будет без помех изучить ситуацию и опробовать различные варианты решения, после чего опять вернуться к сложному проекту.

6.6. МОДЕРНИЗАЦИЯ УСТРОЙСТВ

После того как ваша схема, наконец, заработает, вы захотите добавить к ней новые периферийные устройства и разнообразные функции. При этом у вас, скорее всего, не возникнет особых проблем с электроникой или механическими частями. Самым сложным, как правило, оказывается перераспределение аппаратных ресурсов, встроенных в микроконтроллер. Речь идет о памяти программ и данных, таймерах, обработчиках прерываний, блоках формирования ШИМ-сигнала.

Другая проблема возникает, когда вы хотите заимствовать для своего проекта программный код из какого-либо источника (например, Internet). В таком случае оказывается, что различные участки кода написаны с использованием

разных стилей и методик, поэтому может потребоваться кардинально переработать программу, чтобы заставить все это работать вместе.

Если приложение спроектировано правильно, то внесение необходимых изменений - не слишком сложная задача, но в противном случае проще написать все заново, чем пытаться как-то исправить отдельные фрагменты.

Уже в самом начале работы над проектом желательно закладывать в него возможность модификации. Своего рода образцом правильной разработки приложений могут служить примеры, приведенные в главе 4, где для реализации механических и электронных интерфейсов мы использовали прерывания от таймера, срабатывающего каждую миллисекунду.

При проектировании биологического кода надо помнить, что для его правильной работы в промежутке между запросами на прерывание следует оставлять по крайней мере 25% всех командных циклов, таким образом, отводя для выполнения обработчика прерываний не более 75% времени. Сложные вычисления нельзя реализовывать в процедуре обработки прерываний - их надо выносить в основную программу.

В идеале различные интерфейсные функции должны использовать разные аппаратные средства. Мы нарушили это правило лишь однажды, когда рассматривали приемник команд дистанционного управления, который использовал тот же инфракрасный детектор, что и обнаружитель объектов.

Применяя чужие разработки, не следует пытаться использовать в своем проекте программный код, для которого не имеется исходного текста. Дело в том, что в новом окружении даже самая правильная программа может потребовать некоторых переделок, а это невозможно, если имеется только объектный код.

Все описанные в данной книге устройства мы реализовали с использованием только одного микроконтроллера, но при модернизации уже имеющихся схем наилучшее решение иногда заключается в добавлении еще одного контроллера. Поэтому никогда не помешает оставить свободными какие-нибудь два вывода, которые затем пригодятся для обеспечения их взаимосвязи. Две линии нужны для реализации синхронного обмена, так как в этом случае не требуется загружать таймеры обоих микроконтроллеров.

Даже если в схеме использовано несколько микроконтроллеров, желательно, чтобы только один из них выполнял биологический код. В противном случае программное обеспечение заметно усложняется, поскольку требуется обеспечить взаимодействие отдельных частей биологического кода, выполняемого на разных процессорах. Кроме того, отладить такое распределенное приложение будет не очень-то просто.

И последнее замечание: не забывайте, что следует отключать двигатели на то время, пока микроконтроллер работает с периферийными устройствами. В течение опроса инфракрасных, световых и звуковых датчиков надо стремиться к тому, чтобы уровень помех был минимальным.

ПРИЛОЖЕНИЕ 1

Глоссарий

Здесь приведен список всех использованных в книге терминов и расшифровка аббревиатур.

Адрес — порядковый номер регистра, ячейки памяти команд или данных в пределах пространства памяти процессора.

Адресное пространство - диапазон возможных адресов ячеек памяти или портов ввода-вывода, с которыми может работать процессор.

Активные элементы - интегральные микросхемы, транзисторы и другие элементы электронных схем, для функционирования которых требуется внешний источник энергии.

Ампер - величина силы тока: $1 \text{ А} = 1 \text{ Кл} / 1 \text{ с}$. См. Кулон.

Аналоговый сигнал - сигнал, который может принимать любые значения в некотором диапазоне.

Анод - положительный электрод. См. Катод.

Асинхронная последовательная передача - метод обмена данными, в котором передача сообщений происходит побитно (по одной линии) и не используется дополнительная линия для передачи импульсов синхронизации.

Ассемблер - инструментальное программное средство, с помощью которого исходный текст программы, написанной на языке низкого уровня, преобразуется в объектный файл.

АЦП, аналого-цифровой преобразователь (ADC, **Analog-to-Digital Converter**) - микросхема, предназначенная для преобразования входного аналогового напряжения в цифровое представление. См. ЦАП.

Биполярные логические схемы - логические схемы, выполненные на базе биполярных транзисторов (единичные устройства или интегрированные сборки). См. n-p-n транзистор, p-n-p транзистор.

Битовая маска - набор битов, используемый для установки/сброса отдельных разрядов некоторого двоичного числа.

Внутрисхемное программирование (**ICP, In-Circuit Programming**) - загрузка программы в микроконтроллер без его извлечения из целевой схемы.

Восьмеричное число - число, представленное в позиционной системе счисления по основанию 8.

Гарвардская архитектура - архитектура процессора, имеющего отдельные шины для подключения памяти команд и памяти данных.

Герц - единица измерения частоты: $1 \text{ Гц} = 1 \text{ с}^{-1}$.

Гистерезис - зависимость передаточной характеристики от направления изменения входного сигнала. Обычно используется для фильтрации шумов в цифровых схемах.

Графический интерфейс пользователя (GUI, Graphic User Interface) - простой, ориентированный на непрофессионального пользователя интерфейс ввода-вывода информации, который задействует экран, клавиатуру и мышь.

Двоичные числа - числа, представленные в позиционной системе счисления по основанию 2. Например, десятичное число $37 = 32 + 4 + 1 = 2^4 + 2^2 + 2^0 = 10101$.

Демультимплексор - логическая микросхема, используемая для передачи входного сигнала на одну из нескольких выходных линий. См. Мультимплексор.

Десятичное число - число, представленное в позиционной системе счисления по основанию 10.

Дребезг механических контактов - эффект, заключающийся в появлении многочисленных ложных всплесков сигнала при коммутации механических переключателей.

ЖКИ (LCD, Liquid Crystal Display) - жидкокристаллический индикатор, используемый для отображения информации (обычно символьной).

Защита от дребезга - устранение ложных всплесков сигнала, возникающих при коммутации механических переключателей.

Интерпретатор - программа, предназначенная для непосредственного выполнения исходного текста приложения (без его предварительной компиляции).

Инфракрасный свет (IR, infrared) - свет, имеющий длину волны 760 нм и более, невидимый для человеческого глаза.

Катод — отрицательный электрод. См. Анод.

Катушка индуктивности - проволочная обмотка, предназначенная для хранения энергии магнитного поля. Часто используется для генерации или фильтрации электрических сигналов.

Кварцевый резонатор - устройство, используемое для генерации тактовых импульсов с точно заданной частотой.

Керамический резонатор - устройство, используемое для генерации тактовых импульсов определенной частоты (обычно с меньшей точностью, чем у кварцевого резонатора).

КМОП логика (CMOS) - тип логических микросхем, основанных на использовании п- и р-канальных униполярных транзисторов.

Командный цикл - период времени (всегда кратный длительности тактового цикла), в течение которого выполняется одна машинная команда процессора.

Компаратор - электронное устройство с двумя входами, которое используется для сравнения напряжений.

Компилятор - инструментальное программное средство, предназначенное для преобразования исходного текста программы в объектный код.

Компоновщик (linker) - инструментальное программное средство, предназначенное для объединения нескольких объектных модулей и/или библиотек и формирования исполняемого файла.

Конденсатор - устройство для сохранения электрического заряда. Часто используется в электронных схемах для фильтрации сигналов. Существуют керамические, танталовые, электролитические и др. конденсаторы; танталовые и электролитические являются поляризованными, то есть при их подключении необходимо соблюдать полярность. Короткий вывод поляризованного конденсатора — это катод (отрицательный электрод).

Конечный автомат — аппаратное устройство или логическая модель системы с конечным набором возможных состояний. Переходы между состояниями осуществляются при выполнении заданных условий и отображаются с помощью графа переходов (состояний).

Конкатенация - объединение двух битовых или строковых последовательностей.

Константа - числовое значение, которое не может быть изменено в программе.

Кросс-ассемблер - инструментальное программное средство, позволяющее производить ассемблирование исходного текста программы, написанной для другой аппаратной платформы. См. Ассемблер.

Кросс-компилятор - инструментальное программное средство, позволяющее производить компиляцию исходного текста программы, написанной для другой аппаратной платформы. См. Компилятор.

Кулон - единица электрического заряда: $1 \text{ Кл} = 1 \text{ А} \times 1 \text{ с}$. См. Ампер.

Логический анализатор - инструмент для проверки работы логических схем.

Логический вентиль (логический элемент) - электронная схема, используемая для выполнения логических операций.

Логический пробник - инструмент, позволяющий узнать уровень напряжения в заданной точке схемы.

Масса (земля, ground) - электрическая линия, относительно которой измеряются все напряжения в схеме.

Массив - упорядоченный набор однотипных переменных, доступ к которым производится с помощью номера (индекса).

Машинная команда — операция, входящая в систему команд микропроцессора (микроконтроллера).

Метка (label) - символьный идентификатор, используемый для адресации участка памяти или машинной команды.

Мультиплексор - логическая микросхема, используемая для передачи одного из входных сигналов на единственный выход. См. Демultipлексор.

Нечеткая логика (fuzzy logic) - логика, оперирующая «размытыми», нечетко определенными понятиями, например *высокий*, *низкий*, *большой*, *малый*.

Объектный файл - результат работы компилятора (в том числе ассемблера); входной файл для компоновщика.

Открытый коллектор (сток) - выход, который необходимо соединять с положительной шиной питания при помощи внешнего резистора. Выходы нескольких элементов с открытым коллектором (стоком) могут объединяться.

ОСРВ, операционная система реального времени - комплекс программных средств, предназначенных для поддержки выполнения программ в системах, где время реагирования на внешние воздействия строго ограничено.

Отладчик (debugger) - инструментальное программное средство, предназначенное для поиска ошибок в программах.

Отрицательная логика - соглашение, использующее 0 для обозначения высокого уровня напряжения (или **активного** состояния сигнала) и 1 — для низкого (или неактивного состояния).

Пассивные элементы - резисторы, конденсаторы, катушки индуктивности, диоды и другие элементы электронных схем, для функционирования которых не требуется внешний источник энергии.

Период сигнала - длительность одного полного колебания; величина, обратная частоте сигнала. Измеряется в секундах.

ПИД-регулятор - автоматический регулятор, выходной сигнал которого является суммой трех составляющих: пропорциональной, интегральной и дифференциальной.

Позиционная система счисления - метод представления числовых величин, в котором значение каждой цифры зависит от ее позиции в последовательности цифр, используемых для записи числа.

Положительная логика - соглашение, использующее 1 для обозначения высокого уровня напряжения (или активного состояния сигнала) и 0 - для низкого (или неактивного состояния).

Преобразователь уровней - устройство, используемое для сопряжения логических сигналов микросхем различных семейств (например, из ТТЛ в ЭСЛ или наоборот).

Прерывание - временная приостановка процесса выполнения основной программы и переход на специальную процедуру обработки прерывания.

Программирование микроконтроллера - загрузка программы в микроконтроллер с помощью специального программатора.

Программный опрос (**поллинг**) - периодический опрос состояния внешнего устройства в программном цикле.

Программный счетчик (РС, Program Counter) - специальный регистр микропроцессора, в котором хранится адрес выполняемой команды.

Редактор — приложение, используемое для ввода и изменения (редактирования) исходного текста программ. Обычно редакторы являются частью интегрированной среды разработки программ.

Рекурсия - вызов подпрограммы из этой же подпрограммы.

Светодиод (LED, Light-Emitted Diode) - диод, который при прохождении тока излучает электромагнитные волны (необязательно видимого диапазона).

Сила электрического тока - измеряется числом заряженных частиц, проходящих через поперечное сечение проводника в единицу времени. Единица измерения силы тока - ампер (см.).

Симулятор - программное средство, предназначенное для моделирования выполнения программ без использования настоящего микроконтроллера.

СППЗУ, стираемое программируемое постоянное запоминающее устройство (EPROM, Erased Programmable Read Only Memory) - энергонезависимая микросхема памяти, содержимое которой может быть стерто и перезаписано.

Стек - см. LIFO.

Сторожевой таймер - используется для защиты программного обеспечения от зависания. Если вовремя не выполнится команда сброса сторожевого таймера, это означает, что программа зависла; в результате будет сформирован сигнал сброса микроконтроллера.

Тактовый генератор - используется для генерации импульсов, которые управляют работой микропроцессора (микроконтроллера).

Триггер - логическое устройство, используемое для хранения одного бита информации.

Указатель (pointer) - переменная, ссылающаяся на другую переменную.

Функция - подпрограмма, возвращающая результат своего выполнения в вызвавшую ее программу.

ЦАП, цифро-аналоговый преобразователь (DAC, Digital-to-Analog Converter) - устройство, предназначенное для преобразования цифрового кода на входе в аналоговое напряжение на выходе. См. АЦП.

Цифровой сигнал - сигнал, который может принимать значения из заданного набора допустимых уровней. Часто имеется всего два разрешенных уровня, условно обозначаемые 0 и 1.

Частота сигнала - количество полных колебаний в единицу времени. Измеряется в герцах: $1 \text{ Гц} = 1 \text{ с}^{-1}$.

Шестнадцатеричное число - число, представленное в позиционной системе счисления по основанию 16. Для отображения цифр в шестнадцатеричной системе счисления используются десятичные цифры от 0 до 9 и латинские буквы от А до F.

ШИМ, широтно-импульсная модуляция - представление информации в виде импульсов одинаковой амплитуды и периода, но различной длительности.

Шина - набор линий для передачи информации.

Эмулятор - устройство, имитирующее поведение микроконтроллера в целевой схеме. Используется для отладки приложений.

Язык ассемблера - низкоуровневый язык программирования, ориентированный на систему команд конкретного процессора.

AND (И) - логическая операция, результатом которой является 1, только если все операнды равны 1.

ANSI, American National Standard Institute - Американский национальный институт стандартов.

ASCII, American Standard Character Interchange Interface - стандартный код для представления символьной информации, в котором каждый символ кодируется одним байтом.

ASCIIZ - строка символов в коде ASCII, заканчивающаяся нулевым байтом.

BCD, Binary Code Decimal (двоично-десятичный код) - код для представления числовых значений, в котором каждые четыре бита двоичного числа кодируются десятичными цифрами.

BOR, Brown-Out Reset - детектор понижения напряжения питания; схема, которая формирует сигнал сброса микроконтроллера, если его питание понижается до определенного значения.

CCP, Capture/Compare/Pulse Width Modulation (PWM) - модуль захвата, сравнения и широтно-импульсной модуляции (ШИМ).

CMOS, Complementary Metal-Oxide Semiconductor - комплементарная металл-окисел-полупроводниковая (КМОП) структура.

DIP, Dual In-line Package — корпус микросхемы с двухрядным расположением выводов.

EPROM, Erasable Programmable Read Only Memory - стираемое перепрограммируемое запоминающее устройство, СППЗУ (см.).

FIFO, First In - First Out (первым вошел - первым вышел) - набор данных, организованный в виде очереди, чтение из которого производится в том же порядке, что и запись.

Flash - тип СППЗУ (см.) с поблочным электрическим стиранием.

I2C - протокол последовательного обмена информацией.

ISP, In-System Programming - см. Внутрисхемное программирование.

LCD - см. ЖКИ.

LED - см. Светодиод.

LIFO, Last In - First Out (последним вошел - первым вышел) - набор данных, организованный в виде стека, чтение из которого производится в порядке, обратном тому, в каком производилась запись.

MCU - микроконтроллер.

NOT (НЕ, отрицание) - одноместная логическая операция (или логический элемент с одним входом), инвертирующая логический сигнал.

OR (ИЛИ) - логическая операция, результатом которой является 0, только если все операнды равны 0.

OTP, One-Time Programmable - микросхема, поддерживающая возможность только однократного программирования.

POP - извлечение значения с вершины стека.

PUSH - запись значения в вершину стека.

PWM, Pulse Width Modulation - см. ШИМ.

RTOS, Real Time Operating System - см. ОСРВ.

WDT, WatchDog Timer - см. Сторожевой таймер.

XOR (Исключающее ИЛИ) - двуместная логическая операция (элемент с двумя входами), результатом которой является 0, только если оба операнда равны друг другу.

ZIF, Zero Insertion Force - разъем с «нулевым» усилием; предохраняет выводы микросхемы от повреждений при монтаже/демонтаже.

ПРИЛОЖЕНИЕ 2

СПРАВОЧНЫЕ ДАННЫЕ

Физические константы

В табл. П2.1 приведены значения часто используемых физических констант.

Таблица П2.1. Физические константы

Обозначение	Значение константы	Описание
а.е.	$1,4959787 \times 10^{11} \text{ м}$	Астрономическая единица (расстояние от Солнца до Земли)
c	$2,99792458 \times 10^8 \text{ м/с}$	Скорость света в вакууме
e	2,7182818285	Основание натуральных логарифмов
ж	3,1415926535898	Число «пи» (отношение длины окружности к ее диаметру)
ϵ_0	$8,854\,18782 \times 10^{-12} \text{ Ф/м}$	Электрическая постоянная
μ_0	$1,256637 \times 10^{-6} \text{ Гн/м}$	Магнитная постоянная
g	$9,80665 \text{ м/с}^2$	Ускорение свободного падения
h	$6,626\,176 \times 10^{-34} \text{ Дж/Гц}$	Постоянная Планка
k	$1,380662 \times 10^{-23} \text{ Дж/К}$	Постоянная Больцмана
m_e	$0,910953 \times 10^{-30} \text{ кг}$	Масса покоя электрона
m_n	$1,674954 \times 10^{-27} \text{ кг}$	Масса покоя нейтрона
m_p	$1,672648 \times 10^{-27} \text{ кг}$	Масса покоя протона
K	$8,314 \text{ Дж/(моль} \times \text{К)}$	Универсальная газовая постоянная
v_s	331,45 м/с	Скорость звука в сухом воздухе на уровне моря при 20 °С
	1480 м/с	Скорость звука в чистой воде при 20 °С

Музыкальный звукоряд

В табл. П2.2 приведены частоты звуковых колебаний, соответствующих музыкальным нотам. Заметьте, что частота нот одной октавы отличается от следующей ровно в два раза.

Таблица П2.2. Музыкальный звукоряд

Нота	Обозначение	Частота, Гц
Соль	G	392
Соль-диез	G#	415,3

Таблица П2.2. Музыкальный звукоряд (окончание)

Нота	Обозначение	Частота, Гц
Ля	A	440
Си	B	493,9
До	C	523,3
До-диез	C#	554,4
Ре	D	587,3
Ре-диез	D#	622,3
Ми	E	659,3
Фа	F	698,5
Фа-диез	F#	740,0
Соль	G	784,0
Соль-диез	G#	830,6
Ля	A	880,0
Ля-диез	A#	932,3
Си	B	987,8

Частоты тонального набора телефонных аппаратов

В табл. П2.3 указаны частоты, используемые в тоновых номеронабирателях.

Таблица П2.3. Частоты тонального набора телефонных аппаратов

Частота, Гц	720?	1336	1477
697	1	2	3
770	4	5	6
852	7	8	9
941	*	0	#

Формулы электротехники

В данной книге используются обозначения:

- V - напряжение;
- I - сила тока;
- R - сопротивление;
- C - емкость;
- L - индуктивность.

Закон Ома

$$V = IR.$$

Мощность

$$P = VI.$$

Последовательное соединение сопротивлений

$$R_{\text{общ}} = R1 + R2 + \dots$$

Параллельное соединение сопротивлений

$$R_{\text{общ}} = 1 / ((1 / R1) + (1 / R2) + \dots).$$

Два резистора, включенные параллельно

$$R_{\text{общ}} = (R1 \times R2) / (R1 + R2).$$

Последовательное соединение емкостей

$$C_{\text{общ}} = 1 / ((1 / C1) + (1 / C2) + \dots).$$

Параллельное соединение емкостей

$$C_{\text{общ}} = C1 + C2 + \dots .$$

Мост Уитстона

При выполнении условия

$$R_u = R1 \times R3 / R2$$

(условие баланса моста) ток через измерительный прибор не течет.

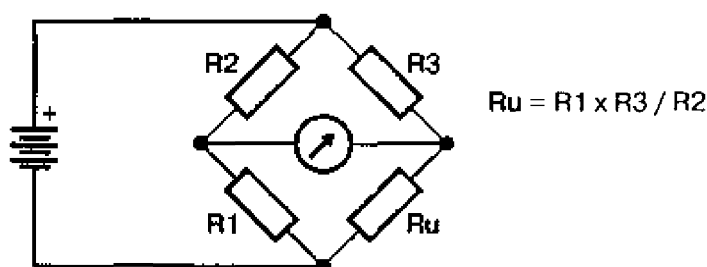


Рис. П2. Т. Мост Уитстона

Резонанс в LC-контуре

$$t = RC.$$

Постоянная времени RC-цепочки

$$t = RC.$$

Постоянная времени RL-цепочки

$$t = L / R.$$

Заряд конденсатора через резистор

$$V(t) = V_0 (1 - e^{-t/\tau});$$

$$I(t) = I_0 (1 - e^{-t/\tau}).$$

Разряд конденсатора через резистор

$$V(t) = V_0 e^{-t/\tau}$$

$$I(t) = I_0 e^{-t/\tau}$$

Трансформатор тока/напряжения

$$k = N_1 / N_2,$$

где N_1 и N_2 - число витков первичной и вторичной обмоток соответственно; k – коэффициент трансформации. Напряжение на вторичной обмотке

$$U_2 = U_1 / k,$$

а ток вторичной обмотки

$$I_2 = I_1 k.$$

Частота и длина волны колебаний

Частота = Скорость / Длина волны.

Уравнение состояния идеального газа

$$PV = nRT.$$

Булева алгебра

$$A \text{ AND } 1 = A$$

$$A \text{ OR } 0 = A$$

$$A \text{ AND } 0 = 0;$$

$$A \text{ OR } 1 = 1;$$

$$A = A;$$

$$\text{NOT} (\text{NOT} (A)) = A$$

$$A \text{ AND NOT}(A) = 0;$$

$$A \text{ OR NOT} (A) = 1;$$

$$A \text{ AND } A = A$$

$$A \text{ OR } A = A$$

Коммуникативный закон

$$A \text{ AND } B = B \text{ AND } A$$

$$A \text{ OR } B = B \text{ OR } A$$

Ассоциативный закон

$$(A \text{ AND } B) \text{ AND } C = A \text{ AND } (B \text{ AND } C) - A \text{ AND } B \text{ AND } C;$$

$$(A \text{ OR } B) \text{ OR } C = A \text{ OR } (B \text{ OR } C) - A \text{ OR } B \text{ OR } C.$$

Дистрибутивный закон

$$A \text{ AND } (B \text{ OR } C) = (A \text{ AND } B) \text{ OR } (A \text{ AND } C);$$

$$A \text{ OR } (B \text{ AND } C) = (A \text{ OR } B) \text{ AND } (A \text{ OR } C).$$

Теорема Моргана (соотношения взаимности)

$$\text{NOT} (A \text{ OR } B) = \text{NOT} (A) \text{ AND } \text{NOT} (B);$$

$$\text{NOT} (A \text{ AND } B) = \text{NOT} (A) \text{ OR } \text{NOT} (B).$$

Замечание. Логическая операция И часто обозначается знаком умножения, а логическое ИЛИ - знаком «плюс».

Соотношения единиц измерений физических величин

$$1 \text{ дюйм} = 2,54 \text{ см}$$

$$1 \text{ миля} = 1,609 \text{ км}$$

$$1 \text{ унция} = 29,57 \text{ г}$$

$$1 \text{ американский галлон} = 3,78 \text{ л}$$

$$1 \text{ атм.} = 29,9213 \text{ мм рт. ст.} = 14,696 \text{ фунта на кв. дюйм} = 101,325 \text{ кПа}$$

10 000 000 000 ангстрем = 1 м
1 ч = 3600 с
1 год = 8760 ч

Кратные и дольные приставки

Тера = 1000 гига
Гига = 1000 мега
Мега = 1000 кило
кило = 1000
1 = 100 санти
1 = 1000 милли

Кодовая таблица ASCII

В табл. П2.4 приведены ASCII-коды символов.

Таблица П2.4. Кодовая таблица ASCII

Биты 6—4		Управляющие символы		Печатные символы				
Биты 3-0	000	001	010	011	100	101	110	111
0000	NUL	DLE	Пробел	0	@	P	*	p
0001	SOH	DC1	!	1	A	O	o	q
0010	STX	DC2	•	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EVT	DC4	\$	4	D	T	d	t
0101	ENO	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	^	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	•	<	L	\	l	
1101	CR	CS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

ПРИЛОЖЕНИЕ 3

Сравнительная характеристика микроконтроллеров

В табл. ПЗ.1 приведены основные сведения о микроконтроллерах, чаще всего используемых в автоматических устройствах.

Таблица ПЗ. 1. Основные параметры микроконтроллеров

Микро-контроллер	Кол-во выводов	Объем памяти программ, инструкций	Размер памяти данных, байт	Среда разработки	Программатор	Комментарии
Microchip PIC12C5xx	6	512 байт – 1 Кб	25-41	Microchip MPLAB	Последовательный ICSP (может быть встроенным)	Используется для периферийных контроллеров. Имеет встроенный генератор 4 МГц
PIC16C505	12	512 байт	72	Microchip MPLAB	ICSP	Для периферийных контроллеров или небольших автоматических устройств. Имеет встроенный генератор 4 МГц
PIC16HV540	12	512 байт	25	Microchip MPLAB	Параллельный	Имеет встроенный регулятор напряжения
PIC16C62x	13	512 байт – 2 Кб	80-96	Microchip MPLAB	ICSP	Имеет встроенный компаратор напряжения
PIC16F628	16	2 Кб	224	Microchip MPLAB	ICSP	Имеет встроенный генератор 4 МГц
PIC16F84(AI	13	1 Кб	68	Microchip MPLAB	ICSP	Имеет Flash-память
PIC16C71x	13	512 байт – 2 Кб	36-128	Microchip MPLAB	ICSP	Содержит четыре АЦП
PICJ6F87x	22, 33	8 Кб	368	Microchip MPLAB	ICSP	Имеет встроенный АЦП, Flash-память, отладчик (ICD)
PIC17C4x	33	8 Кб	232-454	Microchip MPLAB	Параллельный	Может работать с внешней 8/16-разрядной шиной
PIC18Fxx2	23-68	16 Кб	768-3840	Microchip MPLAB	ICSP	Встроенный АЦП, Flash-память, и внутренний аппаратный отладчик (IC)
Parallax BASIC Stamp 1	8	256 байт	14	Parallax stampw	Параллельный порт	Flash-память программ, PBASIC
BASIC Stamp 2	16 + 2 I/O	2 Кб	26	Parallax stampw	RS-232	Flash-память программ, PBASIC
BASIC Stamp 2e	16 + 2 I/O	16 Кб	26 + 63	Parallax stampw	RS-232	Расширенный PBASIC
BASIC Stamp 2sx	16 + 2 I/O	16 Кб	26 + 63	Parallax stampw	RS-232	Расширенный PBASIC/высокоскоростной микроконтроллер

Таблица ПЗ. 1. Основные параметры микроконтроллеров (окончание)

Микро- контроллер	Кол-во выводов	Объем памяти программ, инструкций	Размер памяти данных, байт	Среда разработки	Программатор	Комментарии
BASIC Stamp 2p	16/30 + I/O	16 Кб	26 + 63	Parallax stampw	RS-232	Расширенный PBASIC
Motorola 68HC08	13-51	0-32Кб	128 - 1 Кб			Множество дополнительных функций. Необязательная внешняя программная память
Motorola 68HC11	13-50	0-32Кб	64 - 2Кб			Много дополнительных функций. Необязательная внешняя программная память
Handy (на основе 68HC11)	20	32 Кб	32 Кб		RS-232	Специально для разработки роботов
OOPIC2	40	95 объектов	172	OOPic	I2C	Имеет различные интерфейсы
Atmel AT90S1200 (AVR)	15	1 Кб	64	AVR Studio	Последовательный или параллельный	Для подсистем ввода-вывода
AT90S8515 (AVR)	32	8 Кб	12	AVR Studio	Параллельный	Имеет компараторы и Flash-память
AT90S8535 (AVR)	32	8 Кб	512	AVR Studio	Параллельный	Имеет АЦП и Flash-память
Intel 87C51 и др.	32	4 Кб	128			Микроконтроллер i8051 с СППЗУ, последовательный порт, возможность доступа к внешней 8-разрядной памяти
87C52	32	8 Кб	256			Микроконтроллер i8052 с СППЗУ, последовательный порт, возможность доступа к внешней 8-разрядной памяти
Atmel AT89Cх051 (8051)	15	1-4 Кб	128			Микроконтроллер 8051 с встроенным компаратором напряжения и USART
Kg Systems BASIC-Tiger	38	128 Кб	128 Кб	BASIC-Tiger	Последовательный	Аналоговый ввод-вывод, I2C, последовательный интерфейс, ЖКИ. Поддержка многозадачности
PC/104					Последовательный, Flash, СППЗУ	Шина ввода-вывода, совместимая с ПЭВМ

ПРИЛОЖЕНИЕ 4

Справочное руководство по языку PICC Lite

Представление констант

В табл. П4.1 приведены обозначения констант, представленных в различных системах счисления.

Таблица П4. /. Представление констант в различных системах счисления

Основание системы счисления	Формат PICC Lite	Формат MPASM
2	0bчисло или 0Вчисло	В'число'
8	Очисло или \число	О'число'
10	число	число или D'число' или . 'число'
16	Охчисло или ОХчисло	Н'число' или Охчисло

Комментарии

Два способа выделения комментариев:

1. Двойной слэш:

// Все символы до конца строки считаются комментарием.

2. Слэш со звездочкой:

/* Все символы здесь считаются комментарием. */

Объявления переменных и констант

Объявление переменной или константы представлено в следующем формате:

[квалификатор] тип [* [квалификатор]] метка [= значение];

В квадратных скобках, как обычно, приведены необязательные элементы. В табл. П4.2 перечислены допустимые типы данных.

Таблица П4.2. Типы данных в PICC Lite

Тип	Разрядность, бит	Комментарии
bit	1	Логическое значение
char	8	Целое со знаком
unsigned char	8	Целое без знака
short	16	Целое со знаком
unsigned short	16	Целое без знака
int	16	Целое со знаком
unsigned int	16	Целое без знака
long	32	Целое со знаком
unsigned long	32	Целое без знака
float	24	Вещественное с плавающей точкой
double	24 (по умолчанию)	Вещественное с плавающей точкой (разрядность можно увеличить до 32 с помощью опции компилятора -D32J)

Для явного приведения типа используется конструкция вида

```
(unsigned long) StringAddr = 0x012345678;
```

Описание массива переменных:

```
тип ИмяМассива [КоличествоЭлементов] [= {значение1, значение2, ...}]
```

Строковые переменные объявляются как массивы символов и хранятся в формате ASCII:

```
char ИмяСтроки [МаксимальноеЧислоСимволов] = "Это строка";
```

Последний символ NUL с кодом 0x00 при объявлении строки вставляется автоматически, но программисту приходится помнить о нем, если создание и обработка строк происходят в самой программе.

Для обращения к отдельному символу строковой переменной предусмотрена та же нотация, что и для элементов массива:

```
b = String[n];                    // Выбрали n-й символ строки.
```

Для объявления указателя используется символ * (звездочка):

```
char » String = "Это строка";
```

В микроконтроллере PIC16F84 переменные могут располагаться в первом банке памяти по адресам с 0x0С до 0x4F, что соответствует абсолютным адресам с 0x08С до 0x0CF; они могут быть отображены в нулевой банк памяти. В микроконтроллере PIC16F627 переменные хранятся как в нулевом, так и в первом банке памяти; отображение адресов между банками невозможно.

В языке PICC Lite, в отличие от стандартного C, нельзя использовать символ & для того, чтобы присвоить указателю адрес другой переменной.

Обращение к конкретным адресам регистров в языке PICC Lite представлено в следующем формате:

```
static unsigned char ИмяРегистра @ 0xАдресРегистра;
```


Для обращения к отдельным битам регистра служит запись вида

```
static bit ИмяБита @ (unsigned) &ИмяРегистра • 8 + НомерБита;
```

Если подключить к программе заголовочный файл `pic.h`, то можно использовать стандартные имена регистров и битов.

Чтобы сообщить компилятору, что другая программа (скажем, процедура обработки прерываний) может в любой момент изменить значение некоторой переменной, при ее описании надо указать квалификатор `volatile`, например:

```
volatile unsigned ИмяРегистра @ 0xАдресРегистра;
```

Для описания констант используется квалификатор `const`, например:

```
const int Label = Значение;
```

В результате переменная `Label1` будет размещена компилятором в ПЗУ, и ее значение нельзя будет изменить.

Объявление вида

```
const char * ptr;
```

создает указатель `ptr` на символ. Значение указателя может быть изменено, но сам символ - нет.

Наоборот, если записать

```
char » const ptr;
```

то можно будет изменить значение символьной переменной, но не указателя.

При использовании микроконтроллера PIC16F627 все объявленные в программе переменные размещаются в нулевом банке памяти, но можно использовать квалификатор `bank1`, если требуется хранить переменную в первом банке. Эта возможность недоступна для микроконтроллера PIC16F84.

Структуры и объединения

Формат описания структуры или объединения:

```
struct | union {
    type ИмяПоля1;
    type ИмяПоля2;
    ...
} ИмяОбъединения [ИмяПеременной];
```

Для доступа к полям структуры используется оператор `->`. Для примера рассмотрим следующее определение:

```
struct <
    intdividend;           // Частное.
    intremainder;         // Остаток.
} div_result result;      // Результат деления.
```

Тогда можно записать:

```
result -> dividend = x / y;
result -> remainder = x % y;
```

Приведем пример описания битовой структуры:

```
struct {
    unsigned : 3;           // Старшие три бита не будут использоваться.
    PageSel   : 2;           // Выбор номера страницы.
    OffSel    : 3;           // Смещение внутри страницы.
} PCLATH 9 0x00A;
```

Функции

Главная программа оформляется следующим образом:

```
void main(void) {
    ...                // Текст программы.
>                    // Конец программы.
```

Формат описания функции:

```
ТипРезультата ИмяФункции ( [ Тип1 Параметр1 [, Тип2 Параметр2,...] ] ) {
    ...                // Текст функции.
    return value;       // Возвращаемое значение.
}                      // Конец функции.
```

Для описания обработчика прерываний в заголовке функции надо указать модификатор **interrupt**:

```
void interrupt ИмяФункции (void) {
    ...
}
```

Операторы

В.общем виде формат оператора языка PICC Lite выглядит следующим образом:

```
[ (..) Переменная \ Константа [ Операция [ (..) Переменная \ Константа ] [ ].. ] ]
```

Оператор присваивания значения переменной:

Переменная - *Выражение*;

К управляющим конструкциям относятся следующие: if, while, do, for, switch.

Формат оператора if:

```
if ( Выражение )
    [ Оператор ]; | { Оператор; ... }
[ else [ Оператор ]; | { Оператор; ... } ]
```

Формат цикла while:

```
while ( Выражение ) [ Оператор ]; | { Оператор; ... }
```

Цикл `for` описывается так:

```
for ( [ Инициализация;                // Инициализация параметра цикла.
      [ Условие [, Условие ]];        // Условие продолжения цикла.
      [ Выражение [, Выражение,...]] ) // Изменение параметра цикла.
      Оператор \ { Оператор; ... }    // Тело цикла.
```

Для немедленного выхода из цикла можно использовать оператор `break`.

Команда `continue` пропускает ту часть тела цикла, которая расположена после нее, и переходит к следующему шагу. Эту команду можно указывать в циклах `for`, `while` и `do/while`.

Формат цикла с постусловием (он выполняется, пока условие истинно):

```
do Оператор; [ { Оператор; ... } while ( Условие );
```

Оператор выбора:

```
switch( Выражение ) {
    case Значение1: [ Оператор; ... ] [ break; ]
    case Значение2: [ Оператор; ... ] [ break; ]
    ...
    default: [ Оператор; ... ]
}
```

Инструкция

```
goto Метка
```

используется для перехода на указанный оператор (метка не может быть расположена вне тела функции). Формат описания метки:

```
Метка: Оператор;
```

Для возвращения результата выполнения функции используется оператор `return`:

```
return Выражение;
```

Операции

В табл. П4.3 приведены одноместные (унарные), в табл. П4.4 - двуместные (бинарные), а в табл. П4.5 - составные операции.

Таблица П4.3. Унарные операции

Обозначение	Операция
~	Поразрядная инверсия
++	Инкремент (увеличение на 1)
-	Изменение знака числа
--	Декремент (уменьшение на 1)

Таблица П4.4. Бинарные операции

Обозначение	Операция
!	Логическое отрицание
&&	Логическое И (AND)
&	Поразрядное И (AND) или ссылка
	Логическое ИЛИ (OR)
	Поразрядное ИЛИ (OR)
^	Поразрядное Исключающее ИЛИ (XOR)
+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Остаток от деления
==	Сравнение на равенство
!=	Сравнение на неравенство
<	Меньше
>	Больше
<=	Меньше или равно
>=	Больше или равно
<<	Сдвиг влево
>>	Сдвиг вправо

Таблица П4.5. Составные операции

Символ	КодASCII	Описание
\r	0x00D	Возврат каретки (CR)
\n	0x0 0A	Перевод строки (LF)
\f	0x00C	Перевод страницы (FF)
\b	0x008	Забой (BS)
\t	0x009	Горизонтальная табуляция (HT)
\v	0x00B	Вертикальная табуляция (VT)
\a	0x007	Звонок (BEL)
\'	0x027	Апостроф (')
\"	0x022	Двойная кавычка (")
\\	0x05C	Обратный слэш (\)
\ddd	—	Восьмеричный код
\xdd	0xOdd	Шестнадцатеричный код

Зарезервированные слова

Ниже приведен список зарезервированных слов, которые не могут быть использованы в качестве имен переменных, констант или меток:

auto	if
bank1	int
break	interrupt
case	persistent
char	return
const	signed
continue	static
default	struct
do	switch
else	union
extern	unsigned
fastcall	void
for	volatile
goto	while

Управляющие символы

В табл. П4.6 приведены специальные символы для управления экраном и положением курсора.

Таблица П4.6. Управляющие символы

Символ	Код ASCII	Описание
\r	0x00D	Возврат каретки (CR)
\n	0x00A	Перевод строки (LF)
\f	0x00C	Перевод страницы (FF)
\b	0x008	Забой (BS)
\t	0x009	Горизонтальная табуляция (HT)
\v	0x00B	Вертикальная табуляция (VT)
\a	0x007	Звонок (BEL)
\'	0x027	Апостроф (')
\"	0x022	Двойная кавычка (")
\\	0x05C	Обратный слэш (\)
\ddd	—	Восьмеричный код
\xdd	0xOdd	Шестнадцатеричный код

Директивы компилятора

В табл. П4.7 приведены директивы управления компиляцией. Все директивы начинаются с символа # и выполняются при компиляции программы.

Таблица П4.7. Директивы компилятора

Директива	Функция
#	Пустая директива
#asm . . . #endasm	Ассемблерная вставка
#assert Условие	Генерация ошибки, если условие ложно
#define Имя [(Параметры)] Текст	Определение идентификатора (имени), каждое вхождение которого в текст программы компилятор заменит указанным текстом
#include "Файл" <Файл>	Включение в текст программы содержимого указанного файла. Если имя файла приведено в кавычках, то он ищется в текущем каталоге и во всех каталогах, которые указаны в переменной окружения PATH, а если в угловых скобках, то файл ищется в каталогах, указанных в переменной окружения INCLUDE
#error Сообщение	Генерация ошибки времени компиляции с выводом на экран указанного сообщения
#if Условие #elif, #else, или #endif.	Если условие истинно, компилируется код, следующий до директивы. Если условие ложно, код до #elif, #else или #endif игнорируется
#ifdef Имя #endif,	Если указанное имя определено в программе с помощью директивы #define, то код, следующий до директивы #elif, #else или компилируется
#ifndef Имя #else	Если указанное имя не определено в программе с помощью директивы #define, то код, следующий до директивы #elif, или #endif, компилируется
#elif Условие	Эта директива работает как #else #if и позволяет избежать вложенных директив #if
#else	Используется совместно с директивами #if или #elif
#endif #ifdef	Используется для завершения директив #if, #elif, #else, или #ifndef
#line Номер ИмяФайла	Определяет номер строки и имя файла листинга
#number	Определяет номер строки файла листинга
«pragma interrupt_level 1	Следующая функция обработки прерываний будет вызывать другую функцию. Позволяет подавить сообщение об ошибке, которое в этом случае генерируется по умолчанию
#pragma jis	Разрешает строки JIS (двухбайтовый национальный набор символов)
#pragma nojis	Запрещает строки JIS
#pragma printf_check тип	Определяет, что следующая функция должна принимать строковые данные в формате printf
#pragma psect ИмяСегмента = Имя	Определяет, что данные будут помещены в определенный сегмент памяти
#pragma regused	
Регистр ...	Определяет регистры, которые будут сохранены в процедуре обработки прерывания
#undef Имя	Отменяет определение указанного имени
#warning Сообщение	Выводит предупреждающее сообщение во время компиляции

Параметры командной строки

В табл. П4.8 приведены параметры, которые можно указывать в командной строке при вызове компилятора PICC Lite.

Таблица П4.8. Параметры командной строки компилятора

Параметр	Значение
-processor	Определяет тип процессора (можно указывать только 16C84, 16F84, 16F84A или J6F627)
-A-опция	Передаёт параметр -опция ассемблеру
-AAHEX	Генерирует HEX-файл в символьном формате American Automation
-ASMLIST	Генерирует файл .LST
-BIN	Генерируется двоичный выходной файл
-C	Компиляция будет закончена созданием объектных файлов
-CRфайл	Генерирует файл перекрестных ссылок
-D24	Использует усеченный, 24-разрядный формат вещественных чисел
-D32	Использует полный (стандарт IEEE 754), 32-разрядный формат вещественных чисел
-Dмакрос	Определяет макрокоманду препроцессора (как директива #define)
-E	Определяет расширенный (символьный) формат для ошибок компилятора
-Eфайл	Перенаправляет сообщения об ошибках компилятора в указанный файл
-E+файл	Сообщения об ошибках дописываются в конец существующего файла
-FAKELOCAL	Задаёт формат MPLAB для отладочной информации
-FDOUBLE	Разрешает использовать функции специальной библиотеки для работы с 32-разрядными вещественными числами
-Gфайл	Генерирует расширенную таблицу имен
-HELP	Выводит подсказку по параметрам командной строки
-ICD	Генерирует код для внутрисхемного отладчика
-И путь	Определяет путь к каталогу с подключаемыми файлами
-INTEL	Генерирует выходной файл в шестнадцатеричном формате Intel (по умолчанию)
-Lбиблиотека	Определяет библиотеку, которая будет использоваться компоновщиком
-L-опция	Определяет параметр -опция, который будет использоваться компоновщиком
-Mфайл	Запрашивает создание MAP файла
-MOT	Генерирует выходной файл в шестнадцатеричном формате Motorola S1/S9
-Nчисло	Определяет размер идентификаторов (по умолчанию — 31 символ)
-NORT	Запрещает компоновщику подключать к приложению стандартные модули времени выполнения
-O	Разрешает оптимизацию кода
-Oфайл	Определяет имя выходного файла
-P	Запускает препроцессор для обработки исходного текста программы
-PRE	Создаёт файлы с результатом работы препроцессора
-PROTO	Генерирует информацию о прототипах функций
-PSECTMAP	После компоновки показывает карту памяти
-Q	Определяет режим без вывода сообщений компиляции (данный параметр должен указываться первым)

Таблица П4.8. Параметры командной строки компилятора (окончание)

Параметр	Значение
-RESRAMдиапазон	Резервирует диапазон указанных адресов ОЗУ
-RESROMдиапазон	Резервирует диапазон указанных адресов ПЗУ
-ROMдиапазон	Определяет адресный диапазон для внешнего ПЗУ
-S	Останавливает компиляцию после получения исходных ассемблерных файлов
-SIGNED_CHAR	Определяет для символьных переменных формат «целое со знаком» (если при описании переменной явно не указан другой квалификатор)
-STRICT	Разрешает компилятору использовать строгие соглашения ANSI. В этом случае в программе нельзя использовать битовые переменные
-TEK	Генерирует выходной файл в шестнадцатеричном формате Tektronix
-Uимя	Отменяет определение указанного идентификатора
-UBROF	Генерирует выходной файл в формате UBROF
-V	Показывает в листинге параметры командной строки, заданные при вызове компилятора
-Wчисло	Устанавливает уровень предупреждающих сообщений компилятора
-X	Исключает локальные символы из таблицы имен
-Zчисло	Определяет уровень оптимизации (максимум 3)

Стандартные функции PICC Lite

В табл. П4.9 приведен список встроенных функций PICC Lite.

Таблица П4.9. Стандартные функции PICC Lite

Функция	Заголовочный файл	Комментарии
int abs(int)	stdlib.h	Вычисляет абсолютное значение числа
double acos(double)	math.h	Вычисляет арккосинус (ответ в радианах)
double asin(double)	math.h	Вычисляет арксинус (ответ в радианах)
double atan(double)	math.h	Вычисляет арктангенс (ответ в радианах)
double atan2(double, double)	math.h	Вычисляет арккотангенс (ответ в радианах)
double atof(const char *)	math.h	Преобразует строку ASCII в вещественное число
int atoi(const char *)	stdlib.h	Преобразует строку ASCII в целое число
int atol(const char *)	stdlib.h	Преобразует строку ASCII в длинное целое число
double ceil(double)	math.h	Округляет число с плавающей запятой до ближайшего целого (в большую сторону)
char *cgets(char *)	conio.h	Получает строку из стандартного потока ввода
double cos(double)	math.h	Вычисляет косинус угла (в радианах)
double cosh(double)	math.h	Вычисляет гиперболический косинус
void cputs(char *)	conio.h	Выводит строку символов на экран
void eeprom_write(unsigned char, unsigned char)	pic1684.h	Записывает байт в СППЗУ
unsigned char eeprom_read(unsigned char)	pic1684.h	Считывает байт из СППЗУ

Таблица П4.9. Стандартные функции PICC Lite (продолжение)

Функция	Заголовочный файл	Комментарии
double exp(double)	math.h	Вычисляет экспоненту
double fabs(double)	math.h	Вычисляет абсолютное значение числа
double floor(double)	math.h	Округляет число с плавающей запятой до ближайшего целого (в меньшую сторону)
double frexp(double, int *)	math.h	Вычисляет целую степень вещественного числа
char getch(void)	conio.h	Получает один символ из стандарта потока ввода. Если входной поток пуст, ожидает появления символа
char getche(void)	conio.h	Получает двухбайтовый символ из стандартного потока ввода (для получения старшего байта требуется повторный вызов этой функции)
void init_uart(void)	conio.h	Инициализирует UART
Bit isalnum(char)	ctype.h	Возвращает TRUE, если символ является буквенно-цифровым
Bit isalpha(char)	ctype.h	Возвращает TRUE, если символ является буквой
bit iscntrl(char)	ctype.h	Возвращает TRUE, если символ управляющий (0x000 - 0x01F или 0x07F)
bit isdigit(char)	ctype.h	Возвращает TRUE, если символ является цифрой
bit isgraph(char)	ctype.h	Возвращает TRUE, если символ является печатным, но не пробелом (то есть его код лежит в диапазоне 0x021 - 0x07E)
bit islower(char)	ctype.h	Возвращает TRUE, если символ — буква нижнего регистра
bit ispunct(char)	ctype.h	Возвращает TRUE, если символ является знаком пунктуации
bit isprint(char)	ctype.h	Возвращает TRUE, если символ является печатным (0x020 - 0x07E)
bit isspace(char)	ctype.h	Возвращает TRUE, если код символа лежит в диапазоне 0x09 - 0x0D или является пробелом
bit isupper(char)	ctype.h	Возвращает TRUE, если символ — буква верхнего регистра
bit isxdigit(char)	ctype.h	Возвращает TRUE, если символ - шестнадцатеричная цифра
char kbhit(void)	conio.h	Возвращает код нажатой клавиши
double ldexp(double, int)	math.h	Вычисляет целую степень целого числа
double log(double)	math.h	Вычисляет натуральный логарифм
double log10(double)	math.h	Вычисляет десятичный логарифм
const void* memchr(const void *, int, unsigned int)	string.h	Ищет первое вхождение символа
int memcmp(const void*, const void *, unsigned int)	string.h	Сравнивает две строки
void *memcpy(void *, const void *, unsigned int)	string.h	Копирует заданное число байтов
void *memmove(void *, const void *, unsigned int)	string.h	Копирует заданное число байтов (диапазон адресов источника и адресата могут перекрываться)
void *memset(void *, int, unsigned int)	string.h	Заполняет заданное число ячеек памяти одним и тем же символом

Таблица П4.9. Стандартные функции PICC Lite (продолжение)

Функция	Заголовочный файл	Комментарии
<code>double modf</code> (<code>double</code> , <code>double *</code>)	<code>math.h</code>	Разбивает число с плавающей точкой на целую и дробную части
<code>double pow</code> (<code>double</code> , <code>double</code>)	<code>math.h</code>	Возводит число в степень
<code>int printf</code> (<code>const char *</code> , ...)2	<code>stdio.h</code>	Выводит текстовые строки на экран. Поддерживает стандартные символы форматирования: %d, %i (десятичное число); %o (восьмеричное число); %x, %X (шестнадцатеричное число в нижнем или верхнем регистре); %i (целое число без знака); %c (текстовый символ в коде ASCII); %s (строка ASCII); %f (число с плавающей точкой); %e, %E (число с плавающей точкой, вместо символа # указывается точность); %g, %G (число с плавающей точкой, формат выбирается в зависимости от значения); %p (указатель); %% (символ %)
<code>void putchar</code> (<code>char</code>)	<code>conio.h</code>	Выводит один символ в стандартный поток вывода
<code>int rand</code> (<code>void</code>)	<code>stdlib.h</code>	Возвращает случайное число - см. <code>srand</code>
<code>double sin</code> (<code>double</code>)	<code>math.h</code>	Вычисляет синус угла (в радианах)
<code>double sinh</code> (<code>double</code>)	<code>math.h</code>	Вычисляет гиперболический синус
<code>double sqrt</code> (<code>double</code>)	<code>math.h</code>	Вычисляет квадратный корень числа
<code>void srand</code> (<code>void</code>)	<code>stdlib.h</code>	Инициализирует генератор случайных чисел - см. <code>rand</code>
<code>char* strcat</code> (<code>char *</code> , <code>const char *</code>)	<code>string.h</code>	Конкатенация строк ASCII
<code>const char* strchr</code> (<code>const char *</code> , <code>int</code>)	<code>string.h</code>	Находит заданный символ в строке ASCII
<code>int strcmp</code> (<code>const char *</code> , <code>const char *</code>)	<code>string.h</code>	Сравнивает две строки ASCII
<code>char *strcpy</code> (<code>char *</code> , <code>const char *</code>)	<code>string.h</code>	Копирует строку ASCII
<code>unsigned int strcspn</code> (<code>const char *</code> , <code>const char *</code>)	<code>string.h</code>	Возвращает число символов, которые входят во вторую строку, но отсутствуют в первой
<code>char *strdup</code> (<code>const char *</code>)	<code>string.h</code>	Дублирует строку
<code>int stricmp</code> (<code>const char *</code> , <code>const char *</code>)	<code>string.h</code>	Сравнивает строки (без учета регистра символов)
<code>const char *stristr</code> (<code>const char *</code> , <code>const char *</code>)	<code>string.h</code>	Возвращает указатель на вхождение второй строки в первую (без учета регистра символов)
<code>char *strncat</code> (<code>char *</code> , <code>const char *</code> , <code>unsigned int</code>)	<code>string.h</code>	Добавляет в конец первой строки ASCII заданное количество символов второй
<code>int strncmp</code> (<code>const char *</code> , <code>const char *</code> , <code>unsigned int</code>)	<code>string.h</code>	Сравнивает заданное количество символов двух строк ASCII
<code>int strnicmp</code> (<code>const char *</code> , <code>const char *</code> , <code>unsigned int</code>)	<code>string.h</code>	Сравнивает заданное количество символов двух строк ASCII (без учета регистра символов)
<code>unsigned int strlen</code> (<code>const char *</code>)	<code>string.h</code>	Возвращает длину строки
<code>char *strncpy</code> (<code>char *</code> , <code>const char *</code> , <code>unsigned int</code>)	<code>string.h</code>	Копирует заданное число символов из второй строки в первую
<code>const char* strpbrk</code> (<code>const char *</code> , <code>const char *</code>)	<code>string.h</code>	Возвращает первое вхождение в первую строку любого символа второй строки

Таблица П4.9. Стандартные функции PICC Lite (окончание)

Функция	Заголовочный файл	Комментарии
<code>const char* strchr (const char *, int)</code>	<code>string.h</code>	Возвращает позицию последнего символа в строке ASCIIZ
<code>unsigned int strspn (const char *, const char *)</code>	<code>string.h</code>	Сравнивает две строки и возвращает номер позиции первого несовпадающего символа
<code>const char *strstr (const char *, const char *, const char *)</code>	<code>string.h</code>	Возвращает указатель на вхождение второй строки в первую
<code>double tan(double)</code>	<code>math.h</code>	Вычисляет тангенс угла (в радианах)
<code>double tanh(double)</code>	<code>math.h</code>	Вычисляет гиперболический тангенс
<code>char tolower(char)</code>	<code>ctype.h</code>	Преобразует символ к нижнему регистру
<code>char toupper(char)</code>	<code>ctype.h</code>	Преобразует символ к верхнему регистру
<code>unsigned int xtoi (const char *)</code>	<code>stdlib.h</code>	Преобразует строку ASCII в шестнадцатеричное число

ПРИЛОЖЕНИЕ 5

Информация об авторе

Свои пожелания отправляйте по электронному адресу myke@passport.ca. Вы также можете посетить сайт автора в сети Internet (www.myke.com).

Microchip

Microchip Technology, Inc.

www.microchip.com

На сайте находится полная документация в формате PDF и последние версии MPLAB.

Hi-Tech Software

Hi-Tech Software

www.htsoft.com

На сайте представлена информация о последних версиях компиляторов для различных микроконтроллеров и микропроцессоров (в том числе и о компиляторе PICC Lite).

Компиляторы для микроконтроллеров PICmicro

Custom Computer Services, Inc.

www.ccsinfo.com

ccs@ccsinfo.com

PicBasic и PicBasic Pro

MicroEngineering Labs, Inc.

www.melabs.com

info@melabs.com

Компилятор CC5X

B Knudsen Data
www.bknd.com
sales@bknd.com

ЭЛЕКТРОННЫЕ РЕСУРСЫ ПО РОБОТОТЕХНИКЕ

Seattle Robotics Society

www.seattlerobotics.org

Один из наиболее известных сайтов в Internet.

Silicon Valley HomeBrew Robotics Club

www.wildrice.com/HBRobotics/HBRCBuildersBook.html

Превосходное введение в робототехнику. Представлено большое количество инструкций по проектированию роботов и ссылок на соответствующие ресурсы.

robots.net

<http://robots.net>

Хороший сайт. Имеются ссылки на литературу, список разработок.

The Robot Menu

www.robotics.com/robomenu/index.html

Сайт посвящен человекоподобным роботам.

Robotics FAQ List

www.frc.ri.cmu.edu/robotics-faq

Охватывает практически все, чем занимается робототехника.

Innovatus

www.innovatus.com

Полезное программное обеспечение для разработчика автоматических устройств.

RoboClub

www.roboclub.ru

Практическая робототехника (на русском языке). Роботы домашние, военные, боевые, мобильные; автономные устройства, конструкции, компоненты, схемы, датчики, сенсоры, детекторы, исполнительные устройства, проекты, поведение, обучение, искусственный интеллект, алгоритмы, программирование, руководства, документация, покупка и продажа комплектующих, вопросы, советы, обзоры, соревнования. Каталог ссылок на ресурсы Internet по робототехнике можно найти на страничке <http://links.roboclub.ru>.

Internet-робототехника

www.applmat.ru/pages/i-robotics/rus/rhome.html.

Исследования по Internet-робототехнике были начаты в секторе робототехники ИПМ им. М. В. Келдыша РАН и в Группе компьютерной графики университета Де Монтфорт (Милтон Кейнс, Англия) как продолжение совместного проекта INTAS по космической робототехнике. Практическую цель представляла разработка системы управления через Internet роботом-манипулятором РМ-01 (PUMA 560).

Периодические издания

Circuit Cellar Ink: www.circellar.com

Microcontroller Journal: www.mcjournal.com

Nuts and Volts: www.nutsvolts.com

Everyday Practical Electronics: www.epemag.wimborne.CQuk

Поставщики комплектующих

ООО «МИКРО-ЧИП»

sales@microchip.ru

www.microchip.ru. www.imicro.ru

Тел./факс: (095) 963-96-01

На сайте www.microchip.ru имеется документация по микроконтроллерам PICmicro в формате PDF на русском языке.

Консультационно-технический центр по микроконтроллерам (КТЦ-МК)

www.cec-mc.ru. www.gaw.ru

Тел.: (095) 973-18-55, 942-34-16

Факс: (095) 973-18-64

Магазин «ЧИП и ДИП»

sales@chip-dip.ru

www.chip-dip.ru

Москва, ул. Гиляровского, д. 39

Тел.: (095) 284-56-78, 281-99-17

Факс: (095) 971-31-45

НПО «СИММЕТРОН»

npo@symmetron.ru

www.symmetron.ru

Москва, ул. 8 Марта, д. 8

Тел.: (095) 214-25-55, 212-33-08

Санкт-Петербург, ул. Таллинская, д. 7

Тел.: (812) 278-84-84, 444-02-68

Сведения о других поставщиках вы можете найти в Internet по адресам ww.catalog.gaw.ru. www.einfo.ru. www.efind.ru. www.chipinfo.ru.

СПИСОК ЛИТЕРАТУРЫ

1. Предко М. Руководство по микроконтроллерам. В 2-х тт. - М.: Постмаркет, 2001.
2. Предко М. Справочник по PIC-микроконтроллерам. — М.: ДМК Пресс, 2002.
3. Тавернье К. PIC-микроконтроллеры. Практика применения. - М.: ДМК Пресс, 2002.
4. Миль Г. Модели с дистанционным управлением. - Л.: Судостроение, 1984.
5. Мацкевич В. В. Занимательная анатомия роботов. Серия «Научно-популярная библиотека школьника». - М.: Радио и связь, 1988.
6. Хейзерман Д. Как самому сделать робота. - М.: Мир, 1979.
7. Янг Дж. Ф. Робототехника. - Л.: Машиностроение, 1979.
8. Красковский Е. Я., Дружинин Ю. А., Филатов Е. М. Расчет и конструирование механизмов приборов и вычислительных систем. Учебное пособие для приборостроительных специальностей вузов. — М.: Высшая школа, 1991.
9. Механика промышленных роботов. Учебное пособие для вузов: в 3 кн. / Под ред. К. В. Фролова, Е. И. Воробьева. - М.: Высшая школа, 1988.
10. Смольников Б. А. Проблемы механики и оптимизации роботов. - М.: Наука, 1991.
11. Попов Е. П., Ющенко А. С. Роботы и человек. - М.: Наука, 1984.
12. Юревич Е. И. Основы робототехники. — Л.: Машиностроение, 1985.
13. Новиков Ю. В., Калашников О. А., Гуляев С. Э. Разработка устройств сопряжения для персонального компьютера типа IBM PC. Практик. пособие. - М.: ЭКОМ, 1997.
14. Белоусов И. Р. Управление роботами через сеть Интернет / Новое в управлении и автоматике. - М.: Наука, 2002.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

А

Автомат конечный 336
Алгоритм генетический 352
Анализ ошибок 363
Арбитраж целевых установок 344
Архитектура
 гарвардская 57
 принстонская 57
Ассемблер 40
АЦП 20, 46, 264
АЧХ 281

Б

Банк памяти 90
Бит четности 114
Булева алгебра 376

В

Ввод аналогового сигнала 264
Внутреннее сопротивление
 источника 22
Внутрисхемное
 программирование 57, 120
Выбор периферийных устройств 358
Выражение 47
Выход
 с открытым коллектором (ОК) 56
 с открытым стоком 91

Г

Гармоника 279
Гармонический анализ сигналов 280
Гидролокатор 252
Гистерезис 92
Глобальная переменная 136

Д

Дальномер ультразвуковой 252, 255
Датчик
 звуковой 279
 световой 263
Декодирование команд
 дистанционного управления 245
Декремент 41
Делитель напряжения 265
 балансировка 322
Детерминированное
 поведение 324
Децибел 281
Дисплей
 жидкокристаллический 34, 195
Дифференциальное включение
 фоторезисторов 266

З

Задание входных воздействий
 (стимулов) 66
Закон
 ассоциативный 376
 дистрибутивный 376
 коммутативный 376
 Ома 373

И

Излучатель
 пьезоэлектрический 191
Измерение
 расстояния 305
 сопротивления 272
Индикатор
 жидкокристаллический 195
 звуковой 190

Инкремент 41
Интерпретатор 37, 43
Искусственный интеллект 348
Испытания роботов 361

К

Квитирование 145
КМОП 26, 56
Код
 Manchester 238
 NRZ 56
Компаратор 117
 аналоговый 265
Компилятор 37, 46
Компиляция 40
Компоновщик 38, 50
Конечный автомат 132
Контакт
нормально замкнутый 19
Контроллер 18
 жидкокристаллического
 дисплея 195
 нечеткий 348
Контроллер прерываний 30
Конфигурационный регистр 87

М

Меандр 280
Методы принятия решений 324
Микроконтроллер 18
МК 18
Модели Tamiya 291
Модернизация устройств 364
Модификатор 136
Модуляция
 шиотно-импульсная 178
Монтаж
 накруткой 13
 печатный 13
Мост
 Н-образный 289
 Уитстона 374
Музыкальный звукояд 372
Мысленный эксперимент 364

Н

Нейрон 350
НЗК 19
Низковольтное
 программирование 88
Номеронабиратель
 тоновый 373

О

Обратная польская нотация 48
Обучение нейронной сети 352
Объектный файл 38
Одометр 305
Октава 372
Операционная система
 многозадачная 325
 реального времени 325
ОС 91
ОСРВ 325
Отладка 363
Отладчик внутрисхемный 62

П

Память 26
 динамическая 28
 статическая 28
ПЗУ 26, 337
ПИД-регулятор 307
ПО 36
Поведенческое
 программирование 343
Порт ввода-вывода 32
Постскалер 33
Прерывание 29
Прерывания по изменении уровня
 входного сигнала 278
Прерыватель оптический 305
Прескалер 33
Приемник команд
 дистанционного управления 248
Приоритет задачи 326
Программатор 9
 El Cheapo 38

Программирование нелинейное 324
 Программная модель 51
 Протокол синхронный 162

Р

Радиоуправление 310
 Разрешение коллизий на шине 166
 Распознавание звуковых команд 283
 Регенерация памяти 28
 Регистр
 CCP1CON 108
 EECON 103
 INTCON 90, 94
 OPTION 101
 STATUS 97
 общего назначения 87
 флагов 90
 Регистровые стимулы 69
 Регистровый файл 52
 Резонанс 375
 Резонатор
 кварцевый 25
 керамический 25, 83
 Рекурсия 62
 Релаксационный генератор 24
 Рефлексы 350
 Робот
 мигалка 171
 мышь 295, 343
 пищалка 191
 радиоуправляемый 310, 316
 реагирующий
 на звук 279, 285
 на присутствие человека 344
 на свет 263, 269, 273
 ребенок 343
 с дальномером 252, 257
 с дистанционным
 управлением 238, 242, 248
 с жидкокристаллическим
 индикатором 204
 с инфракрасным детектором
 столкновений 234

с механическими источниками
 входных сигналов 221
 с одометром 305
 с управляемыми
 двигателями 289, 295
 со световой индикацией 182
 таракан 330

РОН 87

Ряд Фурье 281

С

Сброс микроконтроллера 81
 Светодиодный индикатор
 с общим анодом 168
 с общим катодом 168
 Семафор 328
 Серводвигатель 311
 градуировка 322
 Сервопривод 310
 Сеть нейронная 351
 Симулятор 51
 Система навигации INS 308
 Системный ресурс 328
 Скорость звука 253
 Слово
 идентификации 122
 конфигурации 87, 122
 Совмещение функций периферийных
 устройств 246
 Сокет 150
 Сонар 252
 Стабилизация траектории 306
 Стартовый бит 114
 Стекло 47
 Стимул 65
 Стоповый бит 114

Т

Таймер 33
 PWRT 82
 сторожевой 97
 Тактирование 82

Теорема Моргана 376
Терминатор строки 212
Тестирование 361
Техническое задание 357
Токен 46
Трансмиссия 289
Трансформатор 375
Триггер Шмитта 92

У

УВВ 32, 52
Управление
 дистанционное 238, 340
Уровень
 биологический 10
 механический 10
 электронный 10
Уровни программирования 10
Усилитель
 операционный 281, 288

Ф

Файл стимулов 67, 261
ФАПЧ 56, 60, 84
ФВЧ 282
Физические константы 372
Фильтр
 активный 281, 283
 Баттерворта 281
 верхних частот 282
 нижних частот 281
 полосовой 282
 режекторный 282
Фильтрация сигнала 281
ФНЧ 281
Формат
 INXM8 38
 PDF 362
 S9 38
Функция повторно входимая 62

Ц

ЦАП 119
Цепочка
 RC 272, 375
 RL 375

Ч

Частота среза фильтра 281
Червяк в лабиринте 349

Ш

ШИМ 44, 107-109, 168, 178, 289

Э

Эмулятор 52
 внутрисхемный 142
 терминала 148

А

ANSI 148
ANSI C 46
API 26
ASCII 37, 196, 377
ASCIIZ 159, 212

В

BOD 81
BRG 112
Brown-out detection 81

С

CAN 34
CCP 107
Clock Stimulus 68
CMOS 56
CP 88
CPP 101, 108
CTS 145

D

DCD 145
DCE 143
DIP 138
DRAM 28
DSR 145
DTE 143
DTR 145

E

EEPROM 27
EPROM 26

F

Flash 27

G

GPR 87
GPS 213
GUI 162

H

Handshaking 145
HI-TECH 10

I

I2C 33, 34, 162
ICE 64, 142
ICSP 57, 120
IDE 10, 28, 53
Interrupt handler 29
IRF 31

L

LCD 34
Linker 38
LVP 88

M

Microwire 34
MPLAB 10

MPLAB-ICD Debugger 53
Multimastering 163

N

NRZ 34, 114, 143
NTSC 34

O

OTP 27

P

PBasic 46
PC 76
PDF 362
PIC 58
PICC Lite 10
PICmicro 58
PIE1 95
PIR1 95
PISTART Plus 63
PROM 26
PROMATE 64
PWM 107, 109, 178

R

RAM 26, 28
Register Stimulus 69
RI 145
ROM 26
RPN 48
RS-232 34, 143
RS-485 34
RTOS 10, 130, 325
RTS 145

S

SCL 163
SDA 163
Simulator 51

SPI 34
SRAM 28
Stimulus 65
Stimulus file 66

T

TOIE 95
TMRO 95, 98
TRIS 77, 91
TTY 148

U

USART 65, 112
USB 34

W

Warnings 51
WDT 87

Z

ZIF 128, 139