Ю. С. Магда

Микроконтроллеры PIC: архитектура и программирование

Москва, 2009

УДК 621.396.6
ББК 32.872
***

*** **Магда Ю. С.**

Микроконтроллеры PIC: архитектура и программирование. – М.: ДМК Пресс, 2009. – 240 с.: ил.

ISBN

В книге рассматривается широкий круг вопросов, связанных с практическим применением популярных 16-битных микроконтроллеров PIC24 в системах обработки данных и управления оборудованием. Приводятся многочисленные примеры программирования несложных аппаратно-программных систем обработки аналоговой и цифровой информации с применением периферийных модулей микроконтроллеров PIC24F. В контексте разработанных примеров приводятся необходимые сведения из теории, что способствует лучшему пониманию материала книги. Все приведенные в книге аппаратно-программные проекты разработаны и проверены на отладочном модуле Explorer 16 Development Board фирмы Microchip и могут служить основой для создания собственных проектов.

УДК 621.396.6
ББК 32.872

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

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

ISBN

© Магда Ю. С., 2009
© Оформление, ДМК Пресс, 2009

СОДЕРЖАНИЕ

Введение 5
Структура книги 6

1. ОБЗОР 16-БИТНЫХ PIC-МИКРОКОНТРОЛЛЕРОВ 8

2. АРХИТЕКТУРА МИКРОКОНТРОЛЛЕРОВ PIC24F 10

3. СИСТЕМА КОМАНД И ОСНОВЫ ПРОГРАММИРОВАНИЯ МИКРОКОНТРОЛЛЕРОВ PIC24F 18

3.1. Программная модель микроконтроллеров PIC24F 18
3.2. Режимы адресации и система команд 20
3.2.1. Команды перемещения и адресация данных 23
3.2.2. Команды сравнения/выбора и условного перехода 28
3.2.3. Команды работы с битами 32
3.2.4. Команды сдвига и циклического сдвига 35
3.2.5. Команды математических и логических операций 37
3.2.6. Команды условных/безусловных переходов 44
3.3. Разработка и отладка программ на ассемблере в среде MPLAB IDE 46
3.4. Особенности разработки и отладки программ на MPLAB C для PIC24 59

4. ПРОГРАММИРОВАНИЕ ПОРТОВ ВВОДА/ВЫВОДА 73

4.1. Аппаратно-программная архитектура портов ввода/вывода 73
4.2. Программирование портов ввода/вывода 77
4.3. Модуль регистрации событий 81

5. ПРОГРАММИРОВАНИЕ ПРЕРЫВАНИЙ 89

6. ПРОГРАММИРОВАНИЕ ТАЙМЕРОВ 100

6.1. Практическое использование 16-битных таймеров 104
6.2. Работа таймеров в 32-битном режиме 114
6.3. Часы реального времени 118

7. ИНТЕРФЕЙС SPI МИКРОКОНТРОЛЛЕРОВ PIC24F 120

7.1. Аппаратно-программная реализация SPI в микроконтроллерах PIC24F 121
7.2. Практическое программирование обмена данными по SPI 127

8. ИНТЕРФЕЙС I²C МИКРОКОНТРОЛЛЕРОВ PIC24F 140

8.1. Принципы функционирования интерфейса I²C 140
8.2. Модуль интерфейса I²C микроконтроллеров PIC24F 143
8.3. Практическое использование интерфейса I²C 147

9. ПРОГРАММИРОВАНИЕ ИНТЕРФЕЙСА PMP 159

9.1. Режимы работы PMP 161
9.2. Практические примеры программирования интерфейса PMP 164

10. ПОСЛЕДОВАТЕЛЬНЫЙ ИНТЕРФЕЙС МИКРОКОНТРОЛЛЕРОВ PIC24F 182

10.1. Аппаратно-программная архитектура UART 183
10.2. Практическое использование последовательного порта 184

11. ОБРАБОТКА АНАЛОГОВЫХ СИГНАЛОВ В МИКРОКОНТРОЛЛЕРАХ PIC24F 198

11.1. Программная модель интегрированного АЦП 201
11.2. Практическое использование модуля АЦП 205
11.3. Использование внешнего АЦП 217

12. ГЕНЕРАЦИЯ АНАЛОГОВЫХ И ЦИФРОВЫХ СИГНАЛОВ 221

12.1. Модуль генерации цифровых сигналов 221
12.2. Аналоговые компараторы в микроконтроллерах PIC24F 229

Заключение 239

ВВЕДЕНИЕ

Системы управления и контроля на однокристальных микроконтроллерах в настоящее время используются практически во всех сферах человеческой деятельности, причем каждый день появляются все новые и новые области применения таких систем. В последнее время, в связи с бурным развитием электроники расширились возможности и самих микроконтроллеров, позволяющих выполнять многие задачи, ранее недоступные для реализации, такие, например, как обработка и синтез аналоговых аудио- и видеосигналов. Одним из наиболее популярных семейств микроконтроллеров являются устройства, выпускаемые фирмой Microchip и известные под аббревиатурой PIC. В последние годы были разработаны и запущены в производство 16- и 32-битные модели, позволившие существенно расширить области применения систем на базе микроконтроллеров PIC. Для облегчения миграции от 8- к 16-битным устройствам фирма Microchip обеспечила максимальный уровень совместимости аппаратно-программных функций этих микроконтроллеров.

Эта книга посвящена практическим аспектам разработки систем на базе 16-битных микроконтроллеров PIC24F. В книге приводятся основные сведения по аппаратно-программной архитектуре микроконтроллеров PIC24F, а также многочисленные проекты систем обработки данных на базе микроконтроллеров этого семейства. Все примеры, приведенные в книге, разработаны и протестированы на плате «Explorer 16 Development Board» производства Microchip с установленным микроконтроллером PIC24FJ128GA010. Тем не менее, для тестирования приведенных примеров и разработки собственных проектов читатели могут использовать и более дешевую систему начального уровня «MPLAB Starter Kit for PIC24F» той же фирмы или отладочные платы других фирм. Для разработки программного обеспечения проектов, представленных в книге, использовалась интегрированная среда разработки MPLAB IDE версии 8.00 и бесплатная студенческая версия компилятора языка Си, известного под названием MPLAB C для PIC24. Кроме того, при изучении системы команд микроконтроллера PIC24F, а также при анализе процесса отладки некоторых программ на языке Си используется довольно эффективный симулятор/отладчик MPLAB SIM.

Книга рассчитана на широкий круг читателей — от начинающих до опытных разработчиков – и может оказаться полезной для всех, кто желает самостоятельно изучить аппаратно-программную архитектуру 16-битных микроконтроллеров PIC24F и применить эти знания на практике.

СТРУКТУРА КНИГИ

Структура книги рассчитана на последовательное изучение материала, хотя опытные разработчики систем на базе микроконтроллеров PIC могут изучать материал выборочно. Теоретический материал большинства глав подкреплен практическими примерами, что позволяет существенно облегчить изучение.

Книга состоит из 12 глав, краткие сведения о каждой из них приведены далее:

- Глава 1. «Обзор 16-битных PIC-микроконтроллеров». В этой главе рассматриваются общие характеристики 16-битных микроконтроллеров фирмы Microchip и дается краткий обзор инструментальных средств разработки программного обеспечения.
- Глава 2. «Архитектура микроконтроллеров PIC24F». Материал этой главы посвящен обзору аппаратной архитектуры микроконтроллеров PIC24F, организации памяти и функционирования периферийных модулей устройства.
- Глава 3. «Система команд и основы программирования микроконтроллеров PIC24F». В этой главе детально проанализированы принципы реализации системы команд микроконтроллеров PIC24F и приведены многочисленные примеры программного кода на языке ассемблера. Значительная часть главы посвящена вопросам разработки и отладки программного обеспечения в среде MPLAB IDE.
- Глава 4. «Программирование портов ввода/вывода». Эта глава содержит материал по архитектуре и программированию цифровых портов ввода/вывода микроконтроллеров PIC24F. Теоретический материал сопровождается примерами программ на языке Си с детальным анализом программного кода.
- Глава 5. «Программирование прерываний». Материал главы посвящен аппаратно-программной реализации системы прерываний микроконтроллеров PIC24F. На многочисленных примерах программного кода проанализированы различные механизмы вызова и обработки пользовательских прерываний.
- Глава 6. «Программирование таймеров». Глава содержит материал по аппаратно-программной архитектуре модулей таймеров микроконтроллеров PIC24F. Рассматриваются различные режимы работы таймеров и их настройка. Теоретический материал сопровождается примерами программ на языке Си с детальным анализом программного кода.
- Глава 7. «Интерфейс SPI микроконтроллеров PIC24F». В главе рассмотрены принципы функционирования и аппаратно-программная реализация модуля интерфейса SPI, а также приведены примеры программирования систем ввода/вывода дискретных данных с использованием данного интерфейса. Все примеры сопровождаются детальным анализом программного кода.
- Глава 8. «Интерфейс I^2C микроконтроллеров PIC24F». Материал главы посвящен рассмотрению принципов функционирования и аппаратно-программной конфигурации модуля интерфейса I^2C в микроконтроллерах PIC24F. Теоретический материал сопровождается примерами программирования обмена данными по шине I^2C на языке Си с детальным анализом программного кода.
- Глава 9. «Программирование интерфейса PMP». В главе рассматривается аппаратно-программная реализация 8-битного параллельного интерфейса обмена данными PMP. Приводятся примеры разработки систем ввода/вывода цифровых данных с использованием этого интерфейса.
- Глава 10. «Последовательный интерфейс микроконтроллеров PIC24F». В этой главе рассматриваются принципы реализации и настройки модуля последовательного интерфейса микроконтроллеров PIC24F. Теоретический материал сопровождается примерами программирования обмена данными с использованием этого модуля, разработанными на языке Си, с детальным анализом программного кода.
- Глава 11. «Обработка аналоговых сигналов в микроконтроллерах PIC24F». Эта глава посвящена методам обработки аналоговых сигналов в микроконтроллерах PIC24F. Здесь рассматривается широкий круг вопросов, связанных с настройкой и использованием модуля аналого-цифрового преобразователя, а также приводятся примеры программирования ввода данных посредством АЦП.
- Глава 12. «Генерация аналоговых и цифровых сигналов». В этой главе рассматривается широкий круг вопросов, связанных с генерацией цифровых и аналоговых сигналов в микроконтроллерах PIC24F. Теоретические аспекты иллюстрируются примерами программного кода на языке Си.

Автор благодарит коллектив издательства «ДМК Пресс» за помощь при подготовке книги к изданию. Особую признательность автор выражает своей жене Юлии за поддержку и помощь при написании книги.

ГЛАВА 1

ОБЗОР 16-БИТНЫХ PIC-МИКРОКОНТРОЛЛЕРОВ

Разработанные фирмой Microchip 16-битные микроконтроллеры являются очередным шагом на пути повышения производительности и эффективности встроенных и мобильных приложений. Эта 16-битная архитектура изначально разрабатывалась как альтернатива 8-битным решениям и призвана заменить в ближайшее время 8-битные микроконтроллеры в большинстве приложений.

Разработанная фирмой Microchip 16-битная платформа реализована в двух семействах 16-битных микроконтроллеров и в двух семействах цифровых сигнальных контроллеров. Все эти семейства объединяет ряд общих характеристик:

- совместимость по назначению выводов различных 16-битных устройств;
- возможность использования для всех устройств одних и тех же инструментальных средств разработки программного обеспечения;
- аппаратно-программная совместимость всех одноименных периферийных модулей микроконтроллеров;
- общая базовая система команд процессора, используемая во всех семействах.

Выбор той или иной модели микроконтроллера или сигнального контроллера зависит от требований к разрабатываемому приложению. Для большинства недорогих устройств средней производительности подходят микроконтроллеры PIC24F, максимальная производительность которых составляет 16 MIPS. Для устройств, требующих высокой производительности, можно использовать микроконтроллеры PIC24H с максимальным быстродействием 40 MIPS. Микроконтроллеры семейств PIC24F и PIC24H работают с одним и тем же набором инструкций процессора, включают одни и те же периферийные модули, имеют одну и ту же цоколевку, и для работы с ними используются одни и те же инструментальные средства для разработки программного обеспечения.

Если требуются дополнительные возможности по обработке сигналов, то вместо микроконтроллеров семейств PIC24F/H можно применить цифровые сигнальные контроллеры семейства dsPIC30F, которые могут помимо всего прочего работать при напряжении питания 5 В, или высокопроизводительные (40 MIPS) контроллеры dsPIC33F, которые имеют большой объем памяти и используют низковольтное (3.3 В) питание. В качестве инструментального средства разработки программного обеспечения 16-битных микроконтроллеров и цифровых сигнальных контроллеров используется свободно распространяемая интегрированная среда разработки (ИСР) MPLAB IDE фирмы Microchip, которая позволяет разрабатывать и отлаживать 8-, 16- и 32-битные приложения. Программа MPLAB IDE работает под управлением операционных систем Windows 2000/XP/Vista и позволяет выполнить все этапы разработки и отладки программного обеспечения для целевой системы. Среда MPLAB IDE позволяет выполнять тестирование и отладку программ с использованием мощного программного симулятора MPLAB SIM. Кроме того, для разработки программного обеспечения для 16-битных систем в среде MPLAB IDE можно использовать следующие инструментальные средства:

- ассемблер ASM30 — полнофункциональный макроассемблер, в котором можно создавать пользовательские макросы и использовать условное ассемблирование. Многочисленные директивы языка делают макроассемблер очень мощным средством разработки программ;
- компилятор программ, написанных на языке Си, который называется MPLAB C для PIC24. Этот компилятор используется для компиляции и оптимизации программ, написанных для 16-битных микроконтроллеров PIC24F/H и цифровых сигнальных контроллеров dsPIC30/33. Он совместим со стандартом ANSI C и включает полную библиотеку стандартных функций ANSI C, в числе которых функции манипулирования строками, функции работы с динамической памятью, функции преобразования даты/времени и математические функции. В компиляторе MPLAB C для PIC24 имеется мощный оптимизатор, позволяющий почти в 1,5 раза уменьшить размер программного кода по сравнению с компиляторами других фирм-производителей;
- визуальный генератор кода инициализации MPLAB VDI, позволяющий значительно упростить процесс создания инициализационного кода программы. С помощью VDI можно в графическом виде сконфигурировать устройство и по завершении вставить сгенерированный программный код инициализации в программу на языке Си или ассемблере;
- библиотеку периферийных модулей, включающую более чем 270 функций для работы с различными периферийными модулями;
- библиотеку математических функций, совместимую со стандартом IEEE-754, которая включает ряд функций для выполнения операций над обычными вещественными числами и вещественными числами с двойной точностью. Функции этой библиотеки могут использоваться как в программах на языке Си, так и на ассемблере.

Кроме инструментальных средств разработки и отладки программного обеспечения фирмы-производителя на рынке присутствуют и программные средства, выпускаемые многими известными фирмами (Hi-Tech, CCS и т.д.). Из аппаратных средств разработки наиболее известна и популярна отладочная плата «Explorer 16 Development Board» фирмы Microchip, хотя другие фирмы также приступили к выпуску отладочных плат на базе 16-битных микроконтроллеров.

ГЛАВА 2

АРХИТЕКТУРА МИКРОКОНТРОЛЛЕРОВ PIC24F

Микроконтроллеры PIC24F были разработаны как недорогое аппаратно-программное решение для перехода от 8-битных микроконтроллеров семейства PIC18 к 16-битной архитектуре, призванное обеспечить максимальную преемственность как уже разработанных приложений для PIC18, так и вновь создаваемых, более эффективных и недорогих 16-битных решений.

Выбор микроконтроллеров PIC24F оправдан в тех случаях, когда необходимо обеспечить среднюю производительность системы при относительно невысокой стоимости конечного продукта. Для приложений, требующих более высокой производительности (выше, чем 16 MIPS), можно использовать более дорогие микроконтроллеры семейства PIC24H.

Микроконтроллеры PIC24F обладают следующими характеристиками:

- высокая производительность (до 16 MIPS);
- векторная система прерываний с 16 уровнями приоритетов;
- наличие 16 рабочих регистров;
- возможность выполнения 16-битных математических операций;
- возможность выполнения операций умножения с разрядностью 17 × 17 бит за один машинный цикл;
- возможность выполнения сдвига на произвольное количество бит (до 16) за один машинный цикл;
- аппаратно-программная архитектура, оптимизированная для разработки программ на языке Си;
- мощная система команд, которая включает инструкцию повторения `repeat` для циклического выполнения команд, что особенно полезно при использовании команд пересылки данных.

По сравнению с микроконтроллерами PIC18 микроконтроллеры PIC24F обладают многими кардинальными улучшениями. Во-первых, в микроконтроллерах PIC24F расширен объем оперативной и флэш-памяти, что при прочих равных условиях позволяет оптимизировать обработку больших объемов данных и создавать более высокопроизводительные системы управления и обработки данных. Еще одним существенным улучшением стало включение в состав микроконтроллера дополнительных периферийных модулей.

В микроконтроллерах семейства имеется пять 16-битных таймеров, четыре из которых можно каскадировать, получая два 32-битных. Кроме того, на кристалле микроконтроллера находится интегрированный 10-битный аналого-цифровой преобразователь последовательного приближения, который может выполнять преобразование аналоговых сигналов со скоростью до 500 тыс. выборок в секунду. В микроконтроллерах семейства также реализован модуль JTAG, который позволяет выполнять тестирование и программирование микроконтроллера в системе.

Микроконтроллеры PIC24F могут взаимодействовать с различными внешними периферийными устройствами посредством интерфейсов I^{2}C, SPI и UART. Для этого в состав устройства включены соответствующие модули, которые могут настраиваться и работать независимо друг от друга. Функциональность подсистемы асинхронной последовательной передачи данных улучшена за счет включения в UART аппаратно-программного субмодуля IrDA. Существенно улучшает рабочие характеристики модулей интерфейсов SPI и UART сериализация данных в буфере FIFO, которая позволяет снизить непроизводительные траты процессорного времени на обработку передачи данных. В отличие от многих устройств семейства PIC18, в которых реализован порт параллельной передачи данных, работающий только в режиме «ведомого», в микроконтроллерах PIC24F имеется модуль параллельного обмена данными, который позволяет работать как в режиме «ведомого», так и в режиме «ведущего». Это существенно расширяет возможности PIC24F при обмене данными с периферийными устройствами, имеющими параллельный интерфейс (принтеры, сканеры, устройства внешней памяти и т. д.).

Все устройства семейства PIC24F имеют один и тот же набор базовых периферийных модулей и отличаются объемом флэш-памяти. Обобщенная функциональная схема микроконтроллеров семейства PIC24F показана на **Рис. 2.1**.

Периферийные модули микроконтроллеров PIC24F позволяют создавать системы обработки данных и управления для широкого класса задач, решаемых в промышленности и в лабораторных исследованиях. Мы будем детально рассматривать принципы функционирования большинства из этих периферийных модулей в последующих главах, а сейчас проведем краткий обзор и начнем с модуля аналого-цифрового преобразователя.

В микроконтроллерах семейства Microchip используется 10-битный аналого-цифровой преобразователь последовательного приближения. Ниже приводятся отдельные характеристики этого преобразователя:

- скорость преобразования — до 500 тыс. выборок/с;
- количество каналов входных аналоговых сигналов — 16;
- источник опорного напряжения — внешний или внутренний.

Аналого-цифровой преобразователь может работать в режиме автоматического сканирования входов и поддерживает различные режимы синхронизации. Модуль АЦП допускает автономную работу при переходе процессора в «спящий» режим или режим «холостого хода». Аналого-цифровой преобразователь может производить несколько последовательных выборок, накапливая

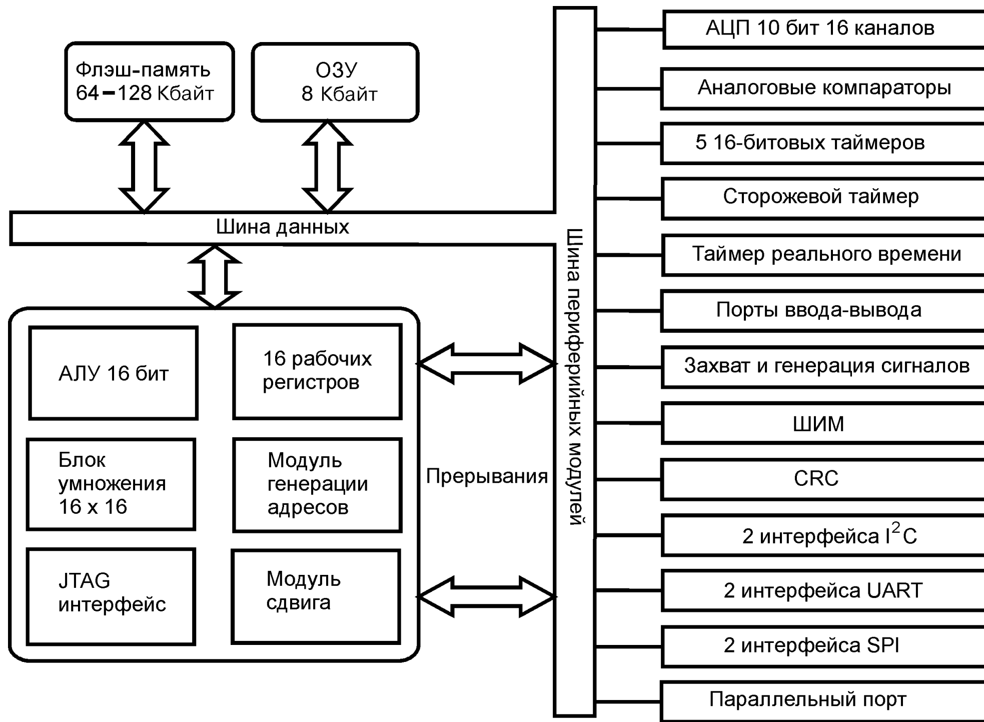

Рис. 2.1. Обобщенная функциональная схема микроконтроллеров PIC24F

результат в 16-уровневом буфере данных, и сохранять результат в одном из четырех форматов.

Следующий периферийный модуль, который мы рассмотрим, — модуль аналоговых компараторов. Это устройство включает в себя два компаратора, которые используются при реализации широкого класса функциональных узлов, например детектора перехода через ноль в схеме синхронизации по переменному току 50 Гц, или при создании более сложных устройств, таких, как 16-битный сигма-дельта аналого-цифровой преобразователь.

Микроконтроллеры PIC24F включают 5 модулей таймеров общего назначения разрядностью 16 бит. Все пять таймеров обладают общими базовыми функциональными возможностями. Регистры периода всех таймеров могут использоваться для генерации прерывания при совпадении содержимого такого регистра с текущим содержимым регистра таймера. Во всех таймерах предусмотрены режим запуска/останова по внешнему сигналу и генерация прерывания по спаду внешнего сигнала. Четыре из пяти таймеров могут объединяться попарно для формирования 32-битных таймеров. С модулями таймеров тесно связан 5-канальный модуль захвата входных сигналов и 5-канальный модуль генерации цифровых сигналов. Модуль захвата входных сигналов используется для измерения интервалов между событиями. Минимальная разрешающая способность при таком измерении равна длительности одного машинного цикла. Для синхронизации временны́х меток модуль захвата входных сигналов использует в качестве базовых Таймеры 2 и 3.

Модуль генерации цифровых сигналов используется для генерации одиночных импульсов и импульсных последовательностей и имеет пять каналов. Для отсчета интервалов времени при формировании таких сигналов модуль генерации цифровых сигналов использует в качестве базовых Таймер 2 и Таймер 3. Выходные сигналы этого модуля могут использоваться для управления обычными (термоэлементы) и индуктивными (электродвигатели) нагрузками, а также для синтеза голосовых сигналов.

В микроконтроллерах семейства PIC24F имеется два модуля универсального асинхронного приемопередатчика (UART), которые позволяют реализовать обмен данными в соответствии со стандартами RS-232 и RS-485. Приемопередатчик обеспечивает обмен 8- или 9-битными данными с контролем четности или без такового с одним или двумя стоповыми битами. Устройство поддерживает функцию аппаратного контроля обмена данными, что обеспечивает высокий уровень надежности и производительности, позволяя подключать к микроконтроллеру различные периферийные устройства, например модемы. Скорость передачи данных может изменяться от 15 бод до 1 Мбод. Повышение производительности операций обмена данными и разгрузка процессора обеспечиваются за счет использования 4-уровневых буферов входных/выходных данных и прерываний. Кроме того, модуль UART поддерживает передачу информации по протоколам IrDA и LIN.

Для обмена данными между различными устройствам в настоящее время очень часто используются протоколы I^2C и SPI. Микроконтроллеры PIC24F имеют интегрированные периферийные модули для поддержки этих протоколов. В модуле I^2C применяется аппаратно-программный алгоритм, позволяющий выбрать режим «ведущего» или «ведомого». Модуль поддерживает как 7-, так и 10-битную адресацию устройств на шине I^2C, при этом тактовая частота шины I^2C может быть задана равной 100 или 400 кГц.

Модули SPI микроконтроллеров PIC24F можно сконфигурировать для работы с двумя, тремя или четырьмя сигнальными линиями. В режиме с двумя линиями (синхронизации и данных) интерфейс SPI можно использовать для приема сигналов датчиков. Для работы с аналого-цифровыми преобразователями, сдвиговыми регистрами и микросхемами памяти EEPROM используется, как правило, 3-проводной интерфейс.

Еще один периферийный модуль — модуль часов реального времени с календарем (RTCC) — предназначен для точного отсчета времени в течение длительных интервалов и оперирует с датами и временем. Он может выполнять функцию будильника, включая внешнее устройство в определенный момент времени в будущем. Модуль RTCC синхронизируется непосредственно от внешнего источника тактового сигнала частотой 32 кГц. Из этого сигнала посредством пределителя формируется внутренний сигнал с периодом 0.5 с, который используется для синхронизации регистров модуля, содержащих эле-

менты даты (год, месяц, день; день недели, часы, минуты и секунды). Данные хранятся в регистрах в удобном BCD-формате. Функция будильника может быть запрограммирована на определенный месяц, день; день недели, час, минуту и секунду. Кроме того, поскольку регистры модуля часов реального времени работают на очень низкой частоте, это позволяет минимизировать энергопотребление устройства. Часы реального времени могут функционировать и в том случае, если процессор находится в «спящем» режиме. Для управления внешними устройствами можно использовать выходной сигнал модуля частотой 1 Гц или сигнал, формируемый при срабатывании будильника.

Модуль параллельного порта позволяет легко реализовывать аппаратно-программный 8-битный интерфейс с внешними устройствами и модулями памяти. Модуль поддерживает мультиплексирование шин адреса/данных, позволяя передавать по 8-битной шине 16-битные данные. В модуле предусмотрена работа с 16 адресными линиями, что дает возможность адресовать до 64 Кбайт памяти, а при использовании дополнительных линий адреса, в качестве которых могут быть задействованы выводы портов общего назначения, — и большее пространство адресов. В модуле предусмотрена функция автоинкремента/автодекремента адреса, что позволяет оптимизировать передачу больших блоков данных.

Последнее периферийное устройство, на котором мы остановимся, — модуль контроля достоверности данных с использованием циклического избыточного кода (CRC). Этот модуль находит применение при контроле ошибок обмена данными с периферийным оборудованием и памятью, особенно при работе с коммуникационным оборудованием по протоколам CAN, USB и Ethernet.

Микроконтроллеры семейства PIC24F имеют 16-разрядную шину данных, что существенно повышает их функциональные возможности по сравнению с устройствами семейства PIC18. Длина инструкции (команды) процессора в микроконтроллерах серии PIC24 равна 24 битам, а частота выполнения команд процессора в 2 раза меньше, чем частота синхронизации устройства. Если использовать принятые в документации фирмы Microchip обозначения, то это соотношение можно выразить формулой:

$F_{CY} = F_{OSC}/2$,

где F_{OSC} — тактовая частота синхронизации микроконтроллера, а F_{CY} — тактовая частота процессора.

Счетчик команд (Program Counter, PC) имеет разрядность 23 бита и позволяет адресовать до 4 миллионов инструкций в памяти программ. Высокая производительность микроконтроллера обеспечивается за счет использования механизма конвейеризации инструкций, при котором выборка и декодирование следующей инструкции процессора осуществляются на этапе выполнения предыдущей. При этом все инструкции выполняются за один машинный цикл, за исключением инструкций передачи управления, инструкций, оперирующих двойными словами, и инструкций табличного чтения/записи. Кроме того, определенную оптимизацию загрузки процессора можно осуществить за счет применения инструкции цикла `repeat`.

Микроконтроллеры семейства PIC24F имеют 16 рабочих регистров, оперирующих данными размером в одно слово (16 бит). Каждый из этих регистров может выступать в роли регистра данных, адреса или смещения. Шестнадцатый по счету регистр (W15) используется в качестве указателя стека (SP, Stack Pointer) в операциях вызова процедур и обработчиков прерываний. В микроконтроллерах PIC24F предусмотрена возможность отображения верхних 32 Кбайт памяти данных на адресное пространство памяти программ с выравниванием по границе слова, для чего используется 8-битный регистр PSVPAG (Program Space Visibility Page).

Разработчики семейства PIC24F предусмотрели обратную совместимость инструкций и режимов адресации процессора с микроконтроллерами семейства PIC18 за счет прямого включения подмножества инструкций PIC18 в систему команд PIC24, а также посредством использования макросов.

В микроконтроллерах семейства PIC24F реализована гарвардская архитектура, в которой память программ и память данных разделены, что позволяет осуществлять прямой доступ к памяти программ из памяти данных во время выполнения программного кода. Организация памяти программ одной из наиболее распространенных линеек микроконтроллеров семейства PIC24FJ128GAxxx со 128 Кбайт флэш-памяти показана на **Рис. 2.2**.

Пользовательским программам доступна область памяти в диапазоне адресов 0x000000...0x7FFFFF, за исключением области конфигурации устройства, доступ к которой осуществляется посредством инструкций `tblrd/tblwt`. Память программ организована в виде блоков, адресуемых пословно. Хотя адрес памяти является 24-битным, более удобно рассматривать любой адрес в виде младшего и старшего слова, при этом старший байт старшего слова не используется и равен 0. Младшее слово всегда располагается по четному адресу, а старшее — по нечетному. Следует сказать, что адреса памяти программ всегда выравниваются по границе слова и при выполнении программного кода всегда инкрементируются и декрементируются на 2.

Область памяти программ между адресами 0x000000 и 0x000200 зарезервирована для векторов прерываний. По адресам 0x000000 и 0x000002 размещается команда перехода к фактическому началу программы, при этом по первому адресу располагается код операции инструкции `goto`, а по второму — собственно адрес точки входа в программу. Также в памяти программ размещаются две таблицы векторов прерываний. Одна из них располагается в диапазоне адресов 0x000004...0x0000FF, а вторая (альтернативная) — в диапазоне адресов 0x000100...0x0001FF.

Адресное пространство памяти данных микроконтроллеров PIC24F представляет собой непрерывную область 16-битных адресов с линейной адресацией. Адрес для доступа к данным формируется двумя модулями генерации адресов, один из которых используется в операциях записи, а другой — в операциях чтения данных. Распределение памяти данных микроконтроллеров PIC24F[1] показано на **Рис. 2.3**.

[1] Моделей с 8 Кбайт ОЗУ. – *Примеч. науч. ред.*

Организация памяти программ PIC24FJ128GA

Область	Адрес
Инструкция GOTO	000000h
Адрес перезагрузки	000002h
Таблица векторов прерываний	000004h 0000FEh
Зарезервировано	000100h
Альтернативная таблица векторов прерываний	000104h 0001FEh
Флэш-память программ (44К инструкций)	000200h
Область конфигурации флэш-памяти	0157FEh
Не используется	015800h 7FFFFEh
Зарезервировано	800000h F7FFFEh
Регистры конфигурации устройства	F80000h F8000Eh
Зарезервировано	F80010h FEFFFEh
Код устройства	FF0000h FFFFFEh

Рис. 2.2. Схема организации памяти программ микроконтроллеров PIC24FJ128GA

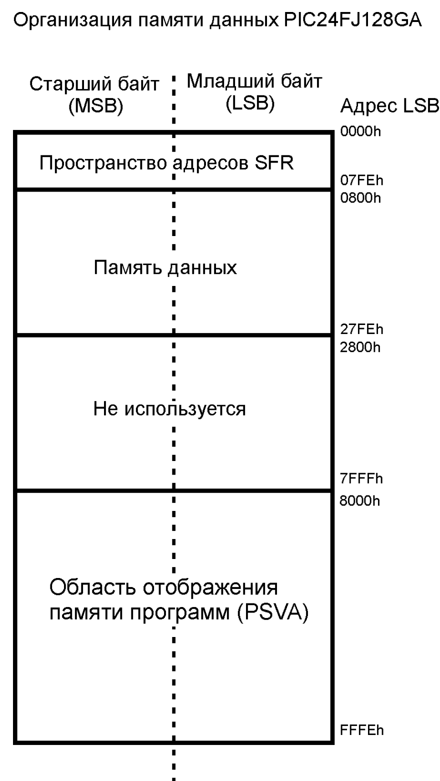

Рис. 2.3. Схема распределения памяти данных

Все действующие адреса памяти данных (Effective Addresses, EA) имеют размер 16 бит, что позволяет адресовать 64 Кбайт памяти. Нижняя половина адресов памяти данных, для которых старший бит действующего адреса равен 0, используется для адресации данных, а старшая половина адресов, для которых старший бит адреса равен 1, — для отображения памяти программ на память данных (Program Space Visibility Area, PSVA). Микроконтроллеры PIC24FJ128GA010 имеют объем памяти данных 8 Кбайт; при обращении к памяти данных по адресу, лежащему за пределами этой области, возвращается нулевой байт или слово.

Данные в памяти выравниваются по границе слова, при этом младшие байты имеют четные, а старшие — нечетные адреса. Для обратной совместимости с 8-битными микроконтроллерами в систему команд PIC24F включены команды, поддерживающие операции как со словами, так и с отдельными байтами. При этом процессор распознает, когда выполняется операция над байтом, а когда — над словом, и генерирует соответствующий адрес. Например, если в команде используется косвенная адресация с пост-инкрементом, такая, как

[W1++], то для байтовой команды генерируется адрес, равный W1+1, а для команды, работающей со словом, — W1+2. При любых операциях со словами данные должны иметь четный адрес, поэтому компилятор будет выдавать ошибку при сборке программ, если в программном коде происходит обращение к слову по нечетному адресу. Если такая ситуация возникает при выполнении программы, то генерируется ошибка адресации и вызывается обработчик этой исключительной ситуации. По этой причине следует соблюдать осторожность, если в программе одновременно присутствуют инструкции, работающие с байтами и со словами.

Для повышения эффективности работы программ в систему команд включен ряд инструкций, позволяющих преобразовывать байты данных в слова. Это инструкция расширения знака `se` (Sign-Extend), приводящая 8-битное число со знаком к 16-битному формату, и инструкция расширения нуля `ze` (Zero-Extend), заполняющая старший байт беззнакового целого нулями. Большинство инструкций процессора могут работать как с байтами, так и со словами, но некоторые инструкции оперируют только со словами.

Первые два килобайта адресного пространства памяти данных (от 0x0000 до 0x7FF) выделяются для регистров специальных функций (Special Function Register, SFR), которые используются ядром и периферийными модулями микроконтроллера PIC24F для выполнения различных операций. Эти регистры распределены между модулями, с которыми они связаны, и, как правило, группируются по принадлежности к тому или иному периферийному модулю. Большинство адресов, выделяемых под регистры специальных функций, на данный момент не задействованы и читаются как 0.

Мы продолжим изучение особенностей аппаратно-программной архитектуры микроконтроллеров PIC24F в следующей главе, в которой подробно рассмотрим особенности программирования этих устройств.

ГЛАВА 3

СИСТЕМА КОМАНД И ОСНОВЫ ПРОГРАММИРОВАНИЯ МИКРОКОНТРОЛЛЕРОВ PIC24F

Микроконтроллеры PIC24 представляют собой качественно новый тип устройств, разработанных фирмой Microchip. Программная архитектура этих микроконтроллеров существенно отличается от предыдущих семейств 8-битных устройств, хотя разработчики сделали все возможное, чтобы обеспечить безболезненный переход от 8-битной архитектуры к 16-битной.

В этой главе рассматривается базовая программная модель микроконтроллеров PIC24F без учета особенностей реализации периферийных модулей, которые детально анализируются в последующих главах.

3.1. Программная модель микроконтроллеров PIC24F

Для разработчика программного обеспечения программную модель микроконтроллеров PIC24F можно представить так, как показано на **Рис. 3.1**.

В качестве рабочих используются 16 регистров процессора (W), которые могут функционировать как регистры данных или адресов. Функция, выполняемая W-регистром, определяется режимом адресации, который используется в данной инструкции.

В регистре W15 содержится указатель стека (Stack Pointer, SP), который автоматически модифицируется при обработке исключений, вызовов подпрограмм и прерываний. Тем не менее, регистр W15 можно использовать в инструкциях в качестве обычного рабочего регистра. Во многих случаях такое использование упрощает манипуляции с указателем стека. Бит 0 этого регистра всегда сброшен в 0 для выравнивания содержимого стека.

При инициализации микроконтроллера в регистр W15 записывается значение 0x0800, и это обеспечивает установку указателя стека на начало области памяти, которая доступна пользователю. Программа пользователя может модифицировать SP так, чтобы он указывал на любую требуемую область памяти. Указатель стека всегда указывает на первое доступное слово и смещается при заполнении от младших адресов к старшим. Указатель стека всегда декремен-

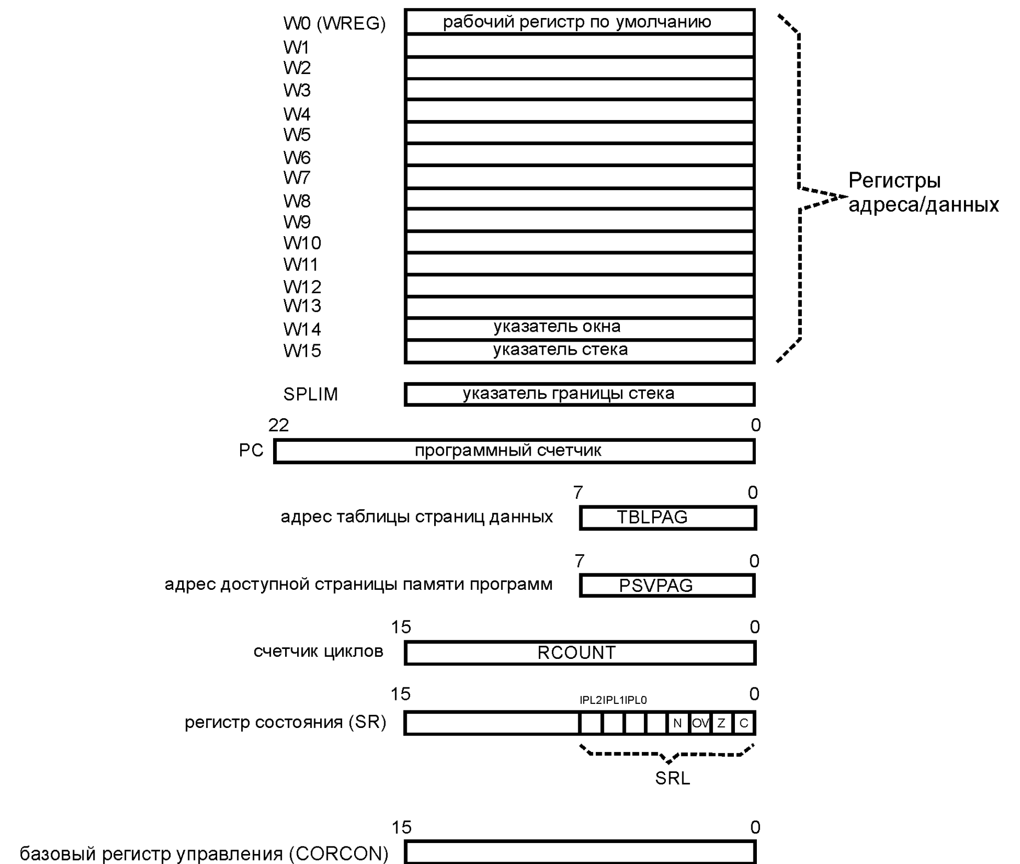

Рис. 3.1. Программная модель микроконтроллера PIC24F

тируется перед извлечением значения из стека и инкрементируется после помещения значения в стек. Функционирование стека показано на **Рис. 3.2**.

Когда содержимое счетчика команд помещается в стек, то биты 15...0 располагаются по первому доступному адресу в стеке, а биты 22...16 помещаются по следующему адресу, при этом старшие биты второго слова дополняются нулями.

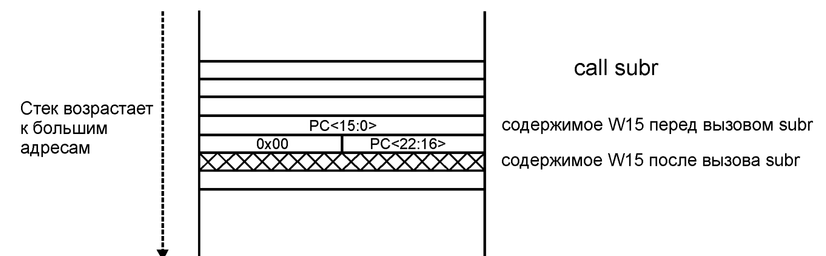

Рис. 3.2. Функционирование стека

Процессор микроконтроллеров PIC24F имеет 16-битный регистр состояния (Status Register, SR). Младшая часть регистра состояния (SRL) содержит флаги арифметических операций (нуля, переноса, переполнения и знака), а также 3 бита приоритета (IPL2...IPL0), в которых устанавливается текущий приоритет прерываний процессора (см. **Рис. 3.1**).

Регистр управления процессором (Core Control Register, CORCON) имеет всего 2 бита, которые можно использовать при разработке программ. Бит 3 (Interrupt Priority Status Level, IPL3) задает уровень приоритета прерываний процессора. Если этот бит установлен, то уровень приоритета прерываний процессора выше 7 — в этом случае пользовательские прерывания однозначно запрещены. Если этот бит сброшен, то уровень приоритета меньше или равен 7 — в этом случае прерывания пользователя могут быть разрешены. Бит 2 (Program Space Visibility, PSV) указывает на возможность доступа к памяти программ из памяти данных: если бит установлен, то заданная область памяти программ доступна из памяти данных, а если сброшен — недоступна.

К программно-доступным регистрам ядра процессора микроконтроллеров PIC24F относятся еще два регистра, которые выполняют специальные функции и могут быть полезны при работе с блоками памяти и таблицами. Регистр TBLPAG используется для хранения старших 8 битов адреса памяти программ при операциях чтения/записи табличных данных. С помощью табличных инструкций процессора можно перемещать данные из памяти программ в память данных. Регистр PSVPAG (Program Memory Visibility Page Address Pointer) позволяет пользователю отобразить страницу памяти программ размером 2 Кбайт на верхние 32 Кбайт адресного пространства памяти данных. С помощью этого регистра можно получить доступ к секции констант программы.

3.2. Режимы адресации и система команд

Систему инструкций (команд) микроконтроллеров PIC24F можно разделить на несколько групп:

- команды перемещения;
- команды математических операций;
- команды логических операций;
- команды сдвига и циклического сдвига;
- команды работы с битами;
- команды сравнения/выбора;
- команды условных переходов;
- команды работы со стеком;
- команды управления.

Мы не будем подробно описывать функционирование каждой команды, а остановимся лишь на общих принципах реализации команд и используемых ими методах адресации. В этом разделе мы рассмотрим многочисленные примеры использования команд микроконтроллеров семейства PIC24F с применением различных методов адресации. Инструкции процессора в PIC24F могут быть ре-

гистровыми и адресными. Основное различие между этими типами инструкций состоит в том, что регистровые инструкции модифицируют содержимое регистров[1], в то время как адресные инструкции используют содержимое рабочих регистров для доступа к данным, находящимся в памяти. В свою очередь адресные инструкции могут быть как с прямой регистровой адресацией (register direct addressing), так и с косвенной регистровой адресацией (register indirect addressing).

Прямая регистровая адресация используется для доступа к содержимому 16 рабочих регистров W0...W15, причем обращение возможно как к целому слову (16 бит), так и к байту.

Косвенная регистровая адресация применяется для доступа к любой ячейке памяти данных посредством формирования исполнительного адреса данных по содержимому одного или нескольких рабочих регистров. Таким образом, содержимое рабочего регистра или регистров является указателем на область памяти данных. Дополнительные возможности такого метода адресации обеспечиваются механизмами пред- и пост-инкремента содержимого регистра, что позволяет последовательно обрабатывать непрерывный диапазон адресов в памяти данных.

Методы формирования эффективного адреса (EA) с помощью косвенной регистровой адресации показаны в **Табл. 3.1**.

Таблица 3.1. Формирование эффективного адреса методом косвенной регистровой адресации

Режим адресации	Байтовая адресация	Адресация слова	Описание
Косвенная без модификации	EA = [Wn]	EA = [Wn]	Эффективный адрес формируется по содержимому регистра Wn
Косвенная с пост-инкрементом	EA = [Wn]+=1	EA = [Wn]+=2	Эффективный адрес формируется по содержимому регистра Wn, затем содержимое Wn инкрементируется
Косвенная с пост-декрементом	EA = [Wn]–=1	EA = [Wn]–=2	Эффективный адрес формируется по содержимому регистра Wn, затем содержимое Wn декрементируется
Косвенная с пред-инкрементом	EA = [Wn+=1]	EA = [Wn+=2]	Перед формированием эффективного адреса содержимое Wn инкрементируется
Косвенная с пред-декрементом	EA = [Wn–=1]	EA = [Wn–=2]	Перед формированием эффективного адреса содержимое Wn декрементируется
Косвенная со смещением, заданным в регистре	EA = [Wn + Wb]	EA = [Wn + Wb]	Эффективный адрес формируется как сумма Wn и Wb
Косвенная со смещением, заданным константой	EA = [Wn + + Slit5]	EA = [Wn + + 2*Slit5]	Эффективный адрес формируется как сумма Wn и константы Slit5 (значение в диапазоне от –16 до +15)

[1] Точнее, области памяти в диапазоне адресов 0x0000...0x1FFF (команда mov — в диапазоне адресов 0x0000...0xFFFE). — *Примеч. науч. ред.*

Регистр W0 является особым регистром, поскольку это единственный из рабочих регистров, который можно использовать в регистровых инструкциях. Регистровые инструкции оперируют с адресами памяти и содержимым регистра W0. Рабочие регистры W1...W15 в таких инструкциях в качестве приемника использовать нельзя. Регистровые инструкции обратно совместимы с инструкциями, которые входят в систему команд микроконтроллеров предыдущих семейств и работают только с одним W-регистром. Синтаксис ассемблера включает обозначение «WREG», призванное подчеркнуть, что регистр W0 используется в регистровой инструкции. Ниже приводятся примеры регистровых инструкций:

```
mov    WREG,0x0100
add    0x0100,WREG
```

Первая инструкция помещает содержимое регистра W0 по адресу 0x0100, а вторая складывает значение по адресу 0x0100 с содержимым регистра WREG и помещает результат в WREG.

Все рабочие регистры отображены на адресное пространство памяти данных, поэтому их можно использовать в таких, например, инструкциях:

```
mov    0x0004,W10
```

Здесь 0x0004 — адрес регистра W2. Эта инструкция эквивалентна следующей:

```
mov    W2,W10
```

Для более глубокого изучения различных режимов адресации и функционирования команд микроконтроллеров PIC24F мы рассмотрим многочисленные примеры программного кода, написанного на языке ассемблера MPLAB ASM30. В качестве целевого устройства будем использовать микроконтроллер PIC24FJ128GA010, хотя программы будут работать и с другими моделями семейства PIC24F.

Изучение функционирования микроконтроллера на уровне команд ассемблера обязательно, если вы намерены стать высококвалифицированным разработчиком программного обеспечения. Даже если в дальнейшем вам придется разрабатывать программы на одном из языков высокого уровня (Си, Бейсик или Паскаль), знание языка ассемблера и программной архитектуры окажет неоценимую помощь при отладке программ. Зачастую, как показывает практика, без знания ассемблера бывает крайне сложно отладить программу, особенно сложную, до требуемого уровня функциональности. Очень часто знание ассемблера помогает решить те или иные проблемы, которые на языке высокого уровня решить довольно трудно (особенно это касается работы с внешними устройствами и интерфейсами). Разумное сочетание языка высокого уровня и ассемблера позволяет решить любую задачу в пределах аппаратных возможностей самого устройства.

Компиляцию тестовых программ на ассемблере мы будем проводить в среде MPLAB IDE, а отладку проводить с помощью программного симулятора MPLAB SIM. Все тестовые программы на ассемблере разработаны с использованием шаблона, который показан далее:


```
#include   "P24FJ128GA010.inc"

.data
           <данные>
.text
.global    reset
__reset:
           <команды>
.end
```

Все примеры из этой книги тестируются на микроконтроллере PIC24FJ128GA010, поэтому в шаблон включен заголовочный файл P24FJ128GA010.inc. Кроме того, во многих примерах используются инициализированные данные, которые объявляются в секции `.data`. Команды размещаются в секции программного кода `.text`. Исходный текст программы на ассемблере следует сохранить в файле с расширением .s, а сам файл включить в проект, созданный с помощью мастера проектов среды MPLAB IDE.

Мы рассмотрим не все команды процессора, поскольку это заняло бы несколько сот страниц, а только часть из них. Детальную информацию обо всех инструкциях процессора читатели смогут найти в технической документации на микроконтроллеры PIC24F фирмы Microchip.

3.2.1. Команды перемещения и адресация данных

В командах перемещения, большинство из которых имеет мнемоническое обозначение mov, используется хотя бы один регистр. Регистровые инструкции могут использовать W-регистр как регистр данных или регистр, в который помещается смещение адреса. Рассмотрим несколько примеров.

Пример 1

```
mov    #0x19C7,W0
mov    W0,W1
```

В первой инструкции константа 0x19C7 командой `mov` помещается в регистр W0. Затем содержимое регистра W0 пересылается в регистр W1. Таким образом, после выполнения второй инструкции в регистре W1 будет содержаться значение 0x19C7.

Пример 2

```
.data
d1:  .word  0x0800
.text
.global  __reset
__reset:
      mov     #0x1234,W0
      mov     #d1,W1
      mov     W0,[W1]
L1:
      goto    L1
.end
```

В этой короткой программе переменная `d1` из секции инициализированных данных (директива `.data`) размещается по адресу 0x0800. В секции программы

.text, которая начинается с метки main, первая команда mov помещает в регистр W0 константу 0x1234. Вторая команда mov помещает адрес переменной d1 (равный 0x0800) в регистр W1. Третья команда помещает содержимое регистра W0 по адресу, находящемуся в W1. Таким образом, после выполнения третьей команды mov в переменной d1 будет содержаться значение 0x1234.

Пример 3

```
.data
d1:  .word  0x0800
d2:  .word  0x0802
.text
.global  __reset
__reset:
      mov      #0x1234,  W0
      mov      #d1,  W1
      mov      W0,  [W1++]
      add      #0x3,  W0
      mov      W0,  [W1]
L1:
      goto     L1
.end
```

Здесь показана методика использования пост-инкрементной адресации. В области данных проинициализированы две переменные – d1 и d2 с адресами 0x0800 и 0x802 соответственно. Как и в предыдущем примере, в регистр W0 помещается значение 0x1234, а в регистр W1 — адрес d1. В третьей команде mov в переменную d1, чей адрес содержится в регистре W1, помещается значение 0x1234, после чего содержимое W1 увеличивается на 2 (команда оперирует словами). Таким образом после выполнения команды регистр W1 будет содержать значение 0x0802, что соответствует адресу переменной d2.

Команда add прибавляет значение 0x3 к содержимому регистра W0, так что перед последней командой mov регистр W0 будет содержать значение 0x1237. Наконец, следующая за add команда помещает это значение по адресу 0x0802, находящемуся в регистре W1.

Пример 4

```
.data
d1:  .word  0x0800
d2:  .word  0x0802
.text
.global  __reset
__reset:
      mov      #0x1234,W0
      mov      #d2,W1
      mov      W0,[W1-]
      add      #0x3,W0
      mov      W0,[W1]
L1:
      goto     L1
.end
```

Пример демонстрирует пост-декрементную адресацию. В регистр W1 командой mov заносится адрес переменной d2 (0x0802), по которому записывается

значение 0x1234, затем содержимое W1 декрементируется, указывая на переменную d1 (адрес 0x0800). По этому адресу записывается значение 0x1237 из регистра W0.

Пример 5

```
.data
d1:  .word  0x0800
d2:  .word  0x0802
d3:  .word  0x0804
.text
.global  __reset
__reset:
      mov      #0x1234,W0
      mov      #d2,W1
      mov      W0,[++W1]
      add      #0x7,W0
      mov      W0,[W1]
L1:
      goto     L1
.end
```

В этой программе показано использование прединкрементной адресации. В регистр W1 командой mov заносится адрес переменной d2 (0x0802). Следующая команда предварительно увеличивает содержимое W1 на 2 (команда оперирует со словами), после чего помещает значение 0x123B из регистра W0 по адресу 0x0804.

Пример 6

```
.data
d1:  .word  0x0800
d2:  .word  0x0802
d3:  .word  0x0804
.text
.global  __reset
__reset:
      mov      #0x1234,W0
      mov      #d2,W1
      mov      W0,[-W1]
L1:
      goto     L1
.end
```

В этом примере показано использование преддекрементной адресации. Вторая команда mov помещает в регистр W1 адрес переменной d2. Следующая команда вначале декрементирует содержимое регистра W1, после чего помещает по адресу 0x0800, содержащемуся в этом регистре, значение 0x1234.

Пример 7

```
.data
d1:  .word  0x0800
d2:  .word  0x0802
d3:  .word  0x0804
.text
```

```
.global  __reset
__reset:
      mov     #0x4321,W0
      mov     #d1,W1
      mov     #4,W2
      mov     W0,[W1+W2]
L1:
      goto    L1
.end
```

Программа демонстрирует применение косвенной адресации со смещением, заданным в одном из рабочих регистров. В качестве базового в данном случае выступает регистр W1, в который помещается адрес переменной `d1`, равный 0x0800 (вторая команда `mov`). Затем в регистр W2 заносится смещение 4. Наконец, последняя команда `mov` помещает содержимое регистра W0 по эффективному адресу, который формируется как сумма содержимого базового регистра W1 и регистра W2. Таким образом, эффективный адрес, по которому будет записано значение 0x4321, получается равным 0x0804, что соответствует переменной `d2`.

Пример 8

```
.data
d1:  .word  0x0800
d2:  .word  0x0802
d3:  .word  0x0804
.text
.global  __reset
__reset:
      mov     #0x4321,W0
      mov     #d1,W1
      mov     W0,[W1+2]
L1:
      goto    L1
.end
```

Данный пример демонстрирует использование косвенной адресации со смещением, заданным константой. Базовый адрес, равный адресу переменной `d1`, загружается в регистр W1. Последняя команда `mov` помещает содержимое регистра W0 по адресу, который формируется как сумма содержимого регистра W1 и константы 2 и будет равен 0x0802. Таким образом, эта команда запишет значение 0x4321 в переменную `d2`.

В ряде случаев необходимо манипулировать с отдельными байтами слова или двойного слова. Для этого в командах перемещения используется суффикс b или q. В следующем примере показано, как записать байт в переменную, расположенную в памяти данных.

Пример 9

```
.data
d1:  .word  0x0800
d2:  .word  0x0802
.text
```



```
.global  __reset
__reset:
      mov     #0x4321,W0
      mov     #d2,W1
      mov.b   W0,[W1]
L1:
      goto    L1
.end
```

Замечу, что во всех командах, манипулирующих с байтами, по умолчанию изменяется младший байт. В этом примере для записи младшего байта (0x21) в младший байт переменной `d2` используется инструкция `mov.b`, в которой применяется косвенная регистровая адресация посредством регистра W1.

В следующем примере в переменную последовательно записываются младший и старший байты значения.

Пример 10

```
.data
d1:  .word  0x0800
d2:  .word  0x0802
.text
.global  __reset
__reset:
      mov     #0x1ACE,W0
      mov     W0,d1
      mov     #d2,W1

      mov     #d1,W0
      mov.b   [W0++],[W1++]
      mov.b   [W0],[W1]
L1:
      goto    L1
.end
```

Первые две команды программы записывают значение 0x1ACE в переменную `d1`. Затем адрес переменной `d1` помещается в регистр W0, а адрес переменной `d2` — в регистр W1. В последних двух командах `mov.b` используется пост-инкрементная адресация для записи старшего байта переменной `d1` в старший байт переменной `d2`.

К командам перемещения данных относится и команда перестановки старшего и младшего байтов или полубайтов, которая эффективно используется в математических вычислениях. Следующий пример демонстрирует перестановку старшего и младшего байтов слова.

Пример 11

```
.text
.global  __reset
__reset:
      mov     #0x4,W1
      mov     #0x9,W2
      add     W1,  W2,W0
      swap    W0
```

```
Loop1:
      bra      Loop1
.end
```

Здесь содержимое регистров W1 (0x4) и W2 (0x9) складывается командой `add W1, W2, W0` и результат (0xD) записывается в регистр W0. В 16-битном регистре W0 старший байт будет равен 0x0, а младший — 0xD. После выполнения команды `swap W0` старший байт станет равным 0x0D, а младший — 0x0. Эту же программу можно модифицировать так, чтобы менять местами значения старшего и младшего полубайта младшего байта регистра W0, применив инструкцию `swap.b`. Исходный текст программы в этом случае будет выглядеть так:

Пример 12

```
.text
.global  __reset
__reset:
      mov      #0x4,W1
      mov      #0x9,W2
      add      W1,W2,W0
      swap.b   W0
Loop1:
      bra      Loop1
.end
```

Перед выполнением команды `swap.b` в регистре W0 содержится значение 0x000D. После выполнения этой команды в регистре W0 будет содержаться значение 0x00D0.

Аналогичным образом можно обменять и содержимое двух 16-битных рабочих регистров с помощью инструкции `exch`. Исходный текст простейшей программы, выполняющей обмен содержимым регистров W0 и W1, приведен ниже.

Пример 13

```
.text
.global  __reset
__reset:
      mov      #0x4,W1
      mov      #0x9,W2
      exch     W1,W2
Loop1:
      bra      Loop1
.end
```

В этой программе содержимое регистра W1 (0x4) пересылается в регистр W2, а содержимое регистра W2 (0x9) пересылается в регистр W1.

3.2.2. Команды сравнения/выбора и условного перехода

К этой группе команд относятся команды вида `BTxx` и `CPxx`, где аббревиатура BT означает «Bit Test», CP означает «Compare», а символы xx — одно из условий. О функциях команд легко догадаться по их названию. Команды `BTxx` проверяют состояние указанного бита (0 или 1) и по результатам проверки

выполняют определенное действие. Команды `CPxx` выполняют сравнение значений операндов, в результате чего устанавливаются флаги в регистре состояния SR процессора. Дальнейшие действия выполняются с помощью команд условных переходов `bra xx`, где xx — условие выполнения перехода (с этими командами мы познакомимся при изучении примеров).

Рассмотрим практические примеры использования этих команд.

Пример 1

```
.data
d1:  .word  0x0800
.text
.global  __reset
__reset:
      mov      #d1,W0
      mov      #0xAAAA,W2
      mov      W2,[W0]

      mov      #0xFFFF,W1
      btss     W1,#0
Loop1:
      bra      Loop1
      mov      #0xFFFF,  W2
      mov      W2,[W0]
Loop2:
      bra      Loop2
.end
```

В этой программе в переменную `d1` записывается значение 0xAAAA, если бит 0 регистра W1 сброшен в 0, и значение 0xFFFF, если бит установлен в 1. В данном примере в регистр W1 заносится значение 0xFFFF, т.е. бит 0 будет установлен. Следовательно, в переменную `d1` будет записано значение 0xFFFF. Если в регистр W1 поместить, например, значение 0xFFFE, то результат программы изменится и в переменную `d1` будет записано 0xAAAA. Команда `btss` выполняет проверку бита 0 в регистре W1: если он равен 0, то выполняется следующая за `btss` команда, а если он равен 1, то следующая за `btss` инструкция пропускается. В этой программе команда `bra` выполняет безусловный переход на указанную в ней метку, образуя «вечный» цикл.

Противоположным образом ведет себя инструкция `btsc`. В следующем примере показан модифицированный исходный текст программы из Примера 1, в котором вместо `btss` применяется инструкция `btsc`.

Пример 2

```
.data
d1:  .word  0x0800
.text
.global  __reset
__reset:
      mov      #d1,W0
      mov      #0xFFFF,W2
      mov      W2,[W0]

      mov      #0xFFFF,W1
```

```
      btsc     W1,#0
Loop1:
      bra      Loop1
      mov      #0xAAAA,W2
      mov      W2,[W0]
Loop2:
      bra      Loop2
.end
```

В этой программе инструкция btsc проверяет на равенство нулю нулевой бит регистра W1. Если он равен 0, то следующая команда пропускается и в переменную d1 записывается значение 0xAAAA; если бит установлен в 1, программа уходит на «вечный» цикл по метке Loop1, реализованный инструкцией безусловного перехода bra, при этом в переменной d1 остается значение 0xFFFF, как в данном конкретном случае.

Пример 3

```
.data
d1:  .word  0x0800
.text
.global  __reset
__reset:
      mov      #d1,W0
      mov      #0xF,W2
      mov      W2,[W0]

      mov      #9,W0
again:
      dec      d1
      cp       d1
      bra      ge,again
Loop1:
      bra      Loop1
.end
```

В этой программе показан простой цикл по метке again, выполненный с применением команд dec, cp и bra ge, again. В начале каждой итерации цикла происходит декремент значения переменной d1, после чего команда cp сравнивает значение, записанное в рабочий регистр по умолчанию W0 (9), со значением в переменной d1. Команда cp изменяет, в зависимости от результата сравнения, младшие биты в регистре состояния процессора SR. Само сравнение выполняется методом вычитания содержимого рабочего регистра из содержимого переменной d1 во внутреннем цикле команды cp, так что значения операндов не меняются. Следующая за командой cp инструкция bra проверяет условие «больше-равно». Пока содержимое переменной d1 остается большим либо равным значению в W0, цикл повторяется. Как только значение в переменной d1 станет равным 8, произойдет выход из цикла.

В этом примере у нас появилась команда условного перехода bra ge, again. Обратите внимание на синтаксис этой команды. После мнемоники инструкции (bra) следует условие в форме аббревиатуры (ge, Great than or Equal) и указывается метка перехода к программе, в случае если условие выполнено.

В нашем случае инструкция bra анализирует биты состояния, установленные предыдущей командой cp.

Команда cp имеет различные модификации, позволяющие организовывать циклы оптимальным образом.

Пример 4

```
.data
d1:  .word  0x0800
.text
.global  __reset
__reset:
      mov      #d1,W0
      mov      #15,W2
      mov      W2,[W0]
again:
      dec      d1
      cp0      d1
      bra      gt,again
Loop1:
      bra      Loop1
.end
```

Здесь для реализации цикла используется инструкция cp0, которая выполняет сравнение значения переменной с нулем и устанавливает соответствующим образом биты регистра состояния SR. В переменную d1 помещается значение 15, после чего ее содержимое декрементируется в цикле again, пока не станет равным 0. Инструкция bra gt, again анализирует флаг нуля регистра SR и после его установки завершает цикл.

Пример 5

```
.data
d1:  .word  0x0800
.text
.global  __reset
__reset:
      mov      #d1,W0
      mov      #15,W2
      mov      W2,[W0]
      repeat   #11
      dec      d1
Loop1:
      bra      Loop1
.end
```

В этой программе демонстрируется использование еще одной весьма полезной инструкции условного перехода процессора repeat. Команда repeat имеет всего один операнд, представляющий собой число в диапазоне 0...16383, которое определяет, сколько раз выполнится следующая за командой repeat инструкция. В данном случае команда repeat будет выполнять следующую за ней инструкцию dec d1 11 раз. После выполнения команды repeat в переменной d1 будет содержаться значение 4, но инструкция dec d1 должна выполниться еще один раз, поскольку является очередной командой после repeat.

В результате перед выполнением последней команды bra программы переменная d1 будет содержать значение 3. Подобные нюансы следует учитывать как при написании кода, так и при анализе дизассемблированных листингов.

Пример 6

```
.data
d1:  .word  0x0800
.text
.global  __reset
__reset:
      mov      #d1,W0
      mov      #32,W2
      mov      W2,[W0]
      call     Increment
Loop1:
      bra      Loop1
Increment:
      repeat   #7
      inc      d1
      return
.end
```

К инструкциям условных/безусловных переходов относят и команду вызова процедуры call. При вызове команда помещает адрес следующей за ней инструкции в стек, а команда return, которая должна быть последней в процедуре, извлекает это значение из стека в счетчик команд. В этом примере вызывается процедура Increment, которая увеличивает значение переменной d1 на 8. В программе на языке ассемблера точка входа в процедуру определяется меткой (в данном случае Increment), а выход из процедуры происходит по команде return.

3.2.3. Команды работы с битами

Команды работы с битами микроконтроллеров PIC24F, как следует из названия, предназначены для манипуляций с отдельными битами операндов. Такая возможность является весьма востребованной, поскольку в большинстве случаев системы на базе микроконтроллеров должны работать с отдельными сигнальными битами внешних устройств. Битовые команды позволяют анализировать состояние отдельных битов, что дает возможность организовать ветвления и переходы в программе, а также устанавливать или сбрасывать отдельные биты операнда.

Пример 1

```
.data
d1:  .word  0x0800
.text
.global  __reset
__reset:
      mov      #d1,W0
      mov      #32,W2
      mov      W2,[W0]
      bset.b   [W0],#0
      bclr.b   [W0],#5
Loop1:
      bra      Loop1
.end
```


Здесь в переменную d1 с адресом 0x0800 записывается значение 32 (0x20), затем команда bset.b [W0], #0 устанавливает бит 0 этой переменной, в результате чего содержимое d1 становится равным 33 (0x21). Следующая за ней инструкция bclr.b [W0], #5 сбрасывает бит 5 переменной d1, в результате чего окончательное значение переменной d1 будет равно 1. В этих командах была использована косвенная регистровая адресация посредством регистра W0, в котором хранится адрес переменной d1.

Для быстрого анализа состояния отдельных битов переменной в систему команд микроконтроллеров PIC24F включены инструкции fbcl, ff1l и ff1r. Эти инструкции очень полезны в системах сбора данных при сканировании, например, сигнальных выводов периферийных устройств. Вот краткое описание этих инструкций:

- fbcl — определяет изменение состояния битов от младшего к старшему (слева направо);
- ff1l — находит первый ненулевой бит при проходе слева направо;
- ff1r — находит первый ненулевой бит при проходе справа налево.

Далее приведен пример использования инструкции ff1r.

Пример 2

```
.text
.global  __reset
__reset:
      mov      #8,W2
      ff1r     W2,W3
Loop1:
    bra        Loop1
.end
```

В этой программе в регистр W2 заносится значение 8. Инструкция ff1r обнаруживает первый единичный бит в регистре W2 на позиции 4 (отсчет ведется от 1) и записывает это значение в регистр W3. Таким образом, регистр W3 будет содержать значение 4.

Пример 3

```
.text
.global  __reset
__reset:
      mov      #d1,W0
      mov      #8,W2
      mov      W2,[W0]
      ff1l     W2,W3
Loop1:
    bra        Loop1
.end
```

В этой программе, в отличие от предыдущей, сканирование первого ненулевого бита в регистре W2 выполняется слева направо инструкцией ff1l.

В этом случае позиция этого бита относительно старшего значащего бита будет равна 13 (0xD). Это значение и будет помещено в регистр W3.

Среди команд битовых операций часто используются инструкции установки (bset, bset.b), сброса (bclr, bclr.b) и переключения (btg, btg.b) битов. Как правило, эти команды применяются для управления внешними исполнительными устройствами, подключенными к портам вывода микроконтроллера.

Рассмотрим учебный пример генератора прямоугольных импульсов на выводе 0 порта A.

Пример 4

```
.text
.global  __reset
__reset:
      mov      #0x0,W0
      mov      W0,TRISA
      clr      PORTA
again:
      btg      PORTA,#0
      call     Delay
      bra      again

Delay:
      clr      TMR1
      mov      #0x8010,W0
      mov      W0,T1CON
  next:
      mov      TMR1,W1
      cp       W1,#0xAFFF
      bra      le,next
      mov      #0,W0
      mov      W0,T1CON
      return

Loop1:
    bra        Loop1
.end
```

В этой программе бит 0 порта A переключается в противоположное состояние с интервалом времени, определяемым функцией задержки Delay. Таким образом, на выводе 0 порта A генерируются прямоугольные импульсы с периодом, равным Delay * 2. Эти операции выполняются в цикле

```
again:
    . . .
    bra        again
```

Собственно переключение бита 0 порта A выполняет инструкция btg PORTA, #0. Для того чтобы это работало, порт A или, по крайней мере, его младший бит должен быть настроен как выход, что достигается сбросом соответствующих битов регистра-защелки TRISA в начале программы. Для реализации задержки в функции Delay используется 16-битный Таймер 1, который запускается на определенное время, определяемое константой в инструкции cp W1, #0xAFFF. Запуск Таймера 1 осуществляется посредством регистра T1CON.

3.2.4. Команды сдвига и циклического сдвига

Эта группа команд часто используется для преобразования последовательного потока битов в параллельный двоичный код и обратно, а также для операций быстрого целочисленного умножения и деления на степень двойки. Рассмотрим несколько примеров.

Пример 1

```
.data
  d1:  .word  0x0800
.text
.global  __reset
__reset:
      mov      #d1,W1
      mov      #0x8888,W2
      mov      W2,[W1]
      mov      #0x2,W0
      lsr      d1
Loop1:
      bra      Loop1
.end
```

В этой программе в переменную d1 записывается значение 0x8888. Затем в регистр W0 помещается число, которое определяет количество сдвигов. Соответственно команда lsr d1 выполняет сдвиг вправо на 2 позиции, в результате чего в переменной d1 окажется число 0x2222.

Пример 2

```
.text
.global  __reset
__reset:
      mov      #0x17,W0
      lsr      W0,  #2,W1
Loop1:
      bra      Loop1
.end
```

В этом примере выполняется сдвиг содержимого регистра W0, которое равно 0x17, на 2 позиции вправо, а результат сохраняется в регистре W1. Таким образом, после выполнения команды lsr W0, #2, W1 в регистре W1 будет содержаться значение 0x5.

В качестве демонстрационного примера рассмотрим реализацию операций целочисленного быстрого умножения при помощи команд сдвига и сложения. Такая возможность продемонстрирована в следующем примере.

Пример 3

```
.text
.global  __reset
__reset:
      mov      #0x4,W0
      mov      W0,W2
      sl       W0,#2,W1
      add      W2,W1,W0
```

```
Loop1:
      bra      Loop1
.end
```

Нам нужно число 4, находящееся в регистре W0, умножить на 5 и результат записать в этот же регистр. Умножение на 5 можно представить в форме 2 × 2 + 1. Иными словами, можно выполнить два сдвига числа 4 влево, в результате чего получим 16, а затем прибавить число 4 к полученному результату. Команда сдвига `sl W0, #2, W1` сдвигает число в регистре W0 на 2 позиции влево и записывает результат в регистр W1. Команда сложения `add W2, W1, W0` складывает содержимое регистров W2 и W1 и записывает результат в регистр W0.

В следующем примере содержимое порта А сдвигается влево, при этом анализируется бит 6 на равенство 1. Если в потоке битов, проходящих через 6-ю позицию, обнаруживается 1, то программа останавливается.

Пример 4

```
.text
.global  __reset
__reset:
      mov      #0x0,W0
      mov      W0,TRISA
      mov      W0,PORTA
      bset.b   PORTA,  #0
again:
      mov      #1,W0
      sl       PORTA
      btsc     PORTA,#6
forever:
      bra      forever
      call     Delay
      bra      again

Delay:
      clr      TMR1
      mov      #0x8010,W0
      mov      W0,T1CON
  next:
      mov      TMR1,W1
      cp       W1,#0x5
      bra      lt,next
      mov      #0,W0
      mov      W0,T1CON
      return

Loop1:
      bra      Loop1
.end
```

В цикле `again` с помощью команды `sl PORTA` выполняется сдвиг влево на 1 бит содержимого порта А. Для команды `sl` величина сдвига задается в рабочем регистре W0 и в данном случае равна 1. Затем анализируется состояние

бита 6 порта А: если он равен 1, то сдвиг влево прекращается, и программа входит в «вечный» цикл по метке `forever`. Если сдвигаемый бит не достиг 6-й позиции, то вызывается функция задержки `Delay`, после чего начинается следующая итерация цикла `again`. Таким образом, в программе будет выполнено 6 сдвигов, после которых в регистре-защелке порта А окажется значение 0x040.

3.2.5. Команды математических и логических операций

Микроконтроллеры семейства PIC24F обладают обширным набором инструкций для выполнения математических и логических действий над операндами. В АЛУ процессора предусмотрено выполнение 4 основных математических действий над целыми знаковыми и беззнаковыми числами: сложения, вычитания, умножения и деления. Для каждого из этих действий предусмотрен ряд инструкций, в которых используются различные типы адресации, делающие выполнение таких операций весьма эффективным. Все математические инструкции оперируют с целыми числами, знаковыми или беззнаковыми. Далее приведено несколько примеров использования этих инструкций.

Пример 1

```
.data
  d1:  .word  0x0800
.text
.global  __reset
__reset:
      mov      #d1,W0
      mov      #67,W1
      mov      W1,[W0]

      mov      #34,W0
      add      d1
Loop1:
      bra      Loop1
.end
```

В этой программе используется одна из версий команды сложения, когда к содержимому ячейки памяти прибавляется содержимое рабочего регистра по умолчанию (W0). Сначала в переменную `d1` записывается значение 67. Затем в регистр W0 заносится значение 34. Команда сложения `add d1` к содержимому `d1` прибавляет содержимое регистра W0, что дает в результате значение 101 (0x65).

Пример 2

```
.text
.global  __reset
__reset:
      mov      #14,W0
      add      W0,#7,W1
Loop1:
      bra      Loop1
.end
```

В этой программе команда add имеет 3 операнда. К содержимому регистра W0 прибавляется 7, и результат записывается в регистр W1. После выполнения операции там будет находиться значение 21 (0x15). Команды математических операций, как и рассмотренные нами команды перемещения, допускают использование прямой и косвенной регистровой адресации. В предыдущем примере в команде add была использована прямая регистровая адресация. Следующий пример демонстрирует использование косвенной регистровой адресации.

Пример 3

```
.data
  d1:  .word  0x0800
.text
.global  __reset
__reset:
      mov      #d1,W0
      mov      #7,W1
      add      W1,#5,[W0]
Loop1:
      bra      Loop1
.end
```

В данном случае команда add складывает содержимое регистра W1 (0x7) с константой 5 и помещает результат по адресу, находящемуся в регистре W0, т.е. в переменную d1. После выполнения операции в d1 будет находиться значение 12 (0xC).

Пример 4

```
.data
  d1:  .word  0x0800
  d2:  .word  0x0802
.text
.global  __reset
__reset:
      mov      #d1,W0
      mov      #7,W1
      mov      #19,W2
      add      W1,W2,[++W0]
Loop1:
      bra      Loop1
.end
```

В этом примере продемонстрировано применение косвенной регистровой адресации с преинкрементом при сложении содержимого двух регистров W1 и W2. В регистр W0 помещается адрес переменной d1 из секции .data, который перед выполнением сложения командой add инкрементируется. Таким образом, результат сложения, выполненного командой add, будет записан по адресу, следующему за адресом d1, т. е. в переменную d2. После выполнения операции в переменной d2 будет содержаться значение 26 (0x1A).

В сочетании с командами условных/безусловных переходов, при помощи математических команд можно реализовать гибкие алгоритмы обработки данных, как это показано в следующих нескольких примерах.

Пример 5

```
.data
  d1:  .word  0x0800
  d2:  .word  0x0802
  d3:  .word  0x0804
.text
.global  __reset
__reset:
      mov      #d1,W0
      mov      #7,W1
      mov      #19,W2
      repeat   #2
      sub      W1,W2,[W0++]
Loop1:
      bra      Loop1
.end
```

В этой программе в ячейки памяти d1…d3 помещается результат вычитания содержимого регистра W2 из содержимого регистра W1, которое равно –12 (0xFFF4). Для последовательного доступа к переменным d1…d3 применяется пост-инкрементная адресация с помощью регистра W0 в команде вычитания sub. Инструкция repeat обеспечивает выполнение следующей за ней инструкции вычитания sub 2 раза. Всего команда sub будет выполнена 3 раза. В результате выполнения этой программы в переменных d1…d3 окажется значение 0xFFF4. Подобный алгоритм при небольшой модификации можно использовать для заполнения области памяти каким-либо значением, например нулем.

Несложно реализовать и алгоритм нахождения суммы элементов какой-либо области памяти, как это показано в следующем примере.

Пример 6

```
.data
  d1:  .word  0x0800
  d2:  .word  0x0802
  d3:  .word  0x0804
  cnt:  .word  0x0806
.text
.global  __reset
__reset:
      mov      #d1,W0
      push     W0
      mov      #7,W1
      mov      W1,[W0++]
      mov      #20,W1
      mov      W1,[W0++]
      mov      #10,W1
      mov      W1,[W0]
      mov      #cnt,W3
      mov      #3,W0
      mov      W0,[W3]         ; cnt = 3

      pop      W0              ; адрес d1 -> W0
```

```
      clr       W4              ; результат накапливаем в W4
next:
      mov       [W0++],W1
      add       W1,  W4,W4
      dec       cnt
      cp0       cnt
      bra       nz,next
Loop1:
      bra       Loop1
.end
```

В этой программе выполняется сложение содержимого ячеек памяти d1…d3, а результат сложения помещается в регистр W4. Переменная cnt выполняет функцию счетчика цикла next, в котором и выполняется операция сложения. В начале программы переменные d1…d3 инициируются следующими значениями: в d1 помещается число 7, в d2 — 20 и в d3 — 10.

Инструкция push W0 сохраняет в стеке адрес переменной d1, который далее будет использоваться в качестве базового адреса при доступе к области памяти. Перед выполнением цикла next в счетчик cnt, адресуемый регистром W3, помещается количество итераций, равное 3, а регистр W4, в котором будет накапливаться результат, обнуляется командой clr W4. Кроме того, перед началом цикла next нам нужно восстановить содержимое регистра W0, а именно адрес переменной d1, что и выполняет команда извлечения из стека pop W0.

Сложение чисел, содержащихся в переменных d1…d3, выполняется инструкцией add W1, W4, W4, при этом к предыдущему значению регистра W4 прибавляется содержимое регистра W1. Регистр W1 содержит значение текущей ячейки памяти, которое помещается в него командой mov [W0++], W1. Цикл выполняется до тех пор, пока значение счетчика cnt не станет нулевым. Это условие отслеживается командами

```
cp0      cnt
bra      nz,next
```

После выполнения цикла next в регистре W4 будет содержаться значение 37 (0x25). Два следующих примера посвящены операциям умножения и их реализации посредством инструкций процессора.

Пример 7

```
.data
  d1:  .word  0x0800
  d2:  .word  0x0802
.text
.global  __reset
__reset:
      mov      #d1,  W0
      push     W0
      mov      #73,  W1
      mov      W1,  [W0++]
      mov      #20,  W1
      mov      W1,  [W0]
```



```
      pop      W0
      mov      [W0++],  W1
      mov      [W0],  W2
      mul.uu   W1,  W2,  W4
Loop1:
      bra      Loop1
.end
```

В этой программе выполняется умножение чисел, содержащихся в переменных d1 и d2. Для умножения беззнаковых чисел используется инструкция mul.uu. В качестве операндов этой инструкции в данной программе выступают регистры W1 и W2, каждый из которых содержит сомножители, извлеченные из переменных d1 и d2 соответственно. В качестве приемника результата умножения выступает пара регистров W4...W5. Первым регистром приемника должен быть обязательно регистр с четным номером (в данном случае W4), в который помещается младший байт результата. Второй регистр приемника в команде явно не указывается, но его номер по умолчанию на 1 больше. В нашем случае это регистр W5, в который помещается старший байт результата. Если бы в качестве младшего регистра приемника использовался регистр W6, то старшим регистром в этом случае был бы W7 и т.д.

В начале программы переменные d1 и d2 инициализируются значениями 73 и 20 соответственно. Для адресации этих переменных используется регистр W0. При указанных значениях переменных после выполнения операции умножения регистр W4 будет содержать значение 1460 (0x5B4).

Для выполнения операций над числами со знаком необходимо использовать другую версию команды умножения, а именно mul.ss. Модифицируем предыдущий пример так, чтобы его можно было использовать для перемножения целых чисел со знаком.

Пример 8

```
.data
  d1:  .word  0x0800
  d2:  .word  0x0802
.text
.global  __reset
__reset:
      mov      #d1,W0
      push     W0
      mov      #73,W1
      mov      W1,[W0++]
      mov      #-10,W1
      mov      W1,[W0]
      pop      W0
      mov      [W0++],W1
      mov      [W0],W2
      mul.ss   W1,W2,W4
Loop1:
      bra      Loop1
.end
```

В этой программе перемножаются положительное число 73 (переменная d1) и отрицательное число −10 (переменная d2). Для умножения используется

команда mul.ss, параметры которой имеют тот же смысл, что и в рассмотренной ранее инструкции mul.uu из предыдущего примера. Результат операции помещается в регистры W4 и W5, при этом в W5 будет содержаться 0, а W4 будет содержать значение 0xFD26, что соответствует отрицательному числу –730, представленному в дополнительном коде.

Группа математических команд содержит инструкции, позволяющие работать с многобайтовыми данными. Эти команды при выполнении операций учитывают флаг переноса, который мог быть установлен как следствие переноса или заема в результате операции с предыдущими словами. К таким инструкциям относятся addc и subb. При желании читатели смогут самостоятельно исследовать работу этих команд с помощью отладчика.

Рассмотрим примеры использования группы команд логических операций. Эти команды позволяют выполнять булевые одноместные и двуместные операции. К таким операциям относятся хорошо известные двуместные операции логическое «И», логическое «ИЛИ», исключающее «ИЛИ» и одноместная операция отрицания «НЕ». В эту группу команд включены инструкции очистки (обнуления) операнда и установки, когда все биты операнда устанавливаются в 1. Инструкции, выполняющие операцию логического «И», имеют мнемоническое обозначение and, логического «ИЛИ» — ior, исключающего «ИЛИ» — xor. Что же касается операции отрицания «НЕ», то она реализована посредством инструкции com. Кроме того, в эту группу команд включена команда neg, позволяющая инвертировать знак числа. Инструкции логических операций весьма полезны при выделении и анализе отдельных битов операндов. Применение этих команд показано в следующих примерах.

Пример 1

```
.text
.global  __reset
__reset:

      mov      #0xFC,W0
      mov      #0x13,W1
      and      W0,W1,W2
Loop1:
      bra      Loop1
.end
```

Здесь выполняется операция логического «И» над операндами, помещенными в регистры W0 и W1. Команда and помещает результат операции в регистр W2. При указанных значениях операндов W0 и W1 итоговое значение в регистре W2 будет равно 0x10. Если инструкцию

```
and  W0,  W1,  W2
```

заменить инструкцией ior W0, W1, W2 (логическое «ИЛИ»), то итоговое значение в регистре W2 станет равным 0xFF. При использовании в программе команды xor (исключающее «ИЛИ») в регистре W2 после выполнения операции будет содержаться значение 0xEF.

Пример 2

```
.text
.global  __reset
__reset:
      mov      #0xC,  W0
      com      W0,  W1
Loop1:
      bra      Loop1
.end
```

В этом примере показано применение инструкции com. Эта инструкция инвертирует все биты исходного операнда. В данном случае инвертируется содержимое регистра W0 (0x000C), а результат, который будет равен 0xFFF3, помещается в W1.

Пример 3

```
.text
.global  __reset
__reset:
      mov      #0xFFF4,  W0
      neg      W0,  W1
Loop1:
      bra      Loop1
.end
```

Программа инвертирует знак числа при помощи команды neg. Команда neg использует стандартный алгоритм получения инверсии числа: сначала все биты операнда инвертируются, затем к младшему биту прибавляется 1. В нашем примере при исходном значении операнда в регистре W0, равном 0xFFF4 (дополнительный код отрицательного числа –12), результат выполнения команды neg, помещенный в регистр W1, будет равен 0x000C, что соответствует положительному числу 12.

Инструкцию neg можно использовать в программе для получения абсолютного значения (модуля) числа, как это показано в следующем примере.

Пример 4

```
.text
.global  __reset
__reset:
      mov      #0x81F4,W0
      mov      W0,W1
      btsc     W0,#15
      neg      W0,W1
Loop1:
      bra      Loop1
.end
```

В этой программе с помощью инструкции btsc W0, #15 проверяется значение старшего бита числа. Если бит равен 0, то число в регистре W0 положительное и не нуждается в преобразовании. В этом случае следующая за btsc инструкция пропускается. Если же число в W0 отрицательное (старший бит установлен в 1), то команда neg преобразует его в положительное. В нашем

случае число 0x81F4 — отрицательное, поэтому оно преобразуется командой `neg` к положительному значению 0x7E0C.

3.2.6. Команды условных/безусловных переходов

Команды этой группы предназначены для управления ходом вычислительного процесса и осуществляют переход к различным секциям программы по результатам проверки флагов состояния процессора. В предыдущих примерах программного кода мы уже использовали некоторые из этих команд, сейчас же остановимся на них более подробно.

Наиболее широко при разработке программного обеспечения используются инструкции `bra xx` (xx — код условия), `call` и `goto`. Команда `bra xx` осуществляет переход по адресу программы, содержащемуся в теле инструкции, при выполнении условия xx. В общей сложности эта команда может проверять выполнение следующих условий:

- C — установлен флаг переноса;
- GE — больше или равно;
- GEU — больше или равно для операций без знака;
- GT — больше;
- GTU — больше для операций без знака;
- LE — меньше или равно;
- LEU — меньше или равно для операций без знака;
- LT — меньше;
- LTU — меньше для операций без знака;
- N — результат предыдущей операции отрицательный;
- NC — перенос (заем) отсутствует;
- NN — результат предыдущей операции неотрицательный;
- NOV — переполнения нет;
- NZ — результат предыдущей операции ненулевой;
- OV — возникло переполнение;
- Z — установлен бит нуля в регистре состояния.

Например, команда `bra z, label1` выполнит переход на метку `label1` программы, если установлен бит Z в регистре состояния процессора.

Кроме того, команда `bra` имеет две модификации, позволяющие выполнять безусловный переход:

- `bra` *метка*, где метка должна находиться в пределах от –32768 до +32767 слов памяти программ относительно команды bra;
- `bra Wn`, где Wn — рабочий регистр.

К командам переходов относятся и команды вызова процедур `call` и безусловного перехода `goto`. Инструкции `call` и `goto` в качестве операнда могут использовать либо метку, либо один из рабочих регистров. Если в качестве адреса перехода использовать содержимое какого-либо из рабочих регистров, то с помощью одной инструкции `bra` или `call` можно организовать несколько ветвлений в программе, например, по принципу оператора `switch...case` языка Си. Рассмотрим несколько примеров подобного использования команд `bra` и `call`.

Пример 1

```
.text
.global  __reset
__reset:
      mov      #0x1,W0
      mov      #handle(Sub1),W2
      btss     W0,#0
      mov      #handle(Sub2),W2
      call     W2
Loop1:
      bra      Loop1
Sub1:
      mov      #0x1111,W1
      return
Sub2:
      mov      #0xFFFF,W1
      return
.end
```

Программа работает следующим образом: с помощью команды `btss W0, #0` анализируется бит 0 регистра W0 и по результату анализа выбирается вариант продолжения. Если бит 0 установлен, то вызывается процедура `Sub1`, если же бит 0 сброшен, вызывается `Sub2`. Для того чтобы вызвать процедуру посредством адреса, содержащегося в рабочем регистре W2, выполняется команда

```
call  W2
```

Значение стартового адреса одной из процедур (`Sub1` или `Sub2`) записывается в регистр с помощью специального оператора `handle` в одной из команд

```
mov      #handle(Sub1),W2
mov      #handle(Sub2),W2
```

В нашем случае бит 0 регистра W0 принудительно устанавливается в 1, поэтому вызывается процедура `Sub1`.

Пример 2

```
.text
.global  __reset
__reset:
      mov      #0x0,W0
      mov      #handle(J1),W2
      btss     W0,#0
      mov      #handle(J2),W2
      goto     W2
J1:
      mov      #0x1111,W1
Loop1:
      bra      Loop1
J2:
      mov      #0xFFFF,W1
Loop2:
      bra      Loop2
.end
```

Это модифицированный вариант предыдущего примера. Здесь вместо вызовов процедур используются команды безусловного перехода `bra` и `goto`. Принцип формирования адреса остался прежним: адрес перехода формируется из меток `J1` и `J2` с помощью оператора `handle` и записывается в регистр W2. Переход на нужную ветвь программы выполняет команда `goto W2`.

3.3. Разработка и отладка программ на ассемблере в среде MPLAB IDE

Для тестирования всех наших программ на ассемблере мы будем использовать возможности среды разработки MPLAB IDE. На данном этапе нам понадобится ассемблер MPLAB ASM30 и отладчик (симулятор) MPLAB SIM. Все этапы разработки и отладки ассемблерных программ в среде MPLAB будут показаны на примере проекта, в котором будет использован исходный текст последнего примера из предыдущего раздела. В этом примере мы рассматривали использование команд передачи управления `bra` и `goto`.

Разработаем новый проект с помощью мастера проектов среды MPLAB IDE. Для этого запустим приложение MPLAB IDE и в меню **Project** выберем опцию **Project Wizard...** (**Рис. 3.3**).

Затем на Шаге 1 следует выбрать модель микроконтроллера, для которого будет разработан данный проект. В данном случае мы будем использовать

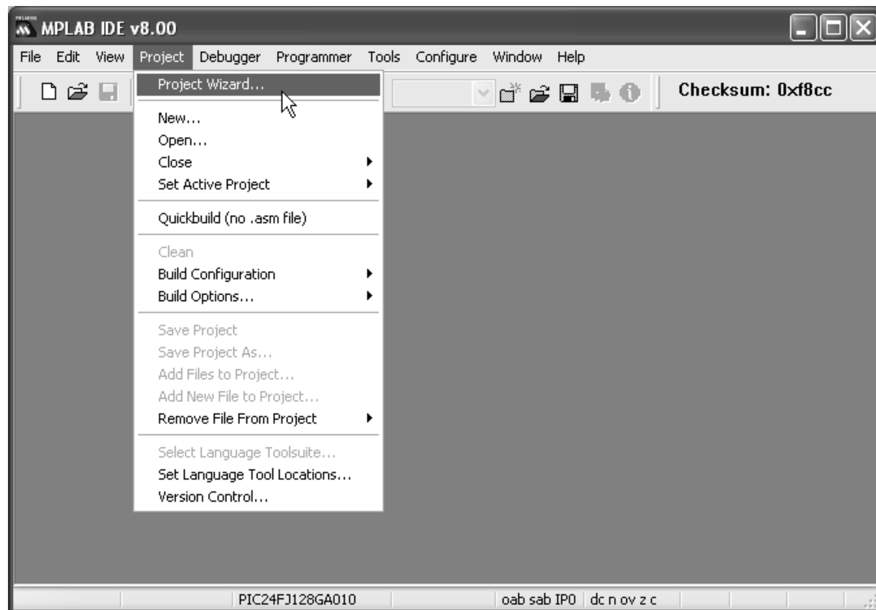

Рис. 3.3. Создание нового проекта в среде MPLAB IDE

PIC24FJ128GA010, поэтому в раскрывающемся списке следует выбрать именно это устройство (**Рис. 3.4**).

Рис. 3.4. Выбор микроконтроллера

На Шаге 2 мастер проектов предложит выбрать требуемый пакет инструментальных средств. Мы будем использовать ассемблер MPLAB ASM30, поэтому выбираем в раскрывающемся списке пункт «Microchip ASM30 Toolsuite» (**Рис. 3.5**).

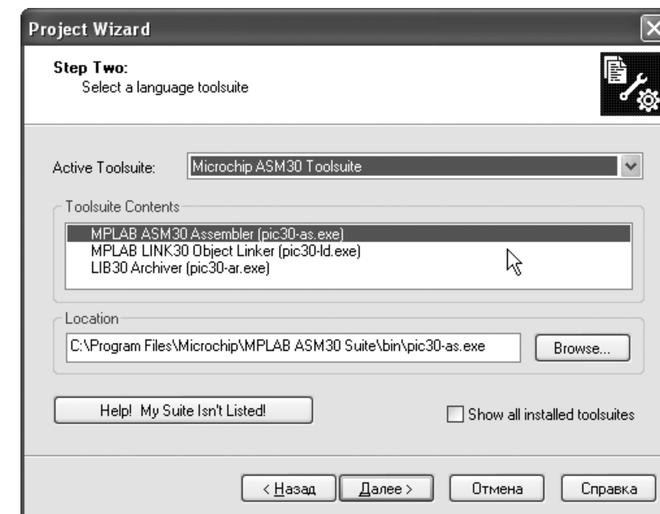

Рис. 3.5. Выбор компилятора MPLAB ASM30

На Шаге 3 следует выбрать имя проекта и каталог, в котором он будет располагаться (**Рис. 3.6**).

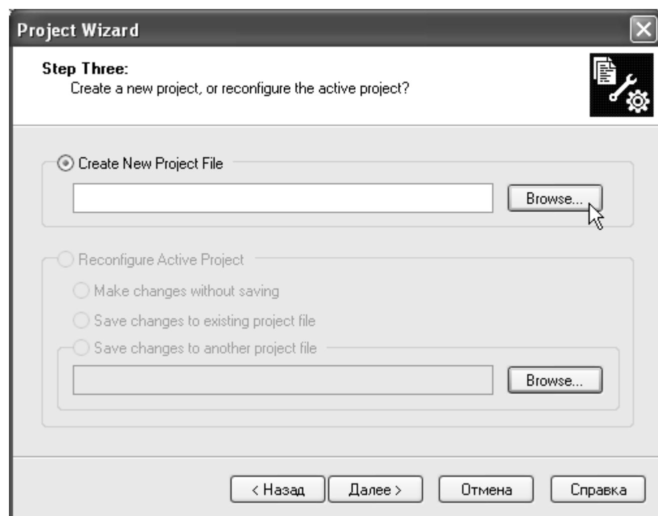

Рис. 3.6. Выбор имени и каталога проекта

Для выбора каталога и имени следует нажать верхнюю на рисунке кнопку **Browse** и, выбрав каталог, ввести имя проекта. В данном случае дадим проекту имя Project1 (**Рис. 3.7**).

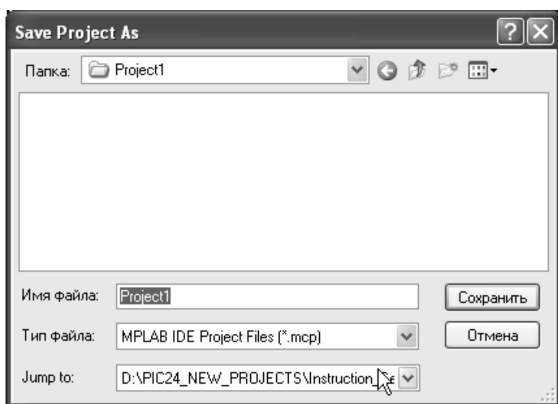

Рис. 3.7. Выбор имени и каталога сохраняемого проекта

После нажатия кнопки **Сохранить** файл проекта с расширением .mcp будет сохранен под именем Project1 и высветится в окне выбранного проекта (**Рис. 3.8**).

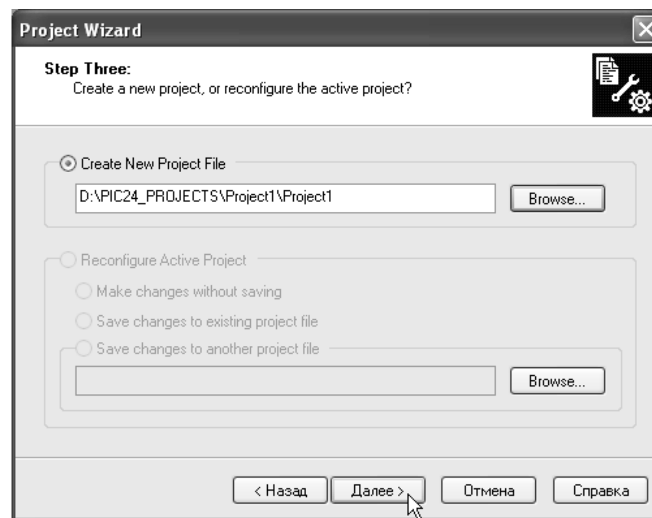

Рис. 3.8. Окно диалога после создания проекта Project1

Еще раз убедимся, что имя проекта и каталог размещения нас устраивают, и нажмем кнопку **Далее**. Созданный проект не содержит ни файлов с исходными текстами программ, ни файлов с библиотеками функций. На следующем, 4-м шаге мастер предложит включить в проект дополнительные файлы (**Рис. 3.9**).

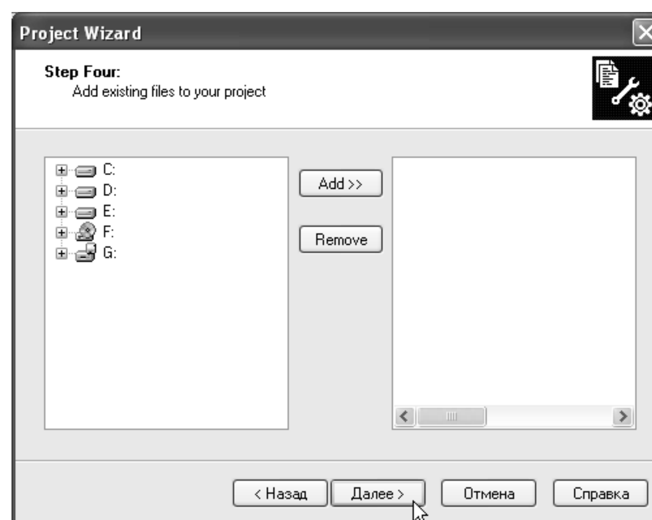

Рис. 3.9. Окно включения дополнительных файлов в проект

Для нашего проекта мы пока не создали ни одного файла и ничего включать не собираемся, поэтому пропустим этот шаг, нажав кнопку **Далее**. После этого в следующем окне, если параметры проекта нас устраивают, нажмем кнопку **Готово** (**Рис. 3.10**).

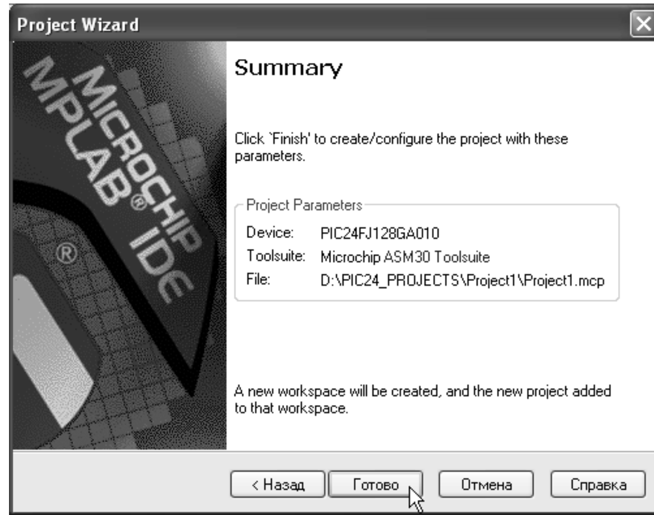

Рис. 3.10. Завершение создания проекта

В окне приложения MPLAB IDE, куда мы возвращаемся после создания проекта, в меню **View** выберем опцию **Project** и посмотрим, что включено в состав проекта (**Рис. 3.11**).

Кроме файла проекта Project1.mcp рабочее пространство пока не содержит файлов. Создадим новый файл, в который включим исходный текст программы на ассемблере. Для этого в меню **File** выберем опцию **New** (**Рис. 3.12**).

В появившемся окне редактирования наберем исходный текст, который сохраним в каталоге проекта в файле project1.s (**Рис. 3.13**).

Для того чтобы включить файл project1.s в проект, нужно установить отметку **Add File To Project**. Хотя, даже если забудете это сделать, сохраненный файл в любой момент можно будет включить в проект, выбрав опцию **Add New File To Project** из меню **File**.

В исходный текст программы с помощью директивы `include` обязательно нужно включить заголовочный файл для данной модели:

```
#include  "P24FJ128GA010.inc"
```

Кроме того, в проект нужно включить файл сценария компоновщика для данной модели микроконтроллера. Файл имеет имя, совпадающее с обозначением модели, и расширение .gld, а располагается в каталоге \...\Program Files\ Microchip\MPLAB ASM30 Suite\Support\gld. После включения файлов в проект окно проекта должно выглядеть так, как показано на **Рис. 3.14**.

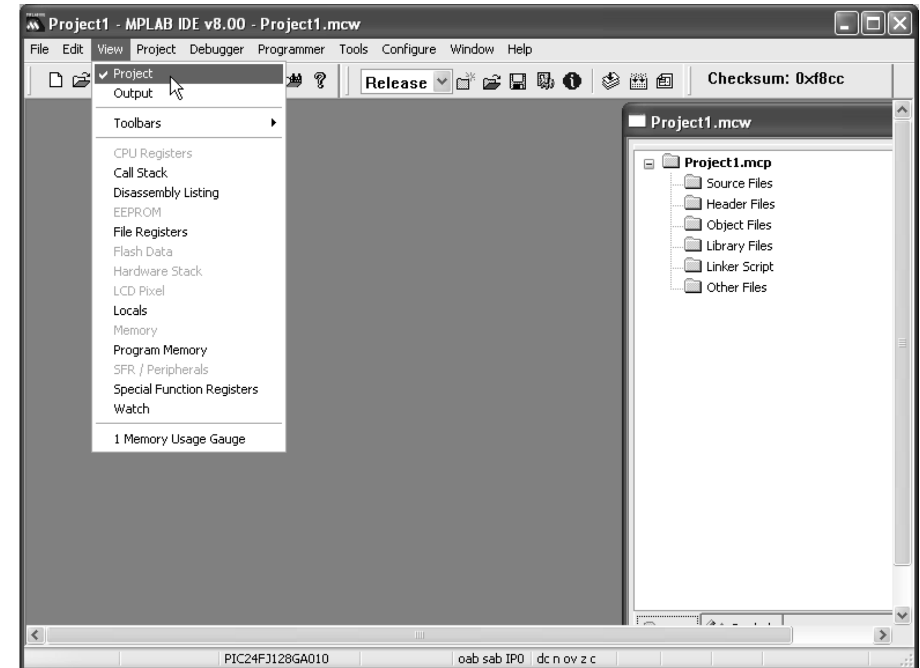

Рис. 3.11. Окно созданного проекта

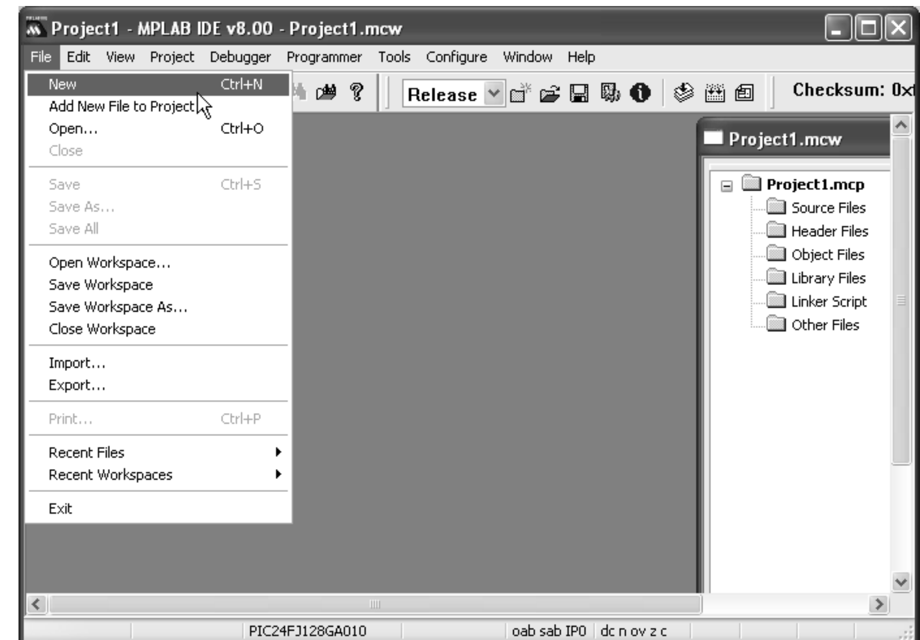

Рис. 3.12. Создание нового файла в среде MPLAB IDE

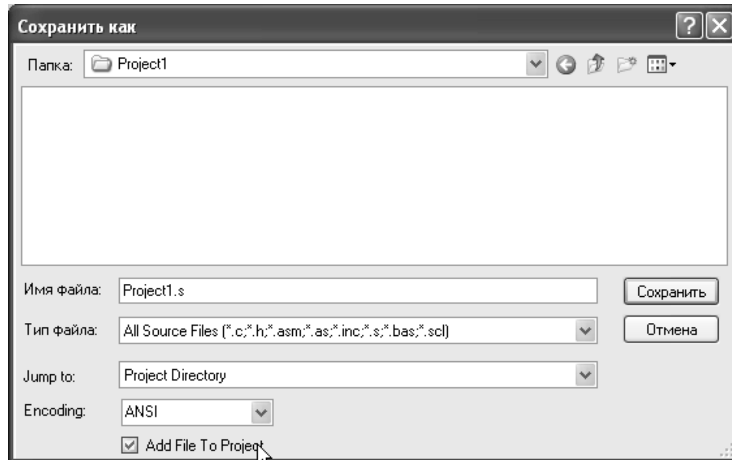

Рис. 3.13. Сохранение исходного текста на ассемблере

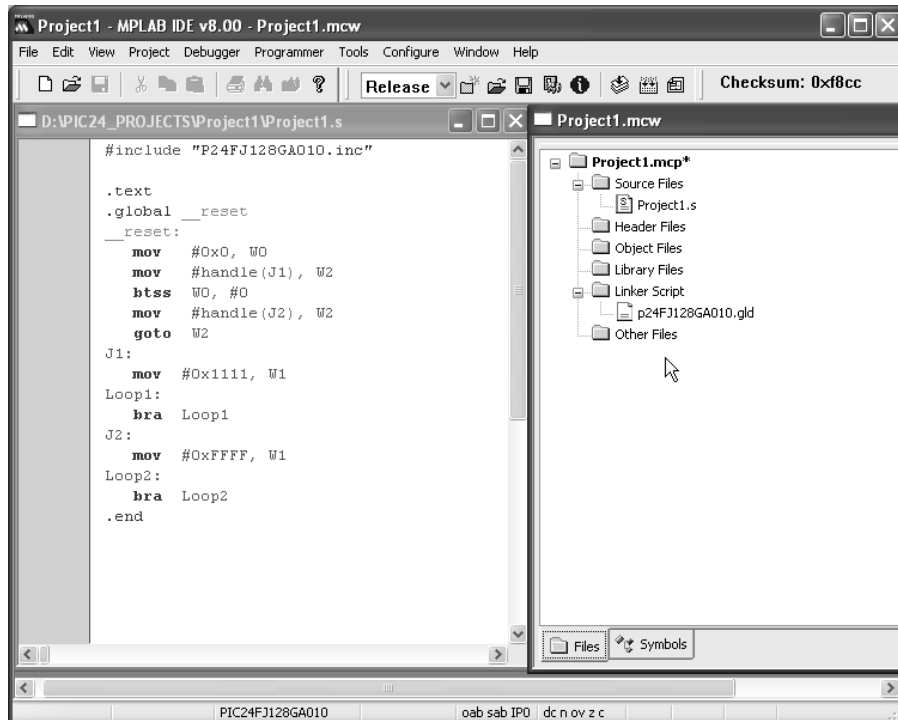

Рис. 3.14. Окно проекта после включения файлов

Сохраним проект и выполним сборку двоичного образа программы. После успешной компиляции и последующей компоновки в каталоге проекта должны появиться файлы с расширением .cof (elf) и .hex. HEX-файл представляет собой готовый к загрузке в память программ целевой системы двоичный образ программы, а COFF-файл представляет собой исполняемый файл, который можно использовать при отладке программы в среде MPLAB IDE. Для сборки приложения в меню **Project** нужно выбрать опцию **Build All**. Если сборка приложения прошла успешно, то можно посмотреть работу программы в отладчике. Для этого нужно выполнить несколько подготовительных действий. Во-первых, необходимо установить опцию **Debug** (**Рис. 3.15**).

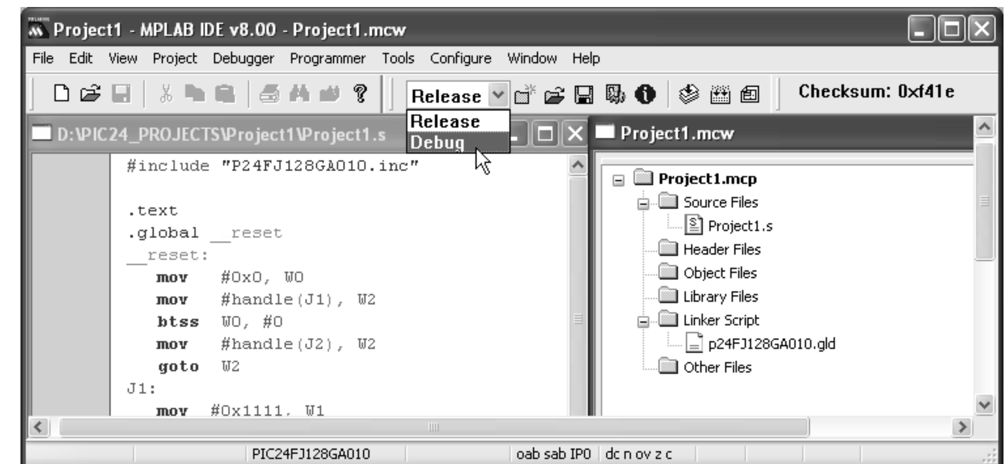

Рис. 3.15. Установка режима компиляции

Если эта опция была установлена до компиляции, можно перейти к следующему шагу. Если нет, то следует выполнить повторную сборку программы. Далее следует выбрать отладчик из меню **Debugger** (**Рис. 3.16**).

Выберем стандартный отладчик MPLAB SIM. После выбора отладчика появится дополнительная панель инструментов для отладки программы с кнопками управления процессом отладки (**Рис. 3.17**).

Теперь все готово для проверки работоспособности программного кода микроконтроллера. Прежде чем мы запустим приложение в отладчике, необходимо сделать одно важное замечание, которое касается достоверности результатов отладки. Программный отладчик (симулятор) никогда полностью не заменит тестирование программы в реальном устройстве, поскольку точно смоделировать работу электроники и обеспечить требуемые временные зависимости между сигналами схемы ни один симулятор никогда не сможет. Тем не менее с проверкой работоспособности программного алгоритма и обнаружением возможных ошибок в программе такой симулятор вполне справится. При

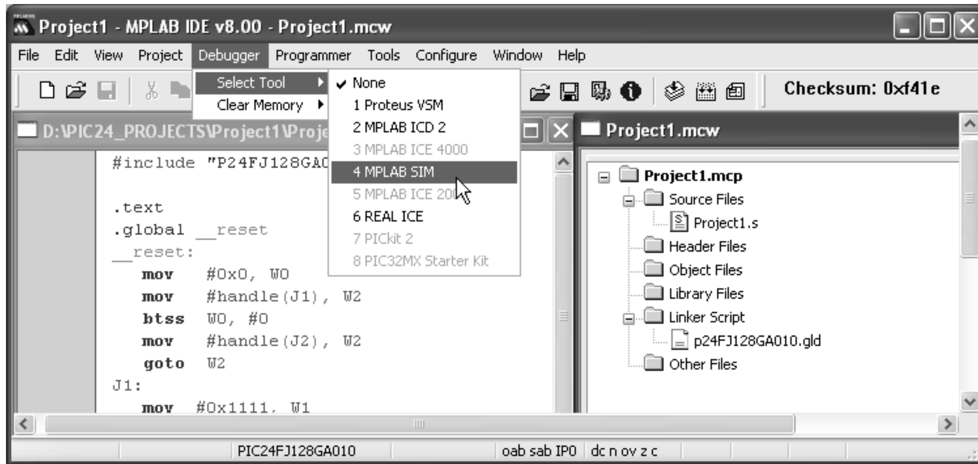

Рис. 3.16. Выбор отладчика

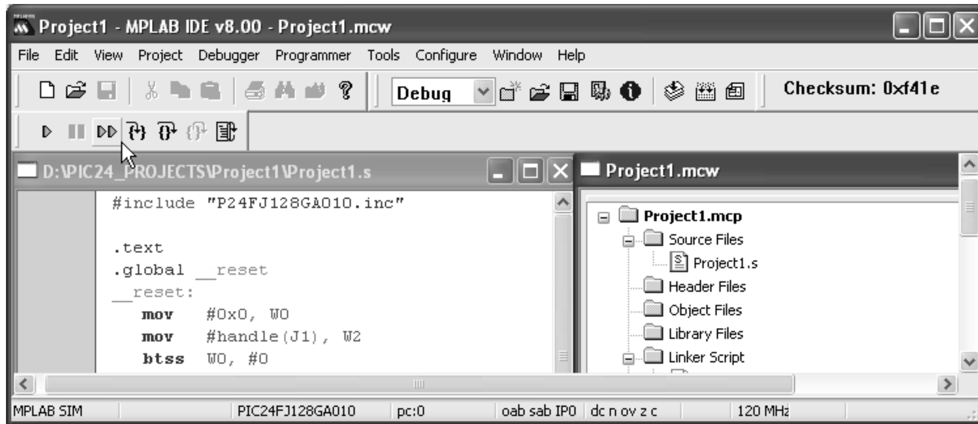

Рис. 3.17. Вид дополнительной панели инструментов отладчика

определенном опыте создания программ и их отладки симулятор сможет стать весьма эффективным средством улучшения качества программы.

Запуск на отладку программы можно выполнить по команде **Run** в меню **Debugger** либо воспользоваться «быстрой» клавишей F9. Можно выполнить отладку в режиме **Animate** — это позволяет визуально отслеживать изменения при выполнении команд программы. В дальнейшем при отладке программ мы будем использовать оба этих режима. Отладку программы можно проводить также в пошаговом режиме и с установкой точек прерывания — эти режимы мы рассматривать не будем, а читатели, интересующиеся деталями механизма отладки, могут обратиться к документации фирмы Microchip.

Перед запуском процесса отладки нужно определить, что именно нас интересует в конкретной программе. Для этого обратимся к исходному тексту, который показан ниже:

```
#include  "P24FJ128GA010.inc"

.text
.global  __reset
__reset:
      mov      #0x0,W0
      mov      #handle(J1),W2
      btss     W0,#0
      mov      #handle(J2),W2
      goto     W2
J1:
      mov      #0x1111,W1
Loop1:
      bra      Loop1
J2:
      mov      #0xFFFF,W1
Loop2:
      bra      Loop2
.end
```

В программе задействованы 3 рабочих регистра W0...W2, поэтому интересно будет отследить, как меняется их содержимое в процессе выполнения программы. Для наблюдения за состоянием регистров выберем опцию **Watch** в меню **View** (**Рис. 3.18**).

После выбора опции появится окно просмотра переменных, в котором можно выбрать регистры и переменные программы, состояние которых нужно отслеживать (**Рис. 3.19**).

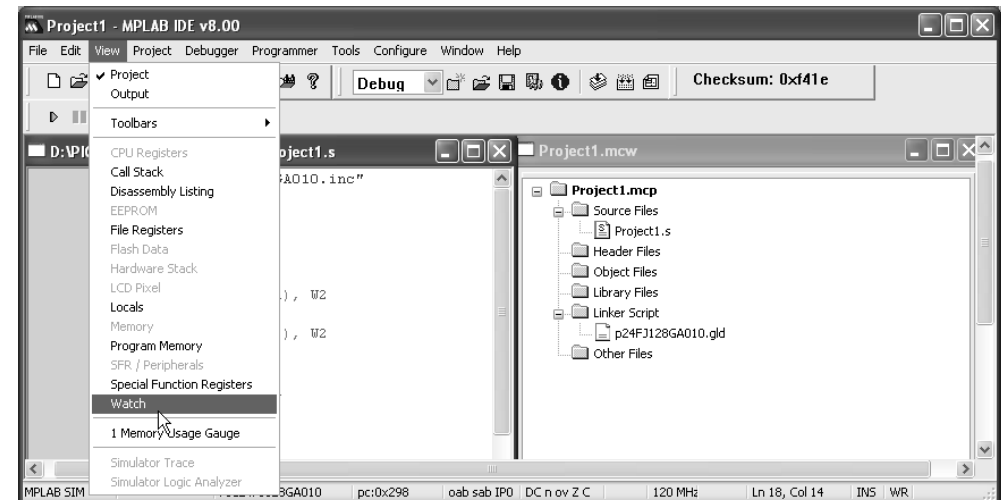

Рис. 3.18. Выбор опции Watch

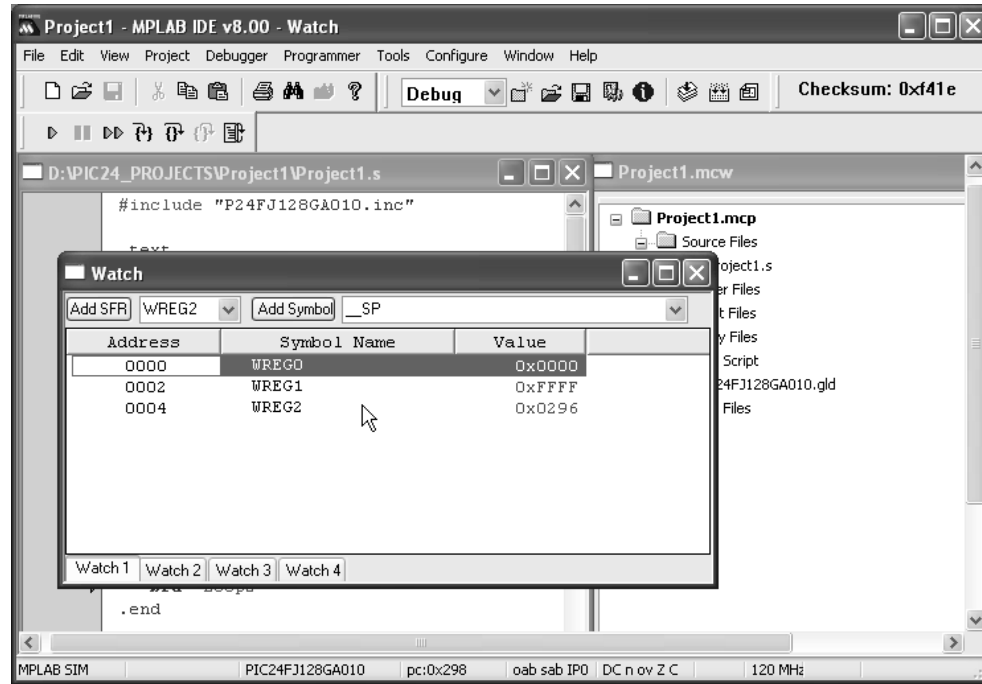

Рис. 3.19. Окно просмотра переменных

Для выбора регистра нужно открыть список в левой верхней части окна и, выбрав нужный регистр, щелкнуть на кнопке **Add SFR**. Под областью меню тут же появится строка состояния, в которой отображается информация о данном регистре. Рабочие регистры в окне просмотра имеют обозначения WREGx (x = 0, 1, 2...) и соответствуют регистрам Wx программы.

Перед отладкой программы желательно всегда делать «аппаратную» перезагрузку симулируемого микроконтроллера, поэтому выберем опцию **MCLR Reset** (**Рис. 3.20**).

После «перезагрузки» рабочие регистры сбросятся в 0. Теперь можно запустить программу на отладку, выбрав режим **Run** или **Animate**. Разница между этими режимами состоит в следующем: в режиме **Run** все команды программы выполняются по возможности с учетом временных параметров устройства, а в режиме **Animate** такая жесткая привязка по времени не делается, зато отображается проход по командам (операторам) программы. Анализ содержимого регистров и переменных программы, выполняющейся в режиме **Run**, можно провести, остановив программу либо установив специальные опции отладки (с этим способом мы познакомимся при изучении процесса отладки программ на языке Си). В режиме **Animate** изменения состояния переменных и регистров можно наблюдать в окне просмотра переменных в процессе выполнения программы, однако следует помнить, что в этом случае программа будет выполняться далеко не в режиме реального времени. Промежуточные состояния пе-

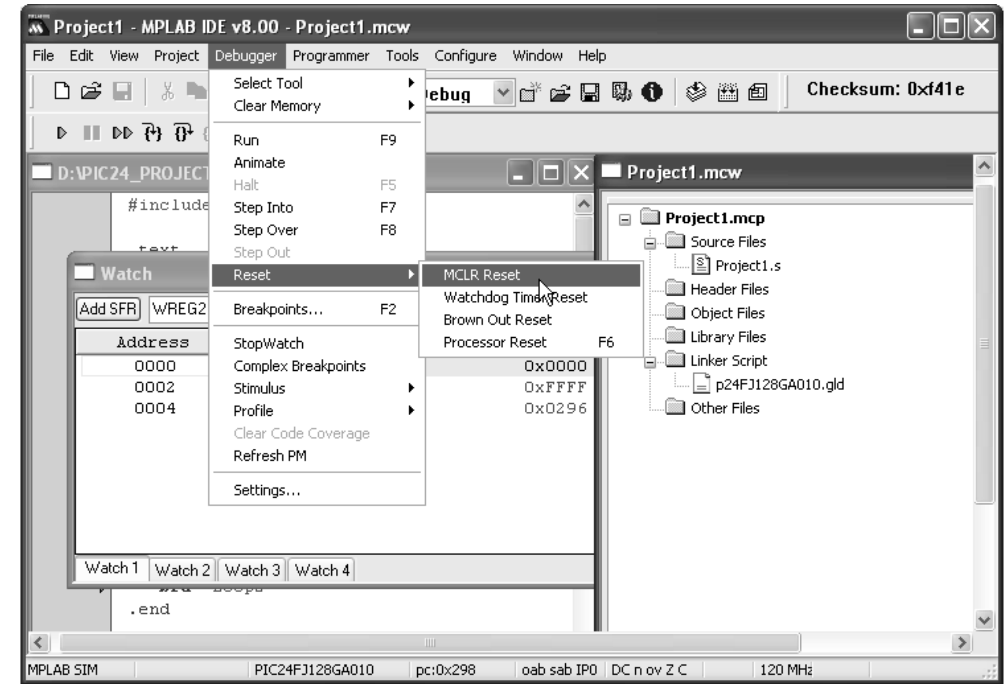

Рис. 3.20. Меню «перезагрузки» микроконтроллера

ременных можно наблюдать, установив в определенных местах программы точки прерывания или же перейдя в пошаговый режим работы.

Еще одно замечание: точность временных интервалов в режиме **Run** зависит от производительности процессора персонального компьютера, его загрузки и выбранной в окне настройки тактовой частоты микроконтроллера (**Рис. 3.21, 3.22**).

Тем не менее, не следует полагать, что если в отладчике установить определенную частоту процессора (например, 8 МГц, как показано на **Рис. 3.22**), то все команды программы и временные зависимости будут точно соответствовать этой частоте. Но в любом случае, временные соотношения между интервалами выполнения команд, вызовы и обработка прерываний и т.д. будут выполняться по возможности максимально точно. Если процессор персонального компьютера загружен выполнением других программ, то отклонения от реального режима при отладке программы в режиме **Run** будут больше.

Вернемся к отладке нашей программы.

Программа выполнится довольно быстро и войдет в «вечный» цикл по метке `Loop2`, а содержимое рабочих регистров будет таким, как показано на **Рис. 3.23**.

Как видно по результатам отладки, регистры W0 и W1 содержат значения, которые мы и должны были получить при запуске программы. Что же касается регистра W3, то в него помещается адрес метки `J2`, который зависит от настроек компилятора и компоновщика, выполняющего сборку программы.

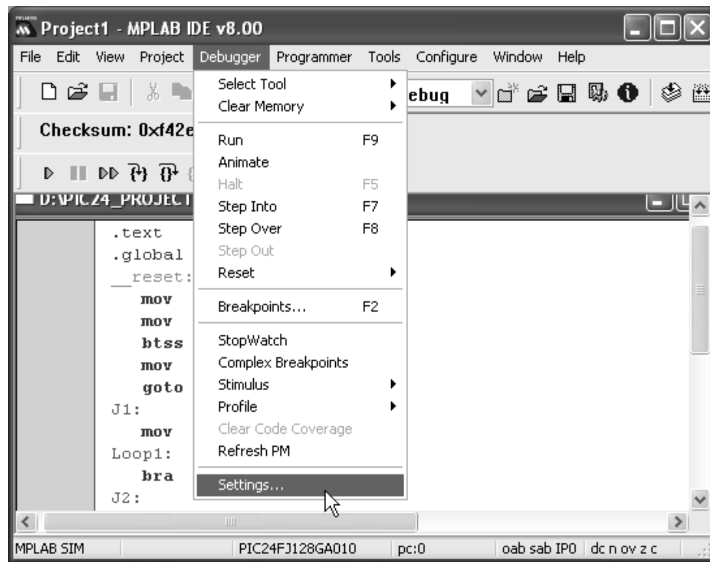

Рис. 3.21. Выбор опции настроек отладчика

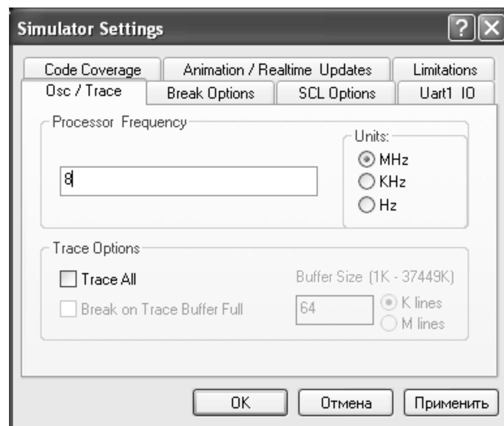

Рис. 3.22. Установка тактовой частоты микроконтроллера в отладчике

Для разработки сложных программ намного больше возможностей предоставляет компилятор языка Си фирмы Microchip, известный под названием MPLAB C для PIC24. Весь последующий материал этой книги будет опираться на методы разработки и отладки программного обеспечения на языке Си. В следующем разделе мы коснемся основных методов разработки и отладки программного обеспечения с использованием данного компилятора.

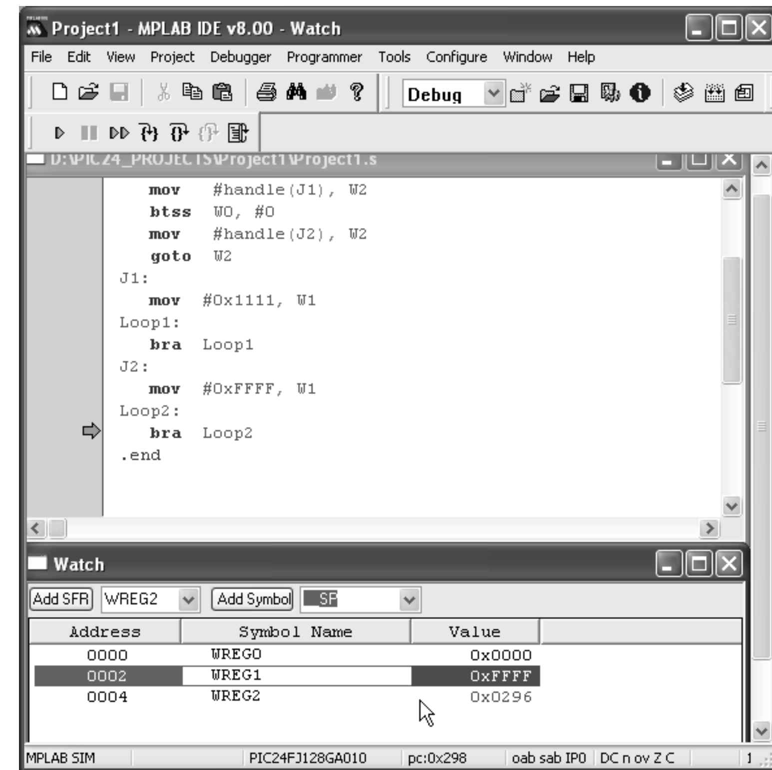

Рис. 3.23. Состояние рабочих регистров после отладки

3.4. Особенности разработки и отладки программ на MPLAB C для PIC24

Компилятор MPLAB C для PIC24 представляет собой эффективный программный инструмент разработки приложений для 16-битных микроконтроллеров фирмы Microchip, базирующийся на стандарте ANSI C.

Компилятор MPLAB C для PIC24 выполняет компиляцию файлов исходных текстов, создавая промежуточные ассемблерные и объектные файлы. Полученные объектные файлы затем компонуются с другими объектными файлами и библиотеками и собираются компоновщиком (линкером) в исполняемый файл программы. Кроме того, объектные файлы в формате COFF или ELF можно использовать при создании библиотек, либо загрузить в отладчик среды MPLAB IDE для дальнейшего тестирования и отладки. С помощью утилиты преобразования форматов можно преобразовать COFF или ELF-файл в формат HEX-файла, пригодный для загрузки в память программ целевой системы. На **Рис. 3.24** показаны все этапы разработки программы с использованием MPLAB C для PIC24.

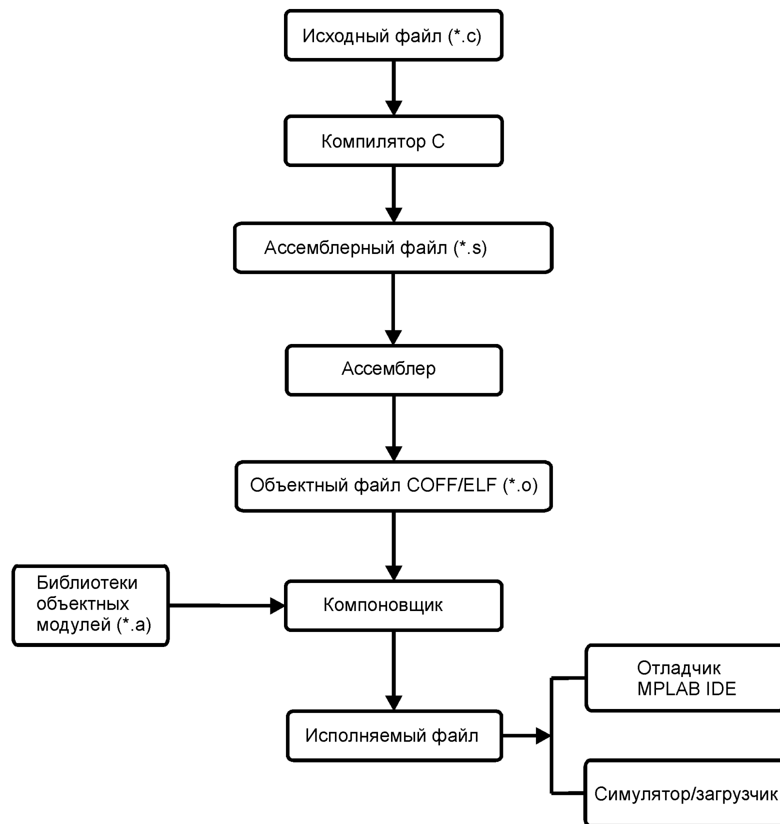

Рис. 3.24. Этапы сборки программы

Компилятор MPLAB C для PIC24 включает средства оптимизации программ при трансляции стандартных ANSI C программ в ассемблерный код микроконтроллера PIC24. В компилятор включены многочисленные расширения, позволяющие эффективно работать с периферийными модулями микроконтроллеров, а также дополнительные библиотеки с готовыми функциями для работы с периферией.

При разработке демонстрационных программ этой книги мы будем использовать среду разработки MPLAB IDE фирмы Microchip и студенческую версию компилятора C30. В этом разделе мы коснемся практических аспектов разработки программ для PIC24, в частности, отладки программ с помощью симулятора MPLAB SIM. Отладчик MPLAB SIM предоставляет намного больше возможностей для отладки и трассировки Си-программ по сравнению с приложениями на ассемблере, что позволяет успешно разрабатывать и тестировать довольно сложные приложения, написанные на языке Си. Мы не будем касаться особенностей синтаксиса языка C30, поскольку он подробно описан в фирменной технической документации, а уделим основное внимание именно

практической стороне разработки. Везде, где это нужно, будет даваться краткое пояснение работы того или иного элемента языка.

Практическое знакомство с MPLAB C для PIC24 начнем с разработки и отладки простого приложения на Си для микроконтроллера PIC24FJ128GA010. Для этого с помощью мастера приложений создадим новый проект точно так же, как это делали для программы на ассемблере. Единственное отличие состоит в том, что на Шаге 2 теперь следует выбрать пакет «Microchip C30 Toolsuite» (**Рис. 3.25**).

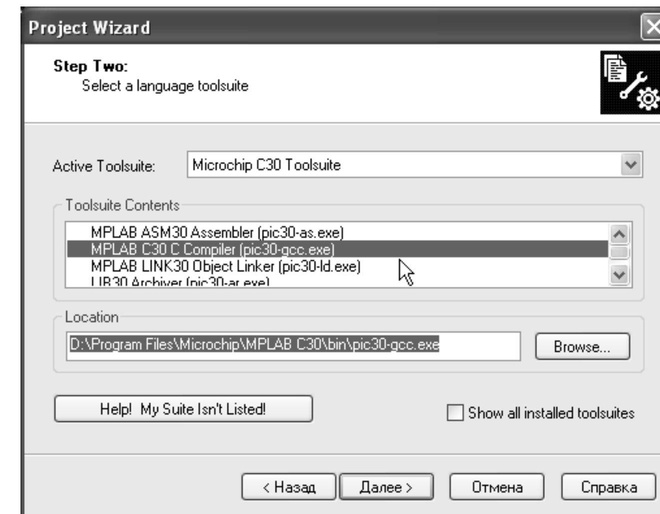

Рис. 3.25. Выбор компилятора проекта

В созданный проект необходимо добавить два файла: файл с исходным текстом на Си и файл сценария компоновщика с расширением .gld, который для данного устройства называется p24fj128ga010.gld и находится в каталоге \...\Program Files\Microchip\MPLAB C30\support\PIC24F[1]. Наше первое приложение будет периодически переключать вывод 0 (бит 0) порта A в противоположное состояние. Создадим и включим в проект файл с расширением .c и следующим исходным текстом:

```
#include  <p24fj128ga010.h>

void  Delay(void)
{
  unsigned  long  cnt  =  10000;
  while  (cnt  !=  0)  cnt-;
}
```

[1] В принципе, файл сценария компоновщика можно и не включать в проект — в этом случае на этапе сборки программы будет автоматически использован файл сценария, соответствующий выбранной модели. — *Примеч. науч. ред.*

```
void  main(void)
{
  TRISA  =  0xfffe;
  PORTAbits.RA0  =  1;

  while(1)
    {
      PORTAbits.RA0  =  ~PORTAbits.RA0;
      Delay();
    }
}
```

Бит 0 порта А здесь переключается в цикле `while(1)` с помощью оператора `PORTAbits.RA0 = ~PORTAbits.RA0`. Задержка между переключениями формируется функцией `Delay`, а величина этой задержки определяется значением счетчика `cnt` в самой функции. Поскольку мы будем выполнять отладку в программном симуляторе MPLAB SIM, то начальное значение `cnt` можно выбрать достаточно большим, чтобы можно было наблюдать изменения состояния бита 0 порта А визуально.

Компиляцию программы нужно выполнить в режиме **Debug**, а для наблюдением за состоянием порта А, вернее, его младшего бита в раскрывающемся списке окна просмотра переменных следует выбрать **PORTA** (**Рис. 3.26**).

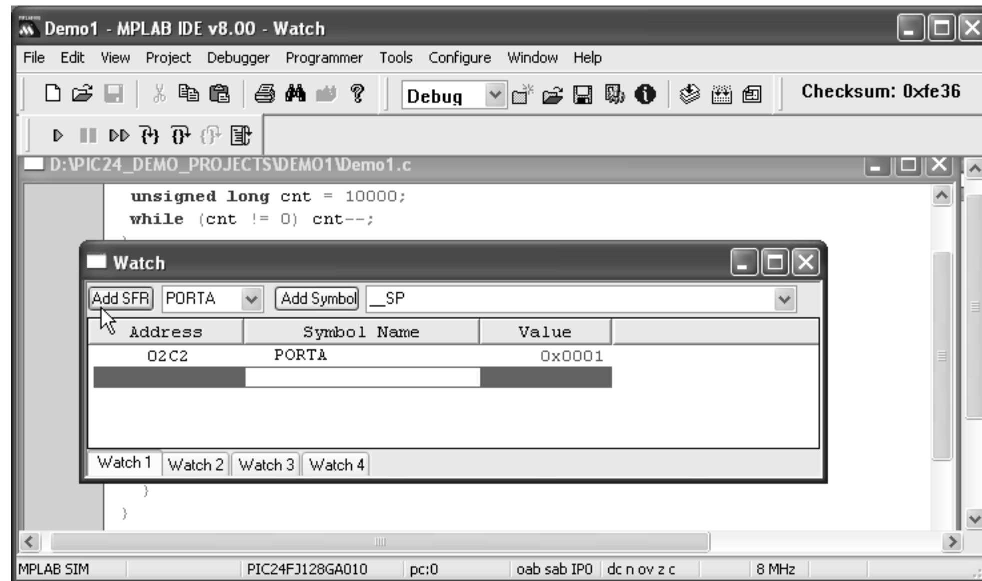

Рис. 3.26. Выбор порта А для просмотра

Изменения состояния бита 0 лучше всего наблюдать в режиме **Animate** отладчика, хотя можно запустить программу и в режиме **Run**, а затем остановить и запустить ее несколько раз для анализа состояния бита. Также можно использовать пошаговое выполнение программы, хотя это потребует больше времени.

Это далеко не самая сложная программа, поэтому процесс отладки довольно очевиден. Намного сложнее протестировать программу, которая должна работать с внешними прерываниями или, например, с точными временными интервалами.

Методы тестирования и отладки подобных программ предполагают хорошие знания аппаратно-программной архитектуры микроконтроллера и возможностей программного симулятора MPLAB SIM. Создадим в MPLAB IDE проект, в котором приложение будет переключать тот же бит 0 порта А, но уже в обработчике прерывания Таймера 1. Таймеры и их применение мы будем детально рассматривать в следующих главах, здесь для демонстрации нам будет достаточно основных сведений.

В микроконтроллерах PIC24F имеется несколько таймеров, но мы будем использовать 16-битный Таймер 1 (его часто называют таймером А-типа). Интервал срабатывания Таймера 1 в режиме таймера при заданной тактовой частоте процессора определяется двумя параметрами: значением, содержащимся в регистре периода PR1, до которого регистр таймера TMR1 инкрементируется, и коэффициентом деления предделителя, который может быть равен 1, 8, 64 или 256. Интуитивно понятно, что, чем больше значение регистра PR1 и коэффициент предделения, тем больше интервал срабатывания Таймера 1. При превышении значения PR1 устанавливается флаг прерывания _T1IF, и если разрешено прерывание Таймера 1, то вызывается функция-обработчик этого прерывания.

Вот исходный текст программы, в которой используется Таймер 1:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

void  __attribute__  ((__interrupt__))  _T1Interrupt(void)
{
  _T1IF  =  0;
  PORTAbits.RA0  =  ~  PORTAbits.RA0;
}

void  main(void)
{
  TRISA  =  0xfffe;
  PORTAbits.RA0  =  1;

  _T1IF  =  0;
  _T1IE  =  1;

  PR1  =  15625  *  3;
  TMR1  =  0;
  T1CON  =  0x8030;

  while(1)
    {
    }
}
```

В начало этой программы включен макрос _CONFIG2, который определяет параметры источника тактового сигнала микроконтроллера. При отладке программы в симуляторе MPLAB SIM этот макрос можно не использовать, поскольку все необходимые настройки производятся в самом симуляторе. При работе с реальным оборудованием, в зависимости от аппаратного решения, выбранного для тактирования устройства, следует правильно указать константы этого макроса. Смысл констант, используемых для настройки источника тактового сигнала, описан в заголовочном файле устройства.

Программа должна переключать бит 0 порта A в противоположное состояние каждые 3 с, для чего мы используем прерывание Таймера 1 микроконтроллера PIC24FJ128GA010. Более подробно прерывания описаны в последующих главах, но понять, как происходит обработка прерывания в этом конкретном примере, несложно. Каждые 3 с инициируется прерывание Таймера 1, и управление передается функции-обработчику _T1Interrupt. В MPLAB C для PIC24 предопределены прототипы функций-обработчиков прерываний для каждого из возможных источников прерывания, поэтому можно воспользоваться таким обработчиком для Таймера 1. Все, что делает функция-обработчик в нашем случае, — это сбрасывает флаг прерывания _T1IF и переключает бит 0 порта A в противоположное состояние.

В основной программе мы должны настроить вывод 0 (бит 0) порта A как выход, что выполняется установкой соответствующего бита в регистре TRISA. Кроме того, нужно разрешить работу прерывания Таймера 1, что выполняет оператор _T1IE = 1. Флаг прерывания перед установкой разрешения прерывания должен быть сброшен (оператор _T1IF = 0).

Операторы

```
PR1  =  15625  *  3;
TMR1  =  0;
T1CON  =  0x8030;
```

устанавливают параметры Таймера 1. При отладке программы в симуляторе MPLAB SIM мы будем использовать тактовую частоту микроконтроллера 8 МГц. Кроме того, зададим коэффициент деления предделителя Таймера 1, равный 256. В этом случае регистр периода PR1 должен содержать значение 15625 × 3, что соответствует интервалу срабатывания 3 с. Перед запуском Таймера 1 очистим регистр таймера, а для запуска установим соответствующие биты в регистре управления T1CON.

Так работает эта программа. Для проверки работоспособности программы в отладчике необходимо выполнить сборку приложения в режиме **Debug**, выбрать в меню **Debugger** симулятор MPLAB SIM и, кроме того, выполнить несколько предварительных настроек в самом отладчике.

Поскольку мы собираемся симулировать работу программы в режиме, приближенном к режиму «реального» времени, то необходимо установить тактовую частоту микроконтроллера, равную 8 МГц, выбрав в меню **Debugger** опцию **Simulator Settings** и установив соответствующее значение в окне **Processor Frequency** (**Рис. 3.27**).

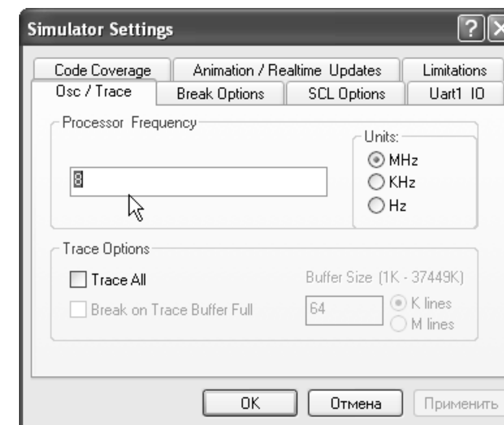

Рис. 3.27. Настройка тактовой частоты микроконтроллера

Кроме того, нужно настроить интервал обновления данных в окне просмотра переменных в режиме **Animate**. Для этого в окне **Simulator Settings** выберем закладку **Animation/Realtime Updates** и установим параметры обновления так, как показано на **Рис. 3.28**.

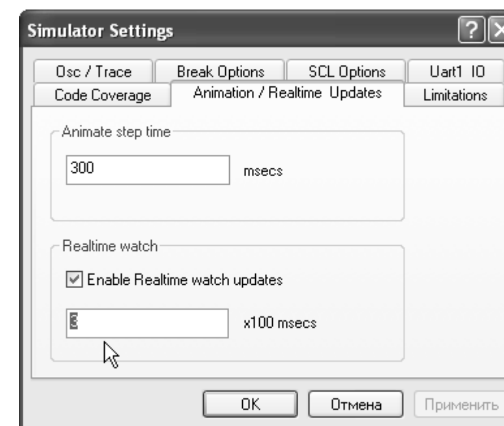

Рис. 3.28. Настройка параметров обновления

В данном случае мы выбрали интервал обновления данных, равный 0.3 с, хотя можно выбрать и другое разумное значение. Не следует выбирать значение интервала слишком малым, поскольку для вывода обновленной информации в окно просмотра требуется определенное время, к тому же процессор персонального компьютера может оказаться перегруженным, что само по себе замедлит работу всей системы и ни о каком «реальном» масштабе времени в этом случае говорить будет нельзя.

В данной программе нас будут интересовать изменения состояния Таймера 1 и порта А, поэтому в новом окне просмотра выберем из списка эти объекты для наблюдения (**Рис. 3.29**).

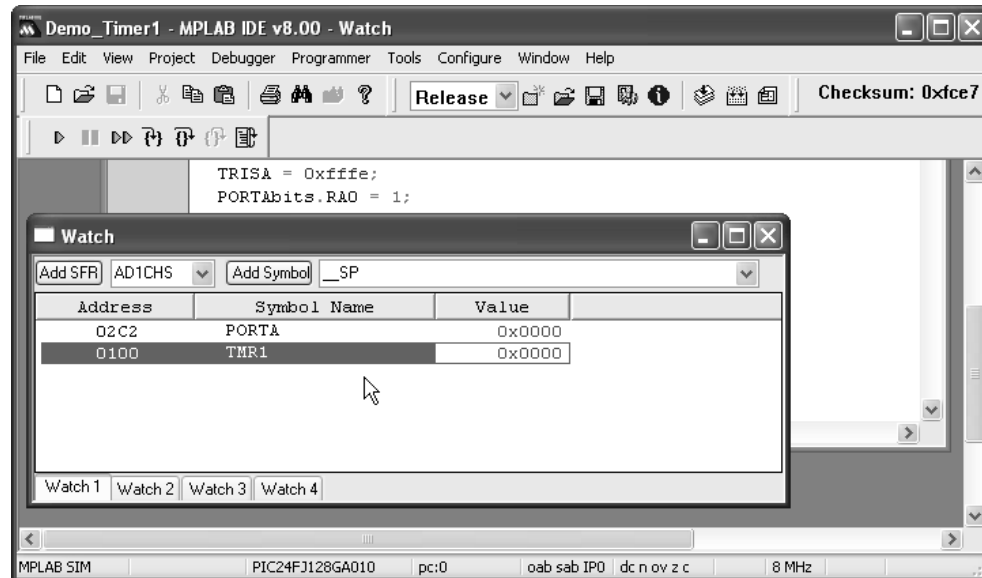

Рис. 3.29. Выбор объектов в окне просмотра

Теперь можно запустить программу в режиме **Animate** и наблюдать за изменениями значений регистра таймера TMR1 и порта А во времени.

До сих пор мы рассматривали примеры программ, в которых не было обработки внешних сигналов, например сигналов прерываний, поступающих на входы микроконтроллера. Подавляющее большинство приложений реального времени в процессе функционирования как раз и должны обрабатывать подобные события. Для тестирования таких программ в отладчике MPLAB SIM предусмотрены специальные средства, позволяющие промоделировать появление и обработку внешних событий. Как это делается, я покажу в следующем примере. Создадим новый проект в среде MPLAB IDE, в который включим файл на Си со следующим исходным текстом:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

void __attribute__ ((__interrupt__)) _INT3Interrupt(void)
{
  _INT3IF = 0;
  PORTAbits.RA0 = ~ PORTAbits.RA0;
}
```



```
void main(void)
{
  TRISA = 0xfffe;
  PORTAbits.RA0 = 1;

  _INT3IF = 0;
  _INT3IE = 1;

  while(1)
    {
    }
}
```

Эта программа переключает бит 0 порта А при возникновении прерывания INT3 (вывод RA14 порта А). Прерывание инициируется перепадом сигнала на выводе INT3 из НИЗКОГО уровня в ВЫСОКИЙ. Как и в предыдущем примере, в программе используется функция-обработчик прерывания, но с предопределенным именем `_INT3Interrupt`. В нее включены два оператора, первый из которых (`_INT3IF = 0`) сбрасывает флаг прерывания, а второй инвертирует бит 0 порта А.

В основной программе с помощью оператора `TRISA = 0xfffe` бит 0 конфигурируется как выход, а остальные (и среди них RA14) — как входы. Затем выход 0 порта А устанавливается в начальное состояние с высоким уровнем сигнала. Операторы

```
_INT3IF = 0;
_INT3IE = 1;
```

инициализируют прерывание INT3.

Компиляцию и сборку программы нужно выполнить в режиме **Debug** и выбрать отладчик MPLAB SIM. Для моделирования внешних событий в отладчике MPLAB SIM можно настроить стимулирующие воздействия, выбрав в меню **Debugger** опцию **Stimulus** (**Рис. 3.30**).

При выборе этой опции появится окно стимулятора, в котором можно выполнить настройку стимулирующих воздействий. Все настройки можно сохранить в файле стимуляции с расширением .sbs и использовать при повторной отладке. Выберем закладку **Asynch** и из раскрывающегося списка под заголовком **Pin/SFR** выберем **INT3** (**Рис. 3.31**).

Во второй колонке (**Action**) выберем опцию **Pulse High** — это означает, что на вывод INT3 (RA14) будет подан кратковременный перепад сигнала 0—1—0, длительностью 1 машинный цикл. Этого достаточно, чтобы сигнал прерывания был зафиксирован микроконтроллером и впоследствии обработан. Для генерации стимулирующего воздействия следует запустить приложение в режиме **Run** или **Animate** и нажать стрелку под заголовком **Fire** в окне стимулятора. Для наблюдения за состоянием порта А при стимуляции INT3 в окне просмотра переменных программы должна присутствовать строка PORTA.

В качестве стимулирующего воздействия можно задать и синхронный сигнал с определенной частотой (стимуляция синхронным сигналом). Это позволяет промоделировать, например, работу программы измерения временных пара-

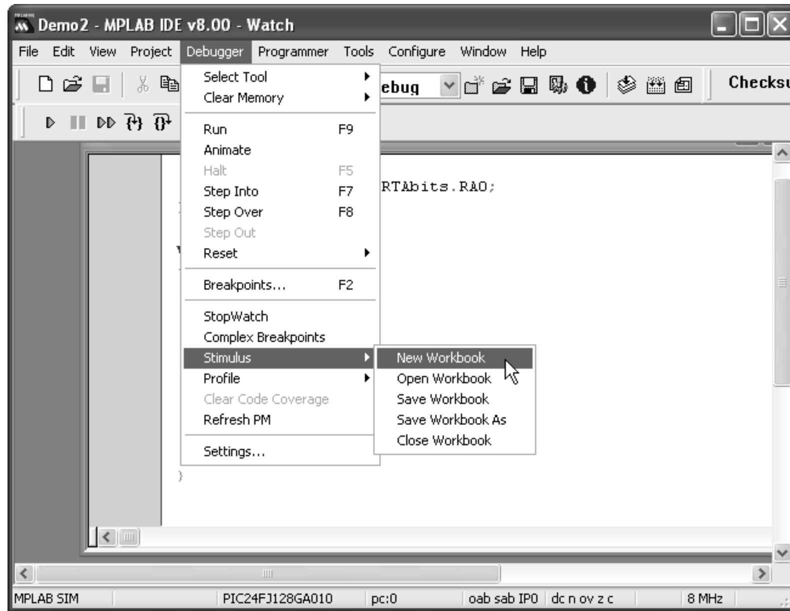

Рис. 3.30. Задание стимулирующих воздействий

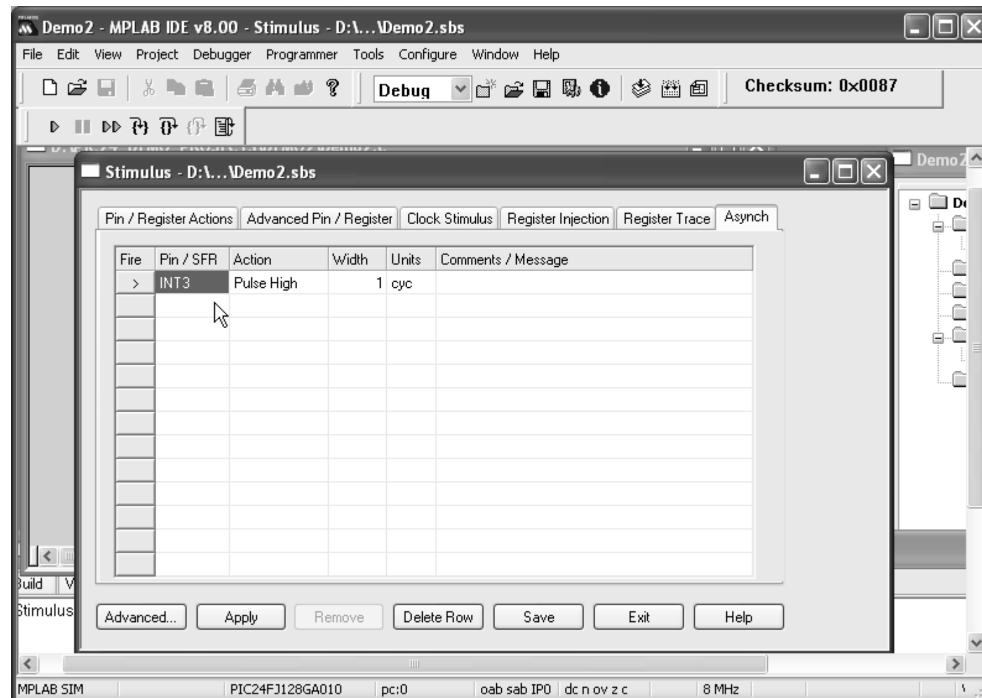

Рис. 3.31. Выбор асинхронного во времени сигнала стимуляции

метров внешних сигналов. Можно выбрать и довольно изощренный алгоритм стимуляции, позволяющий отлаживать намного более сложные программы.

Мы рассмотрим еще одно, очень полезное во многих случаях, свойство стимулятора — возможность вывода и отображения данных программы в «виртуальном» окне. Вывод данных можно осуществить в любой точке программы, используя хорошо знакомую всем программистам библиотечную функцию `printf`. Чтобы это сделать, нужно выбрать опцию **Settings...** меню **Debugger** и в окне **Simulator Settings** перейти к закладке **Uart1 IO** (**Рис. 3.32**).

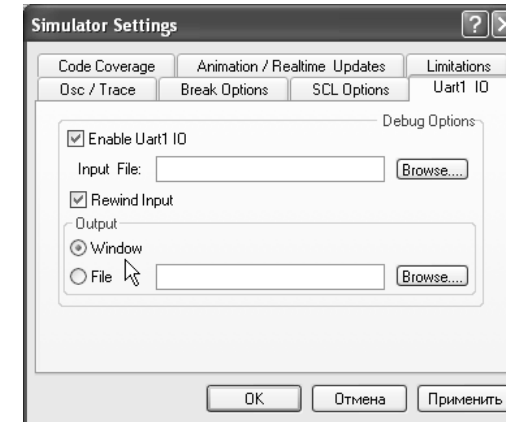

Рис. 3.32. Настройка виртуального терминала

В этом окне нужно установить опцию **Enable Uart1 IO**, а под заголовком **Output** отметить опцию **Window**. После установки этих опций в окно **Output** будут выводиться данные, присутствующие в функции `printf` где-либо в отлаживаемой программе. В самом окне **Output** после установки «виртуального» вывода появится закладка **SIM Uart1** (**Рис. 3.33**).

Для демонстрации «виртуального» вывода данных создадим новый проект и включим в него Си-файл с исходным текстом из предыдущего примера, который немного модифицируем, так, чтобы в виртуальное окно можно было выводить количество вызовов прерывания INT3.

Исходный текст нашей новой программы будет выглядеть следующим образом:

```
#include  <p24fj128ga010.h>
#include  <stdio.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

int  cnt  =  0;

void  __attribute__  ((__interrupt__))  _INT3Interrupt(void)
{
```

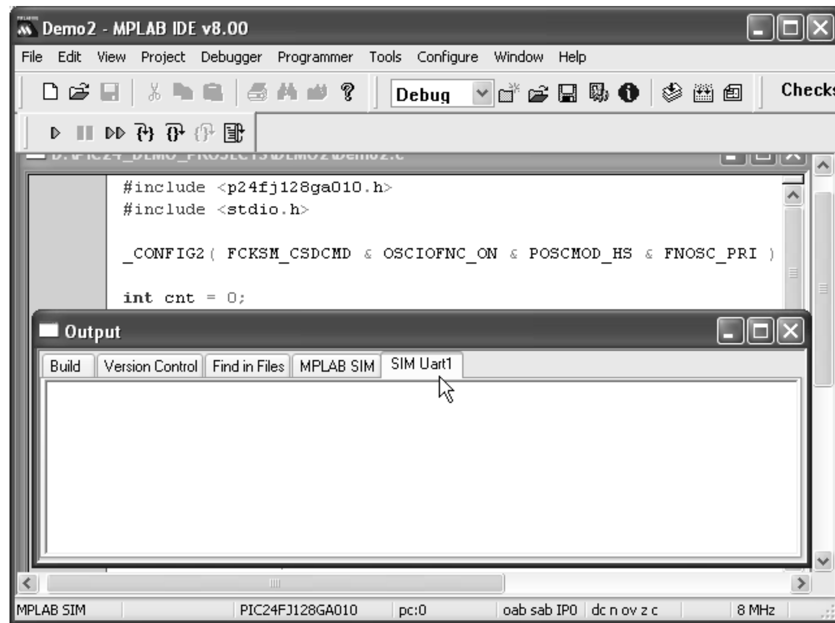

Рис. 3.33. Установка дополнительного окна вывода данных программы

```
  _INT3IF  =  0;
  PORTAbits.RA0  =  ~  PORTAbits.RA0;
  cnt++;
  printf("INT3  occured  %d  times!\n",  cnt);
}

void  main(void)
{
  TRISA  =  0xfffe;
  PORTAbits.RA0  =  1;

  _INT3IF  =  0;
  _INT3IE  =  1;

  while(1)
    {
    }
}
```

Для использования библиотечной функции `printf` в исходный текст нужно включить заголовочный файл `stdio.h`. Все изменения в тексте программы касаются только функции-обработчика `_INT3Interrupt`. При каждом вызове прерывания в «виртуальное» окно с помощью функции `printf` посылается строка, в которой отображается количество вызовов INT3 (переменная `cnt`). Использование функций ANSI C (в данном случае — `printf`) требует выделения неинициализированной области памяти, для чего перед компиляцией и сборкой программы необходимо выбрать опцию **Build Options**... в меню **Project** (**Рис. 3.34**).

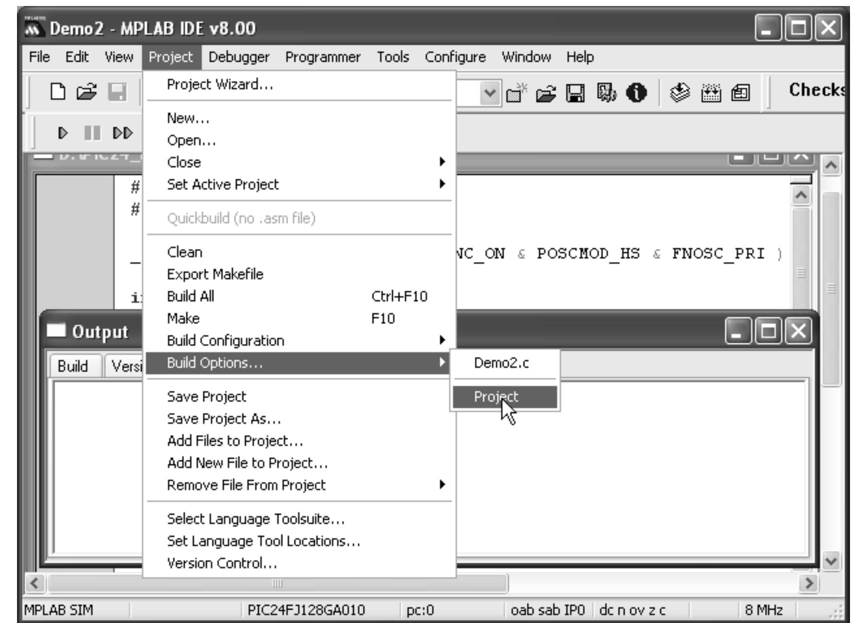

Рис. 3.34. Выбор настроек текущего проекта

Затем на странице **Build Options For Project**... нужно выбрать закладку **MPLAB LINK30** и установить подходящее значение в окне **Heap size**. В нашем случае большого объема памяти не требуется, поэтому установим размер «кучи», равный 512 байтам (**Рис. 3.35**).

После выполнения остальных необходимых настроек запустим программу на симуляцию. Окно работающего приложения может выглядеть так, как показано на **Рис. 3.36**.

Каждый раз при вызове прерывания INT3 (кнопка **Fire**) окна стимуляции в окне **Output** будет отображаться очередная строка.

Мы рассмотрели далеко не все возможности компилятора MPLAB C для PIC24 и отладчика MPLAB SIM. Но из приведенных примеров видно, что эти инструменты разработки являются весьма эффективными на первых этапах создания и тестирования программ. Обширная информация по применению инструментальных средств собрана в технической документации фирмы Microchip, к которой можно обратиться для более детального изучения данной темы.

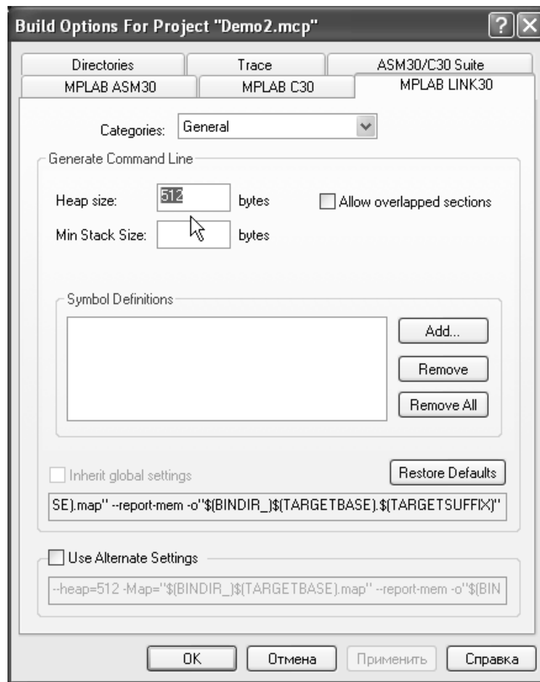

Рис. 3.35. Установка размера «кучи»

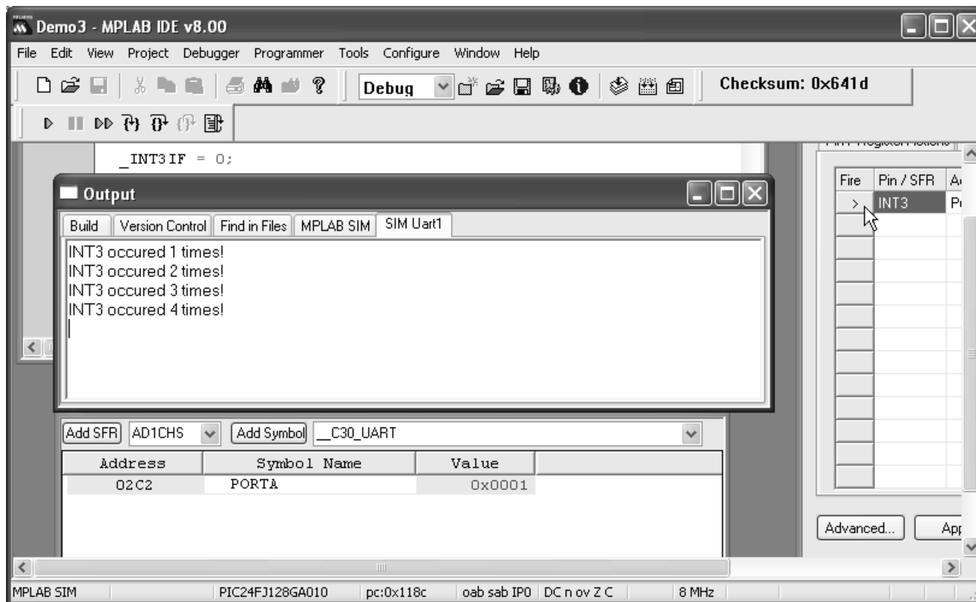

Рис. 3.36. Окно работающего приложения

ГЛАВА 4

ПРОГРАММИРОВАНИЕ ПОРТОВ ВВОДА/ВЫВОДА

Практически ни один проект с микроконтроллерами не обходится без операций ввода/вывода через цифровые порты. Микроконтроллеры семейства PIC24F предоставляют разработчику широкие возможности по работе с дискретными сигналами благодаря наличию достаточного количества портов ввода/вывода и некоторым функциям, расширяющим функциональность самих портов.

4.1. Аппаратно-программная архитектура портов ввода/вывода

Порты ввода/вывода общего назначения — это простейшие периферийные устройства, с помощью которых PIC-микроконтроллер может получать цифровые данные и управлять другими устройствами. Дополнительную гибкость при работе с портами ввода/вывода обеспечивает и то, что их можно настроить для выполнения альтернативных функций.

В общем виде функциональную схему стандартного порта ввода/вывода, который не используется периферийными модулями, можно представить так, как показано на **Рис. 4.1**.

Эта схема довольно точно отображает функционирование стандартного порта ввода/вывода. Порт может быть настроен на ввод или вывод цифрового сигнала с помощью защелки TRIS, которая представляет собой D-триггер. Если порт должен работать как выход, то на вход D защелки TRIS подается сигнал лог. 0, который по тактовому импульсу записывается в триггер. В этом случае инверсный выход TRIS-защелки разрешает работу схемы «И» и открывает тристабильный буфер, через который сигнал с выхода защелки данных (см. **Рис. 4.1**) проходит на вывод порта.

Если на вход D защелки TRIS подать сигнал лог. 1, то на инверсном выходе триггера появится 0, который запрещает прохождение сигнала с защелки данных на выход порта. В этом случае данные с вывода порта можно будет только читать. Здесь не следует смешивать чтение выходных данных порта и чтение вывода порта. Для чтения бита данных, записанного в порт, нужно прочитать содержимое защелки данных.

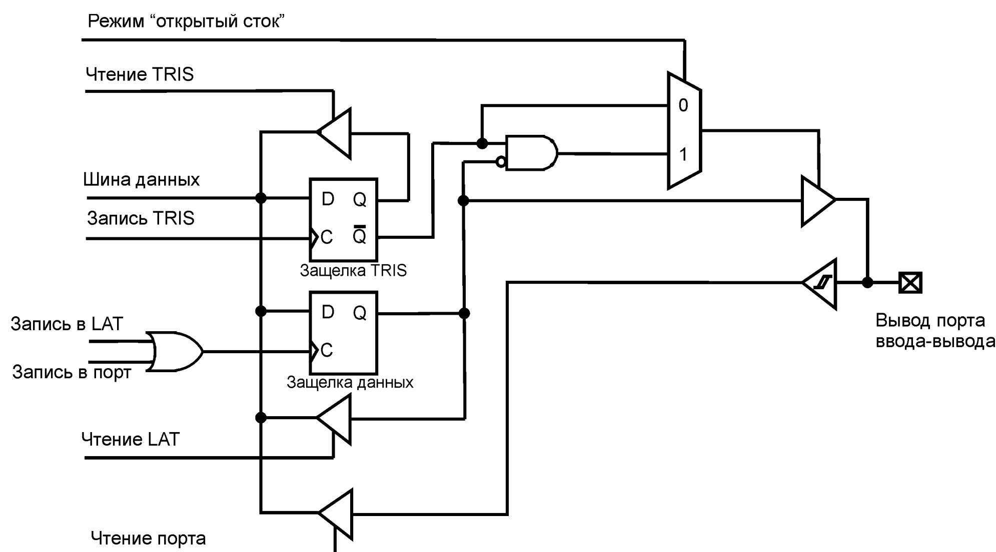

Рис. 4.1. Схема порта ввода/вывода

Каждый порт ввода/вывода управляется с помощью четырех регистров:

- TRISx — установка режима работы (чтение или запись) для порта PORTx (здесь x = A, B, C и т. д. и обозначает конкретный порт);
- PORTx — регистр ввода/вывода порта;
- LATx — регистр-защелка данных порта PORTx;
- ODCx — регистр режима работы с открытым стоком.

Каждому выводу порта ввода/вывода соответствует отдельный бит в регистрах TRISx, PORTx, LATx и ODCx.

Общее количество портов ввода/вывода и количество выводов зависит от конкретной модели микроконтроллера. Вполне возможно и то, что отдельные биты портов ввода/вывода в конкретном устройстве задействованы не будут. В любом случае лучше обратиться к технической документации на микроконтроллер. Рассмотрим назначение каждого из регистров порта ввода/вывода.

Регистр TRISx позволяет сконфигурировать направление обмена данными для каждого вывода порта. Если для какого-либо вывода порта в регистре TRIS на соответствующей позиции установлен 0, то вывод будет работать на передачу данных. Если же соответствующий бит в регистре TRIS установлен в 1, то вывод порта будет работать на прием (чтение).

Доступ к данным порта осуществляется посредством регистра PORTx. При чтении этого регистра считываются значения сигналов, присутствующих на выводах порта, а при записи этого регистра данные заносятся в защелку данных порта.

Многие команды микроконтроллера, такие, например, как BSET и BCLR, выполняют операции, известные как «чтение–модификация–запись». При этом

содержимое регистра-операнда (в данном случае регистра порта) вначале считывается, полученное значение модифицируется и затем обратно записывается в регистр. Если вывод порта сконфигурирован как выход (на запись), то никаких проблем не возникает. Однако если выводы порта сконфигурированы как входы, то здесь нужно соблюдать осторожность и записывать данные в порт до того, как направление обмена данными изменится с чтения на запись.

У каждого порта имеется регистр LATx, использование которого позволяет исключить проблемы, связанные с выполнением операций типа «чтение–модификация–запись». При чтении этого регистра возвращается значение бита, записанного в защелку порта, а не значение на выводе порта. Операции типа «чтение–модификация–запись», выполненные над регистрами LAT, дают правильный результат. Вкратце все вышеизложенное можно резюмировать следующим образом:

- при записи в регистр PORTx данные записываются в защелку порта;
- при записи в регистр LATx данные записываются в защелку порта;
- при чтении регистра PORTx данные читаются с вывода порта;
- при чтении регистра LATx данные читаются с защелки порта.

Каждый вывод порта может работать как в обычном режиме, так и в режиме с «открытым стоком». Для переключения выводов в этот режим используются регистры ODCx (Open Drain Control). Если соответствующий бит регистра ODCx установлен в 1, то вывод порта работает как выход с открытым стоком. Если же данный бит сброшен в 0, то вывод работает как обычный цифровой выход. При сбросе микроконтроллера все биты регистра ODCx сбрасываются в 0. Конфигурация с «открытым стоком» применяется в случаях, когда к выводу порта требуется подключить нагрузку, которая должна питаться от напряжения, большего или меньшего напряжения питания микроконтроллера.

Следует соблюдать осторожность при подключении устройств к «открытому стоку» с повышенным напряжением питания, поскольку это может повредить кристалл микроконтроллера. В таких случаях следует обращаться к документации на микроконтроллер.

Выводы портов микроконтроллеров семейства PIC24F можно сконфигурировать для работы как с цифровыми, так и с аналоговыми сигналами. При использовании порта для ввода цифровых данных входной сигнал поступает на TTL-буфер или триггер Шмита. При настройке порта на выход для передачи выходного сигнала используется буферный усилитель или же применяется конфигурация с «открытым стоком».

Один и тот же вывод порта ввода/вывода может использоваться и при работе периферийных модулей микроконтроллера, при этом он не может работать как обычный вывод порта. При этом в большинстве случаев вывод должен конфигурироваться как обычный вывод порта ввода/вывода, хотя некоторые периферийные устройства могут переопределять содержимое регистра TRIS. На **Рис. 4.2** показана функциональная схема такого «разделяемого» порта ввода/вывода, который используется периферийными модулями.

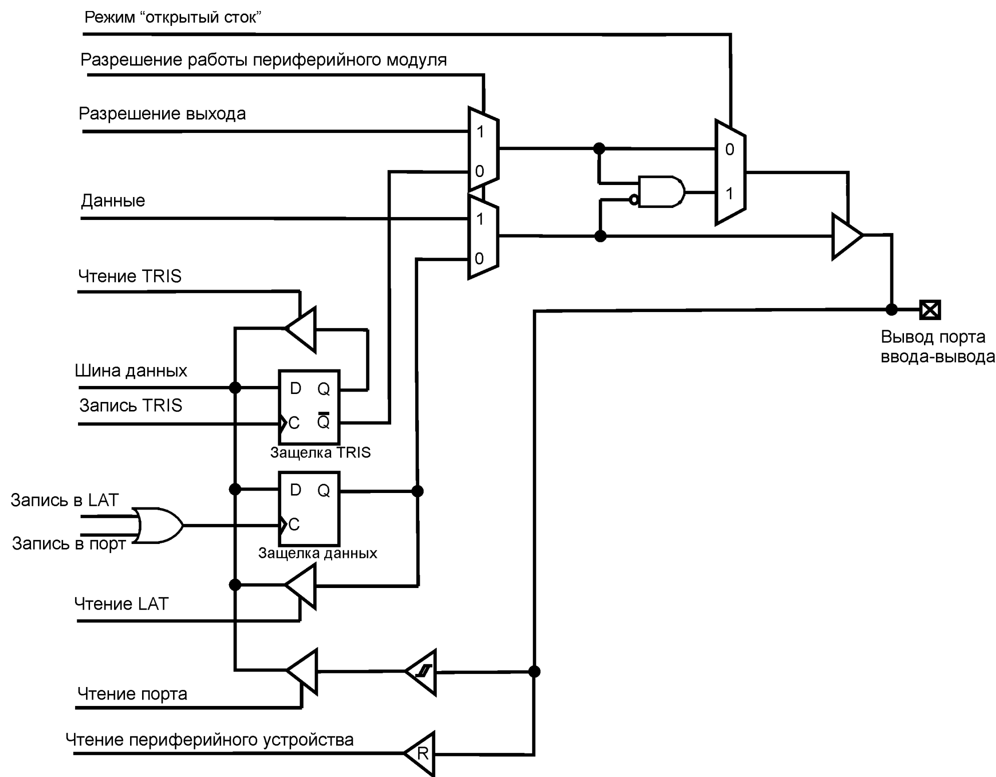

Рис. 4.2. Схема разделяемого порта ввода/вывода

При чтении разделяемого порта ввода/вывода нужно помнить несколько правил. Во-первых, если порт управляется периферийным модулем (**Рис. 4.2**), то состояние его вывода можно определить, прочитав регистр PORTx. Во-вторых, периферийный модуль может прочитать состояние вывода независимо от цифрового модуля ввода/вывода.

При разделении порта ввода/вывода, настроенного на передачу (запись) сигнала, между разными модулями следует помнить, что:

- если выход управляется периферийным устройством, то запись данных в регистр PORTx не возымеет никакого эффекта;
- данные можно прочитать из порта при помощи регистра PORTX;
- параметры выходного сигнала порта (токи нагрузки, скорость нарастания выходного сигнала и т. д.) определяются настройками периферийного модуля, который использует этот порт;
- для конфигурирования вывода порта в качестве выхода необходимо сбрасывать соответствующий бит в регистре TRISx;
- если выход порта может устанавливаться в тристабильное состояние, то это осуществляет сам периферийный модуль.

При работе порта ввода/вывода с модулем аналого-цифрового преобразования установки регистра TRIS не влияют на состояние порта. Для того чтобы использовать порт, используемый аналого-цифровым преобразователем, в качестве стандартного цифрового порта ввода/вывода, соответствующий бит в регистре AD1PCFG должен быть установлен в 1, независимо от того, включен модуль АЦП или выключен.

Следует сказать, что некоторые дополнительные функции, присвоенные портам ввода/вывода, не требуют перехвата управления выводом порта, как, например, в случае с модулем аналого-цифрового преобразователя. Если порт ввода/вывода должен работать как вход прерывания, вход синхронизации таймера или вход регистрации события, то достаточно установить соответствующий бит в регистре TRISx. При этом модули прерывания, таймера и регистрации событий не берут управление портом на себя.

Другие периферийные модули (SPI, I[2]C и UART) при функционировании полностью перехватывают управление портом ввода/вывода, так что конфигурирование такого порта посредством регистров PORTx и TRISx не требуется.

4.2. Программирование портов ввода/вывода

В этом разделе мы рассмотрим несколько проектов, в которых будут продемонстрированы характерные особенности программирования портов ввода/вывода, работающих в стандартном режиме обмена цифровыми данными. Аппаратная часть первого проекта показана на **Рис. 4.3**.

В этой схеме к выводу RA0 (бит 0 порта PORTA) подключен светодиод, который должен включаться/отключаться при ВЫСОКОМ уровне напряжения на выводе RA7 (бит 7 порта PORTA). В исходном состоянии порт RA0 настроен как вход.

Программная часть проекта разработана в среде MPLAB IDE с помощью мастера проектов. Для большей наглядности при разработке мы будем использовать язык ассемблера, поэтому в окне выбора инструментальных средств мастера проектов следует выбрать ассемблер MPLAB ASM30 (**Рис. 4.4**).

В пустой проект следует добавить файл с расширением .s (стандартное расширение для ASM-файла), поместив в него следующий исходный текст:

```
.include  "p24fj128ga010.inc"
.bss
cnt  :  .space  2
cnt1:  .space  2
.text
.global  __   reset
```

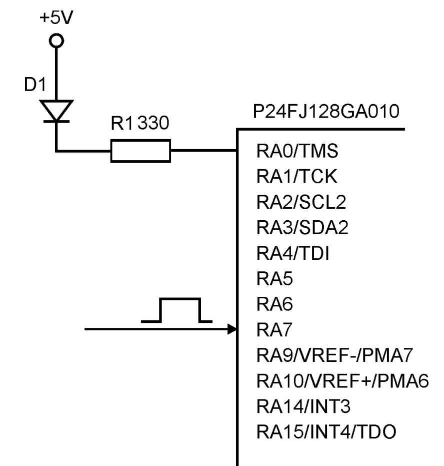

Рис. 4.3. Аппаратная часть проекта

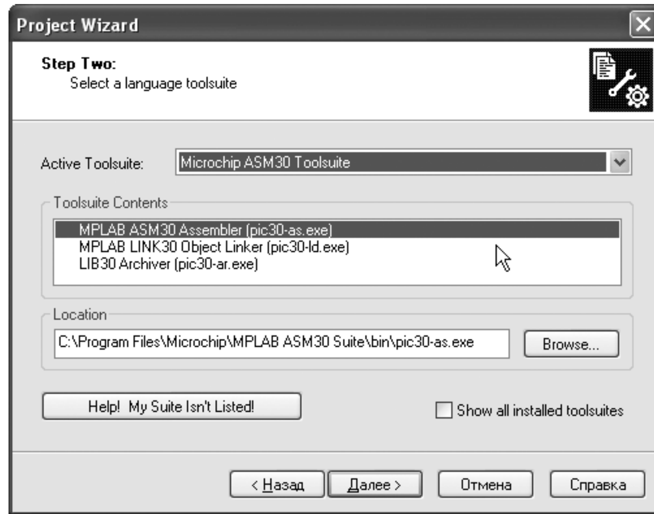

Рис. 4.4. Окно выбора компилятора

```
__reset:
    mov         #0xffff,w0
    mov         w0,cnt
    mov         #0xffff,w0
    mov         w0,TRISA
    bclr.b      PORTA,#0

wait:
    btss        PORTA,#7
    goto        wait
    btsc        TRISA,#0
    bclr        TRISA,#0

    mov         #2,0
    mov         w0,cnt1
Main_Loop:
    mov         #0xffff,w0
    mov         w0,cnt

Loop:
    dec         cnt
    bra         NZ,Loop
    dec         cnt1
    bra         NZ,Main_Loop

    btg.b       PORTA,#0

    goto        wait
.end
```

В этой программе переменные `cnt` и `cnt1` используются для формирования небольшой задержки переключения светодиода при удержании на входе RA7 ВЫСОКОГО уровня сигнала. Обе переменные имеют размер в одно слово (2 байта) и при запуске программы получают максимальные значения 0xffff. Впоследствии эти переменные декрементируются в циклах `Loop` и `Main_Loop`, формируя тем самым определенный интервал времени задержки. Микроконтроллер работает на частоте 8 МГц, так что задержка получается достаточно ощутимой, чтобы увидеть результат работы программы.

В начале программы все выводы порта А конфигурируются как входы:

```
mov      #0xffff,w0
mov      w0,TRISA
```

В этом случае запись в бит 0 любого значения не изменит состояния выхода, поскольку инверсный выход защелки TRIS (см. **Рис. 4.1**) сброшен в 0 и блокирует выходной буфер. Таким образом, команда

```
bclr.b    PORTA,#0
```

хоть и сбрасывает бит 0 защелки данных порта А, но на выход эти данные не попадают. По этой причине светодиод на выходе RA0 не включается, поскольку падение напряжения на нем практически равно 0.

Далее в цикле `wait` программа ожидает поступления сигнала ВЫСОКОГО уровня на вывод RA7 (бит 7 порта А):

```
wait:
    btss      PORTA,#7
    goto      wait
  .  .  .
```

При поступлении сигнала ВЫСОКОГО уровня на этот вывод бит 0 регистра TRISA сбрасывается:

```
bclr      TRISA,#0
```

Это означает, что выходной буфер бита 0 порта А разблокируется, и на выход поступает сигнал с выхода защелки данных (LAT) (**Рис. 4.1**), который был ранее установлен в 0 командой `bclr.b PORTA, #0`. Это вызывает включение светодиода D1. Далее, с определенной задержкой, выход RA0 периодически переключается, до тех пор, пока на входе RA7 присутствует ВЫСОКИЙ уровень.

Эта программа демонстрирует, как работает порт ввода/вывода в режиме передачи цифровых данных при изменении конфигурации.

Многие функции микроконтроллера, такие, как вызов прерывания, синхронизация таймера и т. д., работают со стандартными настройками портов ввода/вывода. Это означает, например, что если вывод INT3 настроен как вход, то сигнал, поступающий на этот вход, будет вызывать соответствующее прерывание (если это прерывание разрешено).

В следующем проекте демонстрируется работа с прерыванием INT3, вход которого использует вывод RA14 (бит 14 порта А). Светодиод D1 переключается каждый раз при вызове обработчика прерывания. Аппаратная часть проекта показана на **Рис. 4.5**.

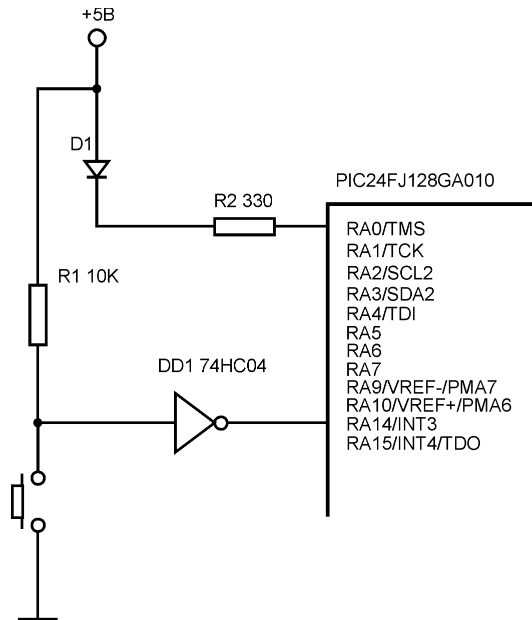

Рис. 4.5. Схема прерывания INT3 на порту ввода-вывода

Напомню, что порты ввода/вывода во многих случаях можно задействовать и для реализации альтернативных функций микроконтроллера. На **Рис. 4.5** как раз и показано использование вывода 14 порта А для приема сигнала прерывания. В такой конфигурации вывод порта должен быть настроен для чтения входного сигнала с помощью установки соответствующего бита в регистре TRISA. При желании или необходимости можно создать и такую аппаратно-программную конфигурацию, в которой, допустим, в определенные моменты времени бит 14 будет принимать сигнал прерывания, а остальную часть времени работать как выходной для управления какой-либо нагрузкой.

Программная часть проекта реализована с помощью мастера проектов MPLAB IDE, но исходный текст написан уже на Си:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

void __attribute__ ((interrupt)) _INT3Interrupt(void)
{
  _INT3IF = 0;
_RA0 = ~_RA0;
}

void main(void)
{
  TRISAbits.TRISA0 = 0x0;
```



```
  _RA0 = 0x1;

  _INT3IF = 0;
  _INT3IE = 1;

  while(1);
}
```

Это очень простой пример. Обработчик прерывания _INT3Interrupt сбрасывает флаг прерывания _INT3IF, а затем инвертирует бит 0 порта А. Поскольку при инициализации микроконтроллер настраивает порты на чтение, то нам нужно переключить бит 0 порта А на запись:

```
TRISAbits.TRISA0 = 0x0;
```

Кроме того, нужно разрешить внешнее прерывание INT3:

```
_INT3IF = 0;
_INT3IE = 1;
```

В данном примере мы работаем с портом А, который не используется периферийными модулями. Если, например, светодиод был бы подключен к порту В, который используется модулем аналого-цифрового преобразователя, то ситуация была бы иной. При инициализации микроконтроллера порт В сразу настраивается для использования модулем АЦП, поэтому, чтобы задействовать все или отдельные выводы порта для ввода/вывода цифровых сигналов, в основной программе пришлось бы вначале выполнить оператор

```
AD1PCFG = 0xffff;
```

В любом случае перед использованием в проекте того или иного порта следует ознакомиться с документацией на конкретный микроконтроллер.

4.3. Модуль регистрации событий

В микроконтроллерах PIC24F предусмотрена еще одна полезная функция, которая позволяет программе реагировать определенным образом на изменения уровней сигналов на входах портов. Эта функция основана на использовании механизма «захвата» или регистрации событий. В данном контексте под событием понимают любое изменение сигнала на входе порта ввода/вывода.

Этот механизм используется при работе с выводами портов, для которых предусмотрен специальный вариант обработки событий при изменении состояния сигнала на входе порта. Такие выводы микроконтроллера имеют дополнительное обозначение CNx (Change Notification) и реализованы посредством отдельного аппаратного модуля в микроконтроллере. При возникновении события задействуется механизм прерываний, при этом обработчик прерывания позволяет оперативно обработать событие. Количество таких выводов зависит от конкретной модели микроконтроллера, поэтому за более подробной информацией следует обращаться к технической документации на устройство.

С CN-модулем связаны три группы регистров управления:

- регистр CNENn — содержит биты разрешения прерывания от соответствующих выводов, которые обозначаются как CNxIE, где x обозначает номер CN-вывода;
- регистры CNPUn/CNPDn — содержат биты управления CNxPUE/CNxPDE. Эти биты разрешают/запрещают подтяжку выводов к питанию (CNPUn) и общему проводу (CNPDn). Это позволяет отказаться от использования внешних подтягивающих резисторов при подключении таких узлов, как кнопки (клавиатура).

Функциональная схема одного из каналов CN-модуля показана на **Рис. 4.6**.

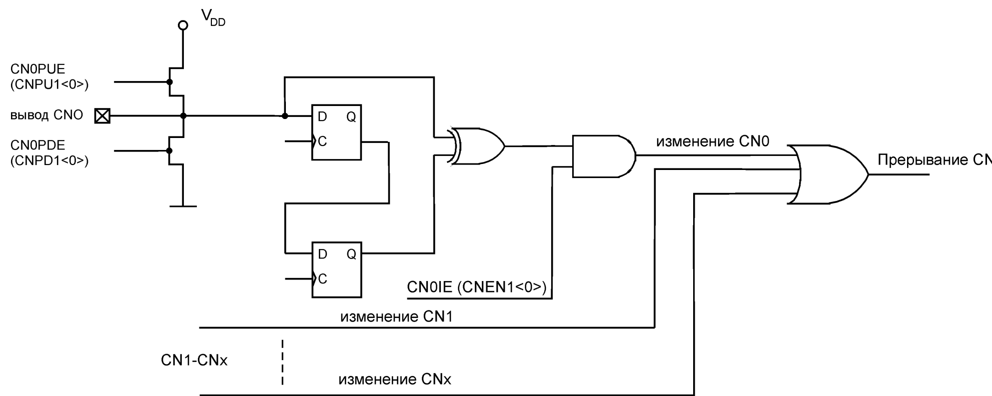

Рис. 4.6. Схема CN-модуля

На этом рисунке изображена функциональная схема канала 0 CN-модуля. При любом изменении сигнала на выводе CN0 схема исключающего ИЛИ будет формировать сигнал ВЫСОКОГО уровня, который, попадая на один из входов оконечного элемента ИЛИ, приведет к генерации прерывания. Точно так же работают и другие входы CN1...CNx.

Программирование обработчиков событий на входах CNx включает несколько шагов:

1. Требуемый вывод CNx переключается в режим цифрового входа посредством установки соответствующего бита в регистре TRISx.
2. Разрешается при необходимости подключение внутренней подтяжки путем установки соответствующих битов регистров CNPUx.
3. Сбрасывается флаг прерывания CNxIF.
4. Устанавливается требуемый уровень приоритета от CN-модуля путем установки CNxIP<2:0>.
5. Разрешаются прерывания от CN-входов путем установки соответствующих битов CNxIE.

Программа пользователя при необходимости должна сама определить, какое событие произошло (перепад 0—1 или 1—0), и выполнить требуемые действия.

Рассмотрим пример проекта, в котором показаны основные принципы обработки событий на цифровых входах микроконтроллера. Аппаратная часть проекта показана на **Рис. 4.7**.

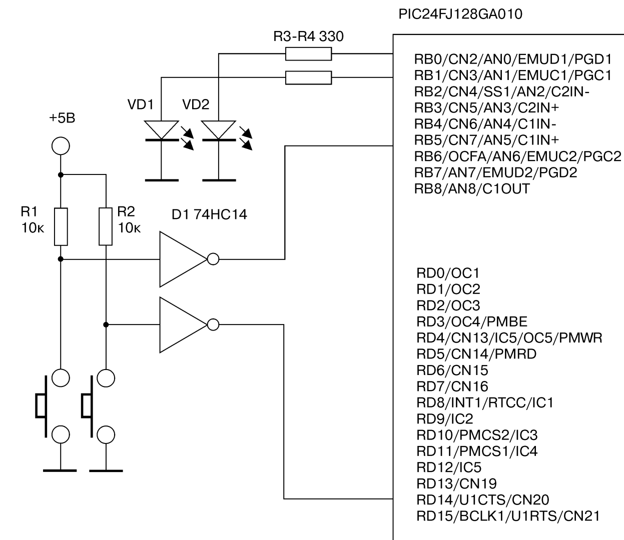

Рис. 4.7. Аппаратная часть проекта

В этой схеме контролируемые сигналы поступают на входы CN7 и CN20 микроконтроллера PIC24FJ128GA010. Программная часть проекта обеспечивает синхронное переключение светодиодов при одновременном присутствии сигналов ВЫСОКОГО уровня на входах CN7 и CN20. Например, переключение светодиодов будет осуществляться при ВЫСОКОМ уровне сигнала на входе CN7 и изменении уровня с НИЗКОГО на ВЫСОКИЙ на входе CN20. Тактовая частота схемы равна 8 МГц.

Программная часть проекта разработана в среде MPLAB IDE с помощью мастера проектов. Исходный текст программы на языке Си приведен далее:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  TRISB_5  TRISBbits.TRISB5
#define  TRISD_14  TRISDbits.TRISD14

#define  TRISB_0  TRISBbits.TRISB0
#define  TRISB_1  TRISBbits.TRISB1

#define  CN7  CNEN1bits.CN7IE
#define  CN20  CNEN2bits.CN20IE

#define  SYSCLK       8000000
```

```
void  __attribute__  ((interrupt))  _CNInterrupt(void)
{
  _CNIF  =  0;
  if  (_RB5  ==  1  &&  _RD14  ==  1)
    {
    _RB0  =  ~_RB0;
    _RB1  =  ~_RB1;
    }
}

void  main(void)
{
  TRISB_0  =  0;
  TRISB_1  =  0;
  TRISB_5  =  1;
  TRISD_14  =  1;

  _CNIF  =  0;
  _CNIP  =  0x4;
    CN7  =  1;
    CN20  =  1;
  _CNIE  =  1;

  while(1);
}
```

Переключение светодиодов, подключенных к выводам 0 и 1 порта B, выполняется в обработчике прерывания `_CNInterrupt`. Для инициализации выводов прерывания CN7 и CN20 в основной программе устанавливаются соответствующие биты в регистрах CNEN1 и CNEN2:

```
CN7  =  1;
CN20  =  1;
```

Кроме того, необходимо, как и в случае других источников прерываний, разрешить прерывания от CN-модуля:

```
_CNIE  =  1;
```

Программный код должен настроить также выводы RB5(CN7) и RD14(CN20) на ввод цифровых сигналов, что выполняется операторами:

```
TRISB_5  =  1;
TRISD_14  =  1;
```

В остальном программный код, думаю, понятен.

Возможность регистрации событий позволяет создавать системы сигнализации и оповещения различной степени сложности. В нашем следующем проекте мы рассмотрим простейшую систему регистрации нескольких событий и индикации источника события. Мы проанализируем взаимосвязь аппаратной и программной частей, а также возможные модификации этого проекта. Разработчики встраиваемых устройств часто сталкиваются с дилеммой: решить задачу с максимальным использованием аппаратных средств или использовать большей частью программный вариант решения. На этом примере мы проанализируем преимущества и недостатки обоих подходов.

Аппаратная часть проекта представлена схемой на **Рис. 4.8**.

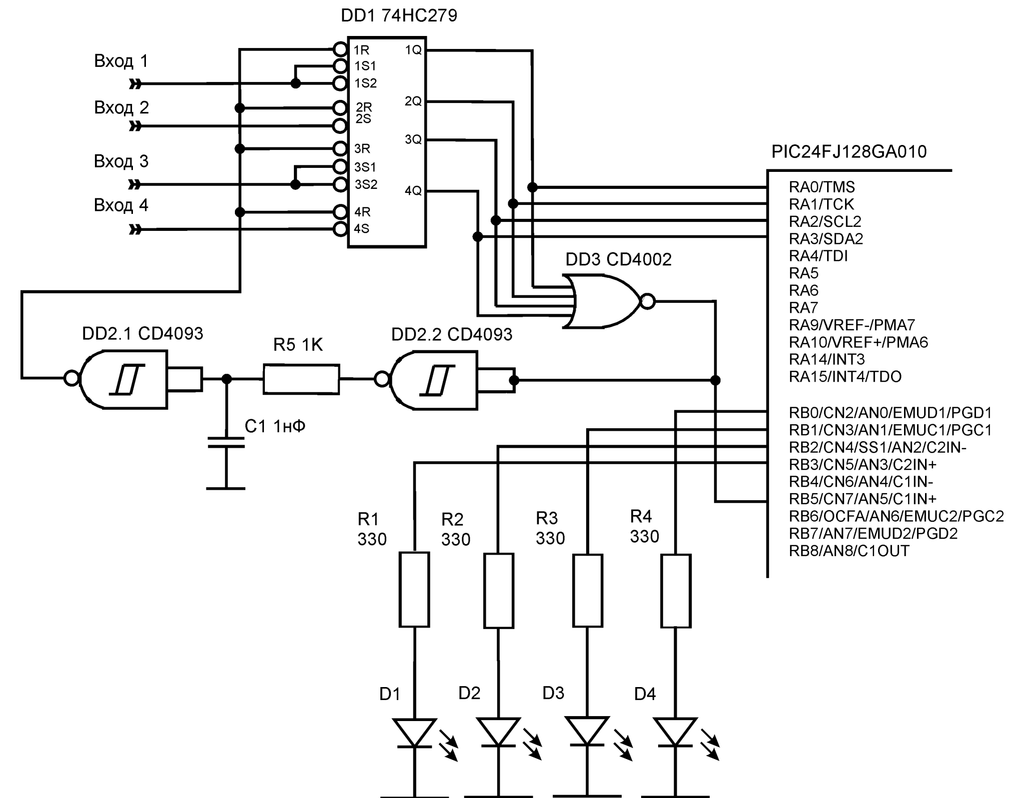

Рис. 4.8. Аппаратная часть устройства

В данном проекте используется микроконтроллер PIC24FJ128GA010 с тактовой частотой 8 МГц. На входы 1...4 RS-триггера 74HC279 (DD1) могут подключаться источники сигналов (дискретные датчики, переключатели и т. д.), НИЗКИЙ уровень напряжения которых приводит к появлению сигнала лог. 1 на одном из входов логического элемента ИЛИ-НЕ (микросхема DD3). Логический элемент DD3 переключается из состояния НИЗКОГО уровня в состояние ВЫСОКОГО уровня при наличии напряжения логической единицы на любом из 4 входов, изменяя тем самым состояние входа CN7, что приводит к генерации прерывания.

Далее программа считывает состояние выходов Q1...Q4 триггера DD1 и выводит его в биты 0...3 порта B, к которым подключены светодиоды D1...D4. Триггеры Шмита (микросхема DD2) выполняют функцию автоматического сброса состояния входов через определенное время задержки, определяемое RC-цепочкой R5C1. Эта задержка нужна для того, чтобы функция-обработчик прерывания успевала считать состояние выходов Q1...Q4.

В программную часть проекта, разработанную в MPLAB IDE, нужно включить Си-файл со следующим исходным текстом:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

char c1;

void __attribute__ ((interrupt)) _CNInterrupt(void)
{
  _CNIF = 0;
  c1 = PORTA & 0xf;
  PORTB = c1;
}

void main(void)
{
  AD1PCFG = 0xffff;
  TRISB = 0xfff0;
  TRISA = 0x7fff;

_CNIF = 0;
  _CNIP = 0x4;
  _CN7IE = 1;
  _CNIE = 1;

  while(1);
}
```

Программа очень проста. Обработка входных сигналов осуществляется в функции-обработчике прерывания _CNInterrupt. Здесь в переменную c1 считываются 4 бита порта A, которые тут же передаются в младшие биты порта B. В основной программе выполняется инициализация портов A и B, а также настройка прерывания обработки событий.

Обработка одних и тех же внешних сигналов может осуществляться и с помощью других аппаратно-программных решений. В каждом конкретном случае нужно определить оптимальную конфигурацию. Если в системе необходимо добиться использования минимального количества внешних компонентов, то основной упор должен делаться на программную часть микроконтроллера (firmware). Однако здесь могут возникнуть ограничения, связанные с быстродействием системы. Если сравнивать один и тот же алгоритм, реализованный оптимальным образом внешними элементами и аппаратно-программными средствами микроконтроллера, то вариант с внешними элементами будет работать быстрее. Тем не менее, для многих систем с невысоким или средним быстродействием упор на программную реализацию алгоритмов вполне оправдан. Например, наш предыдущий проект почти целиком можно реализовать с помощью одного микроконтроллера, оставив в системе минимум внешних компонентов.

Модифицированная аппаратная часть предыдущего проекта представлена на **Рис. 4.9**.

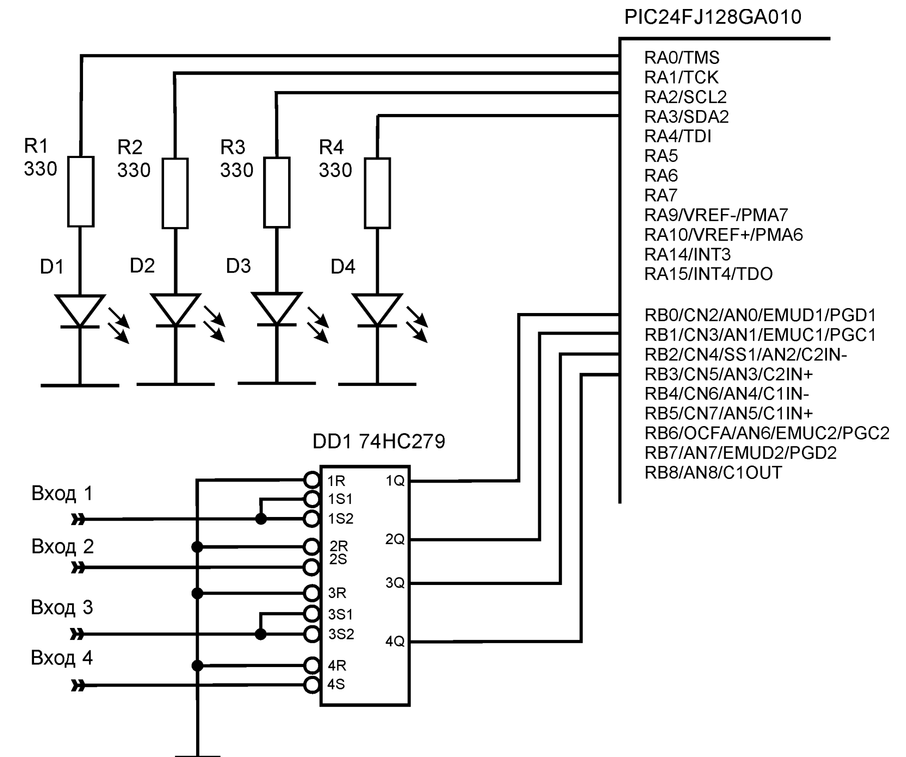

Рис. 4.9. Аппаратная часть модифицированного проекта

В этой схеме для считывания данных мы оставили одну внешнюю микросхему DD1. Выходы Q1...Q4 триггера в этой схеме соединены со входами регистрации событий CN2...CN5 микроконтроллера. Такое упрощение схемы требует модификации программной части проекта, поскольку нужно разрешить обработку событий по входам CN2...CN5. Вот исходный текст основной программы для данной аппаратной конфигурации:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

char c1;

void __attribute__ ((interrupt)) _CNInterrupt(void)
{
  _CNIF = 0;
  c1 = PORTB & 0xf;
  PORTA = c1;
}

void main(void)
{
```

```
  AD1PCFG  =  0xffff;
  TRISB  =  0xffff;
  TRISA  =  0x0;

  _CNIF  =  0;
  _CNIP  =  0x4;

  _CN2IE  =  1;
  _CN3IE  =  1;
  _CN4IE  =  1;
  _CN5IE  =  1;

  _CNIE  =  1;

  while(1);
}
```

Здесь нужно установить выводы порта B для работы в качестве цифровых портов, что выполняется двумя операторами:

```
AD1PCFG  =  0xffff;
TRISB  =  0xffff;
```

Кроме того, нужно установить разрешение прерывания при возникновении изменений сигналов на входах CN2...CN5:

```
_CN2IE  =  1;
_CN3IE  =  1;
_CN4IE  =  1;
_CN5IE  =  1;
```

В принципе, если нет жестких ограничений по количеству задействованных сигнальных выводов, то удобно реализовать большую часть проекта (если это не какой-то специальный случай) программным способом. Если необходима повышенная производительность, то в этом случае можно взять более быстродействующий микроконтроллер, например, семейства PIC24H.

ГЛАВА 5

ПРОГРАММИРОВАНИЕ ПРЕРЫВАНИЙ

Микроконтроллеры PIC24F имеют развитую систему прерываний, позволяющую использовать это устройство в приложениях реального времени различной степени сложности. Модуль прерываний микроконтроллеров PIC24F обладает следующими характеристиками:

- обеспечивает обработку 8 немаскируемых системных прерываний;
- позволяет задавать до семи уровней приоритетов пользовательских прерываний;
- позволяет задавать до 118 векторов прерываний в таблице векторов прерываний IVT. При этом каждому прерыванию назначается уникальный вектор;
- позволяет настроить дополнительную таблицу векторов прерываний (AIVT), которую можно использовать при отладке программ.

Таблица векторов прерываний IVT размещается в памяти программ, начиная с адреса 0x000004, и содержит 126 векторов прерываний, из которых 8 используются системными прерываниями, а остальные 118 могут использоваться периферийными модулями. Системные прерывания позволяют выполнить обработку серьезных аппаратно-программных сбоев в системе, поэтому их нельзя запретить или замаскировать. Векторы системных прерываний описаны в **Табл. 5.1**.

Таблица 5.1. Векторы немаскируемых ловушек

Номер вектора прерывания	Адрес в таблице IVT	Источник ошибки
0	0x000004	Зарезервирован
1	0x000006	Сбой генератора тактовой частоты
2	0x000008	Ошибка адресации
3	0x00000A	Ошибка обращения к стеку
4	0x00000C	Ошибка математических вычислений (как правило, деление на 0)
5	0x00000E	Зарезервирован
6	0x000010	Зарезервирован
7	0x000012	Зарезервирован

Каждый вектор прерывания содержит 24-битный адрес, по которому располагается соответствующая функция-обработчик прерывания (Interrupt Service Routine, ISR).

Любое пользовательское прерывание может быть разрешено или запрещено посредством установки или сброса соответствующего бита в одном из регистров разрешения прерываний IECn. Установка бита соответствующего прерывания в 1 разрешает прерывание, а сброс этого бита — запрещает. В микроконтроллерах PIC24F предусмотрен ряд механизмов, позволяющих разрешить или запретить все прерывания одновременно (кроме, естественно, немаскируемых прерываний).

При сбросе микроконтроллера все биты разрешения прерываний сбрасываются в 0, запрещая генерацию прерываний, поэтому программа пользователя должна сама устанавливать необходимые разрешения.

При разработке приложений было бы очень неудобно изменять биты разрешения прерываний, явно указывая регистры IECn, поскольку каждый раз пришлось бы просматривать все эти регистры в поисках соответствующего бита. К счастью, в компиляторе MPLAB C предусмотрен очень удобный механизм работы с битами разрешения прерываний посредством макросов, определенных в заголовочных файлах. Например, для разрешения/запрета прерывания Таймера 1 используется макрос _T1IE, для INT3 используется _INT3IE и т. д. Так, разрешить прерывание Таймера 1 в программе на Си можно с помощью оператора

```
_T1IE  =  1;
```

Каждому пользовательскому прерыванию может быть присвоен тот или иной уровень приоритета. Биты управления приоритетом располагаются на трех младших позициях каждой тетрады регистров IPCn. Биты 3 тетрад не используются и читаются как 0. Соответственно уровень приоритета каждого прерывания может изменяться от 1 (самый низкий приоритет) до 7 (наивысший приоритет). Если все биты приоритета прерывания сброшены в 0, то прерывание запрещено. Как и в случае с битами разрешения прерываний, в заголовочных файлах описаны макросы, облегчающие программные манипуляции при установке приоритетов. Например, для установки приоритета 4 прерывания Таймера 1 можно использовать оператор

```
_T1IP  =  4;
```

Для установки приоритета 2 прерывания INT4 можно использовать оператор

```
_INT4IP  =  2;
```

По умолчанию (после сброса микроконтроллера) приоритеты пользовательских прерываний устанавливаются равными 4. Для более подробной информации читатели могут обратиться к соответствующим заголовочным файлам.

Рассмотрим несколько проектов, в которых демонстрируются различные варианты реализации прерываний в системах с микроконтроллерами PIC24F. Вначале вкратце рассмотрим, каким образом организуется обработка прерываний в компиляторе MPLAB C.

Каждое прерывание, используемое в программе, должно иметь соответствующую функцию-обработчик, прототип которой в MPLAB C для PIC24 можно представить следующим образом:

```
void  __attribute__  ((__interrupt__))  isr0(void);
```


Из объявления прототипа функции-обработчика прерывания isr0 видно, что она не принимает никаких параметров и не возвращает никаких значений. При вызове функции-обработчика компилятор автоматически сохраняет в стеке все рабочие регистры, а также регистр состояния процессора SR. Остальные переменные можно сохранить, перечислив их в поле атрибутов. Например, для того чтобы компилятор автоматически сохранял и восстанавливал переменные var1 и var2, можно использовать такой прототип функции-обработчика:

```
void  __attribute__((__interrupt__(__save__(var1,  var2))))  isr0(void);
```

Обратите внимание, что в объявлении функции-обработчика используется двойное подчеркивание. Компилятор MPLAB C «понимает» и слегка упрощенный вариант объявления:

```
void  __attribute__  ((interrupt))  isr0(void);
```

Если функция-обработчик не требует указания дополнительных параметров, для ее объявления можно применить упрощенный синтаксис. Это делается с помощью макросов, которые объявлены в заголовочных файлах устройств. Вот пример такого объявления:

```
#define  ISR  __attribute__((interrupt))
```

Тогда функция-обработчик прерывания Таймера 1 может быть объявлена в программе следующим образом:

```
#include  <p24xxxx.h>
void  _ISR  _INT1Interrupt(void);
```

Дополнительные аспекты обработки прерываний мы рассмотрим в контексте последующих практических примеров.

В первом примере демонстрируется техника обработки прерываний с различным уровнем приоритетов. Аппаратная часть проекта показана на **Рис. 5.1**.

В этой схеме сигнал прерывания (перепад 0—1 на выходе инвертора) подается одновременно на два входа внешних прерываний INT3 и INT4, имеющих разные приоритеты. Функция-обработчик прерывания INT3 инвертирует выход 0 порта A, а обработчик прерывания INT3 то же самое делает с выходом 7 того же порта. В каждый из обработчиков включена небольшая задержка, реализованная на Таймере 1. Присваивая разные приоритеты прерываниям INT3 и INT4, можно наблюдать за изменениями последовательности переключения светодиодов на выходах 0 и 7. Если приоритет прерывания INT3 выше приоритета INT4, то светодиод, подключенный к выходу 0 порта A, переключится раньше, чем светодиод на выходе 7. Если же установить приоритет прерывания INT3 ниже приоритета INT4, то можно будет наблюдать обратную картину: светодиод на выходе 7 при подаче сигнала прерывания будет переключаться раньше, чем светодиод на выходе 0.

Программная часть проекта разработана в среде MPLAB IDE на языке Си и включает файл со следующим исходным текстом:

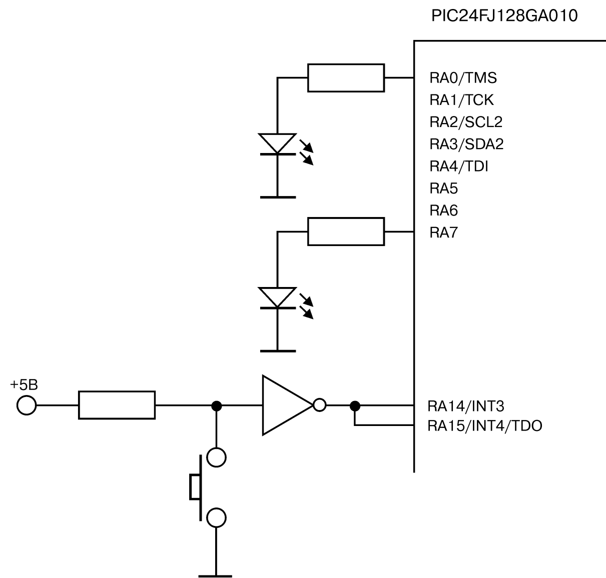

Рис. 5.1. Аппаратная часть проекта

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

void Delay(void)
{
    TMR1 = 0;
    T1CON = 0x8030;
    while (TMR1 < 30000);
    T1CON = 0x0;
}

void __attribute__ ((interrupt)) _INT3Interrupt(void)
{
  _INT3IF = 0;
_RA0 = ~_RA0;
Delay();
}

void __attribute__ ((interrupt)) _INT4Interrupt(void)
{
  _INT4IF = 0;
  _RA7 = ~ _RA7;
Delay();
}

void main(void)
{
    TRISA = 0;
```



```
    TRISAbits.TRISA14 = 1;
    TRISAbits.TRISA15 = 1;

    _INT3IF = 0;
    _INT3IP = 7;
    _INT3IE = 1;

    _INT4IF = 0;
    _INT4IP = 3;
    _INT4IE = 1;

    while(1);
}
```

В программе определены два обработчика прерываний: _INT3Interrupt (прерывание INT3) и _INT4Interrupt (прерывание INT4). В обработчиках прерываний биты 0 (прерывание INT3) и 7 (прерывание INT4) порта A инвертируются каждый раз при генерации прерываний. В каждом из обработчиков присутствует функция `Delay`, реализующая посредством Таймера 1 микроконтроллера небольшую задержку. Первый оператор обоих обработчиков прерываний сбрасывает флаги прерывания (_INT3IF и _INT4IF соответственно).

Инициализация прерывания INT3 выполняется в основной программе с помощью операторов

```
_INT3IF = 0;
_INT3IP = 7;
_INT3IE = 1;
```

Вначале сбрасывается флаг прерывания (первый оператор), затем прерыванию присваивается уровень приоритета, равный 7, после чего устанавливается флаг разрешения прерывания. Точно та же последовательность действий выполняется и для инициализации прерывания INT4, только уровень приоритета для этого прерывания устанавливается равным 3, т. е. ниже, чем для INT3. Таким образом, при возникновении ситуации, когда сигналы прерывания поступают на оба входа одновременно, обработчик прерывания INT3 выполнится первым. Эта ситуация как раз и моделируется схемой, показанной на **Рис. 5.1**.

Выводы прерываний должны быть сконфигурированы как входы, поэтому соответствующие биты регистра TRIS порта A устанавливаются в 1 с помощью операторов

```
TRISAbits.TRISA14 = 1;
TRISAbits.TRISA15 = 1;
```

Задавая разные уровни приоритетов прерываниям INT3 и INT4, можно наблюдать различную очередность переключения светодиодов, подключенных к выходам порта A.

Микроконтроллеры PIC24F позволяют генерировать прерывания программно, устанавливая флаг соответствующего прерывания в регистре IFSn. Например, можно разработать приложение, в котором функция-обработчик прерывания Таймера 1 вызывала бы прерывание INT3. В этом случае прерывание

INT3 будет вложенным по отношению к прерыванию Таймера 1. Микроконтроллеры PIC24F позволяют использовать вложенные прерывания, при этом вложенные прерывания должны иметь более высокий приоритет по сравнению с текущим прерыванием. Это означает, что прерывание INT3, вызванное из обработчика прерывания Таймера 1, будет обработано раньше только в том случае, если его приоритет будет выше, чем у текущего прерывания. Если приоритет прерывания INT3 будет ниже, чем у прерывания Таймера 1, то обработчик INT3 будет вызван по завершении обработчика прерывания Таймера 1.

В нашем следующем проекте будет продемонстрирована последовательность обработки вложенных прерываний.

Аппаратная часть проекта показана на **Рис. 5.2**.

В этой схеме светодиод, подключенный к выводу RA0 порта A, будет переключаться каждый раз при вызове прерывания INT3. Светодиод, подключенный к выводу RA7, будет переключаться при вызове прерывания Таймера 1. Визуально очередность переключения светодиодов будет зависеть от приоритета вложенного прерывания INT3.

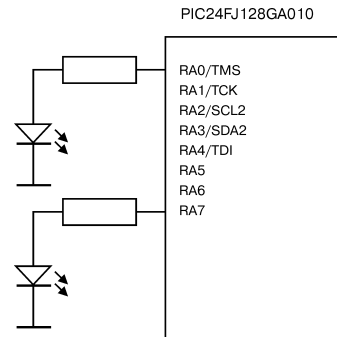

Рис. 5.2. Аппаратная часть проекта

Программная часть демонстрационного проекта разработана в MPLAB IDE и включает файл с программой, исходный текст которой приводится далее:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define SYSCLK 8000000
#define t1 1
#define PREG   SYSCLK/2/256*t1

void Delay(void)
{
  long cnt = 100000;
  while (cnt != 0)cnt-;
}

void __attribute__ ((interrupt)) _T1Interrupt(void)
{
  _T1IF = 0;
  _INT3IF = 1;
  _RA7 = ~_RA7;
  Delay();
}

void __attribute__ ((interrupt)) _INT3Interrupt(void)
{
  _INT3IF = 0;
  _RA0 = ~_RA0;
  Delay();
}

void main(void)
{
    TRISA = 0;
    TRISAbits.TRISA14 = 1;

    PR1 = PREG;
    TMR1 = 0;

    IFS0bits.T1IF = 0;
    _T1IP = 5;
    _T1IE = 1;
    T1CON = 0x8030;

    _INT3IF = 0;
    _INT3IP = 3;
    _INT3IE = 1;

    while(1);
}
```


В этой программе определены две функции-обработчики прерываний. Функция `_T1Interrupt` обрабатывает прерывание таймера, которое при тактовой частоте микроконтроллера 8 МГц, указанных настройках регистра периода PR1 и предделителя будет вызываться с интервалом в 1 с. Функция `_INT3Interrupt` вызывается из обработчика `_T1Interrupt` путем установки флага прерывания _INT3IF. Функция `Delay()` реализует временную задержку и используется для того, чтобы пользователь мог визуально отследить очередность переключения светодиодов при различных настройках параметров прерываний. Исходный уровень приоритета для прерывания Таймера 1 устанавливается равным 5, а уровень приоритета для прерывания INT3 — равным 3. Поскольку приоритет INT3 ниже, чем приоритет Таймера 1, то, несмотря на установку флага прерывания _INT3IF в функции-обработчике `_T1Interrupt`, обработчик INT3 не сможет выполниться, пока не завершится обработка прерывания Таймера 1.

Визуально светодиод на выводе 7 порта A будет переключаться первым, и только затем, с определенной задержкой, переключится светодиод на выводе 0. Если теперь изменить приоритеты прерываний так, чтобы INT3 имело высший приоритет, например 6, то картина изменится. Первым будет переключаться светодиод на выводе 0 порта A, поскольку вначале будет выполняться обработчик вложенного прерывания INT3, и только затем продолжит выполнение обработчик прерывания Таймера 1.

Еще несколько слов о программе. Параметры Таймера 1 настраиваются с помощью операторов

```
PR1 = PREG;
TMR1 = 0;
T1CON = 0x8030;
```

В регистр периода PR1 Таймера 1 записывается значение, соответствующее времени срабатывания Таймера 1 и равное 1 с. Здесь учитывается то, что тактовая частота микроконтроллера, равная 8 МГц, делится на 2 и на коэффициент деления предделителя, равный 256. В результате получаем необходимое значение для регистра периода (оно равно константе PREG). Таймер 1 запускается при установке бита 15 (TON) регистра управления T1CON и битов 5 и 4 предделителя (отношение 1:256). Настройки прерывания Таймера 1 выполняются по стандартной схеме с помощью операторов

```
IFS0bits.T1IF  =  0;
_T1IP  =  5;
_T1IE  =  1;
```

Здесь уровень приоритета прерывания Таймера 1 устанавливается равным 5. Обработку вложенных прерываний можно запретить, если установить бит15 (NSTDIS) регистра управления INTCON1. В этом случае пользовательские установки приоритетов будут игнорироваться. Если, например, в исходный текст нашей программы включить оператор

```
INTCON1bits.NSTDIS  =  1;
```

то все прерывания будут работать с одинаковым приоритетом, а вложенные прерывания работать не будут. В нашем случае установка флага прерывания _INT3IF в обработчике Таймера 1 не вызовет прерывания его выполнения, т. е. обработчик прерывания INT3 всегда будет выполняться после обработчика Таймера 1.

В ряде случаев возникает необходимость запретить на некоторое время пользовательские прерывания, например при выполнении критических участков программного кода. Можно применить один из двух способов. Простейший вариант — использовать инструкцию `disi` микроконтроллера, которая позволяет на определенный интервал времени запретить прерывания с приоритетами 1...6. Прерывания с приоритетом 7 и выше с помощью этой инструкции запретить нельзя.

Второй способ, более эффективный, позволяет запретить все прерывания без каких-либо ограничений по времени. Для этого можно использовать следующую последовательность операций:

- сохранить регистр состояния микроконтроллера в стеке с помощью инструкции `push SR`;
- записать в биты приоритета IPL2...IPL0 процессора (биты 7...5 регистра состояния процессора SR) значение <111>, что позволит установить уровень 7 приоритета процессора и запретить все пользовательские прерывания. Для установки этих битов можно выполнить, например, инструкцию `ior` (логическое ИЛИ) над содержимым младшего байта (SRL) регистра состояния SR и значения 0xE0.

Если критический участок программного кода пройден, можно восстановить состояние процессора и разрешить прерывания инструкцией `pop SR`.

Рассмотрим практический пример, иллюстрирующий запрещение/разрешение прерываний. Аппаратная часть проекта показана на **Рис. 5.3**.

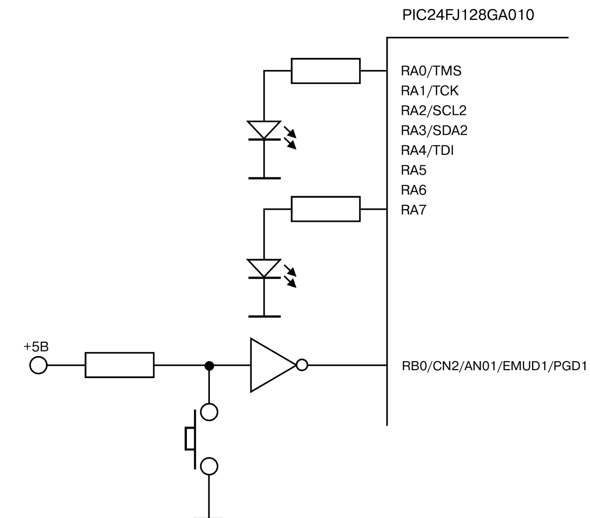

Рис. 5.3. Аппаратная часть проекта

Работа схемы во многом напоминает функционирование аппаратной части из предыдущего проекта, но имеется и существенное отличие: при нажатии на кнопку разблокируется работа прерываний Таймера 1 и INT3. Прерывание INT3 вызывается из обработчика `_T1Interrupt`. При запуске программы все прерывания блокируются, поэтому никакого изменения состояния светодиодов наблюдаться не будет. При нажатии на кнопку все пользовательские прерывания разблокируются, и светодиоды начнут мигать.

Исходный текст программы, написанной на языке Си, показан далее:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  SYSCLK  8000000
#define  t1  1
#define  PREG    SYSCLK/2/256*t1

void  Delay(void)
{
  long  cnt  =  100000;
  while  (cnt  !=  0)cnt-;
}

void  __attribute__  ((interrupt))  _T1Interrupt(void)
{
  _T1IF  =  0;
  _INT3IF  =  1;
```

```
  _RA7 = ~_RA7;
  Delay();
}

void __attribute__ ((interrupt)) _INT3Interrupt(void)
{
  _INT3IF = 0;
_RA0 = ~_RA0;
  Delay();
}

void main(void)
{
    AD1PCFG = 0xffff;
    TRISB = 0xffff;
    TRISA = 0;
    TRISAbits.TRISA14 = 1;

    PR1 = PREG;
    TMR1 = 0;
    _T1IF = 0;
    _T1IP = 5;
    _T1IE = 1;
    T1CON = 0x8030;

    _INT3IF = 0;
    _INT3IP = 6;
    _INT3IE = 1;

    asm ("push SR"
         "\n mov #0xE0, w0"
         "\n ior SR");
// Сюда можно поместить критический участок
// программного кода

  while (_RB0 != 1);
  asm("pop SR");
  while(1);
}
```

Большую часть этого программного кода мы уже анализировали, поэтому остановлюсь только на сделанных изменениях. Для запрещения/разрешения пользовательских прерываний используются две ассемблерные вставки. Группа команд

```
push SR
mov #0xE0, w0
ior SR
```

выполняет сохранение в стеке содержимого регистра состояния (`push SR`), устанавливает уровень приоритета 7 для процессора, запрещая тем самым пользовательские прерывания (`mov #0xE0, w0` и `ior SR`). Таким образом, при запуске программы никакие прерывания обрабатываться не будут. Для разрешения прерываний нужно просто извлечь из стека старое содержимое регистра

состояния SR (`pop SR`). Это произойдет при нажатии на кнопку (см. **Рис. 5.3**), инициирующую подачу сигнала ВЫСОКОГО уровня на вывод 0 порта B. После этого можно будет наблюдать выполнение обработчиков прерываний по смене состояния светодиодов.

Вместо ассемблерной вставки можно воспользоваться макросами, специально предусмотренными для запрещения/разрешения всех прерываний. Тогда ту часть программного кода, в которой используются команды ассемблера, можно переписать следующим образом:

```
int current_cpu_ipl;

. . .

SET_AND_SAVE_CPU_IPL(current_cpu_ipl, 7);
// Сюда можно поместить критический участок
// программного кода
while (_RB0 != 1);
RESTORE_CPU_IPL(current_cpu_ipl);
```

Переменная `current_cpu_ipl` будет содержать текущее на момент обращения значение приоритета процессора. Установка уровня приоритета равным 7 выполняется макросом SET_AND_SAVE_CPU_IPL, а восстановление предыдущего уровня приоритета и разблокирование пользовательских прерываний осуществляется макросом RESTORE_CPU_IPL.

ГЛАВА 6

ПРОГРАММИРОВАНИЕ ТАЙМЕРОВ

Микроконтроллеры семейства PIC24F, в зависимости от исполнения, могут иметь несколько 16-битных таймеров, которые принято обозначать как Таймер 1, Таймер 2, Таймер 3 и т. д. Аппаратно-программная архитектура каждого 16-битного таймера включает несколько регистров, доступных для чтения/записи. К ним относятся:

- TMRx — 16-битный регистр-счетчик (x = 1, 2, 3, ...);
- PRx — 16-битный регистр периода;
- TxCON — 16-битный регистр управления Таймера x.

Каждый модуль таймера может генерировать прерывание, поэтому ему назначены следующие программные ресурсы:

- бит TxIE — флаг разрешения прерывания Таймера x;
- бит TxIF — флаг состояния прерывания Таймера x;
- биты TxIP — установка приоритета Таймера x.

Для 16-битных таймеров предусмотрено три основных режима (хотя не каждый таймер может работать во всех режимах):

- режим «таймера», когда таймер синхронизируется внутренним или внешним тактовым сигналом и инкрементируется до значения, установленного в регистре периода PRx;
- режим запуска/остановки внешним сигналом (Gated Time Accumulation), при котором запуск таймера осуществляется перепадом 0—1 внешнего сигнала, а остановка осуществляется по перепаду 1—0 этого же источника;
- асинхронный режим работы от внешнего источника тактового сигнала.

В зависимости от режима работы 16-битного таймера прерывание генерируется или по превышению значения регистра периода PRx (если таймер не работает в режиме внешнего запуска/останова), или по спадающему фронту внешнего сигнала запуска/останова. Вне зависимости от того, разрешено прерывание таймера или нет, при возникновении любой из этих ситуаций будет установлен флаг прерывания TxIF, который должен сбрасываться программой. Если необходимо работать с прерываниями таймера, то соответствующий флаг разрешения прерывания TxIF должен быть установлен. Биты приоритетов TxIP устанавливаются в соответствии с требованиями к приложению (по умолчанию это значение равно 4).

Кроме указанных режимов работы, существуют и определенные отличия функциональных возможностей 16-битных таймеров, поэтому все таймеры разделены на три базовых типа: A, B и C. Таймер базового типа A, который присутствует во всех микроконтроллерах PIC24F, может работать в режиме «таймера» и внешнего запуска/останова, а к особенностям таймеров типа B и C следует отнести возможность их конфигурирования для работы в 32-битном режиме. К таймеру типа A относится Таймер 1, функциональная схема которого показана на **Рис. 6.1**.

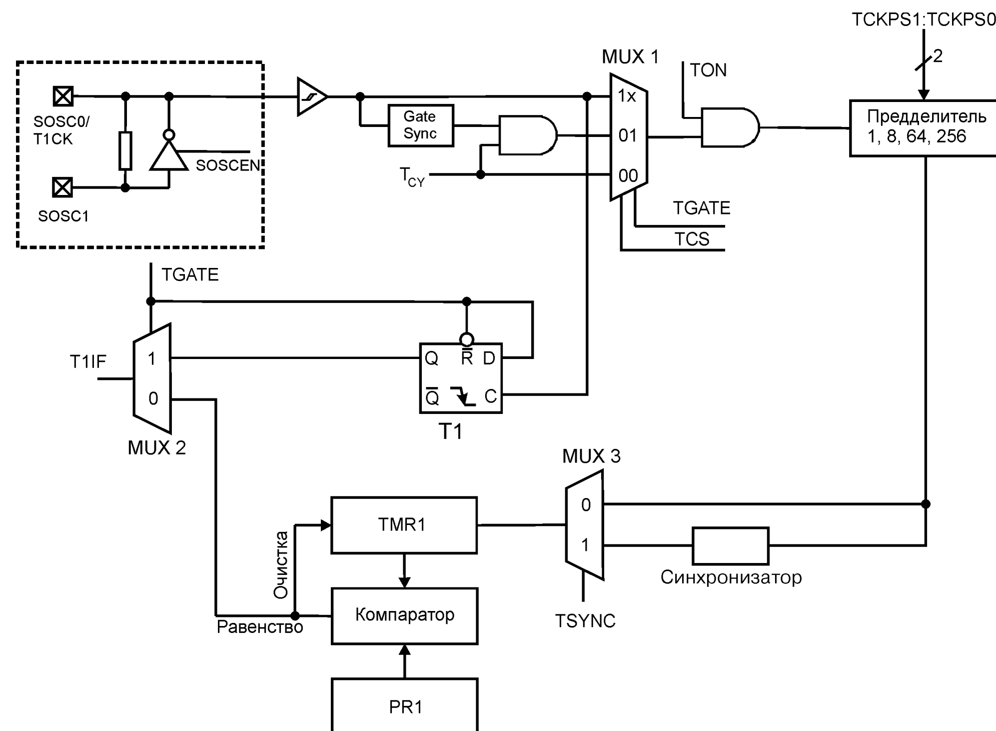

Рис. 6.1. Функциональная схема 16-битного таймера (базовый тип A)

Из **Рис. 6.1** довольно легко понять работу таймера в зависимости от установок отдельных битов регистра управления TxCON. Рассмотрим более подробно, какие биты регистра управления используются при различных режимах работы таймера типа A. Сразу оговорюсь: функциональная схема с элементами цифровой логики, показанная на **Рис. 6.1**, далеко не полностью соответствует принципиальной схеме модуля Таймера 1, скорее это достаточно хорошее приближение к реальной схеме. Тем не менее, такая функциональная схема в точности отображает поведение таймера, поэтому мы рассмотрим ее более

подробно — это значительно облегчит понимание принципов программирования таймера.

Предположим, что Таймер 1 типа A работает в режиме «таймера» с коэффициентом деления тактовой частоты 1:8. В этом случае бит TGATE = 0, TCS = 0 и мультиплексор MUX 1 пропускает на свой выход тактовый сигнал T_{CY}, частота которого равна половине частоты тактового генератора микроконтроллера. Для дальнейшего прохождения сигнала бит TON регистра управления должен быть установлен в 1. В этом случае сигнал попадает на предделитель, коэффициент деления которого определяется значениями пары битов TCKPS1:TCKPS0. Если, например, требуется поделить частоту сигнала T_{CY} на 8, то TCKPS1:TCKPS0 = 01.

Поскольку бит TCS = 0, то значение бита синхронизации от внешнего источника TSYNC игнорируется, и сигнал через мультиплексор приходит на схему инкремента регистра таймера TMR1. Содержимое регистра таймера TMR1 постоянно сравнивается с содержимым регистра периода PR1, и если значения равны, то регистр таймера вместо инкрементирования сбрасывается в 0, а на выходе компаратора формируется сигнал, который, проходя через мультиплексор MUX 2, вызывает установку флага прерывания T1IF.

Второй режим, который мы рассмотрим, — режим запуска/остановки Таймера 1 внешним сигналом с вывода T1CK. При работе в этом режиме таймер, как и в только что рассмотренном режиме «таймера», синхронизируется импульсами тактового сигнала T_{CY}, но запуск и остановка таймера выполняются внешним сигналом: запуск таймера осуществляется по фронту (перепад 0—1) сигнала, а остановка — по спаду (перепад 1—0). Для работы в этом режиме бит TGATE должен быть установлен в 1, а бит TCS должен быть равен 0 (работа от внутреннего источника тактового сигнала).

При этой комбинации сигнал T_{CY} проходит логику совпадения и через мультиплексор MUX 1 поступает на предделитель и далее с коэффициентом деления 1:8 — на логику управления регистром таймера TMR1. В данном режиме флаг прерывания устанавливается по спаду внешнего сигнала, который переключает D-триггер T1, работа которого разрешается ВЫСОКИМ уровнем вследствие установки бита TGATE. Сигнал с выхода Q триггера T1 инициирует установку флага прерывания T1IF. В этом режиме установка флага прерывания вызывается не переполнением Таймера 2, а только перепадом 1—0 внешнего сигнала.

Третий режим — работа от внешнего источника тактового сигнала. В качестве источника может быть использован генератор (выделен пунктирной линией), к выводам SOSC0 и SOSC1 которого подключается кварцевый резонатор. Как альтернатива, к выводу SOSC0 можно подключить внешний тактовый генератор. В любом случае биты конфигурации микроконтроллера должны быть установлены соответствующим образом. В этом режиме бит TCS должен быть установлен в 1, тогда внешний тактовый сигнал проходит через мультиплексор MUX 1 и предделитель на логику управления регистром таймера. Если дополнительно установлен бит синхронизации TSYNC, то внешний сигнал будет синхронизирован внутренним тактовым сигналом. Логика формирования сигнала прерывания будет работать так же, как и в режиме «таймера».

Назначение битов регистра управления T1CON Таймера 1 (тип A) приводится в **Табл. 6.1**.

Таблица 6.1. Назначение битов регистра T1CON

Позиция бита	Обозначение	Описание
15	TON	Запуск/остановка таймера: 1 – запуск таймера; 0 – остановка таймера
14		Читается как 0
13	TSIDL	Режим остановки в режиме «холостого хода» микроконтроллера: 1 – работа таймера прекращается в режиме «холостого хода»; 0 – таймер продолжает работать
12–7		Читается как 0
6	TGATE	Разрешение запуска/остановки таймера внешним сигналом. При установленном бите TCS значение этого бита игнорируется. Если TCS = 0, то установка бита в 1 разрешает запуск/остановку таймера, а сброс в 0 запрещает работу в этом режиме
5–4	TCKPS1–TCKPS0	Биты установки предделителя: 11 – соотношение 1:256 10 – соотношение 1:64 01 – соотношение 1:8 00 – соотношение 1:1
3		Читается как 0
2	TSYNC	Если TCS = 1, то: 1 – выполняется синхронизация внешнего сигнала; 0 – не выполняется синхронизация внешнего сигнала. Если TCS = 0 (используется внутренний источник тактовой частоты), то бит игнорируется и читается как 0
1	TCS	Выбор источника тактовой частоты: 1 – используется сигнал на выводе T1CK; 0 – используется внутренний генератор тактовой частоты (F_{OSC}/2)
0		Читается как 0

Кроме Таймера 1, который относится к типу A, в большинстве микроконтроллеров PIC24F есть Таймер 2 и Таймер 4, которые относятся к типу B. Таймер типа B позволяет (вместе с таймером типа C) организовать 32-битный таймер. Для этого в таймерах типа B предусмотрен бит T32 (бит 3 регистра TxCON), который, будучи установлен, разрешает функционирование в 32-битном режиме.

В большинстве микроконтроллеров PIC24F имеются также и таймеры типа C (Таймер 3 и Таймер 5). Таймер типа C имеет следующие специальные возможности:

- вместе с таймером типа B может образовывать 32-битный таймер;
- как минимум один таймер типа C может управлять процессом преобразования АЦП.

Формирование 32-битного таймера выполняется с помощью либо Таймера 2 типа B и Таймера 3 типа C, либо Таймера 4 типа B и Таймера 5 типа C. За исключением бита T32 в регистре управления таймера типа B, все остальные биты таймеров типов B и C идентичны по назначению битам регистра управления таймера типа A.

Остановимся на некоторых практических аспектах программирования таймеров микроконтроллеров PIC24F.

6.1. Практическое использование 16-битных таймеров

Рассмотрим несколько практических примеров программирования таймеров микроконтроллеров PIC24F. В нашем первом проекте мы будем переключать светодиод, подключенный к выводу 0 порта B, каждые 10 секунд, используя метод опроса Таймера 1. В этом проекте тактовая частота микроконтроллера выбрана равной 8 МГц. Аппаратная часть проекта показана на **Рис. 6.2**.

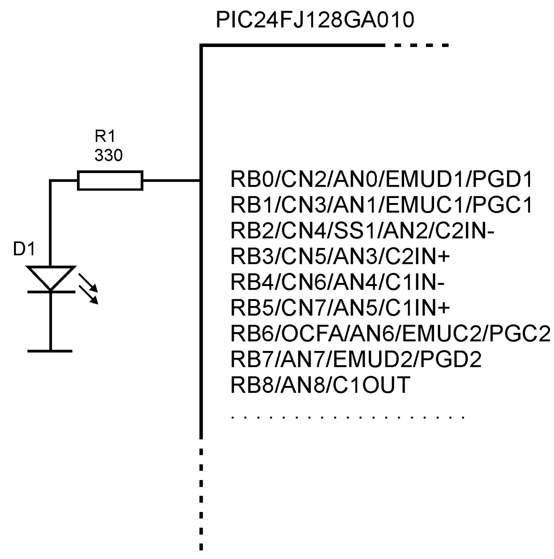

Рис. 6.2. Аппаратная часть проекта

Программную часть нашего проекта создадим с помощью среды MPLAB IDE. В созданный проект включим исходный текст программы на языке Си, приведенный далее:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  SYSCLK   8000000
```



```
#define  t1       0.5
#define  PREG     SYSCLK/2*t1/256

#define  DELAY    20
#define  PORTB_0  PORTBbits.RB0

void  main(void)
{
    int  cnt  =  0;
    AD1PCFG  =  0xffff;
    TRISB  =  0xfffe;

    PR1  =  PREG;
    TMR1  =  0;
    T1CON  =  0x8030;

    while  (1)
      {
          if  (_T1IF  ==  1)
            {
              _T1IF  =  0;

              if  (cnt  ==  DELAY)
                {
                  cnt  =  0;
                  PORTB_0  =  ~PORTB_0;
                }
              cnt++;
            }
      }
}
```

Для установки временных интервалов в программе нам нужно просчитать некоторые константы. Мы приняли, что светодиод должен переключаться каждые 10 секунд. Если выбрать интервал срабатывания таймера `t1` равным 0.5 с, то программе понадобится 20 циклов опроса, прежде чем светодиод переключится. Таким образом, мы определяем в программе две константы:

```
#define  t1      0.5
#define  DELAY   20
```

Здесь `t1` определяет время срабатывания Таймера 1, а `DELAY` определяет общий интервал задержки, равный 20 * 0.5 = 10 с. Далее нам нужно определить число, которое следует записать в регистр периода Таймера 1, чтобы получить интервал перезагрузки, равный 0.1 с (константа `t1`). Значение регистра периода (обозначим его `PREG`) по известному интервалу времени `t1` можно вычислить из формулы:

$$PREG = (t1 \times Fosc/2)/Prescaler,$$

где PREG — значение регистра периода; t1 — заданный интервал времени; Fosc — тактовая частота (во всех наших примерах Fosc = 8 МГц); Prescaler — значение предделителя (выберем значение, равное 256).

После того, как значение регистра периода определено, мы можем запустить таймер:

```
PR1 = PREG;
TMR1 = 0;
T1CON = 0x8030;
```

Собственно для запуска таймера нужно установить бит 15 (TON) регистра управления T1CON и значение коэффициента деления в битах 5:4 (TCKPS1:TCKPS0 = 11 для значения 256). Помимо этого мы обнуляем регистр таймера TMR1.

При указанных настройках Таймер 1 будет перегружаться каждые 0.1 с, при этом будет устанавливаться флаг прерывания `_T1IF` (независимо от того, разрешено или нет прерывание таймера). Именно по состоянию флага и можно будет определить, перегрузился ли таймер, т. е. прошло ли 0.1 с. Когда значение счетчика `cnt` в цикле `while` достигнет 20, то бит 0 порта B инвертируется и светодиод переключится.

Команда

```
AD1PCFG = 0xffff;
```

переводит порт B в режим ввода/вывода цифровых сигналов, а команда

```
TRISB = 0xfffe;
```

переключает вывод 0 на выход.

Данная программа позволяет работать с относительно большими интервалами времени. Если требуются интервалы времени порядка единиц секунд, то предыдущий листинг исходного текста можно значительно упростить. Например, если требуется временная задержка 2 с, то можно использовать такую программу:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define SYSCLK  8000000
#define t1      2
#define PREG    SYSCLK/2*t1/256

#define DELAY   31250
#define PORTB_0 PORTBbits.RB0

void main(void)
{

    AD1PCFG = 0xffff;
    TRISB = 0xfffe;

    PR1 = PREG;
    TMR1 = 0;
    T1CON = 0x8030;
```



```
    while (1)
      {
          while (TMR1 < DELAY);
          TMR1 = 0;
          PORTB_0 = ~PORTB_0;
      }
}
```

Здесь мы указываем значение интервала времени прямо в переменной `t1` (2), а для константы `DELAY` устанавливаем значение 31250, что соответствует значению регистра периода `PR1` для 2-секундного интервала. В этом случае ожидание реализуется в цикле

```
while (TMR1 < DELAY)
```

что исключает необходимость использования флага прерывания и счетчика `cnt`.

Для приложений реального времени цикл ожидания в основной программе — непозволительная роскошь, поэтому в таких случаях можно использовать прерывание Таймера 1. Модифицируем текст только что рассмотренной программы так, чтобы инверсия бита 0 порта B осуществлялась в обработчике прерывания. Новая программа будет выглядеть следующим образом:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define SYSCLK  8000000
#define t1      2
#define PREG    SYSCLK/2*t1/256

#define PORTB_0 PORTBbits.RB0

void __attribute__ ((interrupt)) _T1Interrupt(void)
{
  _T1IF = 0;
  PORTB_0 = ~PORTB_0;
}

void main(void)
{
    AD1PCFG = 0xffff;
    TRISB = 0xfffe;

    _T1IF = 0;
    _T1IP = 3;
    _T1IE = 1;

    PR1 = PREG;
    TMR1 = 0;
    T1CON = 0x8030;

    while (1)
      {
        . . .
```

```
      // другие операторы программ
      . . .
      }
}
```

Для переключения светодиода в этой программе используется обработчик прерывания `_T1Interrupt`, первая команда которого сбрасывает флаг прерывания, а вторая, собственно, и выполняет переключение бита 0 порта B. В основной программе необходимо установить разрешение прерывания Таймера 1, что выполняет оператор

```
_T1IE = 1;
```

Перед этим нужно сбросить флаг прерывания и, если необходимо (хоть это и не обязательно), установить приоритет прерывания (по умолчанию он равен 4), что и выполняется операторами

```
_T1IF = 0;
_T1IP = 3;
```

В данном случае приоритет прерывания Таймера 1 установлен равным 3. Команды инициализации и запуска Таймера 1 те же, что и в предыдущих листингах, поэтому останавливаться на них я не буду.

Кроме внутренней синхронизации, в Таймере 1 можно применить внешнюю синхронизацию, подавая сигнал на вывод T1CK микроконтроллера (**Рис. 6.3**).

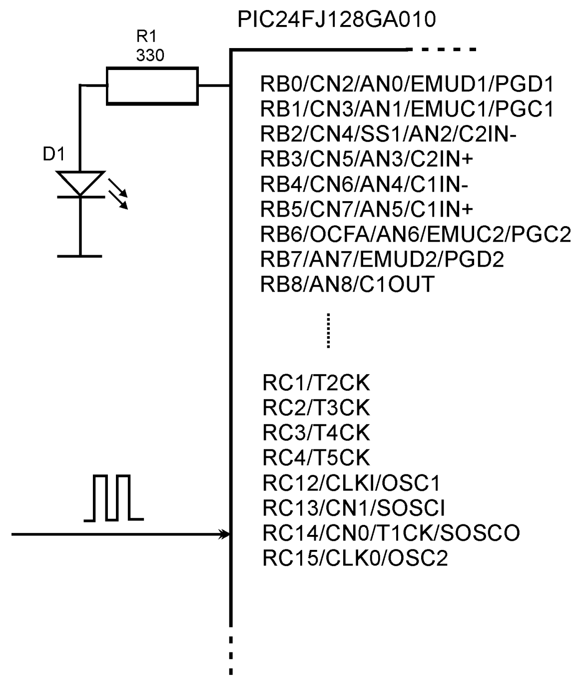

Рис. 6.3. Подключение внешнего источника синхронизации Таймера 1

В этом проекте используется внешний источник сигнала с частотой 1000 Гц. Исходный текст программы нашего модифицированного проекта представлен далее:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define EXTCLK      1000
#define t1          16
#define PREG        EXTCLK*t1

#define PORTB_0  PORTBbits.RB0

void __attribute__ ((interrupt)) _T1Interrupt(void)
{
  _T1IF = 0;
  PORTB_0 = ~PORTB_0;
}

void main(void)
{
    AD1PCFG = 0xffff;
    TRISB = 0xfffe;
    TRISC = 0xffff;

    _T1IF = 0;
    _T1IP = 3;
    _T1IE = 1;

    PR1 = PREG;
    TMR1 = 0;
    T1CON = 0x8006;

    while (1)
      {
      }
}
```

Здесь значение частоты 1000 Гц определяется через константу `EXTCLK`. Выберем время задержки, равное 16 с (константа `t1`), а коэффициент деления частоты равным 1. В этом случае значение регистра периода `PR1` будет вычисляться как `EXTCLK * t1`.

Переключение светодиода будет выполняться, как и в предыдущем проекте, в обработчике прерывания Таймера 1, но сами настройки таймера изменятся. Поскольку мы решили обойтись без предварительного деления частоты и должны установить биты TCS и TSYNC регистра управления T1CON, то команда для настройки и запуска выглядит так:

```
T1CON = 0x8006;
```

То же самое можно выполнить при помощи операторов

```
T1CONbits.TSYNC = 1;
T1CONbits.TCS = 1;
T1CONbits.TON = 1;
```

Кроме того, поскольку внешние синхроимпульсы приходят через порт С, то переключим выводы порта на вход:

```
TRISC  =  0xffff;
```

Во всем остальном исходный текст нам уже знаком и дополнительный анализ кода не нужен. Все наши проекты касались режима работы «таймер» и анализировались на примере работы Таймера 1.

Перейдем к обсуждению следующего режима работы, в котором запуск и остановка таймера осуществляются внешним сигналом на входе TxCK. В этом режиме таймер запускается по нарастающему фронту сигнала (перепад 0—1) и останавливается по спадающему фронту (перепад 1—0). При обнаружении на входе TxCK перепада 1—0 устанавливается соответствующий флаг прерывания _TxIF. Если задан обработчик соответствующего прерывания, то можно выполнить определенные действия по обработке этого события. Этот режим работы мы рассмотрим в следующих примерах. Аппаратная часть проектов, с которыми мы будем работать, показана на **Рис. 6.4**.

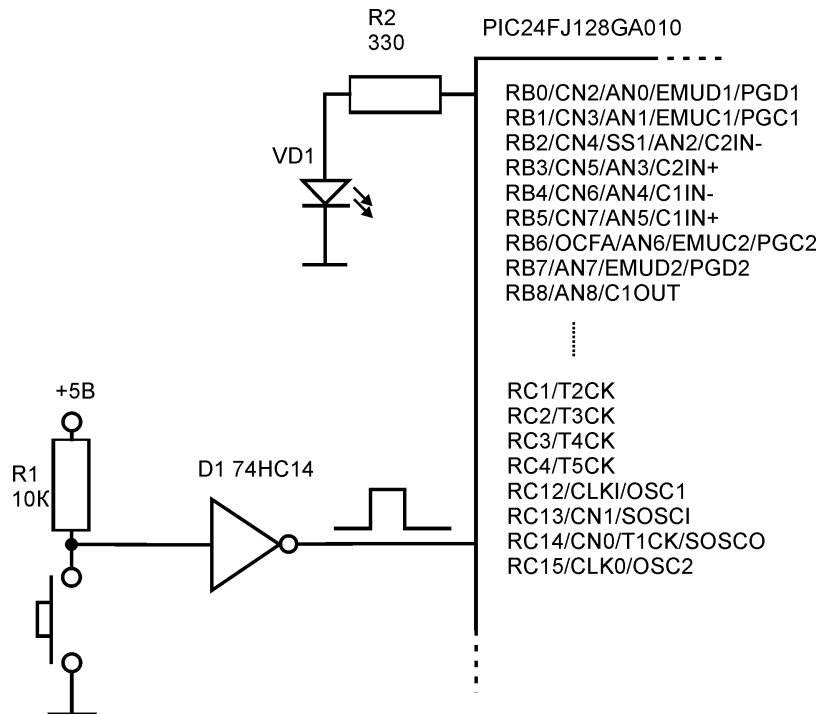

Рис. 6.4. Аппаратная часть проекта

В этом проекте используется внутренний тактовый генератор с частотой 8 МГц. Схема работает следующим образом: при нажатии кнопки на входе T1CK формируется перепад напряжения из НИЗКОГО уровня в ВЫСОКИЙ. При этом запускается Таймер 1, который выполняет инвертирование бита 0 порта B, в результате чего светодиод периодически включается и выключается. Интервал переключения задается программным способом и для нашего проекта устанавливается равным 2 с. Если отпустить кнопку, то на входе T1CK появится НИЗКИЙ уровень и таймер остановится. В результате бит 0 порта B перестанет переключаться и светодиод останется в последнем перед отключением таймера состоянии.

С помощью мастера проектов среды MPLAB IDE создадим проект для микроконтроллера PIC24FJ128GA010 и добавим в него файл со следующим исходным текстом:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  SYSCLK       8000000
#define  t1           2
#define  PREG         SYSCLK/2*t1/256

#define  DELAY        31250
#define  PORTB_0      PORTBbits.RB0

void  main(void)
{
    AD1PCFG  =  0xffff;
    TRISB  =  0xfffe;
    TRISC  =  0xffff;

    PR1  =  PREG;
    TMR1  =  0;
    T1CON  =  0x8070;

    while  (1)
      {
        while  (TMR1  <  DELAY);
        TMR1  =  0;
        PORTB_0  =  ~PORTB_0;
      }
}
```

Для нашего проекта выбираем интервал переключения (константа `t1`), равный 2 с. Значение регистра периода (константа `PREG`) в этом случае будет равно 31250. Это же значение будет иметь и константа `DELAY`, с которой будет сравниваться содержимое регистра Таймера 1.

Как и в предыдущих примерах, записываем значение `PREG` в регистр периода Таймера 1 (PR1) и значение 0 — в регистр Таймера 1 (TMR1). В регистре управления и состояния T1CON устанавливаем биты TON, биты предделителя для коэффициента деления 256 и бит синхронизации таймера внешним сигналом (бит 6, TGATE):

```
T1CON  =  0x8070;
```

Замечу, что сам Таймер 1 в этом режиме работает от внутреннего тактового генератора. Переключение светодиода осуществляется в цикле опроса и реализовано с помощью операторов

```
while (TMR1 < DELAY);
TMR1 = 0;
PORTB_0 = ~PORTB_0;
```

Как обычно, в начале программы следует установить соответствующие режимы работы портов B и C.

Режим работы таймера с запуском/остановкой от внешнего сигнала можно использовать во многих весьма полезных практических приложениях. Например, можно измерить длительность сигнала, подаваемого на вход TxCK. Принцип измерения длительности сигнала показан на **Рис. 6.5**.

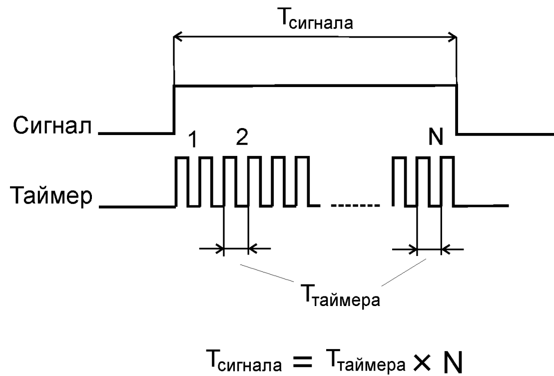

Рис. 6.5. Принцип измерения длительности сигнала

Для измерения длительности сигнала можно использовать следующий принцип: фронт (или спад) сигнала запускает таймер (счетчик), который будет работать до следующего спада (фронта) сигнала. Таким образом, в интервале между изменениями уровней сигнала пройдет определенное число периодов таймера. Если период следования импульсов таймера заранее известен (его можно задать программно) и равен $T_{таймера}$, а количество таких периодов равно N, то длительность сигнала $T_{сигнала}$ можно определить по формуле:

$$T_{сигнала} = T_{таймера} \times N.$$

Разработаем демонстрационный проект, в котором будем измерять длительность импульсного сигнала, подаваемого на вход T1CK микроконтроллера. Аппаратная часть проекта реализуется достаточно просто: к выводу T1CK подсоединяется источник импульсного сигнала. Принципиально несложно измерить длительность любого, не только чисто цифрового сигнала, но для этого на выходе источника сигнала следует включить формирователь сигнала прямоугольной формы. Для этого можно использовать, например, триггер Шмита 74HC14 или аналоговый компаратор с высокой скоростью нарастания выходного напряжения. В качестве источника тактовой частоты нашей схемы используется внутренний генератор 8 МГц.

Программную часть, как обычно, создаем в среде MPLAB IDE. Исходный текст программы представлен далее:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define SYSCLK      8000000
#define t1          0.01
#define PREG        SYSCLK/2*t1/256

#define DELAY       156

unsigned int cnt = 0;
unsigned int time1;


void __attribute__ ((interrupt)) _T1Interrupt(void)
{
    time1 = cnt;
    cnt = 0;
  _T1IF = 0;
}

void main(void)
{
    TRISC = 0xffff;

    _T1IF = 0;
    _T1IP = 3;
  _T1IE = 1;

    PR1 = PREG;
    TMR1 = 0;
    T1CON = 0x8070;
    while (1)
      {
        if (_T1IF != 1)
          {
            while (TMR1 < DELAY);
            cnt++;
            TMR1 = 0;
          }
      }
}
```

В этой программе мы воспользуемся прерыванием Таймера 1. Если запуском/остановкой таймера управляет внешний сигнал, то прерывание возникает только в том случае, если на входе T1CK происходит смена ВЫСОКОГО уровня сигнала на НИЗКИЙ. Прерывание не вызывается переполнением таймера, как это имеет место в обычном режиме. Таким образом, в обработчике прерывания Таймера 1 можно организовать обработку значения счетчика, в кото-

ром на момент возникновения прерывания будет храниться количество периодов (число N, см. **Рис. 6.5**).

В программе объявлено несколько констант и переменных. Константа `t1` равна интервалу срабатывания Таймера 1 (0.01 с). Переменная `cnt` хранит текущее значение количества периодов, прошедших с момента запуска Таймера 1 перепадом напряжения 0—1 на входе T1CK. В переменную `time1` помещается значение счетчика `cnt` в момент возникновения прерывания (перепад 1—0 на входе T1CK). Обработчик прерывания `_T1Interrupt` сохраняет количество периодов в переменной `time1`, обнуляет счетчик cnt и сбрасывает флаг прерывания `_T1IF`.

Длительность импульсного сигнала на входе T1CK (обозначим ее T) после завершения работы обработчика прерывания можно вычислить по формуле:

$$T = t1 \times time1.$$

Если в обработчике прерывания в переменную `time1` записывается значение 178, то длительность импульсного сигнала будет равна $0.01 \times 178 = 1.78$ секунды. Поскольку это демонстрационная программа, то мы не заботимся о высокой точности отсчета временных интервалов. В приложениях реального времени, особенно при измерении небольших интервалов времени, нужно учитывать и другие факторы, влияющие на точность: время выполнения команд микроконтроллера, время инициализации прерывания и т. д.

До сих пор мы рассматривали функционирование 16-битного таймера. Напомню, что в микроконтроллерах PIC24F допускается работа таймеров и в 32-битном режиме.

6.2. Работа таймеров в 32-битном режиме

Для работы в 32-битном режиме могут использоваться таймеры типа B и C. В нашем примере 32-битный таймер будет реализован на Таймере 2 и Таймере 3. Аппаратная часть нашего проекта показана на **Рис. 6.2**. Здесь в качестве источника тактового сигнала используется внутренний тактовый генератор с частотой 8 МГц. Программная часть разработана с помощью мастера проектов среды MPLAB IDE. Текст программы будет следующим:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  SYSCLK       8000000
#define  t1           8
#define  PREGCONST    SYSCLK/2*t1

#define  PORTB_0  PORTBbits.RB0

void  __attribute__   ((interrupt))  _T3Interrupt(void)
{
  PORTB_0  =  ~PORTB_0;
```



```
  _T3IF  =  0;
}

void  main(void)
{
    long  PREG  =  PREGCONST;
    int  *pPREG  =  &PREG;

    AD1PCFG  =  0xffff;
    TRISB  =  0xfffe;

    T2CON  =  0x00;
    T3CON  =  0x00;
    TMR3  =  0x00;
    TMR2  =  0x00;

    PR2  =  *pPREG++;
    PR3  =  *pPREG;

    _T3IP  =  0x01;
    _T3IF  =  0;
    _T3IE  =  1;
    T2CONbits.T32  =  1;
    T2CONbits.TON  =  1;
    while  (1);
}
```

В данной программе мы используем Таймер 2 как базовый таймер типа B, а Таймер 3 — как базовый таймер типа C. При 32-битных операциях с таймерами бит T32 устанавливается в регистре T2CON таймера типа B. Если Таймер 2 и Таймер 3 используются совместно в качестве 32-битного таймера, то регистр T3CON вообще не используется. В качестве регистра управления и состояния используется только регистр T2CON. Тем не менее, если в такой 32-битной конфигурации используется прерывание, то его настройка выполняется для Таймера 3. В регистре Таймера 2 TMR2 будет содержаться младшее 16-битное слово, а в регистре Таймера 3 TMR3 — старшее.

При каждом переполнении регистра Таймера 2 TMR2 будет увеличиваться значение регистра TMR3. Прерывание инициируется при переполнении обоих регистров.

В нашей программе мы переключаем бит 0 порта B каждые 8 секунд. Наш 32-битный таймер синхронизируется тактовой частотой процессора, равной 8 МГц, а интервал времени срабатывания таймера выбран равным 8 секундам. Константа `PREGCONST` определяет значение, которое должно быть записано в регистры периода `PR2` (младшее слово) и `PR3` (старшее слово). В данном случае значение `PREGCONST` равно 28000000.

Для хранения значения `PREGCONST` используется 32-битная переменная `PREG` типа `long`. Младшее слово заносится в регистр периода `PR2`, а старшее — в `PR3`, для чего используется 16-битный указатель `pPREG`:

```
PR2  =  *pPREG++;
PR3  =  *pPREG;
```

Как уже упоминалось, для управления 32-битным таймером используется только регистр T2CON. Здесь устанавливаются значение предделителя 1:1, бит T32 и бит запуска таймера TON:

```
T2CONbits.T32  =  1;
T2CONbits.TON  =  1;
```

Обработка прерывания для Таймера 3 настраивается обычным способом. В функции-обработчике прерывания `_T3Interrupt` всего два оператора: инверсии бита 0 порта A и сброса флага прерывания `_T3IF`. Это, собственно, все, что касается программной части этого проекта.

Как и 16-битный, 32-битный таймер можно синхронизировать внешним источником тактовых импульсов, например, как показано на **Рис. 6.6**.

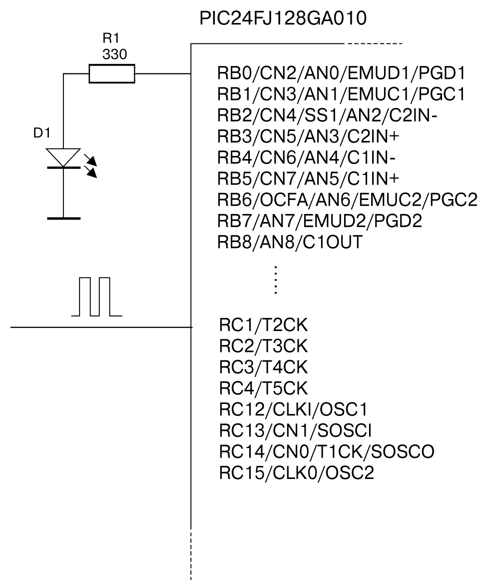

Рис. 6.6. Синхронизация внешним сигналом 32-битного таймера

Здесь внешний сигнал синхронизации подается на вход T2CK (напомню, что 32-битный таймер мы образовали из 16-битных таймеров 2 и 3). Для данной схемы (**Рис. 6.6**) несложно написать программное обеспечение. Если, например, частота внешнего тактового сигнала составляет 1 кГц, а светодиод должен переключаться каждые 13 с, то исходный текст программы будет выглядеть так:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  EXTCLK    1000
```



```
#define  t1            13
#define  PREGCONST     EXTCLK*t1

#define  PORTB_0  PORTBbits.RB0

void  __attribute__   ((interrupt))  _T3Interrupt(void)
{
  PORTB_0  =  ~PORTB_0;
  _T3IF  =  0;
}

void  main(void)
{
    long  PREG  =  PREGCONST;
    int  *pPREG  =  &PREG;

    AD1PCFG  =  0xffff;
    TRISB  =  0xfffe;
    TRISC  =  0xffff;

    T2CON  =  0x00;
    T3CON  =  0x00;
    TMR3  =  0x00;
    TMR2  =  0x00;

    PR2  =  *pPREG++;
    PR3  =  *pPREG;

    _T3IP  =  0x03;
    _T3IF  =  0;
    _T3IE  =  1;
    T2CON  =  0x800A;
    while  (1);
}
```

В этой программе константа `EXTCLK` определяет частоту внешнего сигнала (1000 Гц), константа `t1` — интервал времени переключения (13 с), а `PREGCONST` — содержимое регистров периода `PR2` и `PR3`. Значение константы `PREGCONST` сохраняется в 32-битной переменной `PREG`, а для загрузки регистров периода используются операторы

```
PR2  =  *pPREG++;
PR3  =  *pPREG;
```

Как и в предыдущем примере, переключение бита 0 порта B осуществляется в обработчике `_T3Interrupt`, а само прерывание конфигурируется операторами

```
_T3IP  =  0x03;
_T3IF  =  0;
_T3IE  =  1;
```

Поскольку в данной 32-битной конфигурации для управления используется регистр T2CON, то с помощью оператора

```
T2CON  =  0x800A;
```

устанавливаются бит T32, бит разрешения внешней синхронизации (TCS) и бит запуска (TON). Значение предделителя в этой конфигурации выбирается равным 1:1.

6.3. Часы реального времени

В рассмотренных проектах мы изучили, как можно получать различные временные интервалы для приложений реального времени. С помощью 32-битных таймеров можно получить достаточно длительные интервалы времени – от единиц до десятков и сотен секунд. Но для получения очень длинных интервалов времени, не зависящих от состояния микроконтроллера, в микроконтроллеры семейства PIC24F включен модуль часов/календаря реального времени (Real-Time Clock and Calendar, RTCC), который характеризуется такими параметрами:

- позволяет работать с часами, минутами и секундами;
- позволяет работать в формате календаря: неделя/день/месяц/год;
- имеет функцию «будильника»;
- диапазон задаваемого времени — 2000...2099 год;
- коррекция високосных лет;
- позволяет работать со значениями в формате BCD, что удобно для портативных систем;
- допускает подстройку временных интервалов пользователем;
- погрешность ±2.64 с в месяц.

Для работы модуля RTCC требуется внешний источник тактового сигнала с частотой 32768 Гц. При установленной опции «будильника» на выводе RTCC будет генерироваться сигнал. Особенностью этого модуля является то, что он может работать продолжительное время без «вмешательства» процессора, к тому же он оптимизирован по энергопотреблению, что позволяет длительное время обходиться одной батарейкой. Дискретность отсчета времени составляет 1 с.

Программирование режима реального времени на уровне регистров — занятие довольно утомительное, но здесь можно пойти по другому пути. Компилятор MPLAB C для PIC24 включает целый ряд библиотечных функций, предназначенных для работы с часами/календарем реального времени. Чтобы воспользоваться этой возможностью, следует включить в программный проект библиотечный файл libpPIC24Fxxx-elf.a или libpPIC24Fxxx-coff.a (в зависимости от того, какой тип объектного файла генерирует компилятор). Кроме того, в текст программы следует включить файл rtcc.h.

Приведу исходный текст простейшей программы для работы с часами реального времени:

```
#include  <p24fj128ga010.h>
#include  <rtcc.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_EC&FNOSC_SOSC)

#define  PORTB_0  PORTBbits.RB0
```



```
void  main(void)
{

    rtccTime  wTime;
    rtccTime  rTime;

    AD1PCFG  =  0xffff;
    TRISB  =  0xfffe;


    wTime.f.rsvd  =  0;
    wTime.f.sec  =  0x10;
    wTime.f.min  =  0x07;
    wTime.f.hour  =  0x10;

    RtccInitClock();
    RtccWrOn();

    RtccWriteTime(&wTime,  TRUE);
    do  {
         RtccReadTime(&rTime);
        }  while  (rTime.f.sec  !=  0x15);
    PORTB_0  =  ~PORTB_0;
    while  (1);
}
```

Для работы с интервалами времени в программе следует определить одну или несколько специальных переменных типа `rtccTime`. Такая переменная представляет собой структуру, содержащую элементы `sec`, `min` и `hour`, в которые можно записывать или из которых можно считывать время. Наша программа переключит бит 0 порта B в противоположное значение по прошествии интервала в 5 с. Для этого в переменную `wTime` с помощью функции `RtccWriteTime` записывается время 10 часов 7 минут и 10 секунд, после чего программа считывает текущее значение секунд в другую переменную `rTime` и сравнивает ее со значением 0x15. Обратите внимание, что интервалы времени (часы, минуты, секунды) имеют формат BCD.

При запуске этой программы необходимо инициализировать часы реального времени с помощью функций `RtccInitClock` и `RtccWrOn`. Затем в цикле `do…while` с помощью функции `RtccReadTime` непрерывно считывается время, и по достижении нужного значения выполняется переключение бита 0 порта B. С помощью этих библиотечных функций можно реализовать и другие полезные функции модуля часов реального времени, например установить будильник, который будет вызывать функцию-обработчик в определенный момент времени.

ГЛАВА 7

ИНТЕРФЕЙС SPI МИКРОКОНТРОЛЛЕРОВ PIC24F

Многообразие выпускаемых в настоящее время электронных компонентов и устройств, кроме положительных моментов, добавляет и целый ряд проблем разработчику встроенных систем. Одной из таких проблем является интерфейс микроконтроллера с устройством. В микроконтроллере PIC24 имеется достаточное количество портов ввода/вывода, что теоретически позволяет подключать самые разные, сколь угодно сложные устройства. Тем не менее, для стандартизации подключения своих устройств большинство производителей оборудования используют целый ряд аппаратно-программных интерфейсов, которые обеспечивают унифицированные протоколы обмена данными. Наиболее популярными протоколами обмена данными являются SPI, I²C, SMBus, Bluetooth и др. Ведущими производителями электронных компонентов разработаны многочисленные устройства, совместимые с этими популярными протоколами, которые можно использовать в проектах на базе микроконтроллеров.

В микроконтроллерах большинство этих протоколов можно реализовать «вручную», используя для этого порты ввода/вывода и программное обеспечение. Тем не менее, для упрощения разработки интерфейсов фирмы-производители микроконтроллеров размещают на кристалле специализированные – модули SPI, I²C и др. Не являются исключением и микроконтроллеры семейства PIC24F — все модели семейства имеют как минимум по два, а то и по три интерфейса SPI и I²C. Разработчики микроконтроллеров автоматизируют процесс обмена данными по таким интерфейсам, делая вмешательство программиста минимальным, что значительно экономит время при разработке систем.

Рассмотрим наиболее популярные интерфейсы SPI и I²C и их реализацию в системах на базе PIC24. Эта глава посвящена интерфейсу SPI, а следующая — интерфейсу I²C.

Обмен данными по интерфейсу (протоколу) SPI (Serial Peripheral Interface) можно представить упрощенной диаграммой (**Рис. 7.1**).

Интерфейс SPI обеспечивает обмен данными в синхронном режиме. Поскольку данные передаются синхронно с тактовой частотой, скорость обмена может быть очень высокой. К преимуществам этого протокола следует отнести и возможность обмена данными в полнодуплексном режиме.

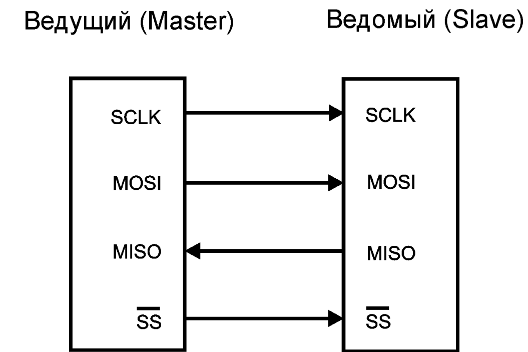

Рис. 7.1. Схема обмена данными по протоколу SPI

В большинстве случаев интерфейс связывает два устройства, одно из которых является «ведущим» (Master), а другое – «ведомым» (Slave). Ведущий инициирует обмен данными и синхронизирует их прием и/или передачу. Несколько устройств могут функционировать как ведомые, при этом ведущий должен выбирать ведомого, подавая сигнал выбора (slave select, chip select) на конкретное устройство.

Сигнал SCLK (Serial Clock) представляет собой последовательность синхронизирующих импульсов, генерируемых ведущим. Передача и прием данных привязаны к передним или задним фронтам синхросигнала. По линии MOSI (Master Output, Slave Input) передача данных осуществляется от ведущего к ведомому, а линия MISO (Master Input, Slave Output) служит для приема данных от ведомого. Сигнал $\overline{SS}$ (Slave Select) генерируется ведущим для конкретного ведомого, разрешая сеанс обмена данными, при этом активным уровнем такого сигнала является уровень логического нуля.

Очень часто ведущие производители оборудования используют и альтернативные обозначения сигналов интерфейса. Например, сигнал синхронизации SCLK часто обозначается как SCK, MISO может обозначаться как SDI, DI или SI. Сигнал MOSI имеет альтернативные обозначения SDO, DO и SO. Наконец, сигнал выбора ведомого часто встречается под альтернативными обозначениями nCS и CS.

Интерфейс SPI является двунаправленным, тем не менее, большинство устройств, выпускаемых промышленностью, обычно используют усеченную (однонаправленную) версию этого протокола. Например, большинство аналого-цифровых преобразователей используют модель, в которой ведущим является устройство, получающее данные (микроконтроллер, компьютер и т. д.). В этих случаях синхронизацию передачи/приема данных выполняет инициатор обмена.

7.1. Аппаратно-программная реализация SPI в микроконтроллерах PIC24F

Микроконтроллеры PIC24F, в зависимости от модели устройства, могут включать два или три модуля интерфейса SPI, которые обозначаются как SPI1, SPI2 и SPI3 (три интерфейса имеют устройства с увеличенным объемом памяти

программ). Все интерфейсы совершенно идентичны по аппаратно-программной архитектуре, а выводы микроконтроллеров, относящиеся к SPI, имеют следующее назначение:

- SDIx — ввод последовательных данных;
- SDOx — вывод последовательных данных;
- SCKx — вход или выход тактовых импульсов;
- $\overline{SS}$x — вход выбора ведомого.

Модуль SPI может быть настроен для работы в 2-, 3- или 4-проводной конфигурации. В 3-проводной конфигурации не используется линия $\overline{SS}$x, а в двухпроводной — не задействованы SDOx и $\overline{SS}$x. Функциональная схема модуля SPIx (x = 1, 2, 3) показана на **Рис. 7.2**.

Рис. 7.2. Функциональная схема модуля SPI

По этой схеме можно понять, как осуществляется обмен данными по интерфейсу SPI. Пусть, например, выполняется передача байта данных из памяти микроконтроллера другому устройству, присоединенному через интерфейс SPI. В этом случае байт данных поступает через внутреннюю шину данных в регистр буфера приема/передачи SPIxBUF. При разрешении передачи байт данных помещается в сдвиговый регистр SPIxSR, откуда каждый из восьми битов (от старшего к младшему) выдвигается на линию SDOx синхронно с тактовым сигналом SCKx, который может быть либо внешним, либо формироваться самим микроконтроллером. В зависимости от конфигурации интерфейса выдача битов может выполняться либо по нарастающему, либо по спадающему фронту импульсов тактового сигнала.

Сам тактовый сигнал может формироваться схемой деления частоты из внутреннего тактового сигнала частотой F_{CY}. Коэффициент деления этой схемы определяется битами 1:0 регистра управления SPIxCON (первый предделитель) и битами 4:2 этого же регистра (второй предделитель). Если устройство работает в роли «ведомого», то сигнал синхронизации приходит от «ведущего».

При приеме данных по интерфейсу SPI последовательность операций иная. В этом случае каждый бит данных по фронту/спаду строба SCKx вдвигается в сдвиговый регистр SPIxSR, откуда сформированный байт (или слово) данных поступает в регистр буфера приема/передачи SPIxBUF. Содержимое буфера далее может быть считано программой.

Если обмен данными должен синхронизироваться «окном», то в этом случае используется сигнал $\overline{SS}$, который разрешает или запрещает прием данных ведомым. Как правило, обмен данными разрешается при НИЗКОМ уровне сигнала $\overline{SS}$ и запрещается при ВЫСОКОМ уровне этого сигнала. Функционирование интерфейса SPI легче понять, если проанализировать две временные диаграммы (**Рис. 7.3** и **7.4**).

Рис. 7.3. Временная диаграмма работы 3-проводного интерфейса SPI

Рис. 7.4. Временная диаграмма работы 2-проводного интерфейса SPI

На **Рис. 7.3** показана временная диаграмма работы интерфейса SPI в 3-проводном режиме. В данном примере интерфейс SPI работает на передачу, поэтому на линию SDO поразрядно выдвигается байт данных. Работа интерфейса ведомого разрешается при НИЗКОМ уровне сигнала $\overline{SS}$ и запрещается при

ВЫСОКОМ уровне. В изображенной конфигурации каждый бит данных, начиная со старшего, выдвигается на линию SDO по фронту SCK, поэтому ведомое устройство может считывать бит после этого перепада. Если сигнал $\overline{SS}$ становится неактивным, то даже при дальнейшем поступлении импульсов синхронизации и наличии данных прием их ведомым будет невозможен. При других настройках биты данных могут, например, фиксироваться по спаду синхронизирующего сигнала SCK. Подобный 3-проводной интерфейс чаще всего используется в аналого-цифровых преобразователях и первичных преобразователях физических величин на основе АЦП.

Окно, генерируемое сигналом $\overline{SS}$, фиксирует только определенное число бит данных (8 в данном примере), отсекая остальные, даже при наличии синхронизирующих импульсов на линии SCK. Если сигналы синхронизации ($\overline{SS}$ и SCK) генерируются микроконтроллером, то он выступает в роли «ведущего». Если же синхронизация выполняется другим устройством, а микроконтроллер работает только с линиями SDI/SDO, то он будет выступать в роли «ведомого».

Интерфейс, состоящий из двух линий — синхронизации (SCK) и входных/выходных данных (SDI/SDO), часто используется для осуществления обмена между самими микроконтроллерами PIC24, поскольку синхронизацию начала/конца обмена данными обеспечить несложно (**Рис. 7.4**).

Двухпроводной интерфейс легко реализовать в режиме «ведущий» — «ведомый» для двух микроконтроллеров, когда импульсы синхронизации SCK «ведущего» синхронизируют прием данных «ведомым». При такой конфигурации вывод тактового сигнала интерфейса SCKx «ведущего» соединяется с выводом тактового сигнала «ведомого», а выводы данных SDI/SDO «ведущего» микроконтроллера соединяются с выводами SDO/SDI «ведомого». Естественно, что такую конфигурацию следует настроить соответствующим образом в обоих микроконтроллерах.

Микроконтроллеры PIC24F могут обмениваться по интерфейсу SPI только 8- или 16-битными данными, поэтому при подключении внешних устройств, требующих иного числа бит (например, 12, 14, 20 или 24, что характерно для микросхем АЦП), необходимо использовать определенные программные ухищрения для обработки данных такой разрядности.

Для программирования обмена данными по интерфейсу SPI в микроконтроллерах PIC24F имеется несколько регистров, функции и назначение отдельных битов которых мы сейчас рассмотрим. Программный интерфейс SPI включает следующие регистры:

- SPIxBUF — буфер данных, который может содержать передаваемые или принимаемые данные. Этот буфер является общим «виртуальным» пространством памяти для регистра передачи SPIxTXB, в который поступают передаваемые данные, и регистра приема SPIxRXB, в который поступают принимаемые с линии данные;
- SPIxCON1 и SPIxCON2 — регистры управления для установки различных параметров обмена;
- SPIxSTAT — регистр состояния, в котором фиксируются состояния интерфейса;

- SPIxSR — 16-битный сдвиговый регистр, который выполняет поразрядную выдачу данных на выход или поразрядное считывание данных, поступающих на вход.

Важное замечание: биты конфигурации в регистрах управления SPIxCON1 и SPIxCON2 невозможно установить при работающем интерфейсе, поэтому перед конфигурированием SPI-интерфейса следует его отключить, сбросив бит SPIEN в соответствующем регистре состояния SPIxSTAT.

Рассмотрим назначение отдельных битов регистров управления и состояния. В **Табл. 7.1** приведено описание битов регистра SPIxCON1.

Таблица 7.1. Назначение битов регистра SPIxCON1

Позиция бита	Обозначение	Описание
15–13		Читаются как 0
12	DISSCK	Позволяет отключить вывод SCKx (только если задан режим «ведущего»): 1 – внутренняя функция тактирования SPI отключена, вывод микроконтроллера работает как цифровой ввод/вывод; 0 – на выводе генерируются тактовые импульсы для SPI-интерфейса
11	DISSDO	Позволяет отключить вывод SDOx: 1 – вывод SDOx не используется в модуле SPI и работает как цифровой ввод/вывод; 0 – вывод SDOx управляется модулем SPI-интерфейса
10	MODE16	Разрядность данных при обмене по SPI: 1 – используются 16-битные данные; 0 – используются 8-битные данные
9	SMP	Определяет режим захвата входных данных. В режиме «ведущего» установки бита имеют такой смысл: 1 – входные данные читаются в конце интервала времени, в течение которого они выставлены; 0 – входные данные читаются посредине интервала. В режиме «ведомого» бит должен быть сброшен
8	CKE	Определяет, на каком шаге происходит изменение данных на выходе: 1 – при переходе из активного состояния в режим ожидания; 0 – при переходе из режима ожидания в активное состояние
7	SSEN	Способ использования бита SSx (в режиме «ведомого»): 1 – бит SSx используется в режиме «ведомого»; 0 – бит SSx не используется и управляется как порт ввода/вывода
6	CKP	Выбор полярности импульса синхронизации: 1 – режим ожидания соответствует высокому, а активный режим – низкому уровню; 0 – режим ожидания соответствует низкому, а активный режим – высокому уровню
5	MSTEN	Разрешение работы в режиме «ведущего»: 1 – работа в режиме «ведущего»; 0 – работа в режиме «ведомого»
4–2	SPRE2–SPRE0	Установка коэффициента деления второго предделителя (режим «ведущего»): 111 – соотношение 1:1

Таблица 7.1. Назначение битов регистра SPIxCON1 (окончание)

Позиция бита	Обозначение	Описание
		110 – соотношение 2:1 . . . 000 – соотношение 8:1
1–0	PPRE1–PPRE0	Установка коэффициента деления второго предделителя (режим «ведущего»): 111 – соотношение 1:1 110 – соотношение 4:1 001 – соотношение 16:1 000 – соотношение 64:1

Назначение битов регистра управления SPIxCON2 приведено в **Табл. 7.2**.

Таблица 7.2. Назначение битов регистра SPIxCON2

Позиция бита	Обозначение	Описание
15	FRMEN	Поддержка оконного интерфейса SPI: 1 – при обмене данными используется «окно»; 0 – при обмене данными не используется «окно»
14	SPIFSD	Режим синхронизации окна: 1 – сигнал «окна» устанавливает «ведомый» при вводе; 0 – сигнал «окна» устанавливает «ведущий» при выводе
13	SPIFPOL	Установка полярности сигнала «окна»: 1 – активный уровень «окна» соответствует высокому уровню сигнала; 0 – активный уровень «окна» соответствует низкому уровню сигнала
12–2		Читается как 0
1	SPIFE	Смещение сигнала «окна» по отношению к стробу тактовой частоты: 1 – устанавливается одновременно с первым импульсом частоты синхронизации; 0 – устанавливается до прихода первого импульса синхронизации
0	SPIBEN	Разрешение работы с расширенным буфером: 1 – использование расширенного буфера разрешено; 0 – использование расширенного буфера запрещено (по умолчанию)

Назначение битов регистра управления SPIxSTAT приведено в **Табл. 7.3**.

Таблица 7.3. Назначение битов регистра SPIxSTAT

Позиция бита	Обозначение	Описание
15	SPIEN	Разрешение работы интерфейса SPI: 1 – разрешает работу SPI-интерфейса и конфигурирует выводы SCKx, SDOx и SSx для SPI; 0 – запрещает работу интерфейса SPI

Таблица 7.3. Назначение битов регистра SPIxSTAT (окончание)

Позиция бита	Обозначение	Описание
14		Читается как 0
13	SPISIDL	Работа в режиме «холостого хода» микроконтроллера: 1 – прекращаются все операции; 0 – обмен данными продолжается
12–11		Читается как 0
10–8	SPIBEC2–SPIBEC0	Подсчет элементов данных в буфере SPI. Если задан режим «ведущего», то количество элементов подсчитывается, если задан режим «ведомого», то подсчет не производится
7	SRMPT	Показывает состояние регистра сдвига (SPIxSR) (только в расширенном режиме): 1 – регистр сдвига пуст и может принимать или отправлять данные; 0 – регистр сдвига содержит данные
6	SPIROV	Бит переполнения при приеме: 1 – получен следующий байт/слово. Микроконтроллер не прочитал предыдущие данные; 0 – переполнение отсутствует
5	SRXMPT	Бит состояния FIFO-буфера приема (только для расширенного режима): 1 – FIFO-буфер приема пуст; 0 – FIFO-буфер приема не пуст
4–2	SISEL2–SISEL0	Установка режима прерывания буфера SPI (только в расширенном режиме): 111 – прерывание инициируется при переполнении буфера передачи; 110 – прерывание инициируется после того, как последний бит сдвинут в регистр сдвига SPIxSR; 101 – прерывание инициируется после того, как последний бит выдвинут из регистра сдвига SPIxSR; . . . 000 – прерывание инициируется после того, как прочитан последний байт буфера
1	SPIxTBF	Состояние буфера передачи: 1 – передача данных не началась, буфер SPIxTXB заполнен; 0 – началась передача данных, буфер передачи SPIxTXB пуст
0	SPIxRBF	Состояние буфера приема: 1 – прием данных завершен, буфер приема SPIxRXB заполнен; 0 – прием данных не завершен, буфер приема SPIxRXB пуст

7.2. Практическое программирование обмена данными по SPI

Рассмотрим некоторые практические аспекты программирования интерфейса SPI в микроконтроллерах PIC24F. Многие устройства, выпускаемые промышленностью, преобразуют последовательный код на входе в параллельный на выходе с использованием тех или иных модификаций протокола SPI. В нашем первом проекте данные будут выводиться через интерфейс SPI1 микроконтрол-

лера PIC24FJ128GA010 в регистр сдвига на микросхеме CD4094, на выходе которой получим параллельный код, управляющий включением/выключением светодиодов по принципу «бегущей строки». Переключение светодиодов осуществляется с интервалом в 1 секунду. На **Рис. 7.5** представлена схема аппаратной части проекта.

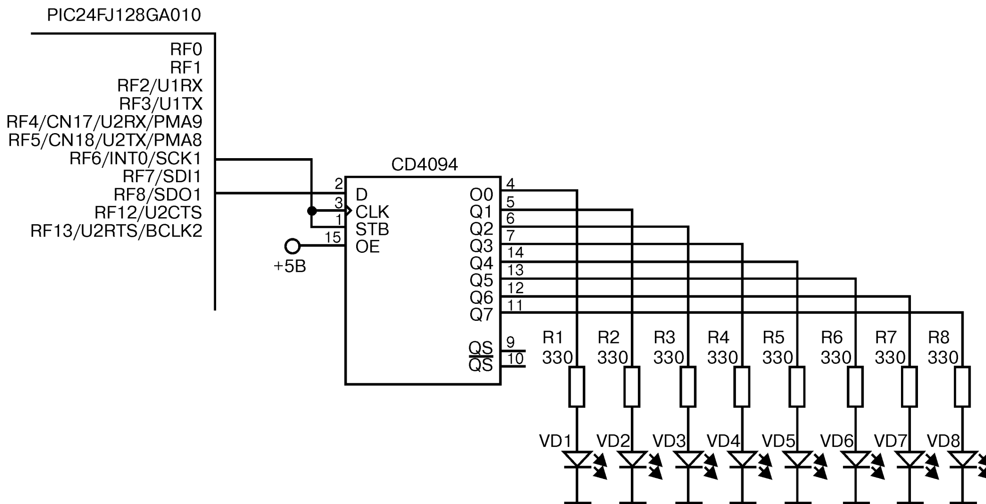

Рис. 7.5. Аппаратная часть проекта

Для обмена данными в этой схеме используется интерфейс SPI1 микроконтроллера. Данные поступают с вывода SDO1 микроконтроллера на вход D регистра сдвига, а синхронизация передачи выполняется микроконтроллером по линии SCK1. Тактовые импульсы поступают на входы CLK и STB регистра CD4094.

Программную часть проекта разработаем в среде MPLAB IDE с помощью мастера проектов. Включим в проект Си-файл со следующим исходным текстом:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  SYSCLK     8000000
#define  t1         1
#define  PREG       SYSCLK/2*t1/256

char  c1;

void  __attribute__ ((interrupt))  _T1Interrupt(void)
{
  _T1IF  =  0;
  SPI1BUF  =  c1;
  while  (_SPI1IF  ==  0);
```



```
  _SPI1IF  =  0;
  c1  =  c1<<1;
  if  (c1  ==  0x0)
      c1  =  0x1;
}


void  SPI1Init(void)
    {
      SPI1STATbits.SPIEN  =  0;
      SPI1CON1  =  0;
      SPI1CON1bits.DISSCK  =  0;
      SPI1CON1bits.DISSDO  =  0;
      SPI1CON1bits.MODE16  =  0;
      SPI1CON1bits.MSTEN  =  1;
      SPI1STATbits.SPIEN  =  1;
    }

void  main(void)
{
      c1  =  0x1;
      PR1  =  PREG;
      TMR1  =  0;
      T1CON  =  0x8030;
      _T1IF  =  0;
      _T1IP  =  4;
      _T1IE  =  1;
      _SPI1IF  =  0;
      SPI1Init();
      while(1);
}
```

В этой программе определена функция `SPI1Init`, которая выполняет инициализацию интерфейса SPI1. Непосредственно перед инициализацией нужно отключить интерфейс SPI путем сброса бита SPIEN регистра состояния SPI1STAT:

```
SPI1STATbits.SPIEN  =  0;
```

Поскольку мы будем по отдельности изменять каждый бит конфигурации, то предварительно желательно очистить регистр управления интерфейса SPI1:

```
SPI1CON1  =  0;
```

В данной конфигурации интерфейса мы будем использовать две сигнальных линии — синхронизации SCK и выходных данных SDO, которые будут управляться самим микроконтроллером. По этой причине биты DISSCK и DISSDO регистра управления SPI1CON должны быть сброшены:

```
SPI1CON1bits.DISSCK  =  0;
SPI1CON1bits.DISSDO  =  0;
```

Далее в нашем проекте используется 8-битный сдвиговый регистр CD4094, поэтому следует указать количество обрабатываемых битов. Для 8-битной посылки данных бит MODE16 регистра управления должен быть сброшен:

```
SPI1CON1bits.MODE16  =  0;
```

Микроконтроллер в данном проекте будет выполнять синхронизацию обмена данными, т. е. работать в режиме «ведущего», поэтому следует установить бит MSTEN:

```
SPI1CON1bits.MSTEN  =  1;
```

На данный момент все необходимые установки закончены, поэтому включаем SPI-интерфейс:

```
SPI1STATbits.SPIEN  =  1;
```

Это все, что касается функции инициализации SPI-интерфейса `SPI1Init`. В нашей программе не используется режим синхронизации кадра, который задается в регистре SPI1CON2, поэтому содержимое этого регистра можно оставить без изменения (по умолчанию покадровая синхронизация не используется).

Поскольку программа выполняет периодические действия по пересылке данных по SPI-интерфейсу, то нам понадобится таймер. В данном случае мы используем 16-битный таймер типа А, в качестве которого будет задействован Таймер 1 микроконтроллера. Пересылка байта данных по SPI-интерфейсу выполняется в обработчике прерывания Таймера 1 `_T1Interrupt`, который будет вызываться каждую секунду.

Собственно пересылка байта данных по SPI осуществляется командой

```
SPI1BUF  =  c1;
```

функции-обработчика. После записи данных в буфер SP1BUF нужно дождаться окончания передачи данных, что и выполняется в цикле `while`:

```
while  (_SPI1IF  ==  0);
```

Здесь анализируется флаг прерывания `_SPI1IF`, который по завершении передачи устанавливается в 1. Затем биты данных (байт `c1`) сдвигаются влево, обеспечивая во времени эффект «бегущей строки». Флаг прерывания `_SPI1IF` должен быть очищен программно в обработчике прерывания таймера.

В основной программе выполняется инициализация Таймера 1 для работы с интервалом времени в 1 с, а также инициализация SPI-интерфейса (функция `SPI1Init`).

В библиотеке 16-битных функций управления периферийными устройствами имеется целый ряд функций, позволяющих облегчить разработку программного интерфейса SPI. Чтобы иметь возможность использовать функции библиотеки, разработчик должен включить в проект файл заголовка spi.h и добавить библиотеку libpPIC24Fxxx-elf.a или libpPIC24Fxxx-coff.a (тип файла зависит от принятого в компиляторе формата объектного файла). Библиотечный файл находится в каталоге \...\Program Files\Microchip\MPLAB C для PIC24\lib.

Можно выполнить небольшую модификацию исходного текста нашей программы, включив туда библиотечные функции для SPI-интерфейса. С учетом таких изменений исходный текст программы будет выглядеть так:

```
#include  <p24fj128ga010.h>
#include  <spi.h>
```



```
_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  SYSCLK       8000000
#define  t1           1
#define  PREG         SYSCLK/2*t1/256

char  c1;

void  __attribute__   ((interrupt))  _T1Interrupt(void)
{
  _T1IF  =  0;
  WriteSPI1(c1);
  while  (_SPI1IF  ==  0);
  _SPI1IF  =  0;
  c1  =  c1<<2;
  if  (c1  ==  0x0)
      c1  =  0x1;
}

void  main(void)
{
      unsigned  int  SPICON1Value;
      unsigned  int  SPICON2Value;
      unsigned  int  SPISTATValue;

      c1  =  0x1;
      PR1  =  PREG;
      TMR1  =  0;
      T1CON  =  0x8030;  //  Преддделитель  1:256
      _T1IF  =  0;
      _T1IP  =  4;
      _T1IE  =  1;
      _SPI1IF  =  0;
      CloseSPI1();

      SPICON1Value  =  ENABLE_SCK_PIN  &  ENABLE_SDO_PIN  &  SPI_MODE16_OFF  &
                       SPI_SMP_ON  &  SPI_CKE_OFF  &
                       SLAVE_ENABLE_OFF  &
                       CLK_POL_ACTIVE_HIGH  &
                       MASTER_ENABLE_ON  &
                       SEC_PRESCAL_7_1  &
                       PRI_PRESCAL_64_1;

      SPICON2Value  =  FRAME_ENABLE_OFF  &  FRAME_SYNC_OUTPUT;
      SPISTATValue  =  SPI_ENABLE;

      OpenSPI1(SPICON1Value,  SPICON2Value,  SPISTATValue  );
      while(1);
}
```

Мы включили в исходный текст программы файл заголовка:

```
#include  <spi.h>
```

а также несколько библиотечных функций. Вот их назначение:

- функция `CloseSPI1()` отключает модуль SPI. Это может понадобиться либо при завершении работы с интерфейсом, либо при необходимости записи новой конфигурации в регистры SPI1CON1 и SPI1CON2;

- функция `OpenSPI1` выполняет запись битов конфигурации в регистры интерфейса. Смысл мнемонических обозначений можно посмотреть в файле spi.h.

В программе определены три целочисленные переменные (`SPICON1Value`, `SPICON2Value` и `SPISTATValue`), в которые помещается логическая комбинация (AND) всех битов конфигурации для регистров SPI1CON1, SPI1CON2 и SPI1STAT соответственно.

Наш следующий проект более сложен по сравнению с предыдущим и демонстрирует обмен данными по протоколу SPI. В этом проекте используется два микроконтроллера PIC24F, один из них будет передавать данные в режиме «ведущего», а другой — принимать данные в режиме «ведомого». Оба микроконтроллера работают на тактовой частоте 8 МГц. После приема символов «ведомый» микроконтроллер отправляет их на другое устройство или персональный компьютер по асинхронному последовательному порту 2. Аппаратная часть проекта (без схемы подключения асинхронного интерфейса) показана на **Рис. 7.6**.

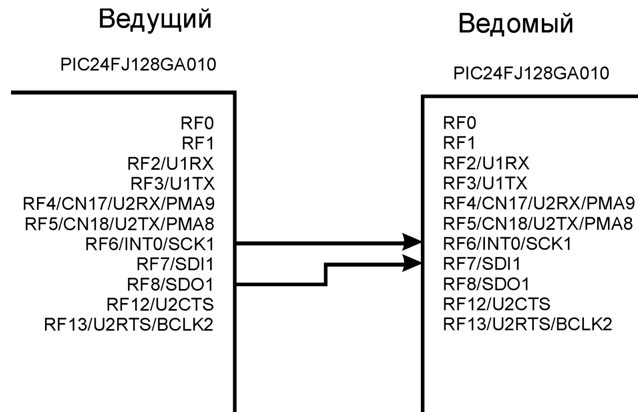

Рис. 7.6. Схема обмена данными по протоколу SPI

Здесь используется двухпроводная конфигурация. Оба микроконтроллера для обмена данными используют интерфейс SPI1. Синхронизация передачи данных выполняется сигналом на выводе SCK1 ведущего, а данные передаются с линии SDO1 ведущего на вход SDI1 ведомого. Программная часть проекта разработана в MPLAB IDE и содержит две отдельные программы: одну – для «ведущего» микроконтроллера, а другую — для «ведомого». Исходный текст программы для ведущего показан далее.

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)
void  SPI1MasterInit(void)
    {
      SPI1STATbits.SPIEN  =  0;
      SPI1CON1  =  0;
```



```
      SPI1CON1bits.DISSCK  =  0;
      SPI1CON1bits.DISSDO  =  0;
      SPI1CON1bits.MODE16  =  0;
      SPI1CON1bits.MSTEN  =  1;
      SPI1CON2bits.FRMEN  =  0;
      SPI1STATbits.SPIROV  =  0;
      SPI1STATbits.SPIEN  =  1;
    }

void  main(void)
{
      char  s1[]  =  "Test  string  for  SPI  Master-Slave  Configuration";
      char  *ps1  =  s1;

      _SPI1IF  =  0;
        SPI1MasterInit();
        while  (*ps1  !=  0)
          {
            SPI1BUF  =  *ps1++;
            while  (_SPI1IF  ==  0);
            _SPI1IF  =  0;
          }
      SPI1STATbits.SPIEN  =  0;
      while(1);
}
```

Программа просто передает текстовую строку (переменная `s1`) от ведущего устройства к ведомому. Настройка канала передачи в режиме «ведущего» выполняется функцией `SPI1MasterInit`. Перед началом конфигурирования необходимо отключить интерфейс SPI, сбросив бит SPIEN. Система сама будет управлять каналом передачи, поэтому в этой функции выбирается режим внутренней синхронизации с помощью следующих установок:

```
SPI1CON1bits.DISSCK  =  0;
SPI1CON1bits.DISSDO  =  0;
```

Программа будет передавать 8-битные символы, поэтому выбирается 8-битный режим работы и функция «ведущего»:

```
SPI1CON1bits.MODE16  =  0;
SPI1CON1bits.MSTEN  =  1;
```

По окончанию настроек следует разрешить работу интерфейса SPI, установив бит SPIEN. После выполнения функции `SPI1MasterInit` в основной программе выполняется последовательная, байт за байтом, передача всех символов строки `s1` до момента, пока не будет достигнут нулевой символ (строка `s1` является стандартной строкой ANSI C с завершающим нулем). Для передачи одного байта достаточно записать его в регистр SPI1BUF, после чего микроконтроллер выполнит все необходимые циклы синхронизации для передачи каждого бита:

```
SPI1BUF  =  *ps1++;
```

Одновременно с записью символа в буфер указатель `ps1` продвигается по строке к следующему символу.

О завершении передачи очередного символа свидетельствует установка флага прерывания `_SPI1IF`, поэтому его значение проверяется в цикле `while`:

```
while  (_SPI1IF  ==  0);
```

По завершению передачи строки «ведомому» интерфейс SPI «ведущего» отключается:

```
SPI1STATbits.SPIEN  =  0;
```

Программа управления приемом данных «ведомого» микроконтроллера выглядит следующим образом:

```
  #include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  SYSCLK          8000000
#define  BAUDRATE2       9600
#define  BAUDRATEREG2    SYSCLK/8/BAUDRATE2-1

#define  UART2_TX_TRIS   TRISFbits.TRISF5
#define  UART2_RX_TRIS   TRISFbits.TRISF4

char  c1;
void  SPI1SlaveInit(void)
    {
      SPI1STATbits.SPIEN  =  0;
      SPI1CON1  =  0;

      SPI1CON1bits.DISSCK  =  0;
      SPI1CON1bits.DISSDO  =  0;
      SPI1CON1bits.MODE16  =  0;
      SPI1CON1bits.MSTEN  =  0;
      SPI1CON1bits.SMP  =  0;
      SPI1CON2bits.FRMEN  =  0;
      SPI1STATbits.SPIROV  =  0;
      SPI1STATbits.SPIEN  =  1;
    }

void  UART2Init()
{
      UART2_TX_TRIS  =  0;
      UART2_RX_TRIS  =  1;
      U2BRG  =  BAUDRATEREG2;

      U2MODE  =  0;
      U2MODEbits.BRGH  =  1;
      U2MODEbits.UARTEN  =  1;
      U2STA  =  0;
      U2STAbits.UTXEN  =  1;
      IFS1bits.U2RXIF  =  0;
}

void   UART2PutChar(char  Ch)
{
```



```
  while(U2STAbits.UTXBF  ==  1);
  U2TXREG  =  Ch;
}

void  __attribute__((__interrupt__))  _SPI1Interrupt(void)
{
_SPI1IF  =  0;
SPI1STATbits.SPIROV  =  0;
}

void  main(void)
{
      int  c1;
      _SPI1IF  =  0;
      _SPI1IE  =  1;

      UART2Init();
      SPI1SlaveInit();
      while  (1)
          {
            while  (_SPI1IF  ==  0);
            c1  =  SPI1BUF;
            if  (c1  ==  0)
                break;
            UART2PutChar(c1);
        }
        SPI1STATbits.SPIEN  =  0;
      while(1);
    }
```

В этой программе, как и в предыдущей, перед приемом данных следует сконфигурировать SPI-интерфейс, для чего используется функция `SPI1SlaveInit`. Здесь устанавливаются почти те же биты, что в функции `SPI1MasterInit`, за исключением бита MSTEN, который, будучи сброшенным, указывает на работу в режиме «ведомого»:

```
SPI1CON1bits.MSTEN  =  0;
```

Для интерфейса SPI1 инициализируется обработчик прерывания, основной функцией которого является очистка флага прерывания и флага переполнения буфера приема SPIROV. Символ, принятый по интерфейсу SPI1, сохраняется в переменной `c1`, и если он равен 0 (конец строки), то происходит выход из цикла чтения `while (1)`. Признаком того, что в буфере приема имеется байт, служит установка флага прерывания `_SPI1IF` в 1, после чего байт можно считывать в переменную `c1`:

```
while  (_SPI1IF  ==  0);
c1  =  SPI1BUF;
```

В программе объявлены и функции управления работой асинхронного последовательного порта 2, который мы подробно рассмотрим в главе 10, а сейчас просто воспользуемся функциями `UART2Init` и `UART2PutChar`. Первая выполняет инициализацию последовательного интерфейса 2, а вторая передает

8-битный символ в асинхронный порт. При выходе из цикла `while` интерфейс SPI1 отключается.

Исходный текст программы можно значительно упростить, если обработку принимаемых по интерфейсу SPI данных выполнить полностью в обработчике прерывания `_SPI1Interrupt`, как показано далее:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  SYSCLK          8000000
#define  BAUDRATE2       9600
#define  BAUDRATEREG2    SYSCLK/8/BAUDRATE2-1

#define  UART2_TX_TRIS   TRISFbits.TRISF5
#define  UART2_RX_TRIS   TRISFbits.TRISF4

char  c1;
void  SPI1SlaveInit(void)
    {
      SPI1STATbits.SPIEN  =  0;
      SPI1CON1  =  0;

      SPI1CON1bits.DISSCK  =  0;
      SPI1CON1bits.DISSDO  =  0;
      SPI1CON1bits.MODE16  =  0;
      SPI1CON1bits.MSTEN  =  0;
      SPI1CON1bits.SMP  =  0;
      SPI1CON2bits.FRMEN  =  1;
      SPI1STATbits.SPIROV  =  0;
      SPI1STATbits.SPIEN  =  1;
    }
void  UART2Init()
{
      UART2_TX_TRIS  =  0;
      UART2_RX_TRIS  =  1;
      U2BRG  =  BAUDRATEREG2;

      U2MODE  =  0;
      U2MODEbits.BRGH  =  1;
      U2MODEbits.UARTEN  =  1;

      U2STA  =  0;
      U2STAbits.UTXEN  =  1;
      IFS1bits.U2RXIF  =  0;
}

void    UART2PutChar(char  Ch)
{
  while(U2STAbits.UTXBF  ==  1);
  U2TXREG  =  Ch;
}

void  __attribute__((__interrupt__))  _SPI1Interrupt(void)
{
IFS0bits.SPI1IF  =  0;
```



```
SPI1STATbits.SPIROV  =  0;
c1  =  SPI1BUF;
if  (c1  !=  0)
    UART2PutChar(c1);
}

void  main(void)
{
      int  c1;
      _SPI1IF  =  0;
      _SPI1IP  =  6;
      _SPI1IE  =  1;

      UART2Init();
      SPI1SlaveInit();
      while(1);
}
```

Как видно из листинга, исходный текст программы значительно упростился. Тем не менее, если система работает в реальном времени, то большие объемы программного кода помещать в обработчик прерывания нежелательно, иначе это скажется на производительности всей системы, особенно если имеются другие обработчики прерываний.

Если разработчик при создании программ данного проекта захочет воспользоваться библиотечными функциями, то тексты программ изменятся. Исходный текст программы для «ведущего» микроконтроллера может выглядеть так:

```
#include  <p24fj128ga010.h>
#include  <spi.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)
void  main(void)
{
      char  s1[]  =  "LIB  SPI-functions  Test!";
      char  *ps1  =  s1;

      unsigned  int  SPICON1Value;
      unsigned  int  SPICON2Value;
      unsigned  int  SPISTATValue;

    CloseSPI1();
      SPICON1Value  = ENABLE_SCK_PIN  &  ENABLE_SDO_PIN  &  SPI_MODE16_OFF  &
                      SPI_SMP_ON  &  SPI_CKE_OFF  &
                      SLAVE_ENABLE_OFF  &
                      CLK_POL_ACTIVE_HIGH  &
                      MASTER_ENABLE_ON  &
                      SEC_PRESCAL_7_1  &
                      PRI_PRESCAL_64_1;

      SPICON2Value  =  FRAME_ENABLE_OFF  &  FRAME_SYNC_OUTPUT;
      SPISTATValue  =  SPI_ENABLE  &  SPI_IDLE_CON  &
                                     SPI_RX_OVFLOW_CLR;
      OpenSPI1(SPICON1Value,SPICON2Value,SPISTATValue  );
      _SPI1IF  =  0;
```

```
      while  (*ps1  !=  0)
        {
          WriteSPI1(*ps1++);
          while  (_SPI1IF  ==  0);
          _SPI1IF  =  0;
        }
      CloseSPI1();
      while(1);
}
```

В этой программе используются библиотечные функции `CloseSPI1`, `OpenSPI1` и `WriteSPI1`. С помощью функции `OpenSPI1` конфигурируется и включается SPI-интерфейс 1, функция `CloseSPI1` отключает интерфейс, а функция `WriteSPI1` передает байт (или слово). Исходный текст программы для «ведомого» микроконтроллера выглядит следующим образом:

```
#include  <p24fj128ga010.h>
#include  <spi.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  SYSCLK          8000000
#define  BAUDRATE2       9600
#define  BAUDRATEREG2    SYSCLK/8/BAUDRATE2-1

#define  UART2_TX_TRIS   TRISFbits.TRISF5
#define  UART2_RX_TRIS   TRISFbits.TRISF4

char  datard;

void  UART2Init()
{
      UART2_TX_TRIS  =  0;
      UART2_RX_TRIS  =  1;
      U2BRG  =  BAUDRATEREG2;

      U2MODE  =  0;
      U2MODEbits.BRGH  =  1;
      U2MODEbits.UARTEN  =  1;

      U2STA  =  0;
      U2STAbits.UTXEN  =  1;
      IFS1bits.U2RXIF  =  0;
}

void   UART2PutChar(char  Ch)
{
    while(U2STAbits.UTXBF  ==  1);
    U2TXREG  =  Ch;
}

void  main(void)
{
      unsigned  int  SPICON1Value;
      unsigned  int  SPICON2Value;
```



```
      unsigned  int  SPISTATValue;

      UART2Init();

      CloseSPI1();

      SPICON1Value  =  ENABLE_SCK_PIN  &  ENABLE_SDO_PIN  &  SPI_MODE16_OFF  &
                       SPI_SMP_OFF  &  SPI_CKE_OFF  &
                       SLAVE_ENABLE_OFF  &
                       CLK_POL_ACTIVE_HIGH  &
                       MASTER_ENABLE_OFF  &
                       SEC_PRESCAL_7_1  &
                       PRI_PRESCAL_64_1;

      SPICON2Value  =  FRAME_ENABLE_OFF  &  FRAME_SYNC_OUTPUT;
      SPISTATValue  =  SPI_ENABLE  &  SPI_IDLE_CON  &
                                    SPI_RX_OVFLOW_CLR;

      OpenSPI1(SPICON1Value,SPICON2Value,SPISTATValue);
      while(1)
      {
        while  (DataRdySPI1()  ==  0);
        datard  =  ReadSPI1();
        UART2PutChar(datard);
      }
}
```

Напомню, что при использовании библиотечных функций для работы с интерфейсом SPI в исходный текст программы следует включить объявление заголовочного файла spi.h, а в проект включить один из библиотечных файлов libpPIC24Fxxx-elf.a или libpPIC24Fxxx-coff.a.

ГЛАВА 8

ИНТЕРФЕЙС I²C МИКРОКОНТРОЛЛЕРОВ PIC24F

Интерфейс I²C является одним из наиболее популярных среди разработчиков оборудования (читается как «I—square—C»). В настоящее время выпускается огромное число микросхем, использующих этот интерфейс. Микроконтроллеры семейства PIC24F имеют развитые аппаратно-программные средства для организации обмена данными с периферийными устройствами по шине I²C. Многие разработчики недолюбливают работу с этим интерфейсом из-за его кажущейся сложности. В отличие от интерфейса SPI, который кажется более простым при изучении, I²C требует более серьезных усилий, но программировать его несложно.

8.1. Принципы функционирования интерфейса I²C

Рассмотрим принципы обмена данными по протоколу I²C. Стандарт I²C реализован как двухпроводной последовательный интерфейс, разработанный компанией «Philips Corp.» для работы с максимальной скоростью передачи данных 100 Кбит/с. Впоследствии стандарт стал поддерживать более скоростные режимы работы шины (400 Кбит/с и 1 Мбит/с). При этом к одной шине I²C могут быть подключены устройства с различными скоростями доступа, если скорость передачи данных будет удовлетворять требованиям самого низкоскоростного устройства.

Протокол передачи данных по шине I²C разработан таким образом, чтобы гарантировать надежный качественный прием/передачу данных. При обмене данными одно устройство является «ведущим» и инициирует такой обмен, а также формирует сигналы синхронизации. Другое устройство, «ведомое», может начать передачу/прием данных только по команде ведущего шины.

Протокол I²C использует две сигнальные линии, по одной из которых подается сигнал синхронизации, обычно обозначаемый как SCL, а по другой, обозначаемой обычно как SDA, передаются или принимаются данные. К шине I²C можно подключать несколько устройств, при этом линии SCL и SDA являются общими (**Рис. 8.1**).

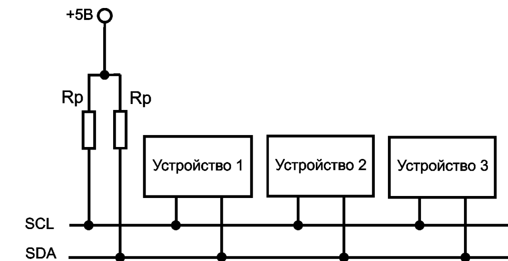

Рис. 8.1. Схема подключения устройств по интерфейсу I²C

К сигнальным линиям необходимо подключить так называемые подтягивающие резисторы, обозначенные на схеме как Rp, присоединив их к источнику питания (для стандартной TTL-логики это +5В). Подтягивающие резисторы (их значение может находиться в диапазоне 1-10 К) нужны для фиксации уровня сигналов, поскольку спецификация I²C предусматривает использование устройств, имеющих выходные каскады с открытым коллектором или стоком (open drain). Кроме того, что более существенно, поскольку линии SCL и SDA являются двунаправленными, то такая аппаратная конфигурация обеспечивает функцию «монтажного» И (AND), что позволяет передавать сигналы в обоих направлениях.

Конфигурация шины I²C позволяет работать с одним или несколькими ведущими, что дает возможность создавать разветвленные высокоскоростные сети обмена данными. В подавляющем большинстве случаев на шине находится только один ведущий и несколько ведомых устройств. «Ведущий» обеспечивает синхронизацию обмена данными, генерируя синхроимпульсы по линии SCL.

Обмен данными по шине I²C инициируется «ведущим» устройством, которое должно обеспечить формирование стартовой (в начале обмена) и стоповой (в конце обмена) последовательностей сигналов. Эти последовательности показаны на **Рис. 8.2**.

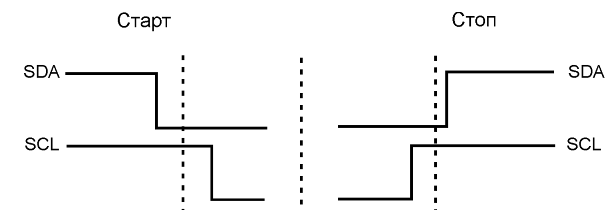

Рис. 8.2. Старт-стоповая последовательность сигналов интерфейса I²C

Протокол I²C требует, чтобы сигнал на линии данных SDA оставался неизменным при ВЫСОКОМ уровне сигнала SCL. Перепад сигнала на линии SDA из ВЫСОКОГО уровня в НИЗКИЙ при ВЫСОКОМ уровне на линии SCL

формируется «ведущим» и указывает на начало обмена данными. Если при ВЫСОКОМ уровне сигнала на линии SCL уровень сигнала на линии SDA меняется с НИЗКОГО на ВЫСОКИЙ, то эта последовательность сигналов интерпретируется как завершение операции обмена данными на шине.

Данные передаются по шине в виде последовательности из 8 бит (хотя в некоторых конфигурациях допускается передача и большего числа бит), причем первым в последовательности идет старший значащий бит (MSB). В практическом плане каждый бит удобно передавать по нарастающему фронту сигнала SCL. Напомню, что при ВЫСОКОМ уровне сигнала SCL сигнал на линии SDA должен оставаться неизменным, если только это не стартовая или стоповая последовательность. После того как все 8 бит переданы, устройство, передающее данные, ожидает от устройства, принимающего данные, подтверждения приема (acknowledge) по линии SDA. Бит ответа фиксируется по нарастающему фронту 9-го импульса SCL. В момент передачи бита подтверждения отвечающее устройство принимает управление линией SDA на себя, а затем возвращает управление инициатору обмена.

Поскольку к шине I²C можно подключать несколько устройств, то для обмена данными с конкретным устройством вначале нужно указать его адрес. В простейшем случае, например, для передачи данных в устройство вначале нужно передать адрес, а затем и сам байт данных. Эта последовательность показана на **Рис. 8.3**.

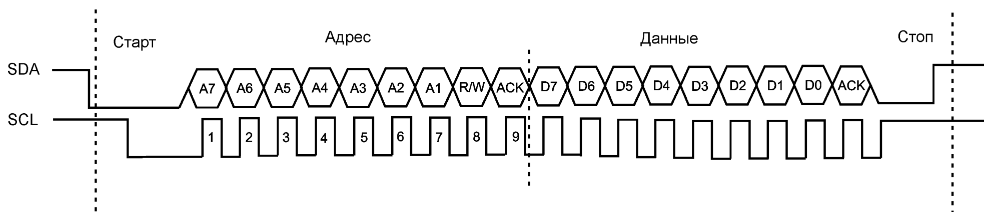

Рис. 8.3. Запись байта данных в устройство на шине I²C

Цикл записи данных в устройство начинается со стартовой последовательности. Затем передаются первые 8 бит, содержащие 7-разрядный адрес устройства (биты A7...A1) и команду чтения-записи (обозначена как $R/\overline{W}$). Обычно команда записи передается НИЗКИМ уровнем, а команда чтения — ВЫСОКИМ. Бит 9 — это ответ устройства (обозначен как ACK), который фиксируется по фронту 9-го синхроимпульса и анализируется «ведущим». Следующие 8 бит являются битами данных. Как и при передаче адреса, 9-й бит также содержит ответ устройства. Цикл записи оканчивается стоповой последовательностью.

Эта временная диаграмма нам очень пригодится, когда мы будем разрабатывать аппаратно-программные интерфейсы микроконтроллера PIC24F и расширителя ввода/вывода, работающего по I²C протоколу PCF8574A. Это устройство работает практически так, как показано на **Рис. 8.3**.

Перейдем к рассмотрению возможностей микроконтроллеров PIC24F в части работы с интерфейсом I²C.

8.2. Модуль интерфейса I²C микроконтроллеров PIC24F

Микроконтроллеры семейства PIC24F предоставляют разработчику удобный аппаратно-программный интерфейс, позволяющий относительно легко разрабатывать системы обмена данными по шине I²C. Далее показана упрощенная функциональная схема одного из модулей I²C, интегрированных на кристалле микроконтроллера (**Рис. 8.4**).

Модуль I²C-интерфейса состоит из передающей части, приемной части и логики управления. Для упрощения анализа на рисунке не показан ряд узлов

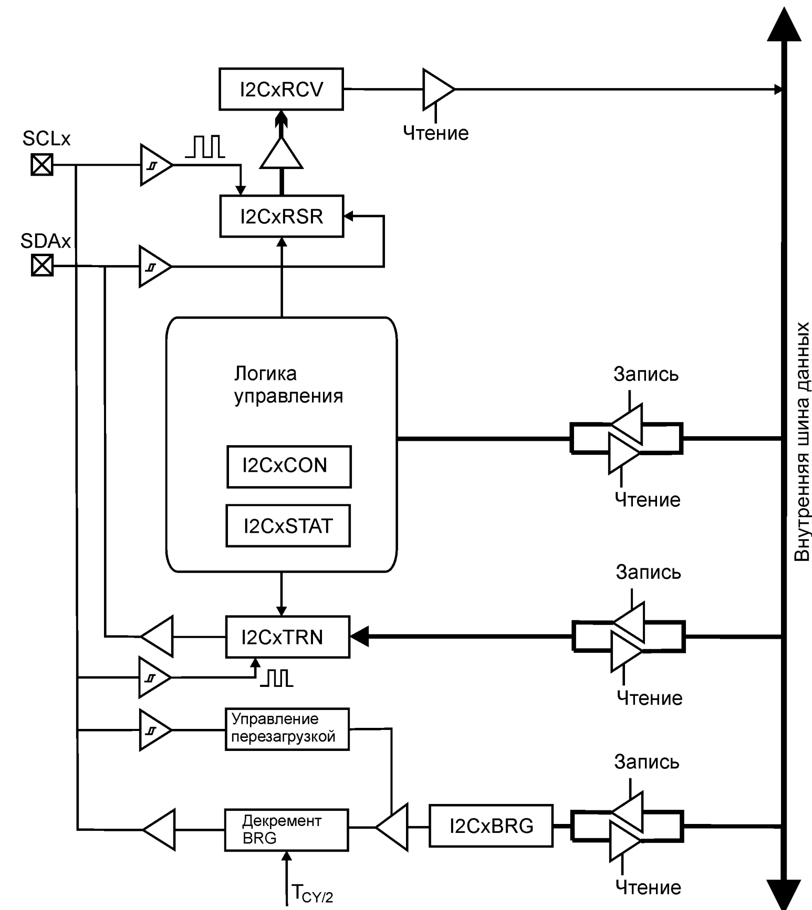

Рис. 8.4. Упрощенная функциональная схема модуля I²C

и программных регистров. Передача данных осуществляется посредством параллельно-последовательного регистра I2CxTRN (x = 1, 2, ...). Данные в регистр записываются программно через интерфейс внутренней шины, после чего побитово выдвигаются на линию SDAx синхронно с импульсами тактового сигнала на линии SCLx. Скорость передачи задается в регистре I2CxBRG и синхронизирована с тактовой частотой шины $T_{CY/2}$.

Прием данных по интерфейсу I²C осуществляется через регистр сдвига I2CxRSR, куда данные вдвигаются побитово с линии SDAx синхронно с тактовыми импульсами SCLx. Принятые данные в параллельном коде помещаются в регистр I2CxRCV, откуда их может прочитать программа. Как правило, данные по шине I²C передаются и принимаются побайтово.

Логика управления включает схемы синхронизации, а также регистры управления и состояния I2CxCON и I2CxSTAT, которые доступны программно и позволяют управлять процессом обмена по шине. Кроме указанных, модуль интерфейса содержит и другие регистры, которые мы при первоначальном знакомстве с функционированием шины I²C рассматривать не будем. Заинтересованным в более глубоком изучении материала читателям можно обратиться к фирменным руководствам фирмы Microchip для конкретных моделей микроконтроллеров.

Далее в таблицах приводятся назначения отдельных битов программно-доступных регистров модуля интерфейса I²C. В **Табл. 8.1** приводится описание битов регистра управления I2CxCON.

Таблица 8.1. Назначение битов регистра управления I2CxCON

Позиция бита	Обозначение	Назначение
15	I2CEN	Разрешение работы интерфейса: 1 – разрешает функционирование I2Cx модуля и устанавливает выводы SDAx и SCLx в режим работы с интерфейсом I²C; 0 – отключает модуль I2Cx. Все выводы интерфейса I²C могут использоваться в качестве дискретных портов ввода/вывода
14		Читается как 0
13	I2CSIDL	Разрешение функционирования I²C-интерфейса в режиме «холостого хода» микроконтроллера: 1 – все операции на интерфейсе в режиме «холостого хода» прекращаются; 0 – операции продолжаются при работе процессора в режиме «холостого хода»
12	SCLREL	Определяет, нужно ли передать управление тактовой частотой на выводе SCLx при функционировании в режиме «ведомого»: 1 – управление передается; 0 – линия SCLx удерживается в состоянии низкого уровня
11	IPMIEN	Разрешение режима расширенного управления: 1 – режим расширенного управления разрешен. В этом режиме «ведущий» получает подтверждения о достоверности каждого адреса, по которому идет обращение; 0 – режим расширенного управления запрещен

Таблица 8.1. Назначение битов регистра управления I2CxCON (окончание)

Позиция бита	Обозначение	Назначение
10	A10M	Разрешение работы с 10-битным адресом «ведомого»: 1 – регистр I2CxADD будет сконфигурирован для работы с 10-битными адресами; 0 – регистр I2CxADD работает с 7-битными адресами
9	DISSLW	Разрешение управления скоростью нарастания тактовых импульсов: 1 – управление запрещено; 0 – управление разрешено
8	SMEN	Разрешение работы с уровнями напряжения шины SMBus: 1 – порты ввода/вывода переключаются уровнями напряжений SMBus; 0 – работа с такими уровнями сигналов запрещена
7–6		Специальное назначение, здесь не рассматриваются
5	ACKDT	Подтверждение приема данных (только при чтении данных в режиме «ведущего»): 1 – послать сигнал NACK; 0 – послать сигнал ACK
4	ACKEN	Разрешение подтверждения (только при чтении данных в режиме «ведущего»): 1 – посылать подтверждение; 0 – не посылать подтверждение
3	RCEN	Разрешение чтения (только в режиме «ведущего»): 1 – разрешить работу в режиме чтения; 0 – чтение не разрешено
2	PEN	Разрешение стоповой последовательности (только в режиме «ведущего»): 1 – инициализировать стоповую последовательность на линиях SDAx и SCLx; 0 – не разрешать стоповую последовательность
1	RSEN	Разрешение повторного старта (только в режиме «ведущего»): 1 – инициировать повторный старт на линиях SDAx и SCLx; 0 – не инициировать повторный старт
0	SEN	Разрешение стартовой последовательности (только в режиме «ведущего»): 1 – инициировать стартовую последовательность на линиях SDAx и SCLx; 0 – не инициировать стартовую последовательность

В **Табл. 8.2** описано назначение битов регистра состояния I2CxSTAT.

Таблица 8.2. Назначение битов регистра состояния I2CxSTAT

Позиция бита	Обозначение	Назначение
15	ACKSTAT	Подтверждение ответа (операции обмена данными в режимах «ведущий» и «ведомый»): 1 – получен сигнал NACK от «ведомого»; 0 – получен сигнал ACK от «ведомого»

Таблица 8.2. Назначение битов регистра состояния I2CxSTAT (окончание)

Позиция бита	Обозначение	Назначение
14	TRSTAT	Состояние операции передачи данных (только при передаче данных в режиме «ведущего»): 1 – выполняется передача (8 бит + ACK); 0 – передача не выполняется
13–11		Читается как 0
10–9		Специальное назначение, здесь не рассматривается
8	ADD10	Подтверждение 10-битного адреса: 1 – адрес подтвержден; 0 – адрес не подтвержден
7	IWCOL	Возникновение конфликта записи: 1 – попытка записи в регистр I2CxTRN была неудачной, поскольку шина I²C занята; 0 – попытка успешна
6	I2COV	Бит переполнения: 1 – байт был получен, когда в регистре I2CxRCV находятся предыдущие данные; 0 – переполнение отсутствует
5	D/A	Индикация типа данных (только в режиме «ведомого»): 1 – указывает, что последнее полученное значение является байтом данных; 0 – указывает, что последнее полученное значение является адресом устройства
4	P	Стоповый бит: 1 – стоповый бит является последним обнаруженным битом; 0 – стоповый бит не является последним
3	S	Стартовый бит: 1 – стартовый бит является последним обнаруженным битом; 0 – стартовый бит не является последним
2	R/W	Направление передачи данных (только в режиме «ведомого»): 1 – передача данных идет от «ведомого»; 0 – передача данных идет «ведомому»
1	RBF	Состояние буфера приема: 1 – прием завершен, в буфере I2CxRCV находятся данные; 0 – прием не завершен, буфер I2CxRCV пуст
0	TBF	Состояние буфера передачи: 1 – передача продолжается, в буфере I2CxTRN имеются данные; 0 – передача завершена, буфер I2CxTRN пуст

При работе в режиме «ведущего» модуль интерфейса I²C должен обеспечить генерацию тактового сигнала для шины. Как правило, шина I²C работает на одной из трех тактовых частот: 100 кГц, 400 кГц или 1 МГц. Период тактовой частоты определяется как сумма минимального интервала времени пребывания сигнала SCLx в состоянии НИЗКОГО уровня и минимального времени пребывания сигнала в состоянии ВЫСОКОГО уровня. Для расчета тактовой частоты шины можно воспользоваться следующим выражением:

$$F_{SCL} = F_{CY}/(I2CxBRG + (F_{CY}/10000000) + 1).$$

Тогда значение, которое нужно записать в регистр установки скорости обмена I2CxBRG, будет равно:

$$I2CxBRG = (F_{CY}/F_{SCL}) - (F_{CY}/10000000) - 1.$$

В этих формулах $F_{CY} = F_{OSC}/2$. Значение, записанное в регистр I2CxBRG, не должно быть меньше 2.

Можно облегчить себе задачу, не рассчитывая значение регистра I2CxBRG, а взяв его из **Табл. 8.3**.

Таблица 8.3. Тактовые частоты шины I²C и их соответствие значениям регистра скорости обмена I2CxBRG

Требуемая тактовая частота шины I²C	FCY	Значение регистра I2CxBRG		Фактическая частота шины I²C
		Десятичное	Шестнадцатеричное	
100 кГц	16 МГц	157	9D	100 кГц
100 кГц	8 МГц	78	4E	100 кГц
100 кГц	4 МГц	39	27	99 кГц
400 кГц	16 МГц	37	25	404 кГц
400 кГц	8 МГц	18	12	404 кГц
400 кГц	4 МГц	9	9	385 кГц
400 кГц	2 МГц	4	4	385 кГц
1 МГц	16 МГц	13	D	1026 кГц
1 МГц	8 МГц	6	6	1026 кГц
1 МГц	4 МГц	3	3	909 кГц

В следующем разделе мы рассмотрим на практике методы программирования обмена данными по интерфейсу I²C.

8.3. Практическое использование интерфейса I²C

Перейдем к практическим аспектам работы с интерфейсом I²C. В нашем первом проекте мы будем использовать микросхему популярного расширителя ввода/вывода PCF8574A, который позволяет преобразовать последовательный сигнал, поступающий по 2-проводному интерфейсу I²C, в параллельный 8-битный выходной код. Аппаратная часть проекта показана на **Рис. 8.5**.

По этой схеме расширитель интерфейса PCF8574A по интерфейсу I²C подключен к микроконтроллеру PIC24FJ128GA010. Сигнальная линия SCL расширителя подключена к линии строба SCL1 первого интерфейса I²C микроконтроллера, а линия данных SDA — к линии данных SDA1. В этом проекте микроконтроллер выступает в роли «ведущего», передающего 8-битные данные «ведомому», в качестве которого выступает расширитель интерфейса PCF8574A. Данные в расширитель передаются каждые 0.5 секунды. На выходах P0...P7 параллельный код включает/выключает светодиоды, создавая эффект «бегущей строки». Резисторы R9 и R10 подключены к выходным каскадам с открытыми стоками микроконтроллера и расширителя, создавая таким образом эффект

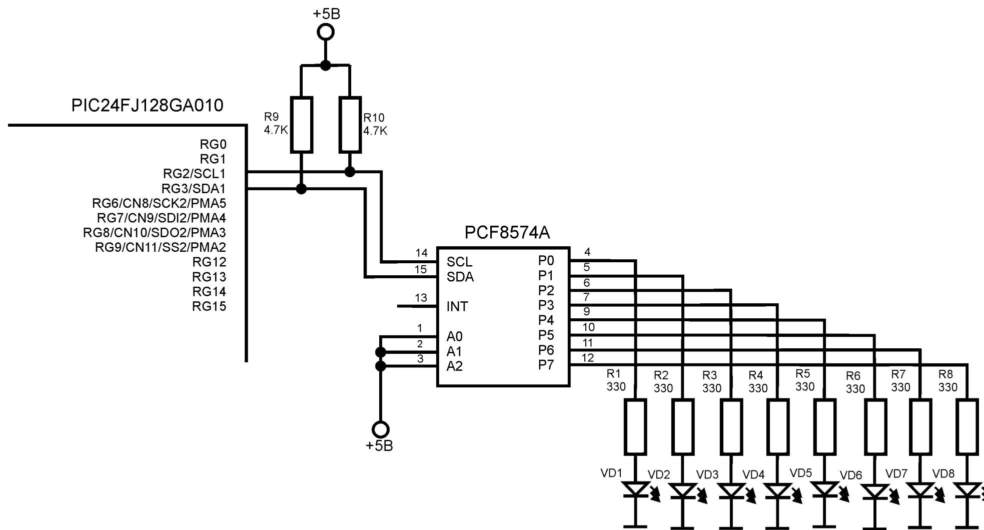

Рис. 8.5. Аппаратная часть проекта

«монтажного И». Расширитель интерфейса PCF8574F имеет три бита адреса (A0...A2), которые установлены в 1.

Полный 8-битный адрес устройства включает, кроме этих бит, оставшиеся старшие биты (они зафиксированы производителем) плюс самый младший бит, значение которого определяется типом операции. Этот бит обычно обозначается $R/\overline{W}$, он равен 0 при выполнении операции записи и единице — при выполнении операции чтения.

Комбинированный адрес I^2C-устройства, сформированный таким образом, будет разным для операций чтения и записи. Например, в нашем случае если выполняется запись в устройство, то его «комбинированный» адрес равен 0x7E. Если выполняется операция чтения, то в цикле передачи адреса на шину будет выставлено значение 0x7F.

Этот пример приведен для конкретного устройства. Для других устройств, работающих с шиной I^2C, принцип формирования адреса может отличаться, хотя, как правило, самый младший бит комбинированного адреса всегда определяется типом операции (чтение или запись).

Обратимся к программной части проекта. Она разработана в среде MPLAB IDE с помощью мастера проектов по стандартной процедуре, которая нам известна. В созданный проект добавим файл программы со следующим текстом:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  SYSCLK       8000000
#define  t1           0.5
```



```
#define  PREG    SYSCLK/2*t1/256

unsigned  char  AddrW  =  0x7e;
unsigned  char  ByteW  =  0x1;

void  __attribute__  ((interrupt))  _T1Interrupt(void)
{
  _T1IF  =  0;
  if  (ByteW  ==  0x0)
        ByteW  =  0x1;

  I2C1CONbits.I2CEN  =  1;
  I2C1BRG  =  0x4E;
  I2C1CONbits.SEN  =  1;

  while  (I2C1CONbits.SEN  !=  0);
  I2C1TRN  =  AddrW;

  while  (I2C1STATbits.TRSTAT  ==  1);

  I2C1TRN  =  ByteW;
  while  (I2C1STATbits.TRSTAT  ==  1);

  I2C1CONbits.PEN  =  1;
  I2C1CONbits.I2CEN  =  0;
  ByteW  =  ByteW  <<  1;
}

void  main(void)
{

  _T1IF  =  0;
  _T1IP  =  4;
  _T1IE  =  1;

  PR1  =  PREG;
  TMR1  =  0;
  T1CON  =  0x8030;

  while(1);
}
```

В этой программе для передачи байта данных от микроконтроллера к расширителю по интерфейсу I^2C используется функция-обработчик `_T1Interrupt` прерывания Таймера 1, которое вызывается каждые 0.5 с. Рассмотрим более детально программный код обработчика, касающийся передачи байта данных по интерфейсу I^2C.

Напомню, для обмена данными с устройством, работающим с протоколом I^2C, необходимо сначала передать адрес устройства, а затем данные. В нашей программе адрес расширителя PCF8574A хранится в переменной `AddrW` (0x7E), а байт данных для передачи — в переменной `ByteW` (его начальное значение равно 0x1). После очередного цикла передачи содержимое байта `ByteW` будет сдвигаться влево (оператор `ByteW = ByteW << 1`) до тех пор, пока не станет равным 0. После этого переменной `ByteW` будет вновь присвоено значение 0x1 (оператор `if` обработчика) и т. д.

Полный цикл передачи данных устройству начинается с оператора

```
I2C1CONbits.I2CEN = 1;
```

Установленный бит I2CEN разрешает работу первого модуля интерфейса I^2C. При выполнении этой команды на обеих линиях шины I^2C устанавливается ВЫСОКИЙ уровень, что свидетельствует о готовности шины к обмену данными. Следующий оператор

```
I2C1BRG = 0x4E;
```

устанавливает частоту синхронизации обмена данными по I^2C, равную 100 кГц, для тактовой частоты микроконтроллера, равной 8 МГц. Это значение можно рассчитать вручную, либо воспользоваться данными из **Табл. 8.3**. Новое значение загружается в тактовый генератор модуля интерфейса I^2C, как только начинается процесс обмена данными.

Следующая команда инициирует стартовую последовательность:

```
I2C1CONbits.SEN = 1;
```

Если установлен бит SEN регистра I2C1CON, то модуль интерфейса I^2C формирует последовательность сигналов SCL и SDA, соответствующую стартовой последовательности. Этот бит сбрасывается аппаратно по завершении старта, поэтому программа может отследить завершение этого этапа обмена с помощью оператора

```
while (I2C1CONbits.SEN != 0);
```

После этого можно передавать адрес устройства, записав в регистр передачи содержимое переменной `AddrW`:

```
I2C1TRN = AddrW;
```

Для определения завершения передачи служит бит TRSTAT регистра состояния I2C1STAT, который устанавливается в 1 в начале обмена данными и сбрасывается в 0 по окончании этой процедуры. Для определения конца передачи байта состояние бита TRSTAT проверяется в цикле `while`:

```
while (I2C1STATbits.TRSTAT == 1);
```

Важное замечание: бит TRSTAT указывает, что обмен данными завершен и получен сигнал подтверждения ACK от «ведомого». Это означает, что никаких дополнительных проверок делать больше не нужно.

Следующий шаг — передача собственно байта данных с проверкой завершения — выполняется двумя операторами:

```
I2C1TRN = ByteW;
while (I2C1STATbits.TRSTAT == 1);
```

По завершении процесса обмена следует сформировать стоповую последовательность сигналов на шине I^2C, что выполняется установкой бита PEN регистра управления I2C1CON:

```
I2C1CONbits.PEN = 1;
```


Затем можно отключить устройство от шины I^2C до следующего цикла с помощью оператора:

```
I2C1CONbits.I2CEN = 0;
```

Собственно, это и есть основная часть работы, выполняемая нашей программой. В основной программе нужно только инициализировать и запустить Таймер 1 в режиме 16-битного таймера с прерыванием:

```
_T1IF = 0;
  _T1IP = 4;
  _T1IE = 1;

  PR1 = PREG;
  TMR1 = 0;
  T1CON = 0x8030;
```

Здесь в регистр периода PR1 Таймера 1 загружается значение `PREG`, необходимое для срабатывания Таймера 1 каждые 0.5 с. В регистр управления T1CON записывается значение, соответствующее коэффициенту деления 1:256, и устанавливается бит запуска Таймера 1.

Чтобы облегчить программирование модуля I^2C, для него, как и для других периферийных устройств, имеется библиотека 16-битных функций. Для ее использования в проект следует включить файл библиотеки libpPIC24Fxxx-elf.a или libpPIC24Fxxx-coff (в зависимости от того, какой объектный файл генерируется компилятором), а в исходный текст программы на языке Си — объявление заголовочного файла i2c.h. Модифицируем исходный текст предыдущей программы так, чтобы в нем использовались функции библиотеки периферийных модулей. Кроме того, для разнообразия изменим интервал срабатывания Таймера 1, установив его равным 2 с. В этом случае исходный текст программы может выглядеть так:

```
#include <p24fj128ga010.h>
#include <i2c.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define SYSCLK 8000000
#define t1     2
#define PREG   SYSCLK/2*t1/256

unsigned char  Done;

void __attribute__ ((interrupt)) _T1Interrupt(void)
{
  _T1IF = 0;
  Done = 1;
}

void main(void)
{

  unsigned int config2, config1;
```

```
  unsigned  char  AddrW  =  0x7e;
  unsigned  char  wByte  =  0x1;

  config2  =  0x4E;

_T1IF  =  0;
  _T1IP  =  4;
  _T1IE  =  1;

  PR1  =  PREG;
  TMR1  =  0;
  T1CON  =  0x8030;


  config1  =  (I2C_ON  &  I2C_IDLE_CON  &  I2C_CLK_HLD  &
                          I2C_IPMI_DIS  &  I2C_7BIT_ADD  &
                          I2C_SLW_DIS  &  I2C_SM_DIS  &
                          I2C_GCALL_DIS  &  I2C_STR_DIS  &
                          I2C_NACK  &  I2C_ACK_DIS  &  I2C_RCV_DIS  &
                          I2C_STOP_DIS  &  I2C_RESTART_DIS  &
                          I2C_START_DIS);
Done  =  0;

while  (1)
{
  if  (Done  ==  1)
    {
      Done  =  0;

      OpenI2C1(config1,config2);
      IdleI2C1();
      StartI2C1();

      while(I2C1CONbits.SEN  );

      MasterWriteI2C1(AddrW);
      while  (I2C1STATbits.TRSTAT);

      MasterWriteI2C1(wByte);
      while  (I2C1STATbits.TRSTAT);

      StopI2C1();

      while(I2C1CONbits.PEN);

      CloseI2C1();
      if  (wByte  ==  0x0)
        wByte  =  0x1;
      else
        wByte  =  wByte  <<  1;
    }
  }
}
```

Программа записывает данные на шину I²C в цикле `while (1)` основной программы при значении переменной `Done`, равном 1. Установка этой переменной выполняется в обработчике прерывания `_T1Interrupt`, который вызывается каждые 2 с.

В этой программе используется несколько библиотечных функций для работы с модулями интерфейса I²C. Функция `OpenI2C1` конфигурирует и включает первый модуль интерфейса I²C. В качестве параметров функция принимает значение, определяющее скорость обмена данными по шине (переменная `config2`), и требуемые настройки регистра управления I2C1CON (переменная `config1`). В данном случае интерфейс настроен на скорость обмена, соответствующую частоте SCL 100 кГц при тактовой частоте микроконтроллера 8 МГц. Смысл макросов переменной `config1` будет понятен, если вы посмотрите содержимое заголовочного файла i2c.h

После включения модуля интерфейса он переводит шину в свободное состояние (состояние ожидания) при помощи функции `IdleI2C1`. В этом состоянии на линиях SCL1 и SDA1 будет присутствовать ВЫСОКИЙ уровень. Из этого состояния можно начинать цикл обмена данными, что и выполняет следующая в листинге за `IdleI2C1` функция `StartI2C1`. Фактически функция `StartI2C1` устанавливает бит SEN регистра управления I2C1CON. Программа должна обязательно дождаться окончания формирования стартовой последовательности (цикл `while`), по завершении которой микроконтроллер сбрасывает бит SEN.

Затем на шину I²C нужно выставить адрес устройства, к которому происходит обращение. В данном случае 7-битный адрес устройства объединяется с битом операции записи (равен 0), и это 8-битное значение (переменная `AddrW`) выставляется на шину при помощи библиотечной функции `MasterWriteI2C1 (AddrW)`.

Завершение передачи адреса фиксируется битом TRSTAT, который сбрасывается по завершении операции. После записи адреса устройства можно передать в устройство байт данных, что и выполняется функцией `MasterWriteI2C1 (wByte)`. Момент завершения передачи байта определяется по сбросу бита TRSTAT.

Операцию обмена следует завершить функцией `StopI2C1`. О завершении обмена данными можно судить по значению флага PEN, который, в случае успешного завершения, должен быть сброшен в 0. Отключение интерфейса выполняется библиотечной функцией `CloseI2C1`.

Чтение данных с шины I²C микроконтроллер выполняет по несколько иному программному алгоритму. Это обусловлено тем, что «ведущий», в роли которого выступает PIC24, в цикле чтения данных с шины I²C должен передавать управление линией данных SDA «ведомому».

Чтение данных по интерфейсу I²C я проиллюстрирую в проекте, схема которого показана на **Рис. 8.6**.

Здесь 8-битный параллельный код на входах P0...P7 микросхемы PCF8574A, который формируется при различных комбинациях нажатых/отжатых кнопок, считывается по интерфейсу I²C в микроконтроллер. Линия SCL1 микроконтроллера соединена с линией SCL расширителя ввода/вывода, а линия SDA — с линией SDA расширителя PCF8574A. Двоичный код на входах расширителя каждые 2 секунды передается в микроконтроллер, а затем — на выход порта A, к которому подсоединены светодиоды. Для реализации периодического

Рис. 8.6. Аппаратная часть проекта

опроса входов используется Таймер 1 микроконтроллера. Адрес расширителя интерфейса PCF8574A изменился и теперь равен 0x7F (последний бит «комбинированного» адреса стал равен 1, поскольку выполняется чтение).

Теперь о программной части. Программный проект реализован в среде MPLAB IDE и содержит программу на Си, исходный текст которой показан далее:

```
#include  <p24fj128ga010.h>
#include  <i2c.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  SYSCLK  8000000
#define  t1      2
#define  PREG    SYSCLK/2*t1/256

unsigned  char  AddrR  =  0x7f;
unsigned  char  ByteR  =  0;

void  __attribute__  ((interrupt))  _T1Interrupt(void)
{
  _T1IF  =  0;

  I2C1BRG  =  0x4E;
  I2C1CONbits.I2CEN  =  1;

  IdleI2C1();
  I2C1CONbits.SEN  =  1;
  while  (I2C1CONbits.SEN  !=  0);

I2C1TRN  =  AddrR;
  while  (I2C1STATbits.TRSTAT  ==  1);

    I2C1CONbits.RCEN  =  1;

    while  (I2C1CONbits.RCEN);
    ByteR  =  I2C1RCV;

    StopI2C1();
```



```
    while(I2C1CONbits.PEN);
    CloseI2C1();
    PORTA  =  ByteR;
  }

void  main(void)
{
  TRISA  =  0x0;

  _T1IF  =  0;
  _T1IP  =  4;
  _T1IE  =  1;

  TMR1  =  0;
  PR1  =  PREG;
  T1CON  =  0x8030;

  while(1);
}
```

В этой программе используются некоторые функции 16-битной библиотеки для периферийных устройств, поэтому в проект нужно добавить один из библиотечных файлов libpPIC24Fxxx-elf.a или libpPIC24Fxxx-coff (в зависимости от того, какой объектный файл генерируется компилятором), а в текст программы включить объявление заголовочного файла i2c.h.

Чтение байта данных выполняется в обработчике прерывания Таймера 1 `_T1Interrupt`. После сброса флага прерывания `_T1IF` включается модуль интерфейса I²C, задается тактовая частота шины, которая выбрана равной 100 кГц при тактовой частоте микроконтроллера 8 МГц, а сама шина переводится в состояние ожидания:

```
I2C1BRG  =  0x4E;
I2C1CONbits.I2CEN  =  1;
IdleI2C1();
```

Цикл чтения данных с шины I²C начинаем с формирования стартовой последовательности:

```
I2C1CONbits.SEN  =  1;
while  (I2C1CONbits.SEN  !=  0);
```

Теперь нужно выставить на шину адрес устройства, к которому происходит обращение (с соответствующим значением бита чтения/записи). Для этого в регистр передачи записывается значение переменной `AddrR`, что автоматически инициирует передачу байта адреса по шине. Чтобы обнаружить конец передачи, нужно дождаться сброса бита TRSTAT. Нулевое значение этого бита свидетельствует о завершении передачи байта адреса в устройство. Эти действия реализованы в двух операторах:

```
I2C1TRN  =  AddrR;
while  (I2C1STATbits.TRSTAT  ==  1);
```

Далее нужно прочитать байт данных из устройства в микроконтроллер. Для этого следует передать управление линией данных SDA расширителю ввода/вывода, установив бит RCEN:

```
I2C1CONbits.RCEN  =  1;
while  (I2C1CONbits.RCEN);
```

Бит RCEN будет оставаться в состоянии 1 до тех пор, пока не закончится прием данных. Как только байт данных окажется в регистре буфера I2C1RCV, бит RCEN аппаратно сбросится в 0, а содержимое регистра I2C1RCV можно будет считать в переменную ByteR:

```
ByteR  =  I2C1RCV;
```

На этом цикл чтения данных по шине I²C можно закончить:

```
StopI2C1();
while(I2C1CONbits.PEN);
CloseI2C1();
```

Наконец, данные из переменной ByteR записываются в порт А

```
PORTA  =  ByteR;
```

Это все, что касается функции обработчика. В основной программе выполняется инициализация Таймера 1 и настройка его прерывания.

Рассмотрим последний проект этой главы, в котором будет показано взаимодействие трех устройств на шине I²C. В этом проекте будут использованы два расширителя ввода/вывода PCF8574A, работающие на одной общей шине I²C в роли «ведомых», и микроконтроллер, выполняющий роль «ведущего» шины. Данные с одного расширителя с базовым адресом 0x7f будут читаться в микроконтроллер, который, в свою очередь, будет отправлять их другому «ведомому» с базовым адресом 0x7c. Таким образом, данные в параллельном коде со входа микросхемы D2 будут переключать светодиоды, подключенные к выходам микросхемы D1. Таймер 1 обеспечивает обмен данными каждые 2 секунды. Схема аппаратной части проекта представлена на **Рис. 8.7**.

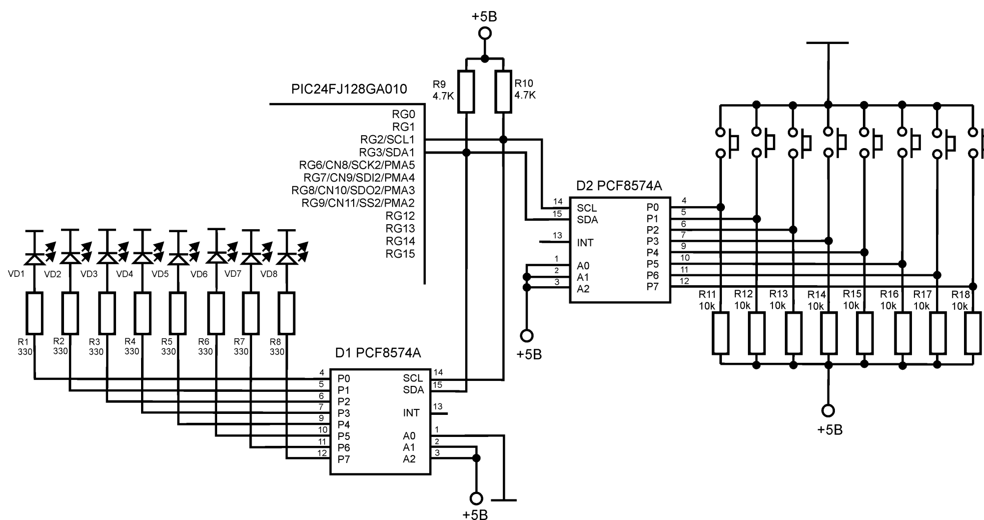

Рис. 8.7. Совместное использование шины I²C несколькими устройствами

Обратите внимание на подключение выводов адреса A0...A2 микросхем D1 и D2 — оно должно соответствовать указанному в программе с учетом бита чтения/записи, который передается после базового 7-битного адреса каждого устройства. Для данной схемы тактовая частота микроконтроллера PIC24FJ128GA010 выбрана равной 8 МГц, а частота синхронизации шины I²C — 100 кГц.

Исходный текст программы на языке Си для данного проекта представлен далее:

```
#include  <p24fj128ga010.h>
#include  <i2c.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  SYSCLK  8000000
#define  t1      2
#define  PREG    SYSCLK/2*t1/256
unsigned  char  AddrR  =  0x7f;
unsigned  char  AddrW  =  0x7c;
unsigned  char  dat  =  0;

char  ReadByteI2C(void)
{
    char  ByteRead;

    I2C1BRG  =  0x4E;
    I2C1CONbits.I2CEN  =  1;

    IdleI2C1();
    I2C1CONbits.SEN  =  1;

    while  (I2C1CONbits.SEN  !=  0);

    I2C1TRN  =  AddrR;
    while  (I2C1STATbits.TRSTAT  ==  1);

    I2C1CONbits.RCEN  =  1;
    while  (I2C1CONbits.RCEN);

    ByteRead  =  I2C1RCV;
    StopI2C1();
    while(I2C1CONbits.PEN);
    CloseI2C1();
    return  ByteRead;
}

void  WriteByteI2C(char  bRead)
{
    I2C1BRG  =  0x4E;
    I2C1CONbits.I2CEN  =  1;

    IdleI2C1();
    I2C1CONbits.SEN  =  1;

    while  (I2C1CONbits.SEN  !=  0);
  I2C1TRN  =  AddrW;
```

```
  while (I2C1STATbits.TRSTAT == 1);

  I2C1TRN = bRead;
  while (I2C1STATbits.TRSTAT == 1);

  StopI2C1();
  while(I2C1CONbits.PEN);
  CloseI2C1();
}

void __attribute__ ((interrupt)) _T1Interrupt(void)
{
  _T1IF = 0;
  dat = ReadByteI2C();
  WriteByteI2C(dat);
}

void main(void)
{
  _T1IF = 0;
  _T1IP = 4;
  _T1IE = 1;

  TMR1 = 0;
  PR1 = PREG;
  T1CON = 0x8030;

  while(1);
}
```

В программе используются 16-битные библиотечные функции, поэтому в проект следует включить файл библиотек libpPIC24Fxxx-elf.a или libpPIC24Fxxx-coff (в зависимости от того, какой объектный файл генерируется компилятором), а в текст программы — объявление заголовочного файла i2c.h.

В этой программе чтение байта из одного устройства выполняется функцией `ReadByteI2C`, а запись байта в другое устройство производит функция `WriteByteI2C`. Обе функции вызываются в обработчике прерывания Таймера 1 `_T1Interrupt`.

Мы уже знакомы с теми действиями, которые выполняются при записи/чтении данных на шине I²C, тем более что почти все операторы этой программы мы уже анализировали в предыдущих примерах. Думаю, читатели сумеют самостоятельно проанализировать исходный текст программы.

Несколько заключительных слов о программировании интерфейса I²C. Приведенные примеры являются относительно простыми — в одном цикле обмена данными выполняется только одна операция записи или чтения данных. Для более сложных устройств, например микросхем памяти, в одном цикле могут выполняться как чтение, так и запись данных. В таких операциях линия данных SDA будет управляться поочередно «ведущим» и «ведомым», поэтому может понадобиться так называемая процедура «рестарта» шины. В любом случае, перед использованием I²C-совместимого устройства следует тщательно изучить техническую документацию производителя.

ГЛАВА 9

ПРОГРАММИРОВАНИЕ ИНТЕРФЕЙСА PMP

В микроконтроллерах семейства PIC24F предусмотрен обмен данными с внешними устройствами по параллельному интерфейсу, называемому PMP (Parallel Master Interface). Этот интерфейс был специально разработан для взаимодействия с многочисленными периферийными устройствами, которые передают данные по параллельному интерфейсу (принтеры, сканеры, жидкокристаллические индикаторы, устройства памяти и др.).

Модуль параллельного интерфейса PMP обладает следующими характеристиками:

- обмен данными выполняется по 8-битной шине;
- можно адресовать до 16 линий адреса;
- можно использовать две линии выбора устройства;
- можно использовать отдельные сигналы чтения/записи либо общий сигнал чтения/записи совместно с сигналом разрешения;
- автоинкремент/автодекремент адреса;
- мультиплексирование линий адреса/данных;
- программирование задержки сигналов.

В общем виде функциональную схему интерфейса PMP можно представить так, как показано на **Рис. 9.1**.

Эта схема довольно наглядно демонстрирует возможности параллельного интерфейса PMP. Рассмотрим эти возможности более подробно. Начнем с режимов адресации. Адресацию внешних устройств можно осуществлять несколькими способами, в зависимости от конкретной реализации интерфейса. Если необходимо выполнять обмен данными с микросхемами памяти, то, как правило, нужно использовать отдельные линии адреса, данных и управления. Кроме линий адреса, в такой конфигурации могут быть задействованы и сигналы выбора устройства. Таким образом, адресация устройств может осуществляться либо с использованием только линий адреса, либо в комбинации с сигналами выбора устройства. Второй вариант часто применяется при ограниченных возможностях адресации, при этом линии сигналов выбора устройства используются как дополнительные адресные линии.

Адресация устройств может осуществляться как с применением мультиплексирования, так и без него. Например, для передачи младших битов 16-битного адреса можно использовать линии данных PMD7...PMD0 (см. **Рис. 9.1**). При

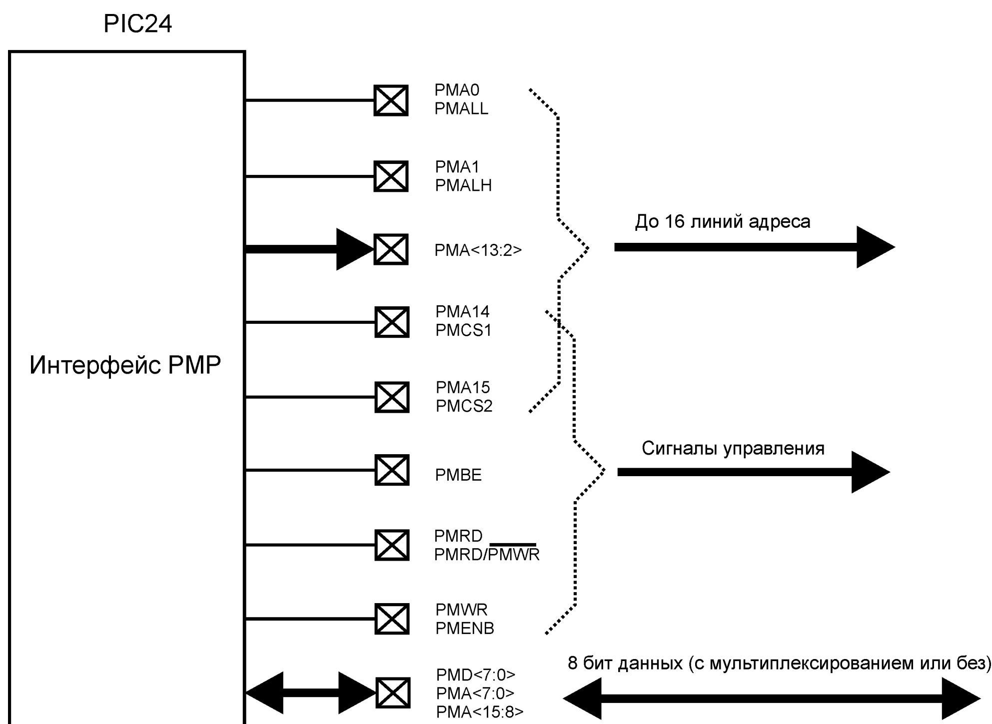

Рис. 9.1. Упрощенная функциональная схема интерфейса PMP

обмене данными с внешними устройствами во многих случаях можно обойтись и без использования линий адреса. Адресация устройств в этом случае может осуществляться с применением сигналов выбора микросхемы PMCS1 и PMCS2, что намного упрощает проектирование несложных систем.

Сигналы управления интерфейса PMP позволяют разрабатывать схемы управления практически для любых внешних устройств с параллельным интерфейсом. Для управления чтением/записью данных можно выбрать один из двух вариантов. В первом варианте для стробов чтения/записи можно использовать отдельные сигнальные линии – PMRD и PMWR соответственно. Такой вариант удобен для обмена данными с устройствами, имеющими отдельные линии чтения/записи.

Второй вариант — использовать совмещенную линию чтения/записи и отдельную линию разрешения. В этом случае чтение данных может выполняться, например, при ВЫСОКОМ, а запись — при НИЗКОМ уровне сигнала на такой совмещенной линии. Соответственно, операции чтения/записи в этом случае будут осуществляться при появлении импульса на линии разрешения.

Интерфейс PMP можно настроить для работы в одном из базовых режимов: в режиме «ведущего» или в режиме «ведомого». Фундаментальное отличие этих режимов состоит в следующем: в режиме «ведущего» микроконтроллер управ-

ляет сигналами интерфейса PMP, обеспечивая необходимую синхронизацию передачи и приема данных. При этом второе устройство работает в режиме «ведомого», обеспечивая прием и передачу данных по сигналам «ведущего». Устройство, выполняющее функцию «ведомого», не генерирует сигналы управления, хотя может выдавать на шину управления сигналы готовности, которые позволяют «ведущему» более эффективно выполнять функции управления.

При обмене данными по интерфейсу PMP в любой момент времени только одно устройство может быть «ведущим», второе будет обязательно «ведомым». Часто применяются конфигурации, когда в процессе функционирования одно и то же устройство может становиться как «ведущим», так и «ведомым». Рассмотрим режимы работы параллельного интерфейса PMP и их настройки.

9.1. Режимы работы PMP

Параллельный интерфейс PMP настраивается и контролируется с помощью группы программно-доступных регистров (всего 8). Вот их назначение:

- регистр PMCON используется для установки основных настроек интерфейса PMP;
- регистр PMMODE используется для установок режимов работы интерфейса («ведущий», «ведомый»), а также для анализа состояния операций обмена данными (бит BUSY);
- регистр PMADDR/PMDOUT1 выполняет, в зависимости от выбранного режима работы («ведущий»/«ведомый»), одну из двух функций. В режиме «ведущий» он работает как регистр адреса PMADDR, в который записывается адрес устройства, к которому происходит обращение. В режиме «ведомый» регистр работает как PDMOUT1 и содержит выходные данные;
- регистр PMDOUT2 используется в режиме «ведомый» для хранения выходных данных;
- регистр PMDIN1 используется в режимах «ведущий» и «ведомый» в качестве буфера входных данных. В режиме «ведомого» в регистре запоминаются данные, переданные «ведущим». В режиме «ведущего» регистр может хранить как входные, так и выходные данные;
- регистр PMDIN2 используется точно так же, как и регистр PMDIN1, но работает только в режиме «ведомого» для хранения входных данных;
- регистр PMAEN используется для управления выводами адреса и выбора устройства модуля PMP. Установка соответствующих битов подключает эти выводы к модулю PMP, а сброс битов разрешает их использование в составе портов ввода/вывода;
- регистр PMSTAT хранит информацию о состоянии выполнения операции интерфейса, работающего в режиме «ведомого».

Остановимся более подробно на двух основных режимах функционирования интерфейса: режиме «ведущего» и режиме «ведомого». Функционирование микроконтроллера в режиме «ведущего» схематически показано на **Рис. 9.2**.

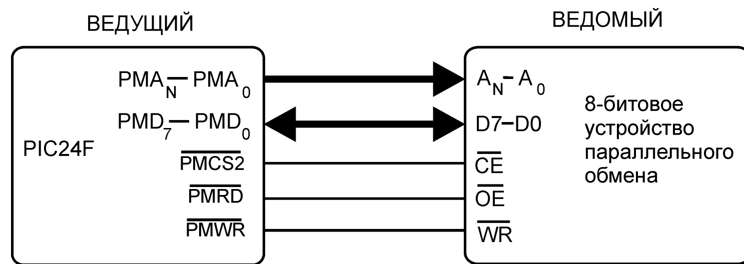

Рис. 9.2. Упрощенная схема взаимодействия «ведущего» и «ведомого»

В режиме «ведущего» микроконтроллер формирует сигналы управления (PMCS2, PMRD или PMWR), устанавливая направление обмена, формат обмена (8 или 16 бит) и адрес «ведомого» устройства на шине адреса PMA. В конкретных вариантах использования интерфейса PMP комбинации сигналов могут варьироваться, например использоваться одна совмещенная линия чтения/записи вместо двух раздельных, применяться мультиплексирование линий адреса и данных и т.д.

Для установки режима «ведущего» нужно выполнить как минимум два действия:

- установить соответствующим образом биты 9:8 регистра режима PMMODE (<11> или <10>);
- разрешить функционирование интерфейса PMP, установив бит 15 (PMPEN) регистра управления PMCON.

Остальные настройки интерфейса в этом режиме зависят от конкретной конфигурации: полярности активных уровней управляющих сигналов (ВЫСОКИЙ/НИЗКИЙ), мультиплексирования адресных линий и т. д. Рассмотрим, как выполняются операции записи и чтения байта «ведущим» устройством.

Чтобы выполнить запись байта в параллельный порт PMP в режиме «ведущего», программа должна записать байт в регистр PMDIN1. Все дальнейшие операции модуль интерфейса выполнит автоматически в следующей последовательности:

1. Модуль интерфейса PMP установит соответствующие значения на линиях адреса и линиях выбора устройства.
2. Байт данных из регистра PMDIN1 будет выставлен на шину данных PMD7...PMD0.
3. Сигнал стробирования PMWR устанавливается в активное состояние, стробируя запись байта. В этой фазе принимающее устройство («ведомый») может считать байт данных с линий PMD7...PMD0. Это удобно сделать по перепаду сигнала PMWR из активного состояния в неактивное, особенно если на входе данных «ведомого» установлены регистры-защелки.

Чтение байта «ведущим» выполняется по схожей схеме, только направление передачи данных меняется на противоположное, а вместо сигнала PMWR используется PMRD. Для приема байта по интерфейсу PMP программа читает

данные из регистра PMDIN1. В этом случае модуль интерфейса PMP автоматически выполнит такую последовательность операций:

1. Модуль интерфейса PMP установит соответствующие значения на линиях адреса и линиях выбора устройства.
2. Сигнал стробирования PMRD устанавливается в активное состояние, стробируя чтение байта с линий данных PMD7...PMD0 в регистр PMDIN1.

Для синхронизации обмена данными между «ведущим» и «ведомым» можно проверять состояние бита BUSY (бит 15 регистра режима PMMODE), который доступен для чтения в режиме «ведущего». Для «ведомого» устройства признаком окончания цикла записи/чтения может служить бит 15 (IBF) или бит 7 (OBE) регистра состояния PMSTAT. Кроме того, можно проверить состояние флага прерывания, который устанавливается по завершении операции.

Интерфейс PMP можно настроить и для работы в режиме «ведомого». Упрощенную схему взаимодействия двух микроконтроллеров, из которых один «ведущий», а другой «ведомый», можно представить схемой на **Рис. 9.3**.

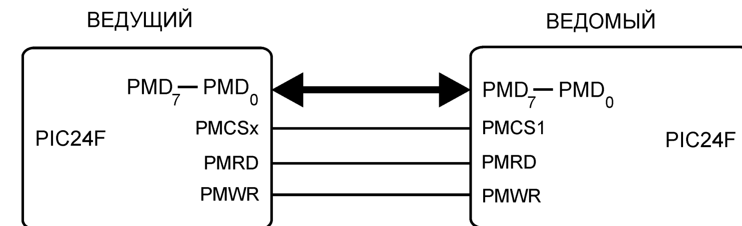

Рис. 9.3. Микроконтроллеры PIC24F в роли «ведущего» и «ведомого»

В режиме «ведомого» управляющие сигналы (PMCSx, PMRD, PMWR) поступают от внешнего микроконтроллера. Кроме того, в режиме «ведомого» можно использовать только сигнал выбора устройства PMCS1.

Для установки режима «ведомого» нужно выполнить как минимум два действия:

- установить биты 9:8 регистра режима PMMODE в <00>;
- разрешить функционирование интерфейса PMP, установив бит 15 (PMPEN) регистра управления PMCON.

Как и в режиме «ведущего», полярности активных уровней управляющих сигналов (ВЫСОКИЙ/НИЗКИЙ), мультиплексирование адресных линий и другие настройки задаются соответствующим образом в регистрах PMCON и PMMODE.

Для записи байта данных в «ведомое» устройство необходимо, чтобы сигнал выбора устройства PMCS1 и сигнал записи PMWR находились в активном состоянии. В этом случае данные, находящиеся на линиях PMD7...PMD0 «ведомого», будут сохранены в младших восьми битах регистра PMDIN1. Об успешном окончании операции записи могут свидетельствовать установленные биты PMPIF (флаг прерывания) и IBF.

Для чтения байта данных из «ведомого» устройства необходимо, чтобы сигнал выбора устройства PMCS1 и сигнал чтения PMRD находились в активном состоянии. В этом случае младшие 8 бит регистра PMDOUT1 будут установлены на линиях PMD7...PMD0. Об успешном завершении чтения байта может свидетельствовать установленный флаг OBE регистра состояния PMSTAT.

Далее мы рассмотрим несколько практических примеров программирования параллельного интерфейса в различных режимах работы.

9.2. Практические примеры программирования интерфейса PMP

В этом разделе мы рассмотрим некоторые примеры программирования обмена данными с использованием интерфейса PMP. Интерфейс можно настроить для работы в различных конфигурациях систем обмена данными. В последующих примерах мы рассмотрим некоторые из них.

Первый проект, который мы разработаем, продемонстрирует работу интерфейса PMP микроконтроллера PIC24F в режиме «ведомого». В этом режиме сигналы синхронизации формируются другим внешним устройством. Наш пример является в достаточной степени искусственным, поскольку сигналы синхронизации будут генерироваться самим микроконтроллером на отдельных линиях цифровых портов ввода/вывода.

В стандартном варианте модуль интерфейса конфигурируется как параллельный интерфейс обмена данными с соответствующими установками портов. При этом бит PMPEN регистра управления PMCON должен быть установлен в 1, а биты 9 и 8 регистра режима PMMODE должны быть сброшены. В этом режиме внешнее устройство, например микроконтроллер или микропроцессор, сможет выполнять асинхронную запись/чтение данных на шине PMD7:PMD0 с использованием сигналов синхронизации PMRD (чтение), PMWR (запись) и PMCSx (выбор устройства).

Для демонстрации возможностей интерфейса PMP микроконтроллеров PIC24F разработаем несколько аппаратно-программных проектов. В первом проекте будет показан вывод строки данных с микроконтроллера PIC24FJ128GA010 через интерфейс PMP на жидкокристаллический дисплей (LCD). Схема аппаратной части проекта представлена на **Рис. 9.4**.

По этой схеме линии данных микроконтроллера PMD0...PMD7 подсоединяются к соответствующим линиям дисплея. Управляющие сигналы на жидкокристаллический дисплей подаются с выводов RA1:RA0 порта A, а сигналы синхронизации интерфейса PMP — с выводов RB0:RB1 порта B микроконтроллера. В данной схеме данные «читаются» дисплеем, поэтому интерфейс PMP работает в режиме «ведомого» с чтением данных. Вывод RB0 управляет сигналом PMCS1, а вывод RB1 — сигналом PMRD. Тактовая частота микроконтроллера выбрана равной 8 МГц.

Программная часть проекта разработана в среде MPLAB IDE и включает программу со следующим исходным текстом:

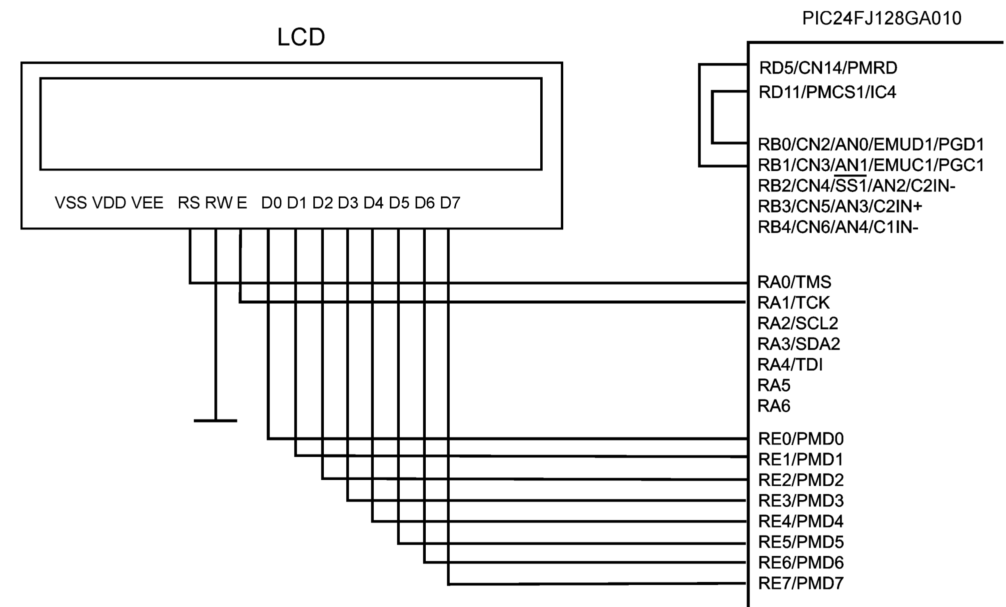

Рис. 9.4. Аппаратная часть проекта

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  EN      _RA1
#define  RS      _RA0
#define  PMCS1   _RB0
#define  PMRD    _RB1

char  status;

void  Delay(void)
{
  _T1IF  =  0x0;
  TMR1  =  0;
  PR1  =  0xffff;
  T1CON  =  0x8000;
  while  (_T1IF  ==  0);
  T1CON  =  0x0;
}

void  DataSet(char  c1)
{
    PMDOUT1  =  c1;
    PMCS1  =  1;
    PMRD  =  1;
    PMRD  =  0;
    PMCS1  =  0;
```

```
}

void  ClearLCD(void)
{
    EN  =  1;
    RS  =  0;
    DataSet(0x1);
    EN  =  0;
    Delay();
}

void  InitLCD(void)
{
    EN  =  1;
    RS  =  0;
    DataSet(0x38);
    EN  =  0;
    Delay();

    EN  =  1;
    RS  =  0;
    DataSet(0x0E);
    EN  =  0;
    Delay();

    EN  =  1;
    RS  =  0;
    DataSet(0x6);
    EN  =  0;
    Delay();
}

void  WriteLCD(char  c1)
{
    EN  =  1;
    RS  =  1;
    DataSet(c1);
    EN  =  0;
    Delay();
}

void  main()
{
    char  *str  =  "Test  of  PMP  interfacing";
    TRISA  =  0x0;
    AD1PCFG  =  0xffff;
    TRISB  =  0x0;

    PMCONbits.PMPEN  =  0x1;
    PMMODE  =  0x0;
    _PMPIF  =  0;

    Delay();

    InitLCD();
```



```
    ClearLCD();
    while  (*str  !=  0)
    {
        WriteLCD(*str++);
        while  (_PMPIF  ==  0);
    }
    while  (1)
        ;
}
```

Анализ программы начнем с констант и переменных, определенных в программе. Следующие директивы:

```
#define  EN        _RA1
#define  RS        _RA0
```

обозначают, какие сигнальные линии жидкокристаллического дисплея каким портам присвоены. В данном случае линия разрешения работы EN будет управляться битом 1 порта A, а линия команд RS будет управляться битом 0 этого же порта. Замечу, что в данной схеме линия чтения/записи LCD (RW, см. **Рис. 9.4**) подключена к общему проводу, т. е. выполняются только команды записи в дисплей.

Константы `PMCS1` и `PMRD` обозначают выводы порта B, с которых сигналы управления подаются на управляющие входы PMCS1 и PMRD интерфейса PMP:

```
#define  PMCS1  _RB0
#define  PMRD    _RB1
```

Функция `DataSet` выполняет чтение байта с линий PMD7:PMD0 в соответствии с временной диаграммой, показанной на **Рис. 9.5**.

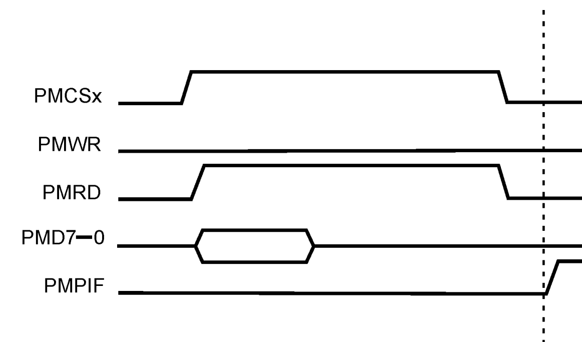

Рис. 9.5. Временная диаграмма чтения данных с интерфейса PMP

Легко проверить, что операторы

```
PMDOUT1  =  c1;
PMCS1  =  1;
PMRD  =  1;
PMRD  =  0;
PMCS1  =  0;
```

этой функции в точности эмулируют временную диаграмму, показанную на этом рисунке. Биты данных на линиях PMD7:PMD0 определяются содержимым регистра PMDOUT1, в который они загружаются из переменной `c1`. Признаком успешного завершения операции является установка флага прерывания PMPIF по завершении операции чтения.

Акцентирую внимание читателей на том, что данные читаются с линий PMD внешним устройством, в данном случае жидкокристаллическим дисплеем, при подаче внешних сигналов синхронизации PMCS1 и PMRD. В этом примере мы эмулируем эти сигналы самим микроконтроллером посредством соответствующих битов регистра PORTB. Обратите внимание на то, что сигнал чтения PMRD устанавливается в активное состояние с некоторой задержкой по отношению к сигналу выбора кристалла PMCS1 и становится неактивным перед снятием сигнала PMCS1. Сигнал записи PMWR на этой диаграмме во время цикла чтения должен находиться в неактивном состоянии (НИЗКИЙ уровень). Точно такая же временная диаграмма характерна и для записи данных по интерфейсу PMP, только в этом случае активным будет сигнал записи PMWR, а PMRD должен удерживаться в состоянии НИЗКОГО уровня.

Кратко остановимся на функциях, выполняющих операции с жидкокристаллическим дисплеем. Функция `ClearLCD` выполняет очистку дисплея, `InitLCD` инициализирует устройство, а с помощью функции `WriteLCD` осуществляется вывод символа данных на жидкокристаллический дисплей. Я не буду подробно останавливаться на принципах программирования жидкокристаллических индикаторов, обширную информацию по этой теме можно обнаружить во многих Интернет-источниках. Замечу только, что при любых операциях с LCD всегда следует проверять бит состояния, который выставляется в разряде D7 индикатора. До тех пор пока он равен 1, никаких последующих операций с устройством выполнять нельзя. В нашем примере (как и во многих реальных устройствах) мы не проверяем бит D7, а формируем определенную задержку, достаточную для того, чтобы текущая операция завершилась. Эта задержка реализуется функцией `Delay` посредством 16-битного таймера Timer 1.

Преимущество такого подхода заключается в следующем: если по какой-либо причине жидкокристаллический индикатор окажется неисправным, то бит D7 может никогда не установиться в 1, что приведет к «вечному» циклу и зависанию программы (если, конечно, не включен сторожевой таймер WDT, но в этом случае произойдет аварийный рестарт программы, что тоже плохо). В любом случае время задержки определяется параметрами конкретного LCD, поэтому всегда полезно предварительно изучить документацию на устройство.

Операторы

```
TRISA  =  0x0;
AD1PCFG  =  0xffff;
TRISB  =  0x0;
```

основной программы конфигурируют порты A и B как выходные. Поскольку порт B по умолчанию используется модулем АЦП, то эту функцию следует отключить, установив в 1 все биты регистра AD1PCFG.

Для инициализации интерфейса PMP в режиме «ведомого» следует установить бит разрешения PMPEN в регистре PMCON и сбросить биты 9:8 регистра режима PMMODE. Если не требуется каких-либо специфичных настроек, можно просто сбросить все биты этого регистра. Эти действия выполняются с помощью следующих операторов:

```
PMCONbits.PMPEN  =  0x1;
PMMODE  =  0x0;
```

О завершении операции чтения данных из микроконтроллера можно судить по установленному биту прерывания _PMPIF. В начале программы он сбрасывается оператором

```
_PMPIF  =  0;
```

Чтение строки `str` для вывода на дисплей осуществляется в цикле `while (*str != 0)` основной программы:

```
while  (*str  !=  0)
{
      WriteLCD(*str++);
      while  (_PMPIF  ==  0);
}
```

Очередной символ читается из микроконтроллера и выводится на дисплей функцией `WriteLCD`, при этом выполняется автоинкремент указателя, который продвигается к следующему байту строки. Далее ожидается установка флага прерывания _PMPIF, после чего выполняется следующая итерация, и так до тех пор, пока не будет обнаружен нулевой символ.

Следующий пример, который мы рассмотрим, демонстрирует работу микроконтроллера в режиме «ведущего» интерфейса PMP. Аппаратная часть проекта показана на **Рис. 9.6**.

В этой схеме байт данных микроконтроллера PIC24FJ128GA010 через выводы интерфейса PMP PMD7:PMD0 выводится в регистр DD1 74HC373, к вы-

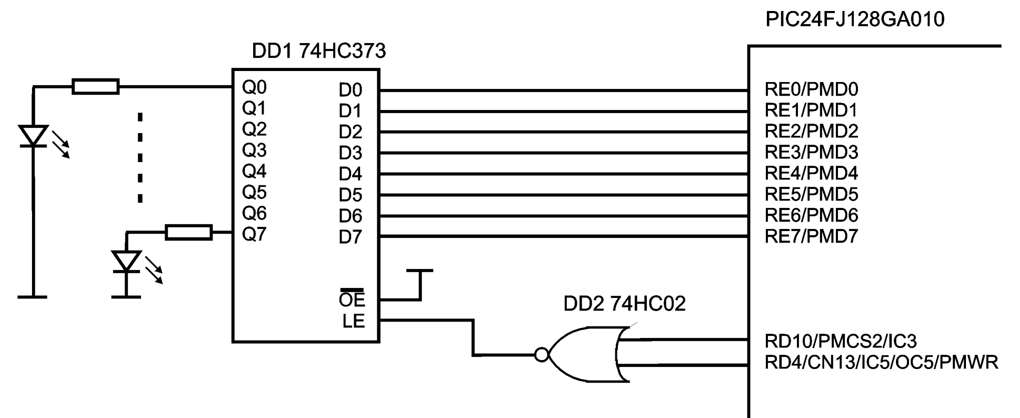

Рис. 9.6. Аппаратная часть проекта

ходам которого подключены светодиоды. Микроконтроллер работает в режиме «ведущего», формируя все необходимые сигналы интерфейса. В данном случае выбрана такая конфигурация шины, при которой для синхронизации передачи данных используются сигналы PMCS2 (выбор кристалла 2) и PMWR (запись). Оба сигнала через схему совпадения на микросхеме DD2 стробируют запись данных на входных линиях D0:D7 регистра-защелки DD1.

Программная часть проекта разработана в среде MPLAB IDE и включает в себя файл с исходным текстом, который представлен далее.

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

void  main()
{
        PMCON  =  0x8392;
        PMMODE  =  0x200;

        _PMPIF  =  0;

        PMDIN1  =  0x5;
        while  (1)
          {
      }
}
```

Программа очень проста и содержит небольшое количество операторов. Вывод байта (0x5) в параллельный интерфейс осуществляется путем записи его значения в регистр PMDIN1; далее модуль интерфейса PMP генерирует все необходимые для операции записи сигналы синхронизации. Для настройки интерфейса PMP в режиме «ведущего» программа должна установить соответствующие биты в регистре управления PMCON и в регистре режима PMMODE.

Признаком окончания цикла записи данных является установка флага прерывания PMPIF. В этой программе мы не проверяем его состояние, но при выводе последовательности байтов во внешнее устройство следует всегда контролировать его значение.

В следующем демонстрационном проекте мы прочитаем данные с внешнего устройства и выведем их на жидкокристаллический индикатор. Интерфейс PMP будет функционировать в режиме «ведущего». Аппаратная часть проекта показана на **Рис. 9.7**.

Эта схема работает следующим образом: при нажатии кнопки вызывается прерывание INT3 микроконтроллера PIC24FJ128GA010, функция-обработчик которого считывает посредством линий PMD7:PMD0 интерфейса PMP байт данных со входов буферных микросхем DD1 и DD2. Затем этот байт в символьном виде выводится на жидкокристаллический индикатор. Линии данных D7:D0 индикатора подключены к порту A, а управляющие сигналы на LCD подаются с выводов 0 (линия E) и 2 (линия RS) порта B. Чтение данных с буферов DD1 и DD2 осуществляется с помощью сигналов PMCS2 и PMRD

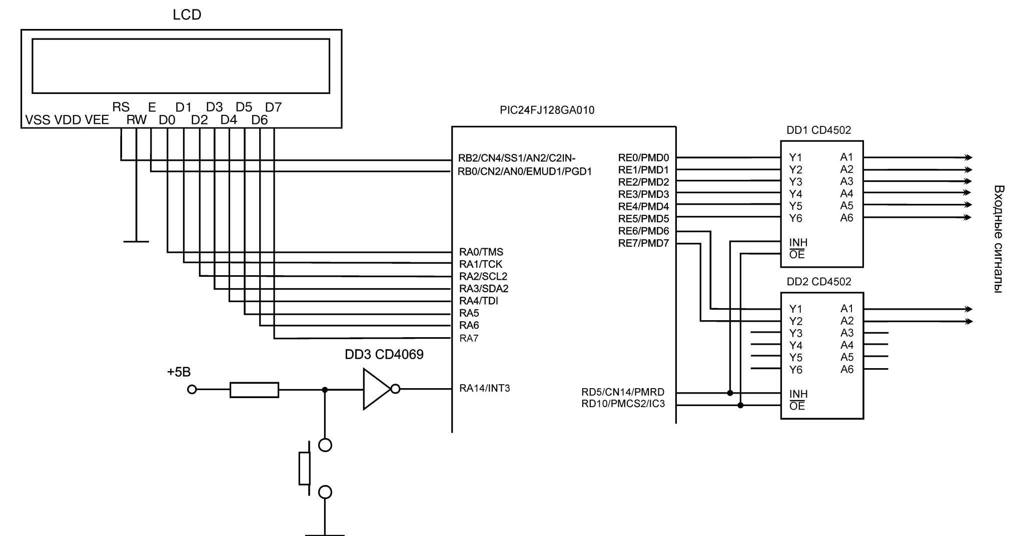

Рис. 9.7. Аппаратная часть проекта

интерфейса PMP. Как и в предыдущих проектах, тактовая частота микроконтроллера равна 8 МГц.

Исходный текст программы содержит следующие строки:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  INT3Tris  TRISAbits.TRISA14
#define  EN        _RB0

#define  RS        _RB2
#define  DATA      PORTA

char  c1;

void  Delay(void)
{
  _T1IF  =  0x0;
  TMR1  =  0;
  PR1  =  0xffff;
  T1CON  =  0x8000;
  while  (_T1IF  ==  0);
  T1CON  =  0x0;
}

void  ClearLCD(void)
{
    EN  =  1;
    RS  =  0;
    DATA  =  0x1;
```

```
    EN  =  0;
    Delay();
}

void  InitLCD(void)
{
    EN  =  1;
    RS  =  0;
    DATA  =  0x38;
    EN  =  0;
    Delay();

    EN  =  1;
    RS  =  0;
    DATA  =  0x0E;
    EN  =  0;
    Delay();

    EN  =  1;
    RS  =  0;
    DATA  =  0x6;
    EN  =  0;
    Delay();

}

void  WriteLCD(char  c1)
{
    EN  =  1;
    RS  =  1;
    DATA  =  c1;
    EN  =  0;
    Delay();

}

void  __attribute__((interrupt))  _INT3Interrupt(void)
{
  _INT3IF  =  0;
  InitLCD();
  ClearLCD();
  c1  =  PMDIN1;
  while  (_PMPIF  ==  0);
  WriteLCD(c1);
}

void  main()
{

  TRISA  =  0x0;
  AD1PCFG  =  0xffff;
  TRISB  =  0x0;
  INT3Tris  =  0x1;

  _INT3IF  =  0;
```



```
  _INT3IE  =  1;

  _PMPIF  =  0;
  PMCON  =  0x8350;
  PMMODE  =  0x20F;
  InitLCD();
  ClearLCD();
  while  (1)
    {
  }
}
```

Прежде всего, для функционирования интерфейса PMP в режиме «ведущего» нужно установить соответствующим образом биты регистра управления PMCON и регистра режима PMMODE. В режиме чтения формируется небольшая задержка (биты 0:3 регистра PMMODE).

Для функционирования требуемых выводов портов A и B в качестве выходов должны быть сброшены соответствующие биты в регистрах TRIS. Для использования прерывания INT3 необходимо разрешить его с помощью операторов

```
_INT3IF  =  0;
_INT3IE  =  1;
```

Собственно чтение байта данных с линий PMD7:PMD0 и вывод символьного представления на LCD выполняется в функции-обработчике прерывания `_INT3Interrupt`. Для выполнения цикла чтения с генерацией соответствующих сигналов шинного интерфейса необходимо прочитать регистр PMDIN1:

```
c1  =  PMDIN1;
```

В результате байт данных сохраняется в переменной `c1` и затем выводится на жидкокристаллический индикатор с помощью функции `WriteLCD`. Как обычно, функция-обработчик прерывания должна сбрасывать флаг прерывания. Для управления LCD и выводом на него данных программа использует функции `InitLCD`, `ClearLCD` и `WriteLCD`.

Настройка интерфейса PMP в каждом конкретном случае зависит от внешнего устройства, с которым происходит обмен данными. Особенно это касается полярности активных уровней сигналов, которые используются для синхронизации обмена данными.

До сих пор мы рассматривали проекты, в которых в качестве активного устройства выступала система с микроконтроллером, а устройство, в которое данные записывались или с которого читались, было реализовано в форме аппаратного модуля (регистр или буфер). В большинстве систем процесс обмена данными по параллельному интерфейсу осуществляется между двумя и более «интеллектуальными» устройствами, в качестве которых используются микроконтроллеры, микропроцессоры или цифровые процессоры сигналов. Следующие проекты, которые мы рассмотрим, демонстрируют подобное взаимодействие. Использование параллельного интерфейса PMP существенно упрощает разработку таких систем, поскольку разработчик не теряет времени на программирование отдельных сигналов шины.

В первом проекте обмен данными будет осуществляться между микроконтроллерами PIC24FJ128GA010. Одно из устройств работает в режиме «ведущего» и считывает байты данных по интерфейсу PMP из другого, работающего в режиме «ведомого». Аппаратная часть проекта представлена на **Рис. 9.8**.

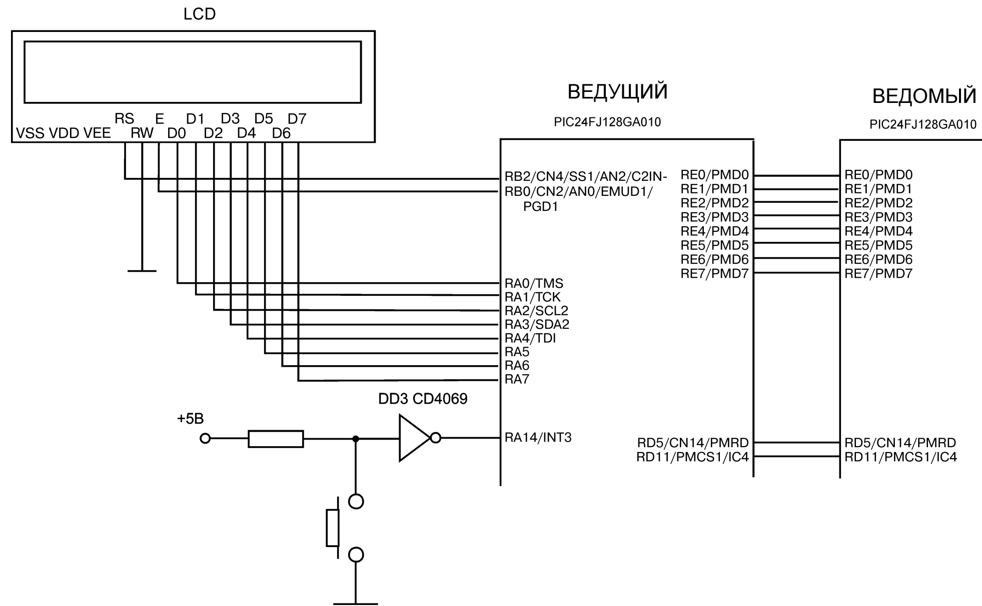

Рис. 9.8. Аппаратная часть проекта

«Ведущий» инициирует чтение байта с «ведомого» при возникновении прерывания INT3 (нажатие кнопки). Считанный байт данных в символьном виде выводится на жидкокристаллический индикатор. Линии данных PMD7:PMD0 и линии управляющих сигналов PMCS1 (выбор устройства) и PMRD (чтение) обоих микроконтроллеров соединены между собой.

Программная часть проекта разработана в среде MPLAB IDE с помощью мастера проектов и включает две отдельные программы для «ведущего» и «ведомого» устройств. Исходный текст программы для «ведущего» имеет следующий вид:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define    EN         _RB0
#define    RS         _RB2
#define    DATA       PORTA

char  c1;
int  cnt  =  0;

void  Delay(void)
```



```
{
  _T1IF  =  0x0;
  TMR1  =  0;
  PR1  =  0xffff;
  T1CON  =  0x8000;
  while  (_T1IF  ==  0);
  T1CON  =  0x0;
}

void  ClearLCD(void)
{
    EN  =  1;
    RS  =  0;
    DATA  =  0x1;
    EN  =  0;
    Delay();
}

void  InitLCD(void)
{
    EN  =  1;
    RS  =  0;
    DATA  =  0x38;
    EN  =  0;
    Delay();

    EN  =  1;
    RS  =  0;
    DATA  =  0x0E;
    EN  =  0;
    Delay();

    EN  =  1;
    RS  =  0;
    DATA  =  0x6;
    EN  =  0;
    Delay();

}

void  WriteLCD(char  c1)
{
    EN  =  1;
    RS  =  1;
    DATA  =  c1;
    EN  =  0;
    Delay();
  }

void  __attribute__  ((interrupt))  _INT3Interrupt(void)
{
  _INT3IF  =  0;
  c1  =  PMDIN1;
  while  (_PMPIF  ==  0);
  if  (cnt  ==  15)
```

```
    {
      InitLCD();
      ClearLCD();
      cnt = 0;
    }
  WriteLCD(c1);
  cnt++;
}

void main()
{
  TRISA = 0x0;
  AD1PCFG = 0xffff;
  TRISB = 0x0;
  TRISAbits.TRISA14 = 0x1;

  _INT3IF = 0;
  _INT3IE = 1;
  _PMPIF = 0;
  PMCON = 0x8340;
  PMMODE = 0x20F;

  InitLCD();
  ClearLCD();
  while (1)
    {
  }
}
```

Для работы микроконтроллера в режиме «ведущего» и установки параметров управляющих сигналов прежде всего настраиваются требуемые биты регистра управления PMCON и регистра режима PMMODE. Это выполняется в основной программе операторами

```
PMCON = 0x8340;
PMMODE = 0x20F;
```

Здесь биты 9:8 регистра PMMODE задают режим «ведущего», а младший байт определяет полярности сигналов управления PMCS1 и PMRD. Кроме того, в регистре режима устанавливается определенная задержка для четкой синхронизации чтения данных на шине (младший полубайт). Выводы портов A и B с помощью регистров TRIS настраиваются в соответствии с направлением сигналов на соответствующих линиях управления/данных жидкокристаллического индикатора и линии прерывания INT3.

Чтение байта данных, полученного от «ведомого», осуществляется в функции-обработчике прерывания INT3, инициализация которого производится с помощью операторов

```
_INT3IF = 0;
_INT3IE = 1;
```

Каждый следующий байт читается только в том случае, если принят предыдущий. С помощью счетчика `cnt` курсор дисплея возвращается в исходную

позицию после вывода 15-го символа. Функции для работы с LCD мы уже рассматривали в предыдущих примерах, поэтому останавливаться подробно на них я не буду.

«Ведомый» микроконтроллер использует те же линии данных и управления, что и «ведущий», но настраивается по-другому. Исходный текст программы для «ведомого» устройства показан далее.

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

void main()
{
  char c1 = '0';
  _PMPIF = 0;
  PMCON = 0x8380;
  PMMODE = 0x0;
  while (1)
  {
    PMDOUT1 = c1;
    while (PMSTATbits.OBE != 1);
    c1 = c1 + 1;
  }
}
```

Программный код для «ведомого» микроконтроллера состоит всего из нескольких строк. Первым выводимым на шину символом будет 0, поэтому переменная `c1` инициируется именно этим значением. При последовательном чтении данных двоичный код выводимого символа будет увеличиваться на 1. Регистр управления PMCON настраивается соответствующим образом с учетом полярности сигналов «ведущего». В регистре режима PMMODE биты 9:8 должны быть сброшены в 0 (режим «ведомого»).

Собственно чтение байта с шины данных интерфейса PMP осуществляется в цикле `while(1)`. Байт данных запоминается в регистре PMDOUT1:

```
PMDOUT1 = c1;
```

При запросе на чтение данных «ведущего» микроконтроллера байт данных из регистра PMDOUT1 выставляется на линии данных PMD7:PMD0, откуда он считывается «ведущим». Признаком окончания передачи байта данных «ведомым» является установленный бит OBE регистра состояния. При равенстве этого бита единице байт считается переданным. Проверка бита OBE выполняется оператором

```
while (PMSTATbits.OBE != 1);
```

По окончании передачи очередного байта значение переменной инкрементируется, и программа ожидает следующего запроса на чтение от «ведущего». Как видно из этого проекта, взаимодействие двух микроконтроллеров PIC24 при обмене данными осуществляется минимальным количеством команд, поскольку внутренние аппаратные средства обоих устройств полностью обеспечивают генерацию сигналов интерфейса.

В том случае, если в качестве одного из устройств обмена данными («ведущего» или «ведомого») выступает система на базе микроконтроллера другого типа, то реализация такого интерфейса в каждом конкретном случае будет полностью зависеть от аппаратно-программной конфигурации используемого микроконтроллера или микропроцессора.

Наш следующий проект демонстрирует, каким образом можно осуществить взаимодействие по параллельному интерфейсу PMP между микроконтроллерами различных типов. В этом проекте в качестве «ведущего» будет выступать микроконтроллер семейства PIC24F, а в качестве «ведомого» — микроконтроллер, совместимый с 8052. В данном случае термин «ведомый» весьма условный, поскольку микроконтроллер 8052 не привязан к настройкам интерфейса PMP, имеющегося в PIC24. Данные с микроконтроллера 8052 будут считываться в микроконтроллер PIC24. Аппаратная часть проекта показана на **Рис. 9.9**.

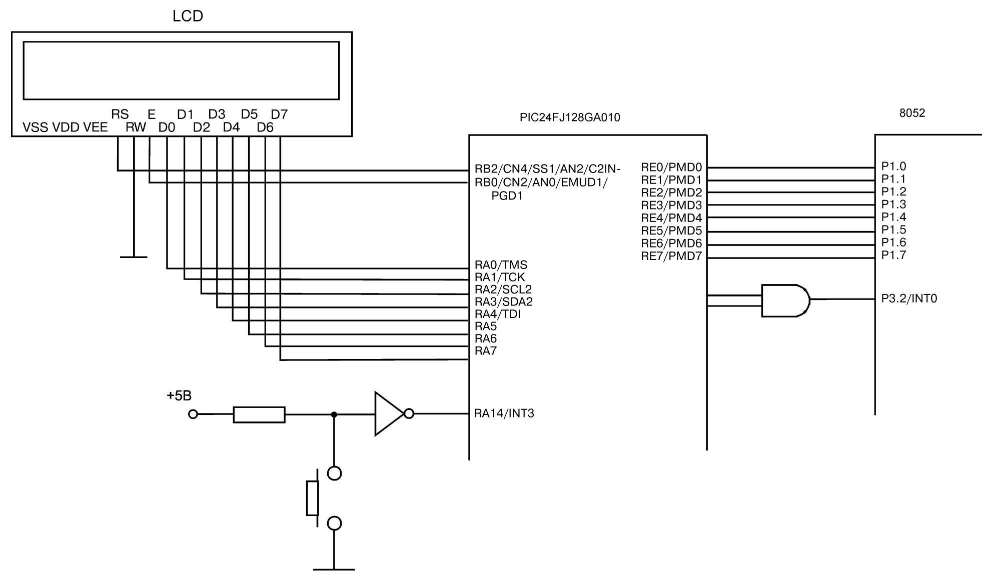

Рис. 9.9. Аппаратная часть проекта

Для читателей, не знакомых с архитектурой систем на базе 8052, поясню некоторые детали работы схемы. Думаю, трудностей здесь не возникнет, поскольку 8052 имеет простую аппаратно-программную архитектуру, которую несложно понять. Для передачи данных в микроконтроллере 8052 используется порт P1, линии 7:0 которого соединены с соответствующими линиями интерфейса PMD7:PMD0 микроконтроллера PIC24. Сигналы PMCS2 и PMRD через элемент «И» поступают на вход прерывания INT0 микроконтроллера 8052. Это прерывание вызывается спадающим фронтом сигнала на линии INT0. Получив сигнал прерывания, микроконтроллер 8052 передает управление функции-обработчику, которая выставляет на линии порта P1

байт данных. Этот байт считывается «ведущим» и отображается на жидкокристаллическом индикаторе.

Программное обеспечение, как и в предыдущем проекте, включает отдельные программы для «ведущего» и «ведомого» устройств. Исходный текст программы «ведущего» микроконтроллера семейства PIC24F показан далее.

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define      EN    _RB0

#define      RS    _RB2
#define      DATA  PORTA

char  c1;

void  Delay(void)
{
  _T1IF  =  0x0;
  TMR1  =  0;
  PR1  =  0xffff;
  T1CON  =  0x8000;
  while  (_T1IF  ==  0);
  T1CON  =  0x0;
}

void  ClearLCD(void)
{
    EN  =  1;
    RS  =  0;
    DATA  =  0x1;
    EN  =  0;
    Delay();
}

void  InitLCD(void)
{
    EN  =  1;
    RS  =  0;
    DATA  =  0x38;
    EN  =  0;
    Delay();

    EN  =  1;
    RS  =  0;
    DATA  =  0x0E;
    EN  =  0;
    Delay();

    EN  =  1;
    RS  =  0;
    DATA  =  0x6;
```

```
    EN  =  0;
    Delay();

}

void  WriteLCD(char  c1)
{
    EN  =  1;
    RS  =  1;
    DATA  =  c1;
    EN  =  0;
    Delay();
  }

void  __attribute__  ((interrupt))  _INT3Interrupt(void)
{
  _INT3IF  =  0;
  c1  =  PMDIN1;
  while  (_PMPIF  ==  0);
  WriteLCD(c1);
}

void  main()
{
  TRISA  =  0x0;
  AD1PCFG  =  0xffff;
  TRISB  =  0x0;
  TRISAbits.TRISA14  =  0x1;
  _INT3IF  =  0;
  _INT3IE  =  1;
  _PMPIF  =  0;
  PMCON  =  0x8340;
  PMMODE  =  0x20F;

  InitLCD();
  ClearLCD();
  while  (1)
    {
    }
}
```

Этот исходный текст во многом напоминает текст программы «ведущего» из предыдущего примера, за исключением обработчика прерывания INT3, который немного модифицирован. Большинство операторов программы уже знакомы читателям, поэтому рассматривать их мы не будем.

Исходный текст программы для микроконтроллера 8052, выполняющего передачу данных «ведущему» PIC24, написан на ассемблере и выглядит следующим образом:

```
      org      0h
      jmp      start
Int0_ISR:
      org      3h
      mov      P1,  A
```



```
      inc      A
      reti
start:
      mov      A,  #'1'
      mov      P1,  #0h
      setb     IT0
      setb     EA
      setb     EX0
      jmp      $
      end
```

В основной программе (метка `start`) в регистр-аккумулятор A заносится числовое значение символа ‘1’, которое затем будет инкрементироваться каждый раз при передаче символа из функции-обработчика в интерфейс PMP. Порт P1 настраивается на вывод данных, для чего во все его биты записываются нули командой

```
mov    P1,  #0h
```

Далее настраивается аппаратное прерывание INT0:

- устанавливается срабатывание по спадающему фронту на линии INT0 (команда `setb IT0`);
- разрешаются все прерывания (команда `setb EA`);
- разрешается прерывание по линии INT0 (команда `setb EX0`).

Программа-обработчик прерывания INT0 располагается по фиксированному адресу 0x3 в сегменте программного кода и включает всего две команды. Команда

```
mov  P1,  A
```

выставляет данные из аккумулятора A на линии порта P1, а команда `reti` завершает обработчик. Как видно из этой программы, механизмом синхронизации для микроконтроллера 8052 в данном случае является вызов прерывания при активных сигналах PMCS2 и PMRD микроконтроллера PIC24.

Параллельный интерфейс микроконтроллеров семейства PIC24F можно настроить и для работы в более сложных конфигурациях, где применяется адресация устройств по отдельным линиям. Такие варианты часто применяются при реализации интерфейсов с запоминающими устройствами и дисковыми накопителями. В любом случае наличие интерфейса PMP существенно упрощает проектирование сложных систем передачи данных с шинным интерфейсом. Для дополнительной информации по данной теме я рекомендую обратиться к фирменной документации Microchip.

ГЛАВА 10

ПОСЛЕДОВАТЕЛЬНЫЙ ИНТЕРФЕЙС МИКРОКОНТРОЛЛЕРОВ PIC24F

Асинхронный последовательный интерфейс является одним из первых интерфейсов, разработанных для обмена данными между устройствами. Однако, несмотря на свой почтенный возраст, этот интерфейс в настоящее время продолжает очень широко использоваться, причем его не смогли вытеснить ни USB, ни другие, разработанные в последнее время, стандарты обмена данными. Причиной популярности асинхронного последовательного интерфейса является его надежность, простота реализации и дешевизна. В промышленных сетях обмена данными и управления этот интерфейс занимает доминирующее положение благодаря относительно новым быстродействующим стандартам, таким, например, как RS-485.

Асинхронный обмен данными лежит в основе нескольких популярных стандартов, наиболее распространенными из которых являются:

- RS-232 — на него часто ссылаются как на «последовательный порт» обмена данными. Данный протокол применяется в персональных компьютерах, модемах и другом коммуникационном оборудовании;
- RS-485 — протокол асинхронного обмена, позволяющий взаимодействовать одновременно с несколькими устройствами. Данный протокол используется в промышленных сетях обмена данными;
- шина LINBUS — используется в недорогих мобильных системах с автонастройкой несущей;
- беспроводные ИК-протоколы обмена данными; обычно работают с частотой модуляции 38...40 кГц.

Микроконтроллеры PIC24F поддерживают все указанные типы протоколов. Функциональные узлы модуля асинхронного обмена данными микроконтроллеров PIC24F можно представить в виде блок-схемы (**Рис. 10.1**).

Каждый микроконтроллер семейства PIC24F может иметь один или более асинхронных портов, поэтому в обозначениях на рисунке присутствует литера «х». В каждом конкретном случае программисту следует обращаться к технической документации на данное устройство.

Рис. 10.1. Упрощенная схема модуля асинхронного обмена данными

10.1. Аппаратно-программная архитектура UART

Для управления параметрами обмена данными, а также контроля передачи/приема данных по асинхронному интерфейсу используется группа программно доступных регистров, которые мы вкратце и рассмотрим.

Назначение всех битов мы обсуждать не будем, а рассмотрим только те, которые наиболее часто используются при настройках. Как правило, при программировании асинхронного обмена данными разработчик использует только некоторые биты регистров, оставляя остальные установки по умолчанию, — этого вполне достаточно для выполнения несложных процедур обмена данными.

Для управления и контроля асинхронным обменом данными служит группа регистров, функции которых перечислены далее:

- UxMODE — регистр установки режимов работы. Он позволяет установить режим работы асинхронного порта «х». При анализе примеров и разработке приложений мы будем использовать бит 15 (обозначается как UARTEN) для разрешения работы порта, биты 8:9 (UEN) для управления выводами портов, бит 3 (BRGH) для установки повышенной скорости обмена, биты 1:2 (PDSEL) для выбора формата данных и четности и бит 0 (STSEL) для выбора количества стоповых битов;
- UxSTA — регистр управления/состояния асинхронного порта «х». Нас будет интересовать бит 10 (UTXEN) разрешения передачи, бит 9 (UTXBF) состояния буфера передачи, бит 0 (URXDA) состояния буфера приема;
- UxRXREG — регистр-приемник данных. Здесь бит 8 используется только при обмене 9-битными данными, а биты 0:7 будут содержать 8 бит данных. Данные доступны только для чтения;
- UxTXREG — регистр передачи данных. Здесь бит 8 используется только при обмене 9-битными данными, а биты 0:7 будут содержать 8 бит данных. Данные доступны только для записи;

- UxBRG — регистр установки скорости обмена. Биты 15:0 содержат значение коэффициента деления частоты.

Прежде чем выполнять обмен данными по асинхронному интерфейсу, следует установить скорость обмена данными. Модуль асинхронного порта включает генератор синхронизации обмена, который задает скорость обмена данными. Для этого в регистр установки скорости обмена следует записать 16-битное значение, которое и будет определять требуемую скорость. Это число зависит от значения бита BRGH регистра режима UxMODE. Если BRGH = 0, то значение, записываемое в регистр UxBRG, вычисляется из выражения:

Скорость обмена = $F_{CY}/(16 \times (UxBRG + 1))$,

откуда

UxBRG = $F_{CY}/16 \times$ Скорость обмена $- 1$

Здесь F_{CY} — значение тактовой частоты, деленное на 2. Например, если процессор работает на частоте 8 МГц, то F_{CY} будет равна 4 МГц. Максимальная скорость обмена в этом случае (BRGH = 0) достигает $F_{CY}/16$ при UxBRG = 0, а минимальная скорость обмена будет равна $F_{CY}/16 \times 65536$.

Если бит BRGH = 1, то выражение для вычисления значения регистра UxBRG будет иметь вид:

Скорость обмена = $F_{CY}/(4 \times (UxBRG + 1))$,

откуда

UxBRG = $F_{CY}/4 \times$ Скорость обмена $- 1$

Максимальная скорость обмена в этом случае (BRGH = 1) достигает $F_{CY}/4$ при UxBRG = 0, а минимальная скорость обмена будет равна $F_{CY}/4 \times 65536$.

При работе модуля в режиме IrDA на выводе BCLKx микроконтроллера будет генерироваться сигнал с частотой, в 16 раз превышающей заданное значение скорости обмена. В данной книге режим IrDA рассматриваться не будет.

10.2. Практическое использование последовательного порта

Рассмотрим на практике программирование последовательного порта микроконтроллеров PIC24F. Как и ранее, будем работать с устройством PIC24FJ128GA010.

В первом примере будем передавать введенный с последовательного порта символ обратно в последовательный порт. Предположим, что обмен данными происходит через порт UART2 микроконтроллера в 8-битном формате, с одним стоповым битом на скорости 9600 бод. Аппаратное управление потоком использовать не будем для упрощения программирования.

С помощью мастера проектов среды MPLAB IDE создадим новый проект и включим в него, как мы это делали в предыдущих примерах, файл p24fj128ga010.gld. Теперь приступим к разработке программы на языке Си, после чего включим этот файл в проект.

Как обычно, в начале программы укажем заголовочный файл:

```
#include  <p24fj128ga010.h>
```

Затем определим источник тактовой частоты микроконтроллера, для чего добавим в исходный текст следующий макрос:

```
_CONFIG2(FCKSM_CSDCMD  &  OSCIOFNC_ON  &  POSCMOD_HS  &  FNOSC_PRI)
```

Теперь укажем значения некоторых констант, которые нам понадобятся при настройке параметров последовательного порта:

```
#define  SYSCLK             8000000
#define  BAUDRATE2          9600
#define  BAUDRATEREG2       SYSCLK/8/BAUDRATE2-1
```

Здесь мы выбираем тактовую частоту, равную 8 МГц (`SYSCLK`), и скорость обмена 9600 бод (`BAUDRATE2`).. Константа `BAUDRATEREG2` используется в регистре установки скорости обмена U2BRG и вычисляется по формуле, приведенной ранее в разделе 10.1 (примем, что бит BRGH регистра режима U2MODE (порт UART2) должен быть установлен в 1 (повышенная скорость обмена)).

Кроме того, для удобства определим биты конфигурации портов ввода/вывода, которые будут использоваться для последовательного обмена данными:

```
#define  UART2_TX_TRIS      TRISFbits.TRISF5
#define  UART2_RX_TRIS      TRISFbits.TRISF4
```

В качестве линии вывода (сигнал TX) в программе будет использоваться RF5, а в качестве линии ввода (сигнал RX) — RF4.

После описания базовых констант нашей программы займемся реализацией алгоритма обмена данными по интерфейсу UART2. Прежде всего, нужно установить параметры обмена данными, записав соответствующие значения в регистры модуля. Этот процесс называется инициализацией порта, и мы реализуем его в отдельной функции, которую назовем `UARTInit`:

```
void  UART2Init()
{
    UART2_TX_TRIS  =  0;
    UART2_RX_TRIS  =  1;
    U2BRG  =  BAUDRATEREG2;

    U2MODE  =  0;
    U2MODEbits.BRGH  =  1;
    U2MODEbits.UARTEN  =  1;

    U2STA  =  0;
    U2STAbits.UTXEN  =  1;

    IFS1bits.U2RXIF  =  0;
}
```

В этой функции с помощью операторов

```
UART2_TX_TRIS  =  0;
UART2_RX_TRIS  =  1;
```

устанавливается направление обмена данными через выводы RF4 и RF5. Затем в регистр установки скорости обмена U2BRG записывается ранее определенная константа BAUDRATEREG2:

```
U2BRG  =  BAUDRATEREG2;
```

С помощью следующей группы операторов устанавливаются соответствующие биты регистра режима U2MODE:

```
U2MODE  =  0;
U2MODEbits.BRGH  =  1;
U2MODEbits.UARTEN  =  1;
```

Здесь установка бита UARTEN разрешает работу модуля. Наконец, в регистре контроля и управления U2STA нужно установить бит разрешения передачи UTXEN:

```
U2STA  =  0;
U2STAbits.UTXEN  =  1;
```

Практически все готово для обмена данными. Теперь возникает вопрос: а как система узнает, что в буфере приема имеется байт или что в буфер передачи помещен байт для отправки? В этой конкретной программе для анализа буфера данных приемника используется флаг прерывания U2RXIF, который устанавливается при поступлении символа в буфер. Этот флаг устанавливается независимо от того, разрешено прерывание последовательного порта или нет. В подпрограмме инициализации мы сбрасываем этот флаг:

```
IFS1bits.U2RXIF  =  0;
```

Для обмена данными нам понадобятся еще две функции: для приема символа из последовательного порта и его отправки обратно в последовательный порт. Функцию приема символа назовем UART2GetChar(), а ее исходный текст приведен далее:

```
char  UART2GetChar()
{
  char  Temp;
  while(IFS1bits.U2RXIF  ==  0);
  Temp  =  U2RXREG;
  IFS1bits.U2RXIF  =  0;
  return  Temp;
}
```

Функция ожидает установки флага прерывания U2RXIF (оператор while), что будет свидетельствовать о помещении байта данных в биты 0:7 регистра приемника U2RXREG модуля. Затем содержимое регистра помещается в переменную Temp, которая и возвращается вызывающей программе. Флаг прерывания должен быть обязательно сброшен во избежание повторного вызова прерывания. Замечу, что в этой программе мы не используем механизм прерываний для обработки данных последовательного порта, но в ситуации, когда прерывание последовательного порта будет разрешено, это приведет к непроизводительной трате машинных циклов.

Для передачи байта данных в последовательный порт используется функция UART2PutChar, исходный текст который представлен далее:

```
void    UART2PutChar(char  Ch)
{
  while(U2STAbits.UTXBF  ==  1);
  U2TXREG  =  Ch;
}
```

Функция имеет единственный параметр, представляющий собой символ (байт) данных, который передает ей основная программа. Функция ожидает освобождения буфера передачи, в качестве которого используется регистр U2TXREG. Если бит UTXBF регистра U2STA установлен, то это свидетельствует о наличии символа в буфере передачи, поэтому следует ожидать его освобождения (оператор while). Если буфер освободился, то можно записать в него байт данных для отправки:

```
U2TXREG  =  Ch;
```

Теперь все это можно собрать вместе в одну программу. Вот ее исходный текст:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  SYSCLK             8000000
#define  BAUDRATE2          9600
#define  BAUDRATEREG2       SYSCLK/8/BAUDRATE2-1

#define  UART2_TX_TRIS      TRISFbits.TRISF5
#define  UART2_RX_TRIS      TRISFbits.TRISF4

void  UART2Init()
{
      UART2_TX_TRIS  =  0;
      UART2_RX_TRIS  =  1;
      U2BRG  =  BAUDRATEREG2;

      U2MODE  =  0;
      U2MODEbits.BRGH  =  1;
      U2MODEbits.UARTEN  =  1;

      U2STA  =  0;
      U2STAbits.UTXEN  =  1;

      IFS1bits.U2RXIF  =  0;
}

void  UART2PutChar(char  Ch)
{
    while(U2STAbits.UTXBF  ==  1);
    U2TXREG  =  Ch;
}

char  UART2GetChar()
{
  char  Temp;
```

```
  while(IFS1bits.U2RXIF == 0);
  Temp = U2RXREG;
  IFS1bits.U2RXIF = 0;
  return Temp;
}

void main()
{
    char c1;

    UART2Init();
    while (1)
      {
        c1 = UART2GetChar();
        UART2PutChar(c1);
      }
}
```

Поскольку микроконтроллер PIC24FJ128GA010 имеет еще один последовательный порт, то, сделав небольшие изменения в исходном тексте программы, можно работать и с портом 1. Рассмотрим следующий пример, в котором мы увидим, как передать строку байтов в последовательный порт. Для этого, как и в предыдущем примере, создадим стандартный проект, но слегка модифицируем исходный текст программы, который будет выглядеть теперь так:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define SYSCLK              8000000
#define BAUDRATE2           9600
#define BAUDRATEREG2        SYSCLK/8/BAUDRATE2-1

#define UART2_TX_TRIS       TRISFbits.TRISF5
#define UART2_RX_TRIS       TRISFbits.TRISF4

void UART2Init()
{
    UART2_TX_TRIS = 0;
    UART2_RX_TRIS = 1;
    U2BRG = BAUDRATEREG2;

    U2MODE = 0;
    U2MODEbits.BRGH = 1;
    U2MODEbits.UARTEN = 1;

    U2STA = 0;
    U2STAbits.UTXEN = 1;
    IFS1bits.U2RXIF = 0;
}

void  UART2PutChar(char Ch)
{

  while(U2STAbits.UTXBF == 1);
  U2TXREG = Ch;
}

char UART2GetChar()
```



```
{
  char Temp;
  while(IFS1bits.U2RXIF == 0);
  Temp = U2RXREG;
  IFS1bits.U2RXIF = 0;
  return Temp;
}

void main()
{
    char c1;
    int i1;

    char s1[] = "ENTER THE CHARACTER:";
    char *ps1 = s1;

    char CRLF[] = {0xd, 0xa};
    char *pCRLF = CRLF;

    int len = strlen(s1);

    UART2Init();

    for(i1 = 0; i1 < len; i1++)
          UART2PutChar(*ps1++);
    UART2PutChar(*pCRLF++);
    UART2PutChar(*pCRLF);
    UART2PutChar('>');
    while (1)
      {
        c1 = UART2GetChar();
        c1 += 1;
               UART2PutChar(c1);
  }
}
```

Текст программы во многом нам уже знаком, поэтому рассмотрим только изменения, которые были внесены. Во-первых, в программе определяется строка символьных данных `s1`, которая будет выводиться в последовательный порт. Если эти данные принимает, например, терминальная программа, работающая на персональном компьютере, то пользователь увидит в окне приложения приглашение к вводу символов. Строка символов и приглашение выводятся следующими операторами:

```
for(i1 = 0; i1 < len; i1++)
    UART2PutChar(*ps1++);
UART2PutChar(*pCRLF++);
UART2PutChar(*pCRLF);
UART2PutChar('>');
```

Каждый байт строки `s1` выводится в последовательный порт при помощи функции `UART2PutChar` в цикле `for`, счетчиком для которого является длина строки `s1`, вычисленная при помощи функции `strlen`. После передачи очередного символа указатель `ps1` инкрементируется, указывая на следующий символ.

Строка CRLF представляет собой набор стандартных символов («возврат каретки» и «перевод строки»). После передачи приглашения из порта в переменную `c1` читается символ, инкрементируется и передается обратно.

До сих пор для приема-передачи байтов через последовательный порт мы использовали обычный метод опроса. Можно не загружать процессор, а использовать для работы с последовательным портом прерывания. В нашем следующем проекте мы будем использовать прерывание для приема байта из последовательного порта и пересылки его обратно. Функция-обработчик прерывания будет пересылать в последовательный порт код символа, увеличенный на 2, и переключать бит 0 порта RA0 в противоположное состояние. Это чисто демонстрационный пример, показывающий технику использования прерываний последовательного порта.

С помощью мастера проектов создадим в среде MPLAB IDE новый проект и добавим в него файл сценария p24fj128ga010.gld. Создадим файл, содержимое которого показано далее:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  PORTA_0             PORTAbits.RA0

#define  SYSCLK              8000000
#define  BAUDRATE2           9600
#define  BAUDRATEREG2        SYSCLK/8/BAUDRATE2-1

#define  UART2_TX_TRIS       TRISFbits.TRISF5
#define  UART2_RX_TRIS       TRISFbits.TRISF4

void  UART2Init()
{
    UART2_TX_TRIS  =  0;
    UART2_RX_TRIS  =  1;
    U2BRG  =  BAUDRATEREG2;

    U2MODE  =  0;
    U2MODEbits.BRGH  =  1;
    U2MODEbits.UARTEN  =  1;

    U2STA  =  0;
    U2STAbits.UTXEN  =  1;

    IFS1bits.U2RXIF  =  0;
    _U2RXIE  =  1;
}

  char  cBuf;

void  __attribute__((__interrupt__))  _U2RXInterrupt(void)
{
    IFS1bits.U2RXIF  =  0;
    while(U2STAbits.UTXBF  ==  1);
    cBuf  =  U2RXREG;
    cBuf++;
    cBuf++;
    U2TXREG  =  cBuf;
    PORTA_0  =~PORTA_0;
}

void  main()
{
    TRISA  =  0xff00;
    UART2Init();
    while  (1)
      {
        }
}
```


В этой программе мы определили функцию-обработчик прерывания последовательного порта при приеме байта данных как

```
void  __attribute__((__interrupt__))  _U2RXInterrupt(void)
{
.  .  .
}
```

Обработчик прерывания вначале сбрасывает флаг прерывания приема последовательного порта 2:

```
IFS1bits.U2RXIF  =  0;
```

После этого выполняется проверка буфера передатчика на готовность к передаче следующего байта:

```
while(U2STAbits.UTXBF  ==  1);
```

Конечно, в реальном приложении обработчик прерывания не должен ждать готовности передатчика, поэтому на практике такой код в обработчике прерывания лучше не применять. Если буфер передатчика пуст, то код обработчика читает байт в переменную `cBuf` и увеличивает код байта на 2:

```
cBuf  =  U2RXREG;
cBuf++;
cBuf++;
```

Затем байт `cBuf` записывается в буфер передатчика, а бит 0 порта RA0 инвертируется:

```
U2TXREG  =  cBuf;
PORTA_0  =~PORTA_0;
```

Для того чтобы прерывание по приему байта порта 2 работало, необходимо его разрешить, для чего в уже знакомую нам функцию инициализации асинхронного последовательного порта 2 `UART2Init` добавляется оператор:

```
_U2RXIE  =  1;
```

Перед установкой разрешения прерывания (не только этого, но и любого) флаг прерывания должен быть предварительно сброшен.

Немного об остальной части программы. В начале программы для удобства работы мы определяем бит 0 порта RA0:

```
#define  PORTA_0  PORTAbits.RA0
```

Кроме того, конфигурируем в основной программе выводы 0:7 порта RA0 как выходы:

```
TRISA  =  0xff00;
```

Собственно, это все, что касается программы.

В следующем примере мы увидим, как можно использовать прерывание асинхронного последовательного порта для передачи строки сообщения. Создадим с помощью мастера проектов каркас приложения и добавим в него файл с исходным текстом программы:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define SYSCLK            8000000
#define BAUDRATE2         9600
#define BAUDRATEREG2      SYSCLK/8/BAUDRATE2-1

#define UART2_TX_TRIS     TRISFbits.TRISF5
#define UART2_RX_TRIS     TRISFbits.TRISF4

char s1[] = "1-st TEST String is passed to the Serial Port!";
char s2[] = "2-nd TEST String is passed to the Serial Port!";
char CRLF[] = {0xd, 0xa, 0x0};

static char *ps;

void UART2Init()
{
    UART2_TX_TRIS = 0;
    UART2_RX_TRIS = 1;
    U2BRG = BAUDRATEREG2;
    U2MODE = 0;
    U2MODEbits.BRGH = 1;
    U2MODEbits.UARTEN = 1;

    U2STA = 0;
    U2STAbits.UTXEN = 1;
  }

void __attribute__ ((interrupt)) _U2TXInterrupt(void)
{
  while(U2STAbits.UTXBF == 1);
  if (*ps != '\0')
    {
      U2TXREG = *(ps++);
    }
  else
    {
      _U2TXIF = 0;
    }
}

void main()
{

  _U2TXIF = 0;
  _U2TXIP = 4;
  _U2TXIE = 1;
```



```
  UART2Init();

  ps = s1;
  _U2TXIF = 1;

while(U2STAbits.TRMT != 1);

ps = CRLF;
  _U2TXIF = 1;

while(U2STAbits.TRMT != 1);
ps = s2;
_U2TXIF = 1;

while (1)
  {
  }
}
```

В этой программе, как и в предыдущих, содержатся стандартные установки режима асинхронного обмена данными для тактовой частоты 8 МГц. В программе определены две строки `s1` и `s2`, которые будут с использованием прерывания передаваться в последовательный порт 2. Строка `CRLF` содержит символы «возврат каретки» и «перевод строки». Перед вызовом прерывания в указатель `ps` помещается адрес одной из строк `s1`, `s2` или `CRLF`.

В нашей демонстрационной программе прерывание асинхронного порта вызывается установкой флага прерывания `_U2TXIF` в основной программе. В этом случае вызывается функция-обработчик прерывания `_U2TXInterrupt`, в начале которой выполняется проверка буфера передачи (оператор `while`). Если в буфер можно поместить следующий байт для передачи, то это делается при помощи оператора

```
U2TXREG = *(ps++);
```

При этом следует проверить значение передаваемого байта. Строки `s1` и `s2` являются строками с завершающим нулем (null-terminated string), поэтому проверяется равенство байта значению 0. Если байт строки не равен 0, то он помещается в регистр U2TXREG, при этом указатель `ps` инкрементируется, чтобы указывать на следующий байт. Флаг прерывания `_U2TXIF` не сбрасывается до тех пор, пока не будет передана вся строка — в этом случае после передачи очередного байта прерывание вызывается повторно. По достижении конца строки, т. е. байта со значением 0, флаг прерывания сбрасывается и передача заканчивается.

При передаче данных из буфера в последовательный порт передача байта данных считается законченной только в том случае, если бит TRMT регистра U2STA устанавливается в 1. Это означает, что регистр сдвига передающего модуля микроконтроллера пуст, что в свою очередь свидетельствует об окончании передачи байта данных. В нашей программе передача следующей строки начинается по завершении передачи предыдущей. Интервал времени между помещением байта данных в буфер передачи U2TXREG и завершением его

отправки незначительный, поэтому можно подождать окончания передачи, применив оператор while:

```
while (U2STAbits.TRMT != 1);
```

Обмен данными по последовательному порту можно с успехом использовать при построении иерархических систем управления и контроля. Например, можно передавать данные по интерфейсу RS-232 в другую систему на базе микроконтроллера или в персональный компьютер. Кроме того, можно использовать последовательный порт и в целях отладки и диагностики работающей программы, пересылая данные из различных точек программы в персональный компьютер. В заключение главы приведу еще один пример. В этом приложении система ожидает внешнего прерывания 3 (INT3) и при его возникновении передает сообщение в последовательный порт (**Рис. 10.2**).

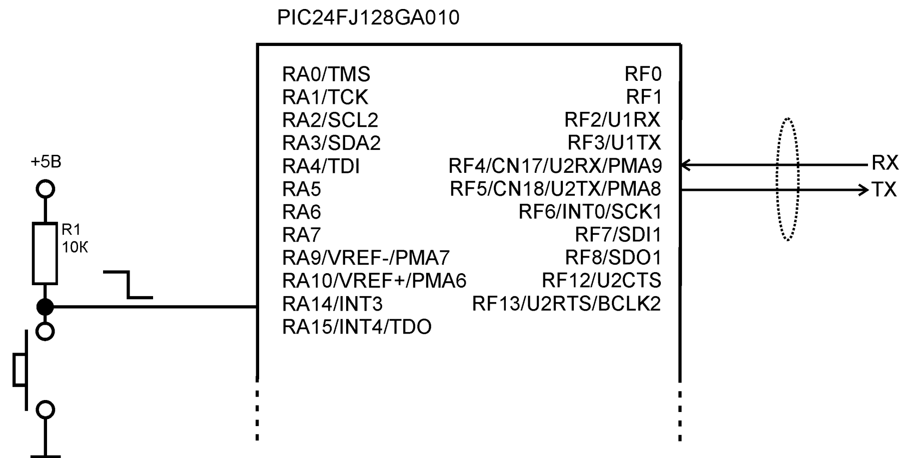

Рис. 10.2. Аппаратная часть проекта

Передача сообщения осуществляется в самом обработчике прерывания. С помощью мастера проектов среды MPLAB IDE создадим каркас приложения и добавим в него файл со следующим исходным текстом:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define SYSCLK            8000000
#define BAUDRATE2         9600
#define BAUDRATEREG2      SYSCLK/8/BAUDRATE2-1

#define UART2_TX_TRIS     TRISFbits.TRISF5
#define UART2_RX_TRIS     TRISFbits.TRISF4

#define INT3port PORTAbits.RA14

char s1[] = "INTERRUPT 3 burnst!";
```



```
char CRLF[] = {0xd, 0xa, 0x0};

static char *ps;

void UART2Init()
{
  UART2_TX_TRIS = 0;
  UART2_RX_TRIS = 1;
  U2BRG = BAUDRATEREG2;

  U2MODE = 0;
  U2MODEbits.BRGH = 1;
  U2MODEbits.UARTEN = 1;

  U2STA = 0;
  U2STAbits.UTXEN = 1;
}

void UART2PutString(char* ps)
{
  while (*ps != '\0')
  {
    while(U2STAbits.UTXBF == 1);
    U2TXREG = *(ps++);
  }
}

void __attribute__ ((interrupt)) _INT3Interrupt(void)
{
  char *ps1 = s1;
  UART2PutString(ps1);
  ps1 = CRLF;
  UART2PutString(ps1);
  _INT3IF = 0;
}

void main()
{
  TRISA = 0xff00;
  INTCON2bits.INT3EP = 1;

  _INT3IF = 0;
  _INT3IP = 4;
  _INT3IE = 1;
  UART2Init();

  while (1)
  {
  }
}
```

В обработчике прерывания _INT3Interrupt строка s1 передается в последовательный порт при помощи функции UART2PutString(ps1). В последней команде обработчика флаг прерывания сбрасывается с помощью оператора

```
_INT3IF = 0;
```

В программе используются те же установки параметров последовательного обмена, что и в предыдущих примерах. Инициализация асинхронного порта 2

осуществляется в основной программе с помощью функции UART2Init. Кроме того, необходимо установить, по какому фронту сигнала на входе INT3 будет генерироваться прерывание. Оператор

```
INTCON2bits.INT3EP = 1;
```

определяет, что прерывание должно генерироваться по спадающему фронту сигнала. Перед установкой разрешения внешнего прерывания 3 сбрасываем флаг прерывания

```
_INT3IF = 0;
```

и устанавливаем приоритет запроса на прерывание:

```
_INT3IP = 4;
```

Здесь в качестве уровня приоритета выбрано значение 4. Наконец, устанавливаем бит разрешения внешнего прерывания 3:

```
_INT3IE = 1;
```

При смене уровня сигнала на входе INT3 с ВЫСОКОГО на НИЗКИЙ будет инициироваться запрос на внешнее прерывание 3, после чего в последовательный порт будет передана строка сообщения s1.

Компилятор MPLAB C для PIC24 содержит ряд специальных 16-битных библиотечных функций, упрощающих работу с периферийными устройствами микроконтроллера PIC24FJ128GA010. Во многих случаях это избавляет разработчика от необходимости написания собственного кода. Для всех библиотечных функций периферийных устройств имеются заголовочные файлы. Например, для функций, работающих с асинхронными портами обмена данных, имеется файл uart.h, для работы с ШИМ имеется файл pwm.h и т. д.

Для работы с 16-битными функциями периферийных устройств в программу необходимо включить соответствующий заголовочный файл. В нашем случае, в начале программы должна присутствовать директива

```
#include <uart.h>
```

Кроме того, в проект нужно включить один из библиотечных файлов: libpPIC24Fxxx-elf.a или libpPIC24Fxxx-coff.a (это зависит от выбранного формата объектного файла). Эти файлы размещаются в каталоге \. . . \MPLAB C30\lib\PIC24F.

Модифицируем предыдущий пример так, чтобы в нем можно было использовать библиотечную функцию putsUART2, записывающую строку байтов, адрес которой передается функции, в последовательный порт 2.

Вот исходный текст программы с учетом сделанных изменений (выделены жирным шрифтом):

```
#include <p24fj128ga010.h>
#include <uart.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define SYSCLK         8000000
```



```
#define BAUDRATE2      9600
#define BAUDRATEREG2   SYSCLK/8/BAUDRATE2-1

#define UART2_TX_TRIS  TRISFbits.TRISF5
#define UART2_RX_TRIS  TRISFbits.TRISF4

#define INT3port  PORTAbits.RA14

char s1[] = "INTERRUPT 3 burnst (16-bit lib functions used)!";
char CRLF[] = {0xd, 0xa, 0x0};

void UART2Init()
{
    UART2_TX_TRIS = 0;
    UART2_RX_TRIS = 1;
    U2BRG = BAUDRATEREG2;

    U2MODE = 0;
    U2MODEbits.BRGH = 1;
    U2MODEbits.UARTEN = 1;

    U2STA = 0;
    U2STAbits.UTXEN = 1;
}

void __attribute__ ((interrupt)) _INT3Interrupt(void)
{
    putsUART2(s1);
    putsUART2(CRLF);
    _INT3IF = 0;
}

void main()
{
    TRISA = 0xff00;
    INTCON2bits.INT3EP = 1;

    _INT3IF = 0;
    _INT3IP = 4;
    _INT3IE = 1;
    UART2Init();
    while (1)
    {
    }
}
```

В числе библиотечных функций для работы с асинхронным портом имеется масса других полезных функций, позволяющих упростить написание программ. Это ни в коей мере не означает, что использование таких функций обеспечит оптимальность программного кода, поэтому в каждом конкретном случае программист сам решает, писать ему собственные функции или воспользоваться библиотечными.

Дополнительную информацию по функционированию асинхронных портов обмена данными можно изучить по фирменным руководствам на конкретный микроконтроллер.

ГЛАВА 11

ОБРАБОТКА АНАЛОГОВЫХ СИГНАЛОВ В МИКРОКОНТРОЛЛЕРАХ PIC24F

Микроконтроллеры семейства PIC24F обладают развитыми возможностями по обработке аналоговых сигналов. Помимо того, что к этим микроконтроллерам, как и к любым другим, можно подключать внешние микросхемы аналого-цифровых преобразователей (АЦП), они уже имеют в своем составе интегрированный модуль 16-канального 10-битного АЦП. Вначале рассмотрим возможности встроенного модуля АЦП. Его упрощенная блок-схема приведена на **Рис. 11.1**.

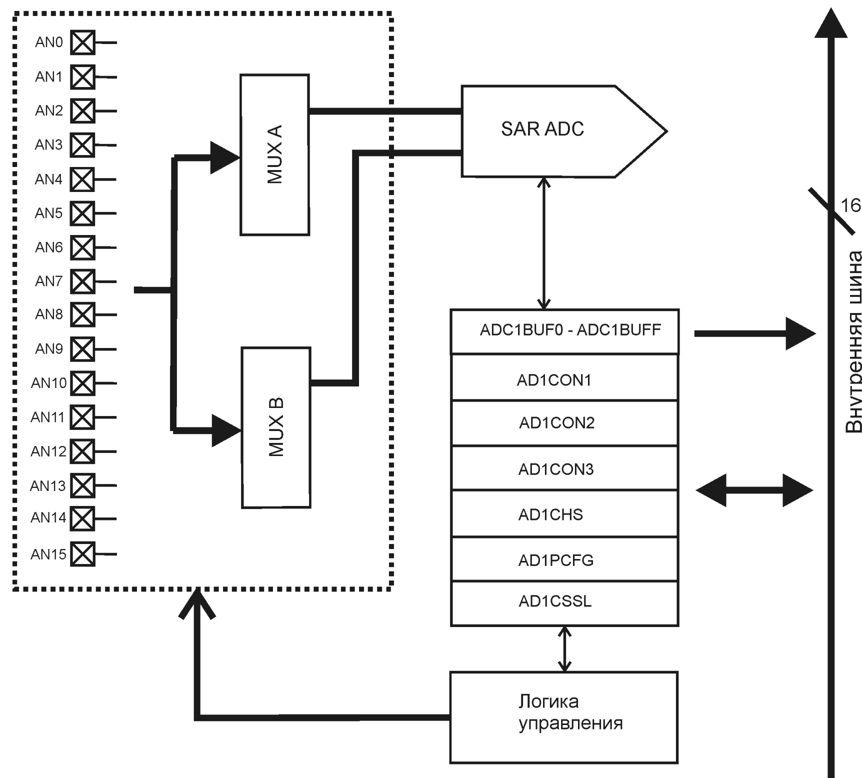

Рис. 11.1. Модуль аналого-цифрового преобразователя

Аналого-цифровой преобразователь микроконтроллера разработан на основе регистра последовательного приближения и позволяет обрабатывать аналоговые сигналы с 16 каналов (обозначаются как AN0...AN15) с использованием двух мультиплексоров MUXA и MUXB (см. **Рис. 11.1**). Результаты преобразований сохраняются в массиве 16-битных слов ADC1BUFx. Установка режимов работы и рабочих параметров преобразователя осуществляется с помощью регистров AD1CON1...AD1CON3, AD1CHS, AD1PCFG и AD1PCFG (мы обсудим их далее).

Аналого-цифровой преобразователь микроконтроллеров PIC24F имеет следующие характеристики:

- разрешающая способность — 10 бит;
- способ преобразования — метод последовательного приближения (Successive Approximation Register, SAR);
- частота преобразования — до 500 тыс. выборок/с;
- количество входных аналоговых каналов — 16;
- возможность подключения внешних источников опорного напряжения;
- униполярный усилитель выборки/хранения входного сигнала;
- возможность сканирования каналов в автоматическом режиме;
- имеется буфер на 16 значений;
- возможность выполнения операций в «спящем» режиме и режиме «холостого хода» микроконтроллера.

Микроконтроллеры семейства PIC24F могут различаться количеством входных каналов, поэтому всегда следует знакомиться с документацией на конкретное устройство. Однако во всех моделях имеются два вывода для подключения опорного напряжения, V_{REF+} и V_{REF-}, а сами варианты выбора источника опорного напряжения можно задавать программным способом. Входы аналоговых сигналов через два независимых мультиплексора (MUXA и MUXB) подключаются к усилителю выборки и хранения аналого-цифрового преобразователя. В преобразователе предусмотрен вариант автоматического сканирования нескольких каналов ввода. Кроме того, процессом преобразования можно управлять как на этапе выборки аналогового сигнала на входе АЦП, так и при выполнении собственно самого преобразования. В преобразователе предусмотрены режимы автовыборки и автопреобразования, что значительно упрощает управление преобразователем и избавляет разработчика от трудоемких расчетов временных характеристик преобразования.

Модуль АЦП можно настроить так, чтобы после каждого полного цикла преобразования генерировалось прерывание, что можно использовать при построении систем реального времени. Данные преобразования сохраняются в виде 16-битных слов во внутреннем буфере.

При написании этой главы я не ставил целью подробное описание всех режимов работы АЦП и использование всех возможностей преобразователя для обработки сигналов. Мы рассмотрим только важнейшие аспекты функционирования АЦП и проиллюстрируем их несложными практическими примерами, что даст возможность научиться быстро и на достаточно хорошем уровне раз-

рабатывать системы обработки аналоговых данных. Освоив основы работы с АЦП микроконтроллеров семейства PIC24F, можно двигаться дальше и изучать обширную документацию, предоставляемую фирмой Microchip.

Процесс преобразования аналогового сигнала в цифровой можно представить в виде последовательности двух фаз или этапов: этапа выборки и этапа собственно преобразования (**Рис. 11.2**).

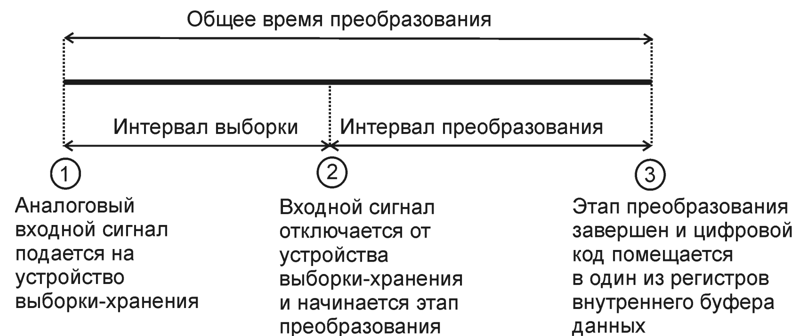

Рис. 11.2. Последовательность выполнения аналого-цифрового преобразования

Преобразование аналогового сигнала в цифровой код начинается с момента подключения одного из входов АЦП к источнику сигнала (1). На программном уровне это подключение выполняется путем установки бита SAMP регистра управления AD1CON1. Устройство выборки-хранения представляет собой матрицу конденсаторов, которые заряжаются напряжением входного сигнала в течение определенного интервала времени (время выборки). Интервал выборки должен быть достаточно продолжительным, чтобы конденсаторы смогли зарядиться до напряжения, соответствующего входному сигналу. Интервал выборки задается программным способом в регистре управления AD1CON3. Вместе с тем, интервал выборки не должен быть слишком большим, поскольку общее время преобразования в этом случае будет увеличиваться. Кроме того, при большом времени выборки может происходить частичная утечка заряда на конденсаторах устройства выборки-хранения, что скажется на точности преобразования. Лучше всего при выборе интервала выборки руководствоваться рекомендациями производителя (это относится не только к данному конкретному случаю, но и к технике аналого-цифрового преобразования в целом).

По завершении этапа выборки источник входного аналогового сигнала отключается от устройства выборки-хранения, и начинается процесс преобразования накопленного на матрице конденсаторов заряда в цифровой код (2). Для данного преобразователя предусмотрен целый ряд режимов, позволяющих управлять процессом преобразования вручную или автоматически. Управление этапом преобразования выполняется путем программирования отдельных битов регистра AD1CON1 (мы рассмотрим это при анализе примеров программного кода).

На этапе преобразования каждый бит результата выдается по отдельному синхронизирующему импульсу. Для 10-битного преобразования требуется 10 импульсов синхронизации плюс два дополнительных импульса, т. е. всего 12 импульсов. Период этих импульсов (обозначается как T_{AD}) выбирается программно путем установки соответствующих битов в регистре AD1CON3. По завершении преобразования 10-битный цифровой код результата помещается в один из 16-битных буферных регистров, которые доступны для чтения (3). Об окончании преобразования свидетельствует либо установленный бит DONE в регистре управления-состояния AD1CON1, либо установленный флаг прерывания АЦП.

В следующем разделе мы ознакомимся с программной моделью аналого-цифрового преобразователя.

11.1. Программная модель интегрированного АЦП

Для программиста модуль аналого-цифрового преобразователя можно представить группой регистров, с помощью которых можно задавать как общие параметры преобразования (интервал выборки, частоту преобразования), так и режимы выполнения самого преобразования (возможность сканирования каналов входных сигналов, автоматического запуска преобразования и т. д.). Думаю, нет смысла подробно описывать абсолютно каждый бит регистров управления и состояния — это все имеется в документации на микроконтроллер. При первоначальном знакомстве с модулем АЦП достаточно понять смысл основных настраиваемых опций — остальные можно оставить по умолчанию. В подавляющем большинстве практических разработок аналого-цифровой преобразователь работает в одном из стандартных режимов, для программирования которого потребуется установка всего нескольких битов в регистрах управления АЦП. Тем не менее, нужно четко себе представлять схемотехнические принципы функционирования АЦП — это позволит создавать самые сложные системы сбора аналоговой информации.

Аналого-цифровой преобразователь имеет 22 регистра, шесть из которых служат для управления АЦП и хранения информации о состоянии устройства. К управляющим регистрам относятся:

- AD1CON1 — регистр управления 1 АЦП;
- AD1CON2 — регистр управления 2 АЦП;
- AD1CON3 — регистр управления 3 АЦП;
- AD1CHS — регистр выбора входного канала;
- AD1PCFG — регистр конфигурации входов АЦП;
- AD1CSSL — регистр настройки сканирования входных каналов.

Регистры AD1CON1...AD1CON3 управляют ходом выполнения преобразования АЦП. С их помощью устанавливается частота преобразования, выбирается конфигурация источников опорного напряжения, а также выполняется программное управление выборкой и преобразованием входного сигнала. Регистр AD1CHS позволяет выбрать входные каналы, которые будут подключены к устройству выборки-хранения сигнала преобразователя, а также мультиплексор для работы с входными сигналами. С помощью регистра AD1PCFG осу-

ществляется настройка выводов микроконтроллера для работы в одном из режимов: либо как аналогового входа, либо как линии ввода/вывода цифровых сигналов. Наконец, регистр AD1CSSL позволяет выбрать каналы входных сигналов для режима сканирования.

Модуль аналого-цифрового преобразователя включает блок из 16-и регистров ADC1BUF. Регистры именуются, начиная с ADC1BUF0 и заканчивая ADC1BUFF. Их содержимое доступно только для чтения.

Перейдем к описанию регистров преобразователя и начнем с регистра управления AD1CON1 (**Табл. 11.1**).

Таблица 11.1. Биты регистра управления и состояния AD1CON1

Позиция бита	Обозначение	Описание
15	ADON	1 – модуль АЦП включен 0 – модуль АЦП отключен
14	–	Не задействован, читается как 0
13	ADSIDL	1 – АЦП прекращает работу в режиме «холостого хода» микроконтроллера 0 – АЦП продолжает работу в режиме «холостого хода»
12	–	Не задействован, читается как 0
11	–	Не задействован, читается как 0
10	–	Не задействован, читается как 0
9–8	FORM1–FORM0	Формат выходных данных (обычно равен 00, что означает беззнаковое целое число)
7–5	SSRC2–SSRC0	Метод инициализации преобразования: 111 – процесс выборки контролируется внутренним счетчиком. По окончании выборки начинается преобразование (автопреобразование); 110 – зарезервирован; 10x – зарезервирован; 100 – зарезервирован; 011 – зарезервирован; 010 – окончание выборки контролируется таймером 3. По окончании выборки немедленно начинается преобразование; 001 – переход в активное состояние сигнала на выводе INT0 завершает преобразование, после чего немедленно начинается преобразование; 000 – очистка бита SAMP приводит к завершению выборки и началу процесса преобразования
4	–	Не задействован, читается как 0
3	–	Не задействован, читается как 0
2	ASAM	1 – процесс выборки начинается сразу после окончания предыдущего преобразования, при этом бит SAMP устанавливается автоматически 0 – выборка сигнала начинается при установке бита SAMP
1	SAMP	Бит разрешения выборки: 1 – устройство выборки хранения начинает выборку сигнала; 0 – устройство выборки находится в режиме хранения сигнала. Если ASAM = 0, то запись 1 в этот бит инициирует начало выборки сигнала. Если SSRC2–SSRC0 = 000, то запись 0 в этот бит приводит к завершению выборки и началу преобразования

Таблица 11.1. Биты регистра управления и состояния AD1CON1 (окончание)

Позиция бита	Обозначение	Описание
0	DONE	Бит состояния преобразования. Установка в 1 свидетельствует об окончании процесса преобразования, установка в 0 означает, что либо преобразование не выполнено по каким-то причинам, либо оно вообще не начиналось

Следующий регистр управления AD1CON2 определяет некоторые аппаратные настройки преобразователя для входных и выходных сигналов. Назначения битов регистра приведены в **Табл. 11.2**.

Таблица 11.2. Биты регистра управления и состояния AD1CON2

Позиция бита	Обозначение	Описание
15 – 13	VCFG2 – VCFG0	Эти биты определяют конфигурацию источника опорного напряжения для АЦП: 000 – в качестве V_{R+} выбирается AV_{DD}, а в качестве V_{R-} – AV_{SS}; 001 – в качестве V_{R+} выбирается источник напряжения, подключенный к выводу V_{REF+}, а V_{REF-} подключается к AV_{SS}; 010 – в качестве V_{R+} выбирается AV_{DD}, а V_{REF-} подключается к AV_{SS}; 011 – в качестве V_{R+} выбирается источник напряжения, подключенный к выводу V_{REF+}, а в качестве V_{REF-} – AV_{SS}; 1xx – в качестве V_{R+} выбирается AV_{DD}, а в качестве V_{R-} – AV_{SS}
12		Зарезервирован, читается как 0
11		Зарезервирован, читается как 0
10	CSCNA	Выбор режима сканирования для канала MUXA: 1 – входы сканируются; 0 – входы не сканируются
9		Зарезервирован, читается как 0
8		Зарезервирован, читается как 0
7	BUFS	Режим записи буфера данных: 1 – АЦП сохраняет результат в ADC1BUF8–ADC1BUFF, а доступ к ADC1BUF0–ADC1BUF7 выполняется вручную; 0 – АЦП сохраняет результат в ADC1BUF0–ADC1BUF7, а доступ к ADC1BUF8–ADC1BUFF выполняется вручную
6		Зарезервирован, читается как 0
5–2	SMPI3–SMPI0	Биты определяют, как часто должно инициироваться прерывание: 1111 – после каждого 16-го преобразования; 1110 – после каждого 15-го преобразования; . . . 0001 – после каждого 2-го преобразования; 0000 – после каждого преобразования
1	BUFM	Выбор режима работы буфера данных: 1 – буфер конфигурируется как две отдельные группы буферов по 8 слов (ADC1BUF0–ADC1BUF7 и ADC1BUF8–ADC1BUFF); 0 – буфер конфигурируется как непрерывный (ADC1BUF0–ADC1BUFF)

Таблица 11.2. Биты регистра управления и состояния AD1CON2 (окончание)

Позиция бита	Обозначение	Описание
0	ALTS	Выбор альтернативного источника входного сигнала: 1 – при первой выборке используется мультиплексор MUXA, затем выполняется переключение между MUXA и MUXB; 0 – всегда используется MUXA

Регистр управления AD1CON3 определяет временные характеристики процесса преобразования. Назначения битов регистра приведены в **Табл. 11.3**.

Таблица 11.3. Биты регистра управления и состояния AD1CON3

Позиция бита	Обозначение	Описание
15	ADRC	Этот бит определяет источник тактового сигнала преобразователя: 0 – используется внутренний RC-генератор; 1 – используется системная тактовая частота
14–13		Зарезервирован, читается как 0
12–8	SAMC4–SAMC0	Биты устанавливают интервал автовыборки входного сигнала: 11111 – 31 T_{AD}; . . . . 00001 – 1 T_{AD}; 00000 – 0 T_{AD} (не рекомендуется)
7–0	ADCS7–ADCS0	Выбор тактовой частоты преобразования: 11111111 – 128 TCY; 11111110 – 127 TCY; . . . . 00000001 – 1 TCY; 00000000 – TCY/2

Регистр управления AD1CHS определяет конфигурацию входных каналов. Назначения битов регистра приведены в **Табл. 11.4**.

Таблица 11.4. Биты регистра управления и состояния AD1CHS

Позиция бита	Обозначение	Описание
15	CH0NB	Этот бит определяет источник отрицательного аналогового входного сигнала для мультиплексора MUXB: 1 – в качестве входа используется AN1; 0 – в качестве входа используется вывод V_{R-}
14–12		Зарезервирован, читается как 0
11–8	SH0SB3–SH0SB0	Биты определяют источник положительного аналогового входного сигнала для мультиплексора MUXB: 1111 – AN15; 1110 – AN14; 1101 – AN13; . . . . 0001 – AN1; 0000 – AN0;

Таблица 11.4. Биты регистра управления и состояния AD1CHS (окончание)

Позиция бита	Обозначение	Описание
7	CH0NA	Этот бит определяет источник отрицательного аналогового входного сигнала для мультиплексора MUXA: 1 – в качестве входа используется AN1; 0 – в качестве входа используется вывод V_{R-}
6–4		Зарезервированы, читаются как 0
7–0	SH0SA3–SH0SA0	Биты определяют источник положительного аналогового входного сигнала для мультиплексора MUXA: 1111 – AN15; 1110 – AN14; 1101 – AN13; . . . . 0001 – AN1; 0000 – AN0

Регистр управления AD1PCFG определяет контакты ввода/вывода, которые будут использоваться для ввода аналоговых сигналов. Назначения битов регистра приведены в **Табл. 11.5**.

Таблица 11.5. Биты регистра управления и состояния AD1PCFG

Позиция бита	Обозначение	Описание
15–0	PCFG15–PCFG0	Этот бит определяет режим работы порта ввода/вывода: 1 – порт сконфигурирован для цифрового ввода/вывода; 0 – порт сконфигурирован как аналоговый

Регистр управления AD1CSSL определяет номера сканируемых аналоговых каналов для мультиплексора MUXA. Назначения битов регистра приведены в **Табл. 11.6**.

Таблица 11.6. Биты регистра управления и состояния AD1CSSL

Позиция бита	Обозначение	Описание
15–0	CSSL15–CSSL0	Биты определяют номера сканируемых каналов аналоговых входных сигналов для MUXA: 1 – соответствующий канал ANxx сканируется; 0 – соответствующий канал ANxx пропускается

11.2. Практическое использование модуля АЦП

Из описания общих принципов функционирования 10-битного аналого-цифрового преобразователя микроконтроллеров PIC24F может показаться, что программировать обработку данных таким модулем очень сложно. На самом деле это не так. При решении практических задач большинство настроек мож-

но оставить такими, какими они задаются по умолчанию, и настроить только несколько ключевых характеристик преобразователя. Как это делается, мы рассмотрим на нескольких практических примерах.

В первом примере микроконтроллер PIC24FJ128GA010 будет преобразовывать в цифровой код аналоговое напряжение, которое подается на вход AN7. Аналоговый сигнал будет преобразовываться каждый раз при переходе напряжения на входе прерывания INT3 из ВЫСОКОГО уровня в НИЗКИЙ. Цифровой код сигнала можно, например, обработать дальше и использовать для генерации сигналов управления или передать его через асинхронный последовательный порт другому устройству или компьютеру. Вот упрощенная схема нашего устройства (**Рис. 11.3**).

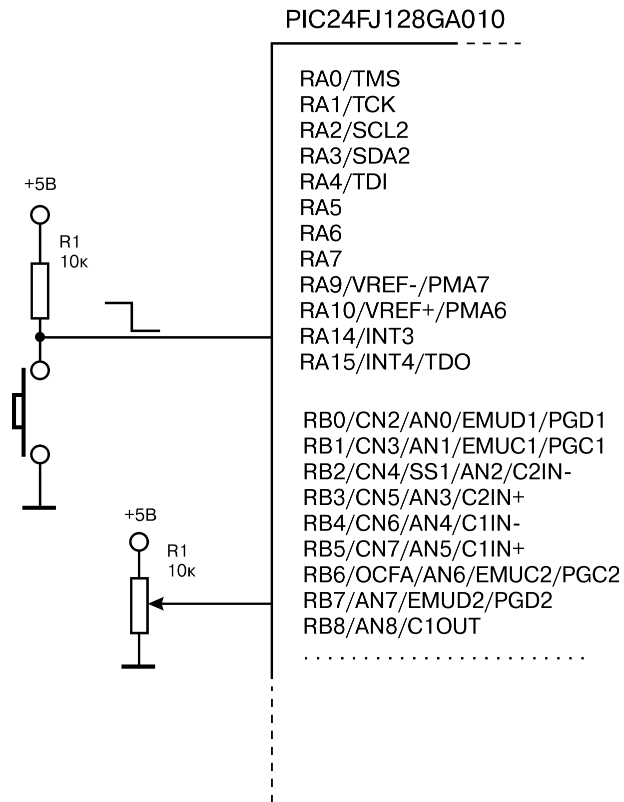

Рис. 11.3. Аппаратная часть проекта

Создадим в MPLAB IDE проект с помощью мастера проектов и добавим в него файл со следующим исходным текстом программы:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)
```



```
#define Channel 7
#define AINPUTS 0x0

unsigned int bReq = 0;

void __attribute__ ((interrupt)) _INT3Interrupt(void)
{
  bReq = 1;
  _INT3IF = 0;
}

void initADC(int adcmask)
{
    AD1PCFG = adcmask;
    AD1CON1 = 0;
    AD1CSSL = 0;
    AD1CON2 = 0;
    AD1CON3 = 0x1F02;

    AD1CON1bits.ADON = 1;
}

int readADC(int channel)
{
    AD1CHS = channel;
    AD1CON1bits.SAMP = 1;

    T1CONbits.TON = 1;
    TMR1 = 0;
    while (TMR1 < 100);

    AD1CON1bits.SAMP = 0;
    while (!AD1CON1bits.DONE);
    return ADC1BUF0;
}

void main()
{
    int a1;
    float res;

    TRISA = 0xff00;
    INTCON2bits.INT3EP = 1;

  _INT3IF = 0;
  _INT3IP = 4;
  _INT3IE = 1;

  initADC(AINPUTS);
  while (1)
  {
    if (bReq == 1)
      {
        a1 = readADC(Channel);
        res = 5.0 / 1024.0 * a1;
```

```
        bReq  =  0;
      }
  }
}
```

Исходный текст программы с первого взгляда может показаться довольно сложным, но мы проанализируем его шаг за шагом, и вскоре смысл операторов станет понятен. Прежде всего, рассмотрим функции, относящиеся к работе аналого-цифрового преобразователя. Это `initADC` и `readADC`. В функции `initADC` выполняется установка основных параметров преобразования. В качестве входного параметра функции передается маска аналоговых входов `adcmask`. Оператор

```
AD1PCFG  =  adcmask;
```

устанавливает в регистре AD1PCFG конфигурацию выводов порта B микроконтроллера, которые используются в качестве аналоговых. В нашем примере все выводы порта являются аналоговыми, что указано в константе `AINPUTS`:

```
#define  AINPUTS  0x0
```

Оператор

```
AD1CON1  =  0;
```

сбрасывает все биты регистра управления 1 АЦП в 0. Помимо всего прочего, такие установки означают, что процесс преобразования должен запускаться вручную. Поскольку наша программа не будет выполнять последовательное сканирование аналоговых каналов ввода, то все биты регистра AD1CSSL тоже сбрасываются в 0:

```
AD1CSSL  =  0;
```

С помощью оператора

```
AD1CON2  =  0;
```

в качестве мультиплексора каналов АЦП выбирается MUXA, а в качестве опорных напряжений V_{ref+} и V_{ref-} выбираются напряжения питания AV_{dd} и земли AV_{ss} соответственно. Далее мы устанавливаем интервал преобразования (T_{samp}) и период тактирования (T_{ad}):

```
AD1CON3  =  0x1F02;
```

В данном случае, при тактовой частоте 8 МГц интервал преобразования выбирается равным $32*T_{ad}$, а период тактирования T_{ad} равным 125 нс. Последний оператор разрешает работу модуля АЦП:

```
AD1CON1bits.ADON  =  1;
```

Собственно преобразование выполняется функцией `readADC`. В качестве параметра функция принимает номер тестируемого канала, который для нашей программы равен 7 и задается константой `Channel`:

```
#define  Channel  7
```


Оператор

```
AD1CHS  =  channel;
```

устанавливает в регистре AD1CHS номер тестируемого канала. После всех этих установок мы можем запустить процесс выборки входного сигнала, что выполняется путем установки бита `SAMP`:

```
AD1CON1bits.SAMP  =  1;
```

Здесь есть очень важный момент. Поскольку преобразование аналогового сигнала контролируется вручную, то необходим определенный интервал времени для того, чтобы конденсатор устройства выборки/хранения АЦП мог полностью зарядиться. В этом случае заряд будет точно соответствовать входному напряжению, и результат преобразования будет достаточно точным. Для задания интервала времени выборки можно воспользоваться Таймером 1. Следующая группа операторов обеспечивает интервал выборки, равный 6.25 мкс (для данной тактовой частоты):

```
T1CONbits.TON  =  1;
TMR1  =  0;
while  (TMR1  <  100);
```

По прошествии этого интервала времени можно начать процесс преобразования, для чего следует сбросить бит `SAMP`:

```
AD1CON1bits.SAMP  =  0;
```

Окончание процесса преобразования определяется по установке бита `DONE` регистра AD1CON:

```
while  (!AD1CON1bits.DONE);
```

Полученный в результате преобразования 16-битный цифровой код помещается в регистр ADC1BUF0, который и возвращается вызывающей программе.

Преобразование аналогового сигнала будет выполняться каждый раз при перепаде напряжения из ВЫСОКОГО уровня в НИЗКИЙ на входе прерывания INT3 микроконтроллера PIC24FJ128GA010, следовательно, в нашу программу мы должны включить код обработчика прерывания INT3. Алгоритм измерения довольно простой: каждый раз при возникновении прерывания функция-обработчик устанавливает переменную `bReq`, значение которой непрерывно проверяется в цикле `while` основной программы. При равенстве `bReq` единице выполняется преобразование, и результат запоминается в переменной `res` в виде числа с плавающей точкой. По окончании преобразования переменная `bReq` сбрасывается, и цикл продолжает выполнение до следующего прерывания.

Обработчик прерывания INT3 предельно прост:

```
void  __attribute__  ((interrupt))  _INT3Interrupt(void)
{
  bReq  =  1;
  _INT3IF  =  0;
}
```

Первая команда устанавливает переменную bReq в 1, а вторая сбрасывает флаг прерывания.

Инициализация прерывания выполняется в основной программе с помощью следующих операторов:

```
TRISA = 0xff00;
INTCON2bits.INT3EP = 1;
_INT3IF = 0;
_INT3IP = 4;
_INT3IE = 1;
```

Бит _INT3EP, установленный в 1, определяет генерацию внешнего прерывания по спаду сигнала на входе INT3. Следующие три оператора сбрасывают флаг прерывания, устанавливают уровень приоритета и разрешают прерывание.

Результат преобразования записывается в целочисленную переменную a1, а окончательный результат вычисляется при помощи следующего оператора:

```
res = 5.0 / 1024.0 * a1;
```

Переменная res будет содержать вещественное число, которое и будет являться результатом преобразования. Здесь выражение 5.0/1024.0 определяет значение младшего значащего бита (LSB) для 10-битного преобразователя ($1024 = 2^{10}$).

Если использовать прерывание АЦП, то результата преобразования можно не дожидаться, а продолжить выполнение программы. Модифицируем исходный текст основной программы из предыдущего примера таким образом, чтобы можно было работать с прерыванием АЦП. Вот исходный текст новой программы:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define Channel 7
#define AINPUTS 0x0

unsigned int bReq = 0;
unsigned int bDone = 0;

void __attribute__ ((interrupt)) _INT3Interrupt(void)
{
  bReq = 1;
  _INT3IF = 0;
}

void __attribute__ ((interrupt)) _ADC1Interrupt(void)
{
  bDone = 1;
  _AD1IF = 0;
}

void initADC(int adcmask)
{
```



```
    AD1PCFG = adcmask;
    AD1CON1 = 0;
    AD1CSSL = 0;
    AD1CON2 = 0;
    AD1CON3 = 0x1F02;

    _AD1IF = 0;
    _AD1IP = 3;
    _AD1IE = 1;

    AD1CON1bits.ADON = 1;
}

void readADC(int channel)
{
    AD1CHS = channel;
    AD1CON1bits.SAMP = 1;

    T1CONbits.TON = 1;
    TMR1 = 0;
    while (TMR1 < 100);

    AD1CON1bits.SAMP = 0;
}

void main()
{
  int a1;
  float res;

  TRISA = 0x0;
  INTCON2bits.INT3EP = 1;

  _INT3IF = 0;
  _INT3IP = 4;
  _INT3IE = 1;

  initADC(AINPUTS);
  while (1)
  {
    if (bReq == 1)
      {
        bReq = 0;
        readADC(Channel);
      }
    if (bDone == 1)
      {
        bDone = 0;
        a1 = ADC1BUF0;
        res = 5.0 / 1024.0 * a1;
      }
  }
}
```

Мы добавили в программу переменную bDone, которая, будучи установленной в 1, указывает на окончание преобразования аналогового сигнала, что

используется основной программой для обработки результата. Установка переменной выполняется в обработчике прерывания модуля АЦП, программный код которого показан далее:

```
void  __attribute__  ((interrupt))  _ADC1Interrupt(void)
{
  bDone  =  1;
  _AD1IF  =  0;
}
```

Кроме установки переменной bDone, в обработчике также сбрасывается и флаг прерывания _AD1IF. Текст функции readADC упростился и теперь выглядит так:

```
void  readADC(int  channel)
{
    AD1CHS  =  channel;
    AD1CON1bits.SAMP  =  1;

    T1CONbits.TON  =  1;
    TMR1  =  0;
    while  (TMR1  <  100);

    AD1CON1bits.SAMP  =  0;
}
```

В этой функции последний оператор запускает процесс преобразования, а окончание преобразования фиксируется в обработчике _ADC1Interrupt. Что же касается изменений в исходном тексте функции initADC, то в нее дополнительно включены три оператора, инициализирующие прерывание АЦП.

Изменения в основной программе читатели без труда смогут проанализировать самостоятельно.

В микроконтроллерах PIC24F предусмотрен очень удобный вариант конфигурирования АЦП, позволяющий обойтись без ручного задания точных временных интервалов, как это мы делали, например, с помощью Таймера 1 в предыдущих двух примерах. Это позволяет в определенной степени автоматизировать процесс аналого-цифрового преобразования и облегчить задачу программисту. Для работы в таком режиме нужно установить биты SSRC2...SSRC0 регистра AD1CON1 в значение 0b111 (0xE). В этом случае временные параметры процесса выборки и преобразования будут определяться внутренним счетчиком, а процесс преобразования будет запускаться автоматически по окончании этапа выборки.

Рассмотрим пример программы, в которой будем использовать такую конфигурацию модуля АЦП. В качестве исходного текста возьмем тот, который мы использовали в первом примере, и модифицируем его. Вот как выглядит текст модифицированной программы:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  Channel  7
```



```
#define  AINPUTS  0x0

unsigned  int  bReq  =  0;

void  __attribute__  ((interrupt))  _INT3Interrupt(void)
{
  bReq  =  1;
  _INT3IF  =  0;
}

void  initADC(int  adcmask)
{
    AD1PCFG  =  adcmask;
    AD1CON1  =  0x00E0;

    AD1CSSL  =  0;
    AD1CON2  =  0;
    AD1CON3  =  0x1F02;

    AD1CON1bits.ADON  =  1;

}

int  readADC(int  channel)
{
    AD1CHS  =  channel;
    AD1CON1bits.SAMP  =  1;

    while  (!AD1CON1bits.DONE);
    return  ADC1BUF0;
}

void  main()
{
    int  a1;
    float  res;

    TRISA  =  0x0;
    INTCON2bits.INT3EP  =  1;

  _INT3IF  =  0;
  _INT3IP  =  4;
  _INT3IE  =  1;


initADC(AINPUTS);
while  (1)
  {
    if  (bReq  ==  1)
      {
        bReq  =  0;
        a1  =  readADC(Channel);
        res  =  5.0  /  1024.0  *  a1;
      }
  }
}
```

Изменения исходного текста программы касаются, в основном, двух функций — initADC и readADC. Оператор

```
AD1CON1 = 0x00E0;
```

в функции initADC устанавливает режим внутренней синхронизации процесса преобразования. По этой причине исходный текст функции readADC также значительно упрощается; фактически для запуска и выполнения полного цикла преобразования достаточно установить бит SAMP, а затем проверить соответствующие флаги (либо DONE, либо флаг прерывания _AD1IF).

При разработке сложных измерительных систем возникает необходимость обработки нескольких аналоговых сигналов. Напомню, что для АЦП можно установить опцию сканирования, позволяющую обрабатывать каналы последовательно. В том случае, если нужно выборочно обработать несколько каналов аналоговых сигналов, это можно реализовать программно. На следующем примере мы посмотрим, как можно обработать три аналоговых сигнала с каналов 1, 2 и 5. Аппаратная часть проекта представлена на **Рис. 11.4**.

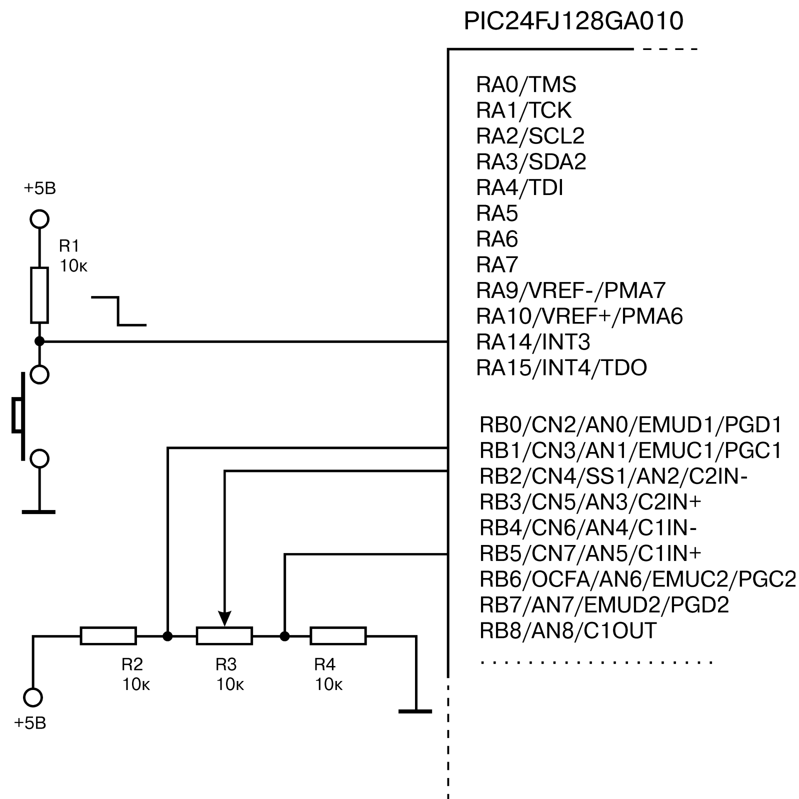

Рис. 11.4. Аппаратная часть проекта обработки аналоговых сигналов 1-го, 2-го и 5-го каналов

Для преобразования аналоговых сигналов выберем полуавтоматическое преобразование с использованием прерывания АЦП. Создадим с помощью мастера проектов среды MPLAB IDE шаблон проекта и включим в него файл со следующим исходным текстом:

```
#include <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)


#define AINPUTS 0x0

int channels[3] = {1, 2, 5};

unsigned int bReq = 0;
unsigned int bDone = 0;

void __attribute__ ((interrupt)) _INT3Interrupt(void)
{
  bReq = 1;
  _INT3IF = 0;
}

void __attribute__ ((interrupt)) _ADC1Interrupt(void)
{
  bDone = 1;
  _AD1IF = 0;
}

void initADC( int adcmask)
{
    AD1PCFG = adcmask;
    AD1CON1 = 0x00E0;

    AD1CSSL = 0;
    AD1CON2 = 0;
    AD1CON3 = 0x1F02;

    _AD1IF = 0;
    _AD1IP = 3;
    _AD1IE = 1;
    AD1CON1bits.ADON = 1;

}

void readADC(int channel)
{
    AD1CHS = channel;
    AD1CON1bits.SAMP = 1;
}

void main()
{
  int a1[3];
  float res[3];
```

```
  int  i1;

  int  *pADC  =  &ADC1BUF0;
  bDone  =  0;

  TRISA  =  0x0;
  INTCON2bits.INT3EP  =  1;

  _INT3IF  =  0;
  _INT3IP  =  4;
  _INT3IE  =  1;

initADC(AINPUTS);
while  (1)
  {
    if  (bReq  ==  1)
      {
        bReq  =  0;
        for  (i1  =  0;  i1  <  3;  i1++)
          {
            readADC(channels[i1]);
            while(bDone  ==  0);
            bDone  =  0;
            a1[i1]  =  *pADC;
            res[i1]  =  5.0  /  1024.0  *  a1[i1];
          }
    }
  }
}
```

За основу этой программы мы взяли программный код из предыдущих проектов, так что большая часть кода нам уже знакома. Остановимся только на изменениях, которые были внесены в эту программу.

Поскольку мы измеряем и преобразуем сигнал трех аналоговых источников, то для хранения результатов нам понадобятся переменные:

```
int  a1[3];
float  res[3];
```

В массиве a1 будут храниться двоичные коды результатов преобразования, а в массиве res — окончательные результаты в виде чисел с плавающей точкой. В указателе pADC будет храниться адрес внутреннего буфера АЦП, в котором сохраняются результаты преобразования. Этот адрес будет равен адресу первого элемента, т. е. адресу регистра ADC1BUF0. Поскольку в программе используется автоматическое преобразование, окончание которого фиксируется установкой флага прерывания _AD1IF, то исходный текст функции readADC существенно упрощается и выглядит так:

```
void  readADC(int  channel)
{
  AD1CHS  =  channel;
  AD1CON1bits.SAMP  =  1;
}
```


В качестве параметра функции readADC будут передаваться номера выбранных каналов из массива channels. Смысл остальных переменных и операторов, думаю, понятен. Преобразование и запись результатов выполняются в цикле while(1) основной программы.

Мы проанализировали только часть возможностей интегрированного 10-битного АЦП. В принципе, возможности модуля позволяют подобрать наилучшую конфигурацию для каждой конкретной системы. Чтобы АЦП обеспечивал высокую достоверность результатов, требуются и довольно серьезные расчеты схемотехнической части устройства. В наших демонстрационных примерах предполагалось, что интегрированный АЦП является идеальным, т. е. не вносит помех и искажений в обрабатываемый сигнал, источники аналоговых сигналов являются идеальными в плане подавления шумов, источник питания дает абсолютно точное напряжение +5 В на выходе, в схеме отсутствуют электромагнитные помехи и т. д. К сожалению, в реальных схемах далеко не все так гладко. Далее в этой главе мы еще вернемся к вопросам схемотехнической (аппаратной) реализации систем с использованием АЦП.

Внутренний АЦП очень удобен и избавляет разработчика от многих проблем, связанных с настройками параметров преобразования и расчетом временных зависимостей, однако разрешения в 10 бит в ряде случаев может оказаться недостаточно. В этом случае можно воспользоваться внешним аналого-цифровым преобразователем. Пример применения такого внешнего 12-битного преобразователя мы рассмотрим в следующем разделе.

11.3. Использование внешнего АЦП

Подключение внешней микросхемы аналого-цифрового преобразователя к микроконтроллеру PIC24F — задача достаточно тривиальная. В нашем следующем проекте мы подключим к порту A аналого-цифровой преобразователь LTC1292 фирмы Linear Technology. Данный преобразователь выполняет передачу данных в микроконтроллер по протоколу SPI и подключается в соответствии со схемой, показанной на **Рис. 11.5**.

Программную часть проекта разработаем по обычной схеме. Создадим в MPLAB IDE новый проект и добавим в него файл со следующим текстом:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  DOUT  PORTAbits.RA0
#define  CLK   PORTAbits.RA1
#define  CS    PORTAbits.RA2

unsigned  int  getBinary(void)
{
    int  cnt;
    unsigned  int  dat  =  0;

    CS  =  1;
```

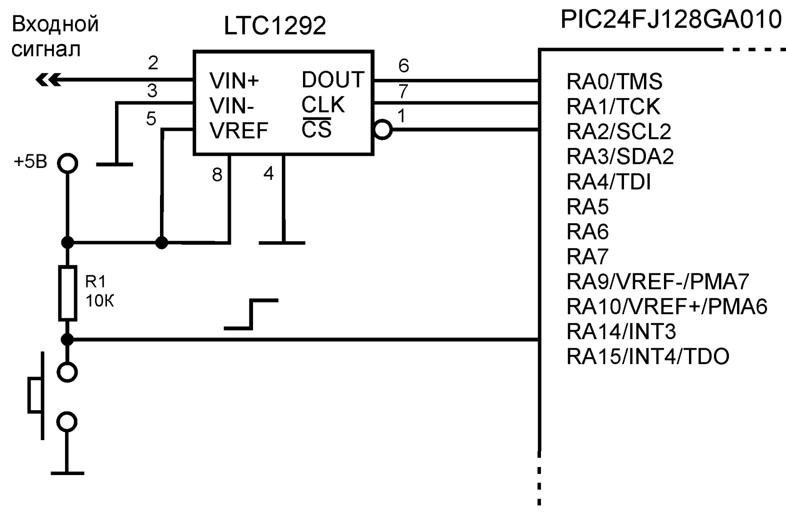

Рис. 11.5. Интерфейс АЦП LTC1292 с микроконтроллером

```
    CLK  =  1;
    CS  =  0;
    CLK  =  0;
    for  (cnt  =  0;  cnt  <  13;  cnt++)
    {
        CLK  =  1;
        CLK  =  0;
        dat  |=  DOUT;
        dat  =  dat  <<  1;
    }
  CS  =  1;
  dat  =  dat  >>  1;
  dat  &=  0xfff;
  return  dat;
}


void  main()
{
  unsigned  int  binData;
  float  res;
  int  temp;

  TRISA  =  0x0f01;
  while  (1)
    {
      if  (_INT3IF  ==  1)
        {
          _INT3IF  =  0;
          binData  =  getBinary();
          res  =  5.0  /  4096.0  *  binData;
          temp  =  res  *  100  —  273.16;
        }
    }
  }
```


Проанализируем текст программы. Собственно преобразование выполняется функцией `getBinary`. Чтобы понять работу функции, нам следует обратиться к временной диаграмме работы микросхемы аналого-цифрового преобразователя LTC1292 (**Рис. 11.6**).

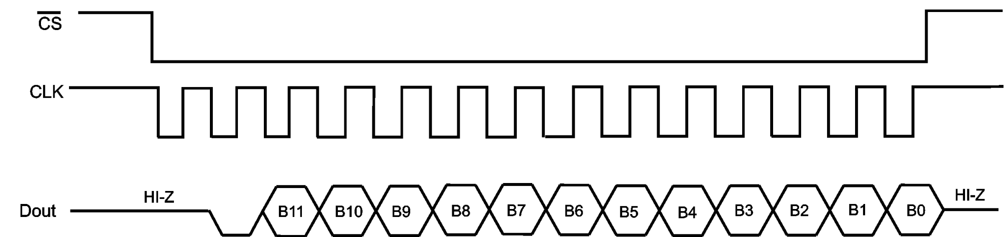

Рис. 11.6. Временная диаграмма работы АЦП LTC1292

Преобразование аналогового сигнала выполняется при НИЗКОМ уровне сигнала CS. После фазы выборки, которая завершается в момент второго спада на линии CLK, начинается фаза преобразования. Данные на линии Dout становятся действительными по спаду синхроимпульсов CLK, а по фронту CLK их можно считывать в микроконтроллер. По окончании преобразования следует установить сигнал CS в ВЫСОКИЙ уровень. Временная диаграмма одного преобразования формируется в функции `getBinary`.

Эта временная диаграмма является типичной для большинства аналого-цифровых преобразователей последовательного приближения, поэтому, поняв принцип работы какого-либо одного устройства, можно успешно применять и другие микросхемы этого типа. Само преобразование запускается при перепаде сигнала на входе прерывания INT3 из НИЗКОГО уровня в ВЫСОКИЙ. В этой программе используется тот факт, что при возникновении запроса на прерывание флаг прерывания `_INT3IF` будет установлен в любом случае, независимо от того, разрешено конкретное прерывание или нет. Двоичный результат преобразования помещается в переменную `binData`, а окончательный результат в формате вещественного числа с плавающей точкой помещается в переменную `res`.

На базе этого проекта можно реализовать несложный измеритель температуры, в котором используется прецизионный аналоговый датчик температуры LM335. Аппаратная часть проекта представлена на **Рис. 11.7**.

В этой схеме используется прецизионный датчик температуры LM335, выходное напряжение которого пропорционально температуре измеряемого объекта или среды с коэффициентом пропорциональности 10 мВ/К°. Например, при температуре 0°С выходное напряжение датчика будет равно 2.73 В. Принципиальная схема, думаю, в объяснениях не нуждается. Что же касается программной части, то практически весь код позаимствован из предыдущей программы, поэтому я остановлюсь только на изменениях. Их два: в основную

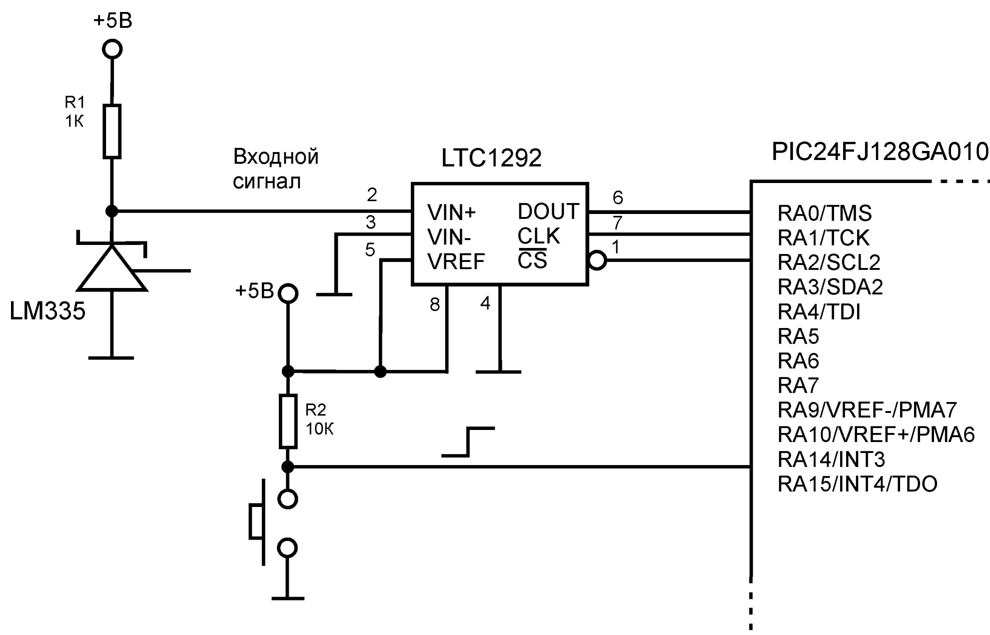

Рис. 11.7. Схема измерителя температуры

программу включена целочисленная переменная `temp`, в которой будет храниться значение температуры, а также добавлен оператор

```
temp = res * 100 — 273.16;
```

В качестве упражнения читатели могут попробовать реализовать такой измеритель температуры на интегрированном АЦП микроконтроллера.

ГЛАВА 12

ГЕНЕРАЦИЯ АНАЛОГОВЫХ И ЦИФРОВЫХ СИГНАЛОВ

В микроконтроллерах PIC24F предусмотрена возможность генерации различных цифровых и аналоговых сигналов, которые можно использовать для управления внешними устройствами. На кристалле микроконтроллеров PIC24F имеется несколько модулей, выполняющих подобные функции: модуль аналоговых компараторов, модуль формирования опорных напряжений аналоговых компараторов, а также модуль генерации цифровых сигналов. Модули, относящиеся к обработке аналоговых сигналов, в процессе функционирования оперируют только с аналоговыми напряжениями и являются, по сути, простейшими преобразователями «цифра—аналог—цифра». С помощью модуля генерации цифровых сигналов формируются последовательности цифровых сигналов на основе временных параметров, задаваемых Таймером 2 или 3. Все перечисленные периферийные модули очень полезны при разработке внешних аналоговых и цифровых интерфейсов, создании различного рода преобразователей сигналов, в измерительных и управляющих схемах. Вначале рассмотрим модуль генерации цифровых сигналов.

12.1. Модуль генерации цифровых сигналов

Модуль генерации цифровых сигналов содержит несколько каналов. Каждый из каналов функционирует следующим образом: в регистры данных (один или оба, в зависимости от выбранного режима работы) заносится значение интервала времени, которое затем будет сравниваться с текущим значением, находящимся в регистре одного из таймеров (Таймера 2 или Таймера 3).

При совпадении значений в регистрах данных и регистре таймера на определенных выходах микроконтроллера будет генерироваться либо одиночный цифровой импульс, либо их последовательность. Параметры импульсов определяются содержимым регистров данных и настройками Таймера 2 или Таймера 3. Характеристики выходных сигналов модуля генерации цифровых сигналов могут варьироваться в очень широком диапазоне, что дает возможность получить импульсные сигналы с различной скважностью и частотой. Такие импульсные последовательности легко преобразовать в аналоговые гармони-

ческие (синус/косинус) сигналы и их комбинации с помощью несложных аналоговых фильтров. Подобная методика используется при синтезе аналоговых сигналов звуковых частот. Как и все периферийные модули микроконтроллера PIC24F, модуль генерации цифровых сигналов представляет собой автономное устройство, которое может настраиваться и функционировать независимо от других модулей. Его можно настроить для работы с использованием прерывания, что позволит разгрузить процессор.

Функциональная схема одного канала модуля показана на **Рис. 12.1**.

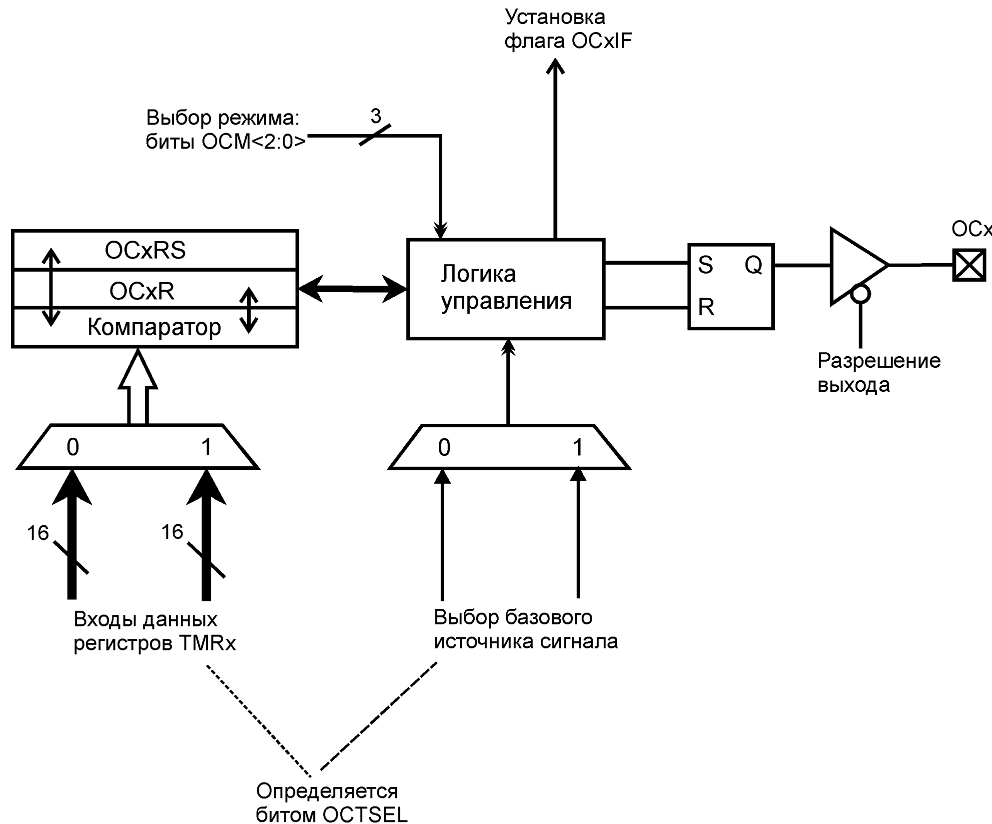

Рис. 12.1. Функциональная схема одного канала модуля генерации цифровых сигналов

Каждый канал модуля содержит два регистра данных – OCxR и OCxRS. В одних режимах работы используется только регистр OCxR, а в других режимах — оба регистра. Например, в режимах однократного сравнения в регистр OCxR помещается значение, которое будет сравниваться с содержимым регистра одного из таймеров. При совпадении значений будет генерироваться единичный импульс или последовательность импульсов. В этом случае длительность такого сигнала будет определяться разностью содержимого регистра OCxR и регистра

периода Таймера 2 или Таймера 3. Таким образом, содержимое регистра OCxR определяет интервал задержки между очередной перезагрузкой таймера и изменением уровня сигнала на выводе OCx.

В режимах двойного сравнения дополнительно задействуется регистр OCxRS, содержимое которого определяет момент второго перепада импульсного сигнала. Фактически содержимое регистров OCxR и OCxRS будет определять длительность импульсного сигнала в этих режимах.

Выбор режима (однократного или двойного сравнения), а также дополнительные характеристики импульсных сигналов определяются битами OCM<2:0> регистра управления модуля OCxCON. Бит 3 (OCTSEL) этого же регистра определяет источник базового сигнала (0 — Таймер 2, 1 — Таймер 3). Количество каналов модуля генератора цифровых сигналов определяется моделью микроконтроллера, поэтому для получения детальной информации нужно обращаться к технической документации на данный микроконтроллер. Конфигурирование прерываний ничем не отличается от настройки прерываний для других модулей, только здесь нужно использовать соответствующие флаги разрешения и приоритета прерываний.

Далее рассмотрим практический пример программирования модуля генерации цифровых сигналов для микроконтроллера PIC24FJ128GA010. Отладку и анализ результатов выполним в среде программирования MPLAB IDE с помощью симулятора MPLAB SIM. Разработаем генератор импульсов, выход которого будет переключаться каждые 3 с, для чего выберем режим однократного сравнения с использованием только регистра OC1R. Выходной сигнал будет генерироваться на выходе OC1 (RD0) микроконтроллера.

Создадим с помощью мастера проектов MPLAB IDE новый проект и включим в него файл со следующим исходным текстом:

```
#include  <p24fj128ga010.h>
#include  <stdio.h>

#define  SYSCLK  8000000
#define  t1      3
#define  PREG2   SYSCLK/2/256*t1

unsigned  int  cnt  =  0;

void  __attribute__  ((__interrupt__))  _OC1Interrupt(void)
{
  _OC1IF  =  0;
printf("OC1  toggled  %d  times\n",  ++cnt);
}

void  main(void)
{

PR2  =  PREG2;
OC1CON  =  0x0000;
OC1CON  =  0x0003;
```

```
OC1R  =  PR2;

IFS0bits.OC1IF  =  0;
IEC0bits.OC1IE  =  1;

TMR2  =  0;
T2CON  =  0x8030;

while(1)
    {
    }
}
```

Анализ исходного текста программы начнем с переменных и констант, объявленных в программе. Константа SYSCLK содержит значение тактовой частоты микроконтроллера (8 МГц). Константа t1 равна 3 и обозначает интервал времени срабатывания таймера, а константа PREG2 определяет значение, которое будет загружено в регистр периода Таймера 2, который мы намерены использовать в этом проекте. Эта константа соответствует времени срабатывания Таймера 2, равному 3 с. В программе объявлена также целочисленная переменная cnt, которая будет равна количеству перепадов импульсного сигнала на выходе OC1.

В программе используется прерывание первого канала модуля с функцией-обработчиком _OC1Interrupt. Эта функция сбрасывает флаг прерывания и выводит содержимое счетчика cnt на «виртуальную» консоль функцией printf. В основной программе мы определяем две настройки: канала 1 модуля генерации цифровых сигналов и Таймера 2. Конфигурирование канала 1 выполняется операторами

```
OC1CON  =  0x0000;
OC1CON  =  0x0003;
OC1R  =  PR2;
```

Первый оператор сбрасывает текущие настройки канала 1 модуля (эту процедуру желательно выполнять каждый раз при перенастройке каналов модуля генерации цифровых сигналов). В регистр управления OC1CON заносится значение 3, т. е. выбирается режим переключения выхода OC1 при каждом срабатывании Таймера 2, который выбран в качестве базового для этой программы. Переключение выхода OC1 должно выполняться всякий раз при перезагрузке Таймера 2, поэтому в регистр данных OC1R помещаем значение регистра периода PR2 Таймера 2.

Настройка Таймера 2 выполняется стандартным образом. Вначале в регистр периода PR2 помещается значение перезагрузки (оно определяется константой PREG2), регистр Таймера 2 TMR2 обнуляется, а в регистр управления T2CON заносится значение 0x8030. Коэффициент деления частоты тактового сигнала Таймера 2 здесь выбран равным 256.

Операторы

```
IFS0bits.OC1IF  =  0;
IEC0bits.OC1IE  =  1;
```


выполняют инициализацию прерывания канала 1 модуля, для чего вначале сбрасывается флаг прерывания OC1IF, а затем устанавливается флаг разрешения прерывания OC1IE.

Для отладки нашей программы скомпилируем ее в режиме **Debug**, затем выберем отладчик MPLAB SIM и выполним некоторые дополнительные действия. В окне настройки отладчика установим значение тактовой частоты устройства, равное 8 МГц (**Рис. 12.2**).

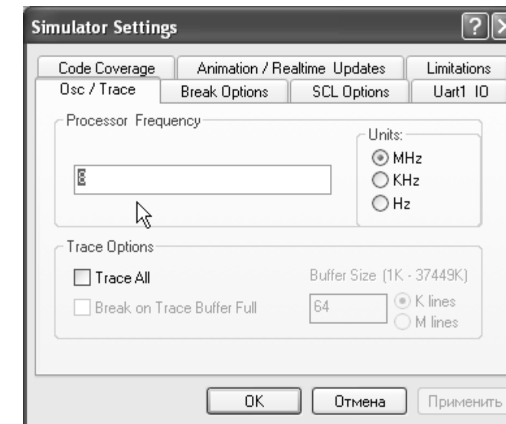

Рис. 12.2. Установка тактовой частоты

Кроме того, установим опцию вывода данных в «виртуальное» окно отладчика, для чего выберем закладку **UART1 IO** и установим опции так, как показано на **Рис. 12.3**.

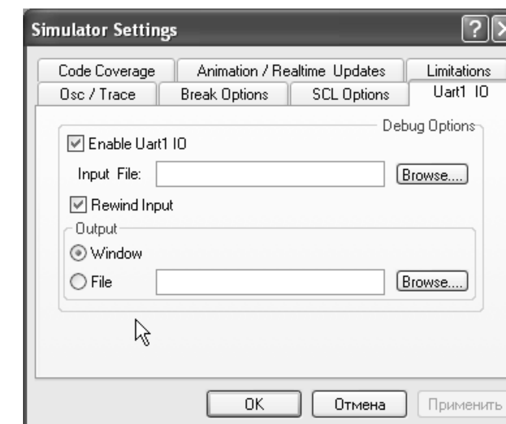

Рис. 12.3. Установка опций вывода в «виртуальное» окно

Напомню, что установка опции «виртуального» вывода позволит выводить данные программы в окно **Output** при отладке. И последняя операция — в настройках проекта нужно выбрать закладку MPLAB LINK30 и выделить память для «кучи», указав объем выделяемой памяти в окне **Heap size** (**Рис. 12.4**). Эта настройка необходима для использования библиотечных функций ANSI C.

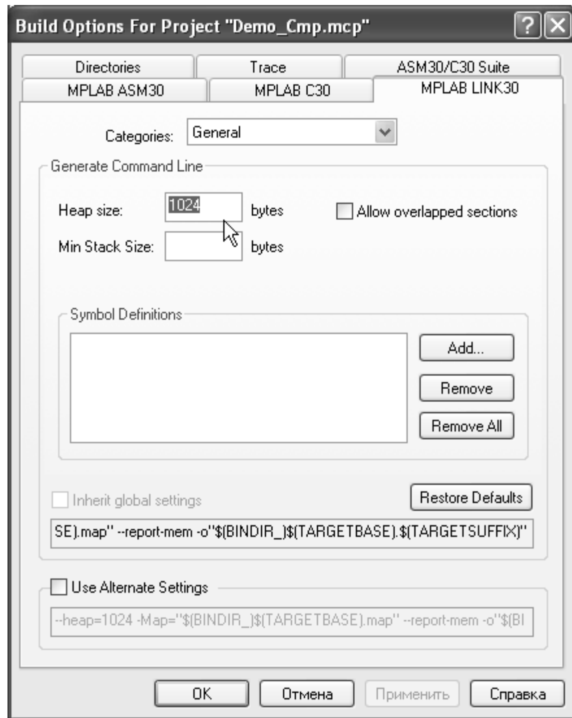

Рис. 12.4. Выделение памяти для «кучи»

Для отладки приложения в окно просмотра переменных нужно вывести переменные TMR2, PORTD (бит 0 соответствует выводу OC1) и `cnt` (**Рис. 12.5**).

После выполнения этих настроек можно запустить программу в режиме **Animate** и наблюдать за изменениями значений переменных в окне **Watch**.

Функционирование канала 1 модуля генерации цифровых сигналов в режиме двойного сравнения я покажу на следующем примере. Для этого разработаем проект, аппаратная часть которого показана на **Рис. 12.6**.

Схема работает следующим образом: на выводе OC1 микроконтроллера каждые 3 с появляется короткий импульс положительной полярности, который переключает защелку DD1 в противоположное состояние. В результате светодиод D1 будет периодически включаться/выключаться. Длительность импульса будет определяться программными настройками канала 1. Микроконтроллер работает на тактовой частоте 8 МГц (частота выполнения команд равна 4 МГц).

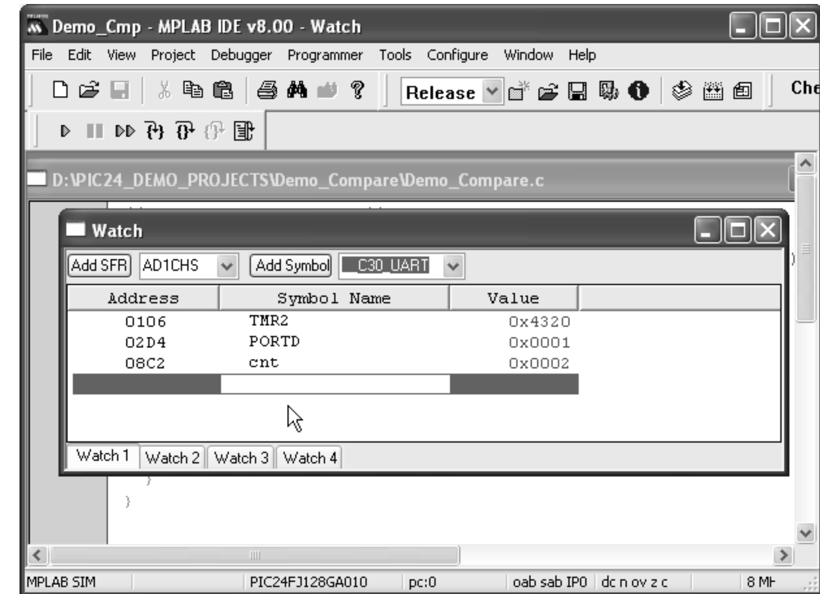

Рис. 12.5. Окно просмотра переменных

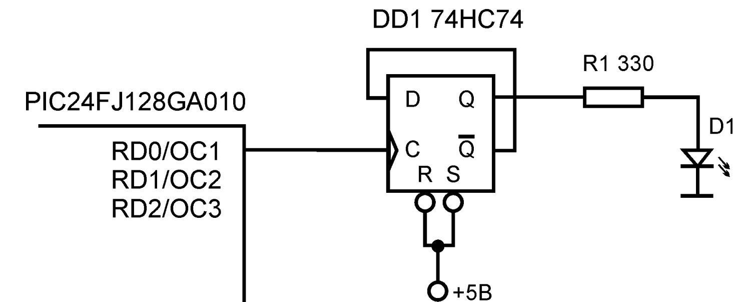

Рис. 12.6. Аппаратная часть проекта

Программная часть проекта разработана в среде MPLAB IDE и содержит файл программы со следующим исходным текстом:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  SYSCLK  8000000
#define  t1      3
#define  PREG2   SYSCLK/2/256*t1

void  __attribute__  ((__interrupt__))  _OC1Interrupt(void)
{
  _OC1IF  =  0;
```

```
}

void  main(void)
{

  OC1CON  =  0x0000;
  OC1CON  =  0x0005;
  OC1R  =  3000;
  OC1RS  =  10000;

  _OC1IF  =  0;
  _OC1IE  =  1;

  PR2  =  PREG2;
  TMR2  =  0;
  T2CON  =  0x8030;

  while(1)
    {
    }
}
```

Здесь используются те же константы, что и в предыдущей программе, поэтому останавливаться на них мы не будем. В качестве базового таймера для канала 1 модуля генерации цифровых сигналов используется Таймер 2. Настройки Таймера 2 соответствуют интервалу его перезагрузки, равному 3 с. В течение этого интервала времени на выводе OC1 будет генерироваться импульс, длительность которого определяется разностью содержимого регистров данных OC1RS и OC1R. Регистр OC1R определяет смещение фронта импульса относительно момента перезагрузки Таймера 1.

Как только содержимое регистра таймера TMR2 превысит содержимое OC1R, сигнал на выводе OC1 перейдет в состояние ВЫСОКОГО уровня. Этот уровень будет удерживаться до того момента, пока содержимое TMR2 не превысит значение во втором регистре данных OC1RS. После этого сигнал на выводе OC1 перейдет в состояние НИЗКОГО уровня. Таким образом, в каждом периоде перезагрузки Таймера 2 на выводе OC1 будет генерироваться импульсный сигнал, положительный фронт которого будет переключать защелку DD1.

Настройка режима двойного сравнения выполняется операторами

```
OC1CON  =  0x0000;
OC1CON  =  0x0005;
OC1R  =  3000;
OC1RS  =  10000;
```

Как обычно, перед настройкой нового режима канал 1 следует отключить, что и выполняет первый оператор. Затем в регистр управления канала 1 модуля записывается число 5, которое и определяет выбранный режим работы. Бит 3 регистра OC1CON сброшен в 0, поэтому в качестве базового источника будет использоваться Таймер 2.

В регистр OC1R записывается значение смещения относительно начала перезагрузки таймера, которое определяет момент формирования нарастающего фронта импульса на выходе OC1. Содержимое второго регистра данных OC1RS определяет момент формирования спадающего фронта импульса. Оба значения в данном примере выбраны произвольно, поэтому читатели могут при желании поэкспериментировать с этой программой, записывая в регистры данных различные значения и наблюдая результат. Функция-обработчик прерывания канала 1 модуля `_OC1Interrupt` используется для сброса флага прерывания _OC1IF, установленного после перепада 1—0 импульса.

Из прямоугольной последовательности импульсов можно синтезировать непрерывный аналоговый сигнал требуемой частоты. Более того, поскольку микроконтроллеры PIC24F включают, как правило, несколько каналов в одном модуле генерации цифровых сигналов, то можно получить непрерывный аналоговый сигнал, используя микширование различных сигналов в смесителях. В простейшем случае для получения, например, синусоидального сигнала частотой 1000 Гц на выходе OC1 можно установить полосовой фильтр со средней частотой 1000 Гц (**Рис. 12.7**).

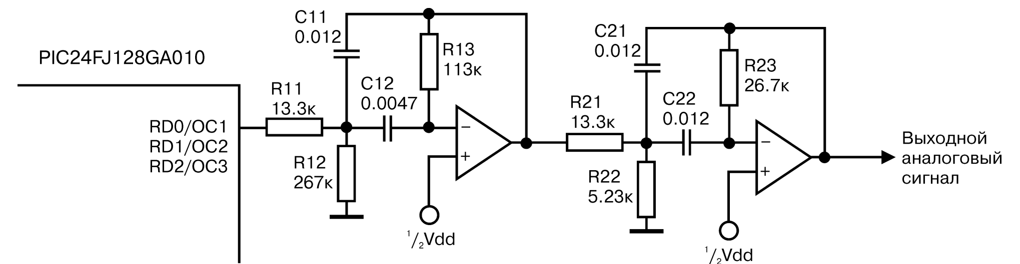

Рис. 12.7. Схема формирования синусоидального сигнала частотой 1000 Гц

В этой схеме используется аналоговый полосовой фильтр Баттерворта 2-го порядка с центральной частотой 1000 Гц и коэффициентом усиления сигнала, равным 1. Входной импульсный сигнал фильтра поступает с выхода OC1 микроконтроллера PIC24FJ128GA010. Операционный усилитель для такого фильтра должен иметь высокую скорость нарастания входного сигнала (slew rate), чтобы избежать появления нелинейных искажений. Используя несколько таких полосовых фильтров для разных выходных сигналов и смеситель, можно получить непрерывный сигнал произвольной формы и частоты.

12.2. Аналоговые компараторы в микроконтроллерах PIC24F

Модуль компараторов аналоговых сигналов и модуль формирования опорного напряжения микроконтроллеров PIC24F часто используются при создании аналого-цифровых интерфейсов различных измерительных и управляющих схем. Модуль формирования опорного напряжения обычно используется в качестве источника смещения (опорного напряжения) для модуля аналоговых

компараторов, хотя его вполне успешно можно применить в качестве управляемого цифро-аналогового преобразователя.

Функциональная схема модуля формирования опорного напряжения показана на **Рис. 12.8**.

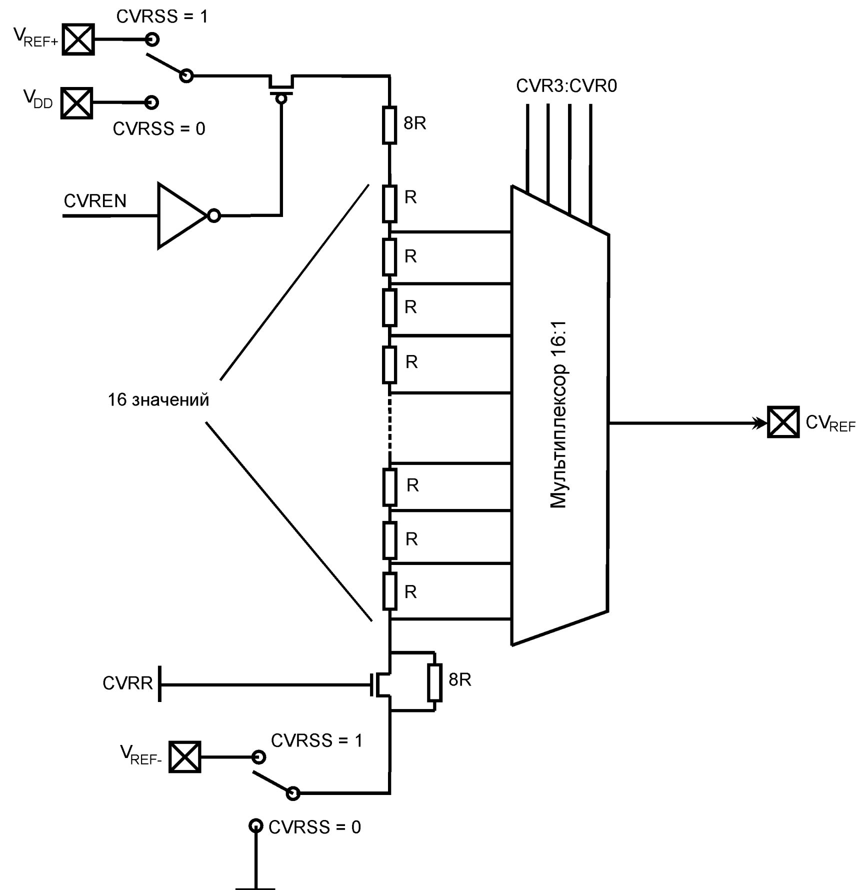

Рис. 12.8. Модуль формирования опорного напряжения

Вкратце рассмотрим работу этого модуля. На схеме обозначены биты регистра управления CVRCON данного модуля, соответствующие тем или иным установкам. Бит 4 (CVRSS) регистра CVRCON определяет источник входного напряжения для модуля формирования опорного напряжения. Если бит установлен, то используется внешний источник напряжения, если же CVRSS = 0, то используется напряжение питания микроконтроллера.

Бит 7 (CVREN) включает (1) или отключает (0) модуль. Бит 5 (CVRR) определяет диапазон изменения и шаг выходного напряжения модуля. Если CVRR = 1, то выходное напряжение изменяется от 0 до значения $0.75CV_{RSRC}$,

при этом шаг составляет $CV_{RSRC}/24$ (здесь CV_{RSRC} — выбранное напряжение питания модуля). При CVRR = 0 выходное напряжение модуля формирования опорного напряжения изменяется от 0.25CV до $0.75CV_{RSRC}$ с шагом $CV_{RSRC}/32$.

Биты 3:0 (CVR3:CVR0) регистра управления CVRCON определяют выходное напряжение модуля. При этом значение выходного напряжения определяется выбранным с помощью бита CVRR рабочим диапазоном.

Если CVRR = 1, то напряжение на выходе модуля CV_{REF} определяется по формуле

$CV_{REF} = (CVR\langle 3:0\rangle/24) \times CV_{RSRC}$.

Если бит CVRR = 0, то выходное напряжение вычисляется следующим образом:

$CV_{REF} = 0.25CV_{RSRC} + (CVR\langle 3:0\rangle/32) \times CV_{RSRC}$.

Кроме этих битов, на функционирование модуля формирования опорного напряжения оказывает влияние бит 6 (CVROE) регистра CVRCON. Если он установлен, то выходное напряжение появляется на выводе CV_{REF} микроконтроллера, если сброшен — выходной каскад вывода CV_{REF} отключается от схем формирования.

Далее рассмотрим пример применения модуля формирования опорного напряжения в простой схеме контроля температуры на микроконтроллере PIC24FJ128GA010. Аппаратная часть проекта показана на **Рис. 12.9**.

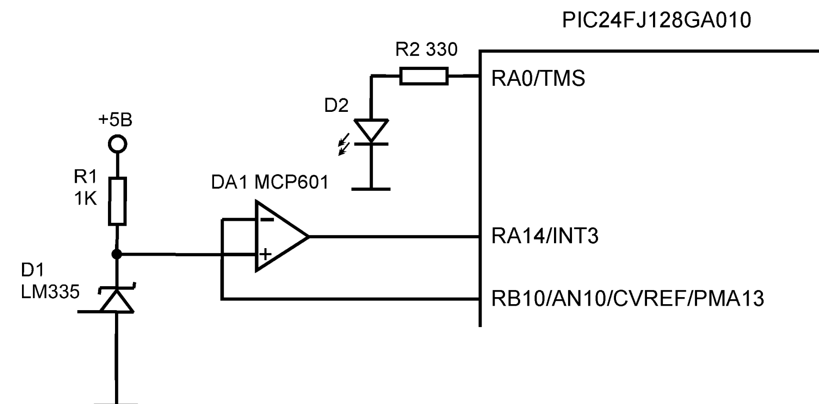

Рис. 12.9. Аппаратная часть проекта

В этой схеме сигнал с датчика температуры LM335 подается на неинвертирующий вход операционного усилителя DA1, который здесь работает в качестве компаратора. На инвертирующий вход DA1 подается напряжение смещения с выхода модуля формирования опорного напряжения CVREF, соответствующее напряжению на выходе датчика температуры при температуре окружающей среды приблизительно 25°C. Величина напряжения на выходе CVREF устанавливается программно. Так, для переключения компаратора при темпе-

ратуре, превышающей 25°C, напряжение CVREF должно быть равным примерно 2.98 В.

При превышении температурой этого значения напряжение на выходе датчика D1 станет выше напряжения на инвертирующем входе и на выходе компаратора DA1 появится ВЫСОКИЙ уровень, что приведет к генерации прерывания INT3.

Программная часть проекта разработана в среде MPLAB IDE и включает файл со следующим исходным текстом:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

#define  SYSCLK  8000000
#define  t1      3
#define  PRG1    SYSCLK/2/256*t1

unsigned  long  del  =  0;

void  __attribute__  ((__interrupt__))  _INT3Interrupt(void)
{
  _INT3IF  =  0;
  _RA0  =  1;
  del  =  0;
  TMR1  =  0;
  T1CON  =  0x8030;
}

void  __attribute__  ((__interrupt__))  _T1Interrupt(void)
{
  _T1IF  =  0;
  del++;
  if  (del  ==  5)
    {
      _RA0  =  0;
      T1CON  =  0x0;
    }
}

void  main(void)
{
  TRISA  =  0xfffe;
  _RA0  =  0;

  _INT3IF  =  0;
  _INT3IE  =  1;

  _T1IF  =  0;
  _T1IP  =  1;
  _T1IE  =  5;

  PR1  =  PRG1;

  CVRCON  =  0;
```



```
  CVRCONbits.CVREN  =  1;
  CVRCONbits.CVROE  =  1;
  CVRCON  |=  0x000B;

  while(1)
      {
      }
  }
```

В процессе функционирования программа обрабатывает два прерывания: внешнее прерывание INT3 и прерывание Таймера 1. Прерывание INT3 вызывается переходом сигнала на выходе компаратора DA1 из НИЗКОГО уровня в ВЫСОКИЙ. Функция-обработчик прерывания `_INT3Interrupt` включает светодиод на выводе 0 порта A и запускает Таймер 1, который предназначен для отсчета времени (15 с) нахождения светодиода во включенном состоянии. По прошествии этого интервала светодиод выключается, а Таймер 1 отключается. Формирование задержки и отключение Таймера 1 выполняется в обработчике прерывания `_T1Interrupt` Таймера 1. Временные параметры рассчитаны для тактовой частоты микроконтроллера, равной 8 МГц.

Конфигурирование модуля формирования опорного напряжения выполняется в основной программе. Оператор

```
CVRCONbits.CVREN  =  1;
```

включает модуль (см. **Рис. 12.8**), а оператор

```
CVRCONbits.CVROE  =  1;
```

подключает выход модуля к выводу CVREF микроконтроллера. Кроме того, по умолчанию, в качестве источника питания модуля используется источник питания устройства (бит CVRSS равен 0) и выбирается диапазон выходных напряжений с начальным значением 0.25CVRSRC (бит CVRR = 0). Чтобы при таких установках на выходе CVREF появилось напряжение порядка 2.98 В, необходимо в младшие 4 бита регистра управления CVRCON записать код, соответствующий аналоговому напряжению 2.98 – 1.25 = 1.72 В. Это соответствует десятичному значению 11 или шестнадцатеричному 0x000B. Это значение следует записать в регистр CVRCON, что выполняется с помощью оператора

```
CVRCON  |=  0x000B;
```

Инициализация прерывания INT3 выполняется операторами

```
_INT3IF  =  0;
_INT3IE  =  1;
```

а инициализация прерывания Таймера 1 — с помощью операторов

```
_T1IF  =  0;
_T1IP  =  1;
_T1IE  =  5;
```

Запуск Таймера 1 осуществляется в функции-обработчике прерывания INT3 обычным способом с использованием операторов

```
TMR1  =  0;
T1CON  =  0x8030;
```

Загрузку регистра периода Таймера 1 требуемым значением мы выполняем в основной программе. Перейдем к анализу работы модуля аналоговых компараторов. В микроконтроллере PIC24FJ128GA010 реализовано два канала аналоговых компараторов (**Рис. 12.10**).

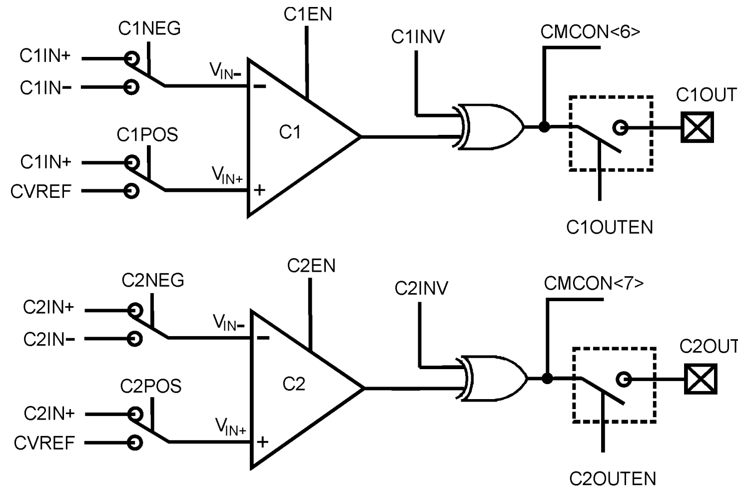

Рис. 12.10. Функциональная схема двухканального модуля аналоговых компараторов

Оба канала модуля содержат аппаратно реализованные компараторы (C1 и C2), которые настраиваются с помощью одного регистра управления CMCON. Настройка каждого из компараторов заключается в выборе нескольких опций, которые устанавливаются с помощью соответствующих битов регистра управления и состояния CMCON. Вот назначение отдельных битов этого регистра:

- биты C1EVT и C2EVT (12, 13) — устанавливаются при изменении состояния (переключении) компаратора C1 или C2 соответственно. Биты доступны для чтения и могут быть сброшены программно;
- биты C1EN и C2EN (10, 11) — установка этих битов разрешает работу компараторов C1 или C2 соответственно, сброс бита запрещает работу соответствующего канала модуля;
- биты C1OUTEN и C2OUTEN (8, 9) — установка этих битов разрешает прохождение выходных сигналов компараторов C1 и C2 на соответствующие выводы микроконтроллера;
- биты C1OUT и C2OUT (6, 7) — доступны для чтения и отражают состояние выходов компараторов C1 и C2 соответственно. Значение этих битов определяется установками битов C1INV и C2INV;
- биты C1INV и C2INV (4, 5) — установка этих битов приводит к инверсии сигнала на выходах компараторов C1 и C2 соответственно. Если биты сброшены в 0, то при условии $V_{IN+} > V_{IN-}$ на выходе компараторов будет ВЫСОКИЙ уровень. Если биты установлены в 1, то при этих же условиях на выходе компараторов будет НИЗКИЙ уровень;
- биты C2POS и C2NEG (2, 3) — определяют источники входных сигналов для компаратора C2. Если C2POS = 1, то вход V_{IN+} компаратора C2 подключается к выводу C2IN+, если же C2POS = 0, то вход V_{IN+} подключается к выходу модуля формирования опорного напряжения CVREF. Если C2NEG = 1, то вход V_{IN-} компаратора C2 подключается к выводу C2IN+, если же C2NEG = 0, то вход V_{IN-} компаратора C2 подключается к выводу C2IN− микроконтроллера;
- биты C1POS и C1NEG (0,1) — определяют источники входных сигналов для компаратора C1. Если C1POS = 1, то вход V_{IN+} компаратора C1 подключается к выводу C1IN+, если же C1POS = 0, то вход V_{IN+} подключается к выходу модуля формирования опорного напряжения CVREF. Если C1NEG = 1, то вход V_{IN-} компаратора C1 подключается к выводу C1IN+, если же C1NEG = 0, то вход V_{IN-} компаратора C1 подключается к выводу C1IN− микроконтроллера.

Для определения момента переключения компараторов и выполнения соответствующих действий можно задействовать прерывание компаратора, поскольку при изменении состояния выходного сигнала любого из компараторов устанавливается флаг прерывания CMIF (бит 2 регистра IFS1 микроконтроллера). Функция-обработчик прерывания компараторов может прочитать биты C1EVT и C2EVT, чтобы определить, какие именно изменения состояния произошли. Даже если бит разрешения прерывания компараторов CMIE (бит 2 регистра IEC1 микроконтроллера) сброшен, т. е. прерывание запрещено, флаг прерывания CMIF все равно будет устанавливаться при изменениях состояний компараторов.

Рассмотрим применение канала 1 модуля аналоговых компараторов на примере схемы сигнализации превышения температуры с датчиком LM335. Аппаратная часть проекта показана на **Рис. 12.11**.

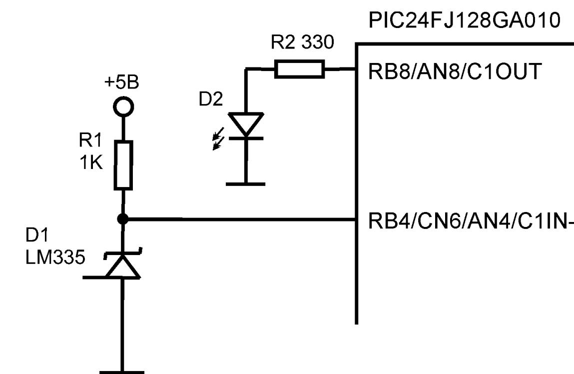

Рис. 12.11. Аппаратная часть проекта

Здесь сигнал с датчика температуры D1 поступает на вход C1IN– модуля аналогового компаратора 1. При превышении температуры среды 54°C сработает внутренний компаратор C1 и на его выходе установится ВЫСОКИЙ уровень, который, поступая на вывод C1OUT микроконтроллера, включит светодиод.

Программно вход C1IN– подключается ко входу V_{IN-} компаратора C1, а на вход V_{IN+} подается напряжение CVREF с модуля формирования опорного напряжения. При таких настройках для получения ВЫСОКОГО уровня на выходе компаратора C1 при превышении порогового значения необходимо инвертировать выход установкой бита C1INV.

Программная часть проекта разработана в среде MPLAB IDE. Исходный текст программы приводится далее:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)


void  main(void)
{

  CVRCON  =  0;
  CVRCONbits.CVREN  =  1;
  CVRCONbits.CVROE  =  1;
  CVRCON  |=  0x000D;

  CMCON  =  0x0F10;

  while(1)
      {
      }
  }
```

Программа очень проста. Поскольку на вход V_{IN+} компаратора 1 мы подаем опорное напряжение CVREF с выхода модуля формирования опорного напряжения, то первые четыре оператора программы выполняют соответствующие настройки. Для получения напряжения CVREF, равного приблизительно 3.278 В (это соответствует температуре датчика 54°C) при напряжении питания +5 В, в младшие 4 бита регистра управления CVRCON записывается значение 0xD:

```
CVRCON  |=  0x000D;
```

Настройка компаратора 1 выполняется в регистре управления CMCON с помощью оператора

```
CMCON  =  0x0F10;
```

Обратите внимание, что для инвертирования выходного сигнала компаратора C1 бит 4 регистра CMCON (C1INV) установлен.

Для демонстрации работы прерывания модуля аналогового компаратора немного модифицируем наш проект. Для сигнализации превышения температуры будем

использовать бит 0 порта A (вывод RA0 микроконтроллера PIC24FJ128GA010), к которому подключим светодиод, а схему подключения датчика оставим прежней (**Рис. 12.12**).

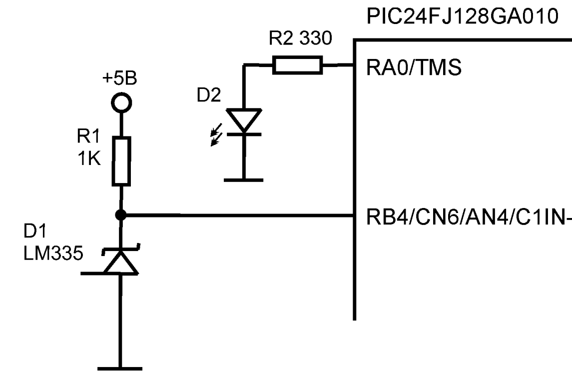

Рис. 12.12. Аппаратная часть проекта

Исходный текст программы на языке Си для данного проекта показан ниже:

```
#include  <p24fj128ga010.h>

_CONFIG2(FCKSM_CSDCMD&OSCIOFNC_ON&POSCMOD_HS&FNOSC_PRI)

void  __attribute__  ((__interrupt__))  _CompInterrupt(void)
{
  _CMIF  =  0;
  _C1EVT  =  0;
  _RA0  =  CMCONbits.C1OUT;
}

void  main(void)
{
  TRISA  =  0xfffe;
  _RA0  =  0;

  CVRCON  =  0;
  CVRCONbits.CVREN  =  1;
  CVRCONbits.CVROE  =  1;
  CVRCON  |=  0x000D;

  CMCON  =  0x0F10;
  _CMIF  =  0;
  _CMIP  =  5;
  _CMIE  =  1;

  while(1)
      {
      }
  }
```

Большая часть исходного текста программы нам уже знакома, поэтому рассмотрим только внесенные изменения. Поскольку компаратор 1 модуля теперь работает в режиме прерывания, то в нашей программе объявлена функция-обработчик прерывания с предопределенным в файле p24fj128ga010.gld именем `_CompInterrupt`. Функция сбрасывает флаг прерывания _CMIF и флаг события _C1EVT. В защелку 0 порта А записывается значение флага состояния C1OUT компаратора 1, при этом на вывод RA0 подается соответствующий уровень.

Операторы

```
_CMIF  =  0;
_CMIP  =  5;
_CMIE  =  1;
```

позволяют настроить параметры прерывания модуля аналоговых компараторов, при этом прерыванию присваивается приоритет 5. Последний оператор разрешает работу прерывания. Для функционирования вывода 0 порта А в режиме выхода защелка TRIS этого вывода сбрасывается в 0.

Мы рассмотрели только часть функций, которые могут выполняться модулями компараторов. Обширная информация по функциям этих модулей и их применению содержится в документации фирмы Microchip.

ЗАКЛЮЧЕНИЕ

В этой книге автор старался охватить как можно более широкий круг вопросов, касающихся программирования систем на базе 16-битных микроконтроллеров PIC. Хотя довольно сложно проанализировать все аспекты проектирования таких систем в одной книге, особенно с учетом бурного развития технологий, тем не менее, автор надеется, что анализ основных принципов разработки и программирования устройств с 16-битными микроконтроллерами фирмы Microchip окажет несомненную пользу и при решении тех задач, которые не были рассмотрены. Хочется надеяться, что даже в таком объеме книга принесет пользу читателю и станет настольной для многих разработчиков, как опытных, так и начинающих.

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС-КНИГА» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Internet-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. **(495) 258-91-94, 258-91-95**; электронный адрес **books@alians-kniga.ru**.

Магда Ю. С.

Микроконтроллеры PIC:
архитектура и программирование

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Перевод *Слинкин А. А.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 13.05.2008. Формат 70×100 1/16.
Гарнитура «Ньютон». Печать офсетная.
Усл. печ. л. 45. Тираж 1000 экз.
№

Издательство ДМК Пресс
Web-сайт издательства: www.dmk-press.ru
Internet-магазин: www.alians-kniga.ru