Ю. С. МАГДА

МИКРОКОНТРОЛЛЕРЫ СЕРИИ 8051: ПРАКТИЧЕСКИЙ ПОДХОД

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

МОСКВА 2008

УДК **621.396.6**
ББК **32.872**
М12

Магда Ю. С.
М12 Микроконтроллеры серии 8051: практический подход. — М.: ДМК Пресс, 2008. — 228 с.

ISBN 5-94074-394-3

В книге рассматривается широкий круг вопросов, связанных с практическим применением популярных микроконтроллеров 8051 и их расширений в системах управления и контроля. Основной упор сделан на практические аспекты разработки цифровых и аналоговых интерфейсов, использования таймеров, визуализации результатов измерений в системах сбора информации. Значительная часть материала посвящена практическому программированию в популярной среде разработки Keil uVision. Приводятся многочисленные примеры разработки несложных аппаратно–программных систем сбора аналоговой и цифровой информации, измерительных систем, систем управления внешними устройствами и т.д. Все приведенные в книге проекты разработаны и проверены на отладочном модуле Rita-51 фирмы Rigel Corp. и могут служить основой при разработке собственных проектов.

ББК 32.872
УДК 621.396.6

Юрий Степанович Магда

МИКРОКОНТРОЛЛЕРЫ СЕРИИ 8051:
практический подход

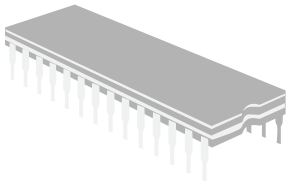
Главный редактор Мовчан Д. А.
dm@dmk-press.ru
Корректор Теренина О. А.
Верстка и графика Старцевой Е. М.
Дизайн обложки Мовчан А. Г.

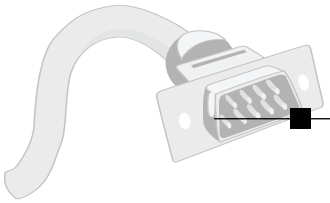
Подписано в печать 02.11.2007. Формат 70x100 1/16.
Гарнитура «Abbat». Печать офсетная.
Усл. печ. л. 21. Тираж 2000 экз.

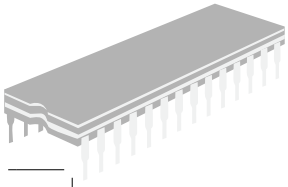
Издательство «ДМК Пресс»
Электронный адрес издательства: www.dmk-press.ru

ISBN 5-94074-394-3

Оглавление







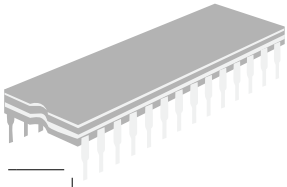
Введение

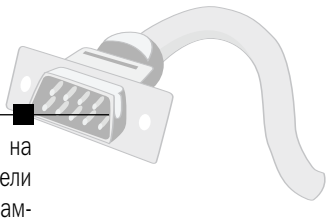
Разработка систем управления и контроля с использованием однокристальных микроконтроллеров в настоящее время переживает настоящий бум. Системы на базе микроконтроллеров используются практически во всех сферах жизнедеятельности человека, и каждый день появляются все новые и новые области применения этих устройств. В последнее время в связи с бурным развитием электроники и схемотехники расширились возможности и самих микроконтроллеров, позволяющие выполнять многие задачи, ранее недоступные для реализации, такие, например, как обработка аналоговых сигналов. Одним из наиболее ранних микроконтроллеров, появившихся на рынке, является микроконтроллер 8051, разработанный фирмой Intel более двадцати лет назад. Несмотря на столь приличный возраст, классический 8051 и его клоны в настоящее время остаются одними из наиболее популярных при разработке систем управления и контроля. Хорошо продуманная архитектура и интуитивно понятная система команд оказывают решающее влияние на выбор многих разработчиков аппаратно-программных систем.

Да и сами микроконтроллеры линейки 8051 постоянно развиваются, предлагая разработчику все новые и новые возможности. На основе базового кристалла 8051 созданы и успешно применяются устройства с развитой периферией и большими объемами памяти. Программирование микроконтроллеров в настоящее время значительно упростилось благодаря инструментальным средствам высокого уровня, разработанным ведущими фирмами. Сегодня микроконтроллеры можно программировать на языках C, Pascal, Basic, Forth и др., что во многом облегчает жизнь программистам, не знакомым с аппаратной частью этих устройств.

Эта книга посвящена практическим аспектам разработки систем на основе микроконтроллера 8051. В отличие от большинства книг по данной тематике, здесь приводятся примеры создания реальных систем и решения задач, с которыми разработчики сталкиваются каждый день. Любой разработчик знает, какие трудности возникают при решении реальной задачи, такой, например, как создание системы сбора и анализа аналоговой информации. В первую очередь это трудности с получением необходимой информации по разработке более-менее сложных систем, поскольку ни одна фирма в мире или разработчик такую информацию так просто не отдадут.

В настоящее время на рынке присутствует очень много литературы и документации по микроконтроллерам для разработчиков начального уровня, но очень мало литературы для более опытных категорий читателей. Предлагаемая вашему вниманию книга призвана восполнить этот пробел. Здесь наряду с основными сведениями об аппаратно-программной архитектуре микроконтроллеров 8051 приводятся многочисленные проекты разработки систем обработки аналоговых и дискретных данных, вывода и отображения информации. Все примеры





аппаратно-программных систем, приведенные в книге, разработаны и протестированы на плате развития Rita-51 фирмы Rigel Corp. и являются полностью работоспособными. Читатели без особого труда смогут адаптировать и усовершенствовать приведенный в книге программный код при разработке собственных систем обработки данных и контроля.

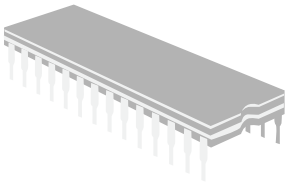
Книга рассчитана на широкий круг читателей – от начинающих до опытных разработчиков – и может оказаться полезной для всех, кто желает самостоятельно изучить аппаратно-программную архитектуру микроконтроллеров 8051 и применить эти знания на практике.

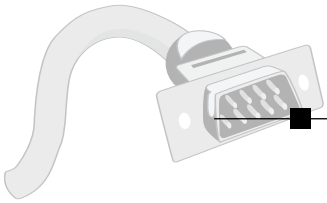
Структура книги

Структура книги такова, что материал можно изучать выборочно, отдельными главами или последовательно, начиная с первой главы. Это позволяет различным категориям читателей изучать тот материал, который им более всего интересен.

Книга состоит из 7 глав; краткий обзор каждой из них:

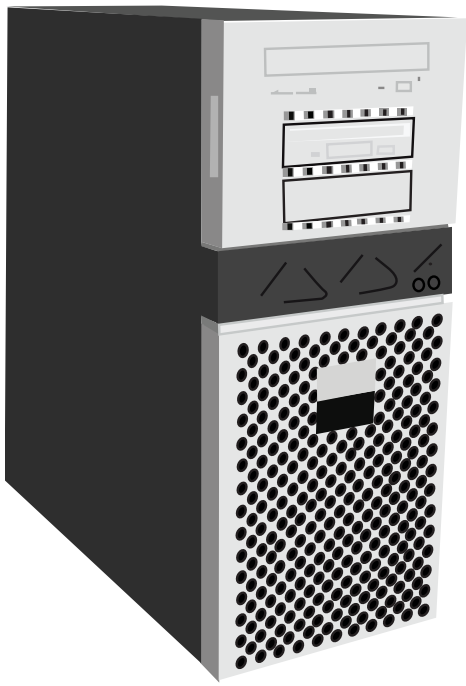
- глава 1 «Программная архитектура микроконтроллеров 8051». В этой главе рассматриваются общие вопросы функционирования микроконтроллеров семейства 8051, аппаратная и программная архитектура базовой модели, включая систему прерываний, ввода/вывода, интерфейсы с внешней памятью программ и данных. Значительная часть материала посвящена описанию системы команд микроконтроллера 8051;
- глава 2 «Программирование и отладка в среде Keil uVision». Материал этой главы посвящен вопросам разработки и отладки программ для 8051 в популярной среде Keil uVision. Рассматриваются вопросы программирования на языке C51 и ассемблере, отладки программного обеспечения. Приводятся практические примеры разработки, компиляции и отладки программ в среде Keil uVision;
- глава 3 «Использование последовательного порта». В этой главе детально проанализированы принципы функционирования последовательного порта микроконтроллера 8051. Рассматриваются многочисленные примеры программного кода обмена данными с использованием последовательного порта. Приводятся практические примеры создания программного интерфейса с персональным компьютером, работающим под управлением операционной системы Windows XP;
- глава 4 «Встроенные таймеры». Эта глава содержит материал по программированию таймеров микроконтроллеров 8051/8052. Дается исчерпывающая информация по аппаратно-программной архитектуре таймеров, подкрепленная многочисленными примерами программирования. Значительная часть главы посвящена применению таймеров в разработке реальных систем с детальным анализом программного кода;
- глава 5 «Обработка дискретных сигналов». Материал главы посвящен вопросам разработки и программирования систем обработки дискретных сигналов. В главе проанализированы основные аппаратно-программные интерфейсы I2C и SPI, а также приведены примеры разработки систем ввода-вывода дискретных данных с использованием этих интерфейсов. Большое внимание уделено разработке пользовательских интерфейсов с детальным анализом их программного кода;
- глава 6 «Ввод/вывод аналоговых сигналов». В этой главе рассматривается широкий круг вопросов, связанных с обработкой аналоговых входных сигналов, а также принципы генерации аналоговых выходных сигналов. На практических примерах показано создание реальных систем сбора данных при использовании аналого-цифровых преобразователей с детальным анализом программного кода. Приводится практический пример разработки цифро-аналогового преобразователя;



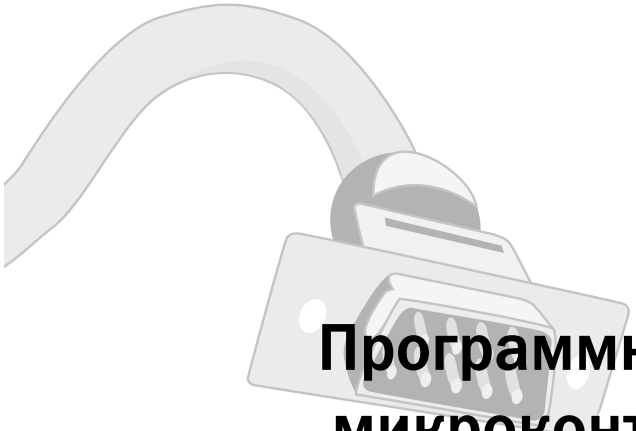


- глава 7 «Отображение информации в системах с микроконтроллерами 8051». Материал главы затрагивает вопросы, связанные с отображением визуальной информации в системах на базе микроконтроллера 8051. Рассмотрены практические примеры построения простых систем отображения информации с использованием семисегментных светодиодных и жидкокристаллических индикаторов. Анализируются вопросы создания пользовательских систем отображения информации с использованием жидкокристаллических дисплеев.

Автор благодарит коллектив издательства «ДМК» за помощь при подготовке книги к изданию. Особая признательность жене Юлии за поддержку и помощь при написании книги.



Программная архитектура микроконтроллеров 8051

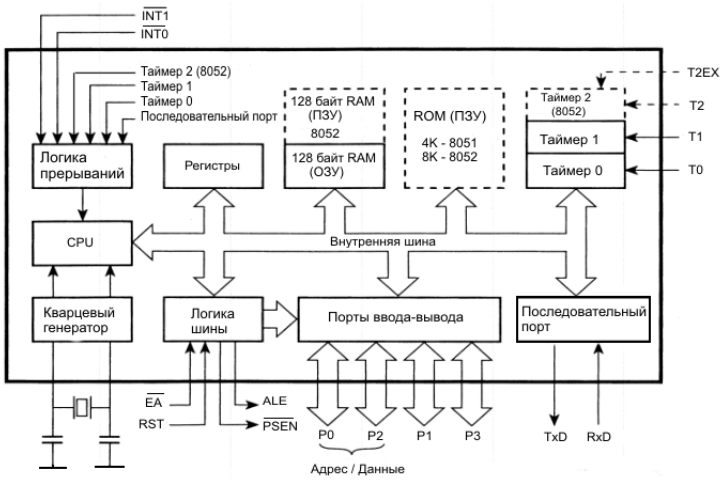


1 Программная архитектура микроконтроллеров 8051

В этой главе мы рассмотрим основные функциональные узлы популярных микроконтроллеров семейства 8051/8052 и принципы их работы. Здесь же вкратце рассмотрим и систему команд 8051, которая нам пригодится при создании аппаратно-программных проектов последующих глав.

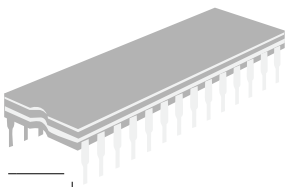
Аппаратная архитектура 8051 представлена на рис. 1.1.

Рис. 1.1. *Функциональная схема аппаратной части 8051*



В микроконтроллере 8051 все вычисления выполняются в арифметико-логическом устройстве, являющемся частью базового процессорного модуля (CPU). Обмен данными, находящимися в оперативной памяти микроконтроллера, а также считывание команд выполняется по внутренней шине 8051. По этой шине осуществляется и обмен данными с портами ввода/вывода P1 – P3, с последовательным портом и таймерами. Внутренний контроллер шины формирует необходимые сигналы (EA, ALE, PSEN, RD/WR) для работы с внешней памятью программ и данных, а также сигнал сброса/начальной установки RST.

Микроконтроллеры 8051 рассчитаны на работу с системами реального времени, которые могут генерировать определенные сигналы, требующие немедленной реакции микроконтроллера. Для обработки таких сигналов (или событий) служит аппаратно реализованная логика прерываний, позволяющая обрабатывать сигналы внешних источников, таймеров и последовательного порта.



Скорость выполнения операций в системе на базе 8051 зависит от тактовой частоты, с которой работает кристалл и которая может варьироваться от единиц до нескольких десятков мегагерц. В архитектуру классического микроконтроллера 8051 были внесены некоторые изменения (к двум существующим таймерам добавлен третий, а также расширена внутренняя память), которые привели к созданию устройства 8052, наиболее популярного в настоящее время.

Микроконтроллер 8051 реализован в виде однокристального устройства с внешними выводами, обозначенными, как показано на рис. 1.2.

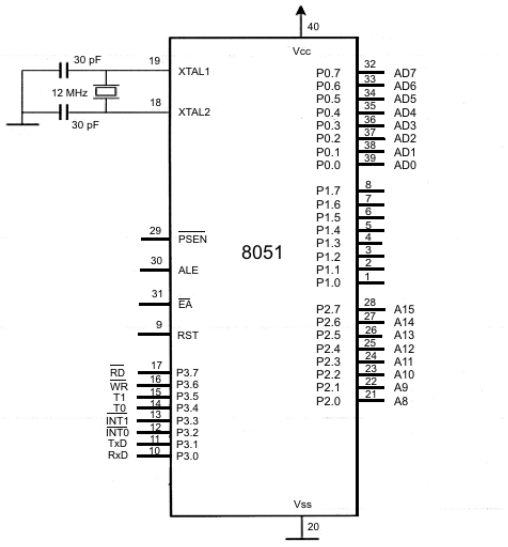


Рис. 1.2.
Схема расположения выводов 8051

Входные и выходные сигналы микроконтроллера 8051 имеют следующие назначения:

- XTAL и XTAL2 – входы подключения кварцевого резонатора для работы генератора тактовой частоты микроконтроллера;
- PSEN – сигнал, используемый при обращении к внешней памяти программ;
- ALE – выходной сигнал разрешения фиксации адреса при обращении к внешней памяти программ/данных;
- EA – сигнал, блокирующий работу с внутренней памятью;
- RST – сигнал общего сброса;
- P0 – P3 – выводы портов ввода/вывода микроконтроллера;
- Vss и Vcc – выводы подачи напряжения питания.

Порты P0, P2 и P3 помимо функционирования в режиме ввода/вывода дискретных сигналов могут выполнять, в зависимости от аппаратной конфигурации, и другие функции. Так, через порт P0 при обращениях к внешней памяти выставляются младшие 8 бит 16-разрядного адреса, а затем, в фазе записи/чтения данных, через этот порт идет обмен данными. Порт P2 при обращениях к внешней памяти служит источником старших 8 бит 16-разрядного адреса.

Выводы порта P3 микроконтроллера 8051 имеют следующие альтернативные назначения:

- P3.0 – вход приема данных в последовательный порт;
- P3.1 – выход передачи данных с последовательного порта;
- P3.2 – вход внешнего прерывания INT0;

- P3.3 – вход внешнего прерывания INT1;
- P3.4 – вход управления таймером 0;
- P3.5 – вход управления таймером 1;
- P3.6 – выход сигнала записи в память;
- P3.7 – выход сигнала чтения из памяти.

Для использования альтернативных функций порта P3 необходимо настроить соответствующим образом программное обеспечение системы 8051.

1.1. Структура внутренней памяти 8051

Микроконтроллеры 8051 оперируют двумя типами памяти: памятью программ и памятью данных. Память данных может быть реализована как комбинация размещенного на кристалле (резидентного или on-chip) статического ОЗУ и внешних микросхем памяти. Для простых аппаратно-программных конфигураций с применением 8051 бывает достаточно резидентной памяти самого микроконтроллера.

Программный код размещается в памяти программ, которая физически может быть реализована в виде однократно программируемого устройства (EPROM), перепрограммируемого устройства (EEPROM) или флеш-памяти. Если для записи программ используется EPROM или EEPROM, то программный код обычно располагается во внешнем по отношению к микроконтроллеру устройстве. В подавляющем большинстве современных микроконтроллеров 8051 память программ располагается во флеш-памяти, находящейся, так же как и резидентная память данных, на одном кристалле.

Память программ и память данных физически и логически разделены, имеют различные механизмы адресации, работают под управлением различных сигналов и выполняют разные функции.

Память программ может иметь максимальный объем, равный 64 Кб, что обусловлено использованием 16-разрядной шины адреса. Во многих случаях емкость памяти программ, размещенной на кристалле 8051, ограничена 4, 8 или 16 Кб. В память программ кроме команд могут записываться константы, управляющие слова инициализации, таблицы перекодировки входных и выходных переменных и т.п. Доступ к содержимому памяти программ осуществляется посредством 16-битовой шины адреса. Сам адрес формируется с помощью либо программного счетчика (PC), либо регистра-указателя данных (DPTR). DPTR выполняет функции базового регистра при косвенных переходах по программе или используется в операциях с таблицами.

Общая структура памяти микроконтроллера 8051 показана на рис. 1.3.

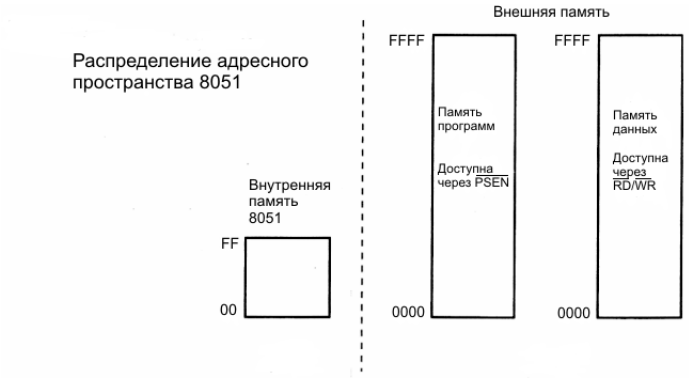


Рис. 1.3. *Общая структура памяти*

Рассмотрим более подробно резидентную (on-chip) память микроконтроллера 8051. Резидентная память, изображенная в левой части рис. 1.3, состоит из двух частей: внутреннего ОЗУ размером 128 байт и памяти, выделяемой для регистров специальных функций (Special Function Registers, SFR). Внутреннее ОЗУ имеет структуру, показанную на рис. 1.4.

Рис. 1.4.
Внутреннее ОЗУ

Для доступа к данным, размещенным во внутреннем ОЗУ, используется однобайтовый адрес. Архитектура внутренней памяти данных 8051 позволяет обращаться к отдельным битам данных в специально выделенной области внутреннего ОЗУ, начиная с адреса 0x20 и заканчивая 0x2F (см. рис. 1.4). Таким образом, в указанном диапазоне адресов можно обращаться к 128-битовым переменным с помощью команд битовых операций SETB и CLR. Битовые переменные нумеруются, начиная с 0x0 и заканчивая 0x7F. Это не означает, что нельзя обращаться к этим ячейкам памяти, как к байтам при обычных операциях с памятью.

Например, для установки бит 0 и 1 в области памяти начиная с 0x20, можно выполнить команды

```
SETB    00h
SETB    01h
```

или команду

```
ORL 20h, 0x3
```

Во втором случае обращение выполняется к байту по адресу 0x20, а установка соответствующих битов выполняется операцией «логическое ИЛИ».

Во внутреннем ОЗУ микроконтроллера 8051 выделены 4 банка регистров общего назначения. При включении микроконтроллера банком по умолчанию становится банк 0 (см. рис. 1.4). При этом регистру R0 соответствует адрес 0x00, регистру R1 – адрес 0x01, наконец, регистру R7 при использовании банка 0 соответствует адрес 0x07. Если банком по умолчанию становится, например, банк 1, то регистру R0 будет соответствовать адрес 0x08, регистру R1 – адрес 0x09 и регистру R7 – адрес 0x0F.

К адресному пространству внутреннего ОЗУ начиная с адреса 0x80 примыкают и адреса регистров специальных функций (рис. 1.5).

Регистры специальных функций (Special Function Registers, SFR) предназначены для управления ходом вычислительных операций, а также отвечают за инициализацию, настройку и управление портами ввода/вывода, таймерами, последовательным портом. Кроме того, регистры специальных функций содержат информацию о приоритетах прерываний, а также биты

управления разрешением прерываний. Регистры специальных функций с указанием их назначения перечислены в табл. 1.1.

Рис. 1.5. *Регистры специальных функций*

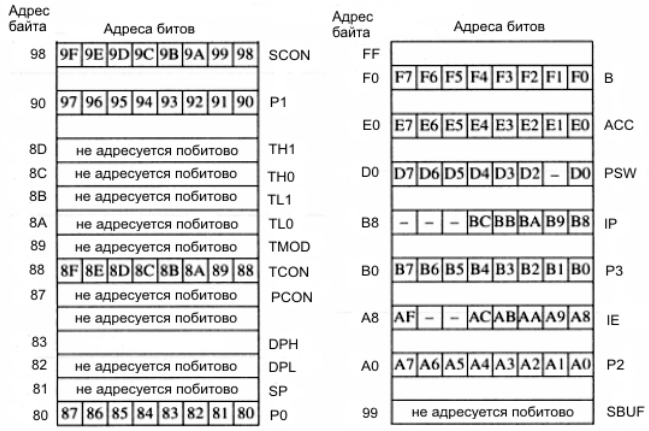
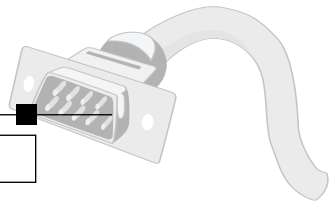


Таблица 1.1. *Назначение регистров специальных функций*

Обозначение	Описание	Адрес
A	Аккумулятор	0E0H
B	Регистр-расширитель аккумулятора	0F0H
PSW	Слово состояния программы	0D0H
SP	Регистр-указатель стека	81H
DPTR	Регистр-указатель данных (DPH)	83H
	(DPL)	82H
P0	Порт 0	80H
P1	Порт 1	90H
P2	Порт 2	0A0H
P3	Порт 3	0B0H
IP	Регистр приоритетов прерываний	0B8H
IE	Регистр маски прерываний	0A8H
TMOD	Регистр режима таймера/счетчика	89H
TCON	Регистр управления/статуса таймера	88H
TH0	Таймер 0 (старший байт)	8CH
TL0	Таймер 0 (младший байт)	8AH
TH1	Таймер 1 (старший байт)	8DH
TL1	Таймер 1 (младший байт)	8BH
SCON	Регистр управления приемопередатчиком	98H
SBUF	Буфер приемопередатчика	99H
PCON	Регистр управления мощностью	87H

Некоторые регистры специальных функций допускают побитовую адресацию. При этом обращение к отдельным битам такого регистра возможно как с помощью обычных функций для работы с байтами, так и с помощью команд побитовых операций. Например, для запуска таймера 0 можно выполнить команду ассемблера



```
ORL TCON, #10h
```

или одну из команд установки бита TCON.4 (TR0):

```
SETB    TCON.4
SETB    TR0
```

Рассмотрим смысл некоторых регистров специальных функций и начнем с аккумулятора и регистра слова состояния (регистра флагов).

Аккумулятор (A) является источником операнда и фиксирует результат при выполнении арифметических, логических операций и ряда операций передачи данных. Кроме того, некоторые операции можно выполнить только с использованием аккумулятора: например, операции сдвигов, проверку на нуль, формирование флага паритета и т.п.

В распоряжении программиста имеются 8 регистров общего назначения R0 – R7 одного из четырех банков. При выполнении многих команд в арифметико-логическом устройстве микроконтроллера формируется ряд признаков операции (флагов), которые фиксируются в регистре PSW. Перечень флагов PSW, их символические имена и условия формирования приведены в табл. 1.2.

Таблица 1.2. *Регистр слова состояния микроконтроллера*

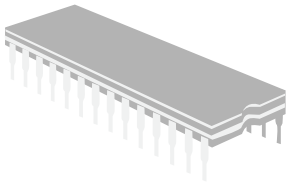
Обозначение	Бит	Описание
C	PSW.7	Флаг переноса. Устанавливается и сбрасывается аппаратно или программно при выполнении арифметических и логических операций
AC	PSW.6	Флаг вспомогательного переноса. Устанавливается и сбрасывается только аппаратно при выполнении команд сложения и вычитания и сигнализирует о переносе или займе в бите 3
F0	PSW.5	Флаг 0. Может быть установлен, сброшен или проверен программой как флаг, специфицируемый пользователем
RS1	PSW.4	Выбор банка регистров. Устанавливается и сбрасывается программно для выбора рабочего банка регистров (табл. 1.3)
RS0	PSW.3	
OV	PSW.2	Флаг переполнения. Устанавливается и сбрасывается аппаратно при выполнении арифметических операций
–	PSW.1	Не используется
P	PSW.0	Флаг четности. Устанавливается и сбрасывается аппаратно в каждом цикле и фиксирует нечетное/четное число единичных битов в аккумуляторе, т.е. выполняет контроль по четности

Установки битов RS0 – RS1 при выборе банка регистров показаны в табл. 1.3.

Таблица 1.3. *Битовые комбинации для выбора банка регистров*

RS1	RS0	Банк	Границы адресов
0	0	0	00h–07h
0	1	1	08h–0Fh
1	0	2	10h–17h
1	1	3	18h–1Fh

Среди регистров специального назначения есть регистры, выполняющие функции адресации данных, находящихся в памяти. К ним относятся 8-разрядный указатель стека (SP) и регистр-указатель DPTR.



Указатель стека может адресовать любую область внутренней памяти данных микроконтроллера, при этом содержимое этого регистра инкрементируется перед выполнением команд PUSH и CALL и декрементируется после выполнения команд POP и RET. В процессе инициализации микроконтроллера после сигнала RST в указатель стека автоматически загружается код 0x07. Это значит, что если программа не переопределяет содержимое указателя стека, то первый элемент данных в стеке будет располагаться в ячейке памяти 0x08.

Нужно отметить одну важную особенность: при загрузке данных в стек адрес растет вверх, поэтому если в программе используются банки 1, 2 или 3, то указатель стека следует проинициализировать адресом из неиспользуемой области памяти, например 0x30, чтобы не перезаписать содержимое регистров одного из банков.

Двухбайтный регистр-указатель данных DPTR обычно используется для фиксации 16-разрядного адреса в операциях с обращением к внешней памяти. При работе с DPTR допускается использование старшего и младшего байтов регистра (DPH и DPL соответственно).

1.2. Подключение внешней памяти программ и данных

Внешняя память данных может быть подключена к микроконтроллеру приблизительно по такой схеме, какая изображена на рис. 1.6.

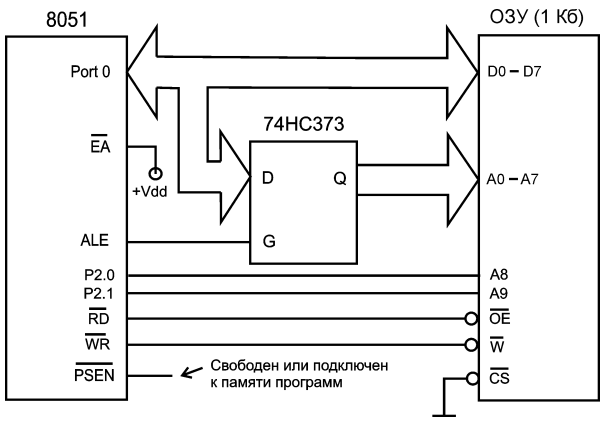


Рис. 1.6. *Подключение внешней памяти данных*

Здесь показан интерфейс микроконтроллера 8051 с внешним модулем ОЗУ емкостью 1 Кб. Адресация памяти по этой схеме реализована следующим образом:

1. Младшие 8 бит адреса выводятся стандартным образом через порт P0 и запоминаются в регистре-защелке 74HC373 по спаду сигнала ALE.
2. На шину адреса подаются старшие биты адреса, из которых используются разряды A8 и A9, устанавливаемые на выводах P2.0 и P2.1 и предназначенные (вместе с установленными в регистре 74HC373 линиями A0 – A7) для выбора адреса в пределах 1 Кб.
3. По низкому уровню одного из сигналов RD (чтение) или WR (запись) осуществляется требуемая операция, при этом байт данных считывается/записывается через порт P0.

Память программ, так же как и память данных, может быть расширена до 64 Кб путем подключения внешних микросхем. Стандартная схема подключения внешней памяти программ осуществляется по схеме, показанной на рис. 1.7.

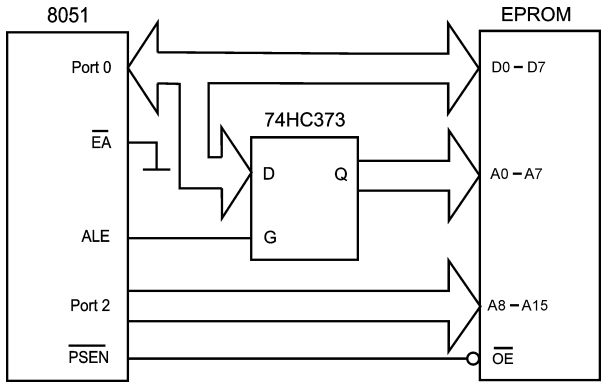


Рис. 1.7. *Подключение внешней памяти программ*

Так же как и при обращении к памяти данных, младшая часть адреса памяти формируется портом P0 и запоминается в регистре-защелке по спаду ALE, а старший байт адреса выводится через порт P2. Считывание команды выполняется при подаче низкого уровня на линию PSEN. Поскольку вывод EA подключен к общему проводу, то внутренняя память программ отключается и микроконтроллер при включении начинает выполнение программы с адреса 0x0000 внешней памяти.

1.3. Система команд микроконтроллера семейства 8051

Микроконтроллеры семейства 8051 являются микропроцессорными устройствами с архитектурой CISC со стандартным набором команд, характерных для данной архитектуры. Система команд 8051-совместимых устройств включает 111 основных команд размером от одного до трех байт, но большая часть этих команд – одно- или двухбайтовая. Почти все команды выполняются за один или два машинных цикла, что по времени приблизительно равно 1–2 мкс при тактовой частоте 12 МГц, за исключением команд умножения и деления, которые требуют для выполнения четыре машинных цикла.

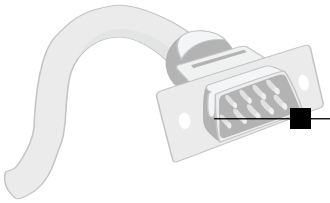
Команды микроконтроллеров 8051 используют прямую, непосредственную, косвенную и неявную адресацию данных. При этом в качестве операндов команд могут выступать отдельные биты, четырехбитовые комбинации (тетрады), байты и слова из двух байт.

В набор команд семейства 8051 входит ряд команд, обеспечивающих выполнение специфичных функций управления, например, манипуляции с отдельными битами. Особенностью системы команд 8051 является возможность адресации отдельных бит в памяти данных, а также отдельных бит регистров специальных функций.

По выполняемым функциям команды микроконтроллера 8051 можно разделить на несколько групп:

- пересылки данных;
- арифметических операций;
- логических операций;
- операций над битами;
- передачи управления.

Рассмотрим эти группы команд более подробно, но перед этим условимся при описании мнемоники команд использовать следующие обозначения:



- Rn (n = 0...7) – регистр общего назначения в выбранном банке регистров;
- @Ri (i = 0, 1) – регистр общего назначения в выбранном банке регистров, используемый для формирования косвенного адреса;
- addr – адрес байта;
- src – адрес байта-источника;
- dst – адрес байта-приемника;
- addr11 – 11-разрядный абсолютный адрес перехода;
- addr16 – 16-разрядный абсолютный адрес перехода;
- label – относительный адрес перехода;
- #direct8 – непосредственный операнд размером 1 байт;
- #direct16 – непосредственный операнд размером 2 байта;
- bit – адрес прямо адресуемого бита;
- ~bit – инверсия прямо адресуемого бита;
- A – регистр-аккумулятор;
- PC – регистр-счетчик команд;
- DPTR – 16-разрядный регистр-указатель данных;
- ( ) – содержимое ячейки памяти или регистра.

Команды пересылки данных микроконтроллера 8051 включают 28 команд, краткое описание которых приведено в табл. 1.4.

Таблица 1.4. *Команды пересылки данных*

Мнемоника	Описание
MOV A, Rn	(A) ← (Rn)
MOV A, addr	(A) ← (addr)
MOV A, @Ri	(A) ← ((Ri))
MOV A, #direct8	(A) ← #direct8
MOV Rn, A	(Rn) ← (A)
MOV Rn, addr	(Rn) ← (addr)
MOV Rn, #direct8	(Rn) ← #direct8
MOV add, A	(addr) ← (A)
MOV add, Rn	(addr) ← (Rn)
MOV dst, src	(dst) ← (src)
MOV addr, @Ri	(addr) ← ((Ri))
MOV addr, #direct8	(addr) ← #direct8
MOV @Ri, A	((Ri)) ← (A)
MOV @Ri, addr	((Ri)) ← (addr)
MOV @Ri, #direct8	((Ri)) ← #direct8
MOV DPTR, #direct16	(DPTR) ← #direct16
MOVC A, @A+DPTR	(A) ← ((A) + (DPTR))
MOVC A, @A+PC	(PC) ← (PC+1), (A) ← ((A)+(PC))
MOVX A, @Ri	(A) ← ((Ri))
MOVX A, @DPTR	(A) ← ((DPTR))

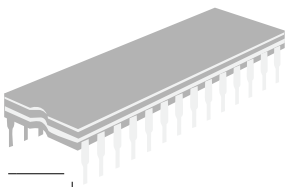


Таблица 1.4. *Команды пересылки данных (окончание)*

Мнемоника	Описание
MOVX @Ri, A	((Ri)) ← (A)
MOVX @DPTR, A	((DPTR)) ← (A)
PUSH addr	(SP) ← (SP) + 1, ((SP)) ← (addr)
POP addr	(addr) ← ((SP)), (SP) ← (SP) – 1
XCH A, Rn	(A) ↔ (Rn)
XCH A, addr	(A) ↔ (addr)
XCH A, @Ri	(A) ↔ ((@Ri))
XCHD A, @Ri	(A0 – 3) ↔ ((@Ri0 – 3))
MOV A, Rn	(A) ← (Rn)

Команда MOV выполняет пересылку данных из второго операнда в первый. Эта команда не работает с данными, находящимися во внешней памяти данных или в памяти программ. Для работы с данными, находящимися во внешней памяти данных, предназначены команды MOVX, а для работы с константами, записанными в память программ, – команда MOVC. Первая из них обеспечивает чтение/запись байтов из внешней памяти данных, вторая – чтение байтов из памяти программ.

Команды XCH выполняют обмен байтами между аккумулятором и ячейкой памяти, а команда XCHD выполняет обмен данными между младшими тетрадами (биты 0–3).

Команды PUSH и POP предназначены для записи данных в стек и их чтения из стека соответственно. При этом размер стека ограничен лишь размером памяти данных, расположенной на кристалле. В процессе инициализации микроконтроллера после сигнала сброса или при включении питающего напряжения в указатель стека SP заносится код 07H. Таким образом, первый элемент стека будет располагаться в ячейке памяти с адресом 08H.

В группе команд пересылок микроконтроллера нет команд для работы с регистрами специальных функций (таймерами, портами ввода/вывода и т.д.). Доступ к таким регистрам осуществляется по их прямому адресу или с использованием их мнемоники, записанной в специальном файле (для программ на ассемблере, например, таким файлом может быть 8051.MCU).

Следует отметить, что регистр-аккумулятор имеет два различных имени в зависимости от способа адресации (A – при неявной адресации, например MOV A, R0; ACC – при использовании прямого адреса).

В группу команд арифметических операций 8051 входят 24 команды, выполняющие операции по обработке целочисленных данных, включая команды умножения и деления. Перечень команд этой группы приведен в табл. 1.5.

Таблица 1.5. *Команды арифметических операций*

Мнемоника	Описание
ADD A, Rn	(A) ← (A) + (Rn)
ADD A, addr	(A) ← (A) + (addr)
ADD A, @Ri	(A) ← (A) + ((Ri))
ADD A, #direct8	(A) ← (A) + #direct8
ADDC A, Rn	(A) ← (A) + (Rn) + (C)
ADDC A, addr	(A) ← (A) + (addr) + (C)
ADDC A, @Ri	(A) ← (A) + ((Ri)) + (C)

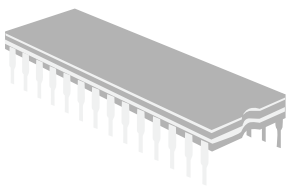


Таблица 1.5. *Команды арифметических операций (окончание)*

Мнемоника	Описание
ADDC A, #direct8	(A) ← (A) + #direct8 + (C)
DAA	Десятичная коррекция аккумулятора
SUBB A, Rn	(A) ← (A) – (Rn) – (C)
SUBB A, addr	(A) ← (A) – (ad) – (C)
SUBB A, @Ri	(A) ← (A) – ((Ri)) – (C)
SUBB A, #direct8	(A) ← (A) – #direct8 – (C)
INC A	(A) ← (A) + 1INC Rn(Rn) < (Rn) + 1
INC Rn	(Rn) ← (Rn) + 1
INC addr	(addr) ← (addr) + 1
INC @Ri	((Ri)) ← ((Ri)) + 1
INC DPTR	(DPTR) ← (DPTR) + 1
DEC A	(A) ← (A) – 1
DEC Rn	(Rn) ← (Rn) – 1
DEC addr	(addr) ← (addr) – 1
DEC @Ri	((Ri)) ← ((Ri)) – 1
MUL AB	(B)(A) ← (A) × (B)

Результат выполнения команд ADD, ADDC, SUBB, MUL и DIV влияет на флаги слова состояния (PSW) следующим образом:

- флаг переноса C устанавливается при переносе из разряда D7 в том случае, если результат операции не помещается в восемь разрядов; флаг дополнительного переноса AC устанавливается при переносе из разряда D3 в командах сложения и вычитания и служит для реализации десятичной арифметики. Этот признак используется командой DAA;
- флаг OV устанавливается при переносе из разряда D6 в случае, если результат операции не помещается в семь разрядов и восьмой не может быть интерпретирован как знаковый. Этот признак служит для организации обработки чисел со знаком;
- флаг четности P устанавливается и сбрасывается аппаратно. Если число единичных битов в аккумуляторе нечетно, то P = 1, в противном случае P = 0.

Следующая группа команд, которую мы рассмотрим, – это команды логических операций. Группа содержит 25 команд (табл. 1.6).

Таблица 1.6. *Команды логических операций*

Мнемоника	Описание
ANL A, Rn	(A) ← (A) & (Rn)
ANL A, addr	(A) ← (A) & (addr)
ANL A, @Ri	(A) ← (A) & ((Ri))
ANL A, #direct8	(A) ← (A) & (#direct8)
ANL addr, A	(addr) ← (addr) & (A)
ANL addr, #direct8	(addr) ← (addr) & (#direct8)
ORL A, Rn	(A) ← (A) I (Rn)
ORL A, addr	(A) ← (A) I (addr)

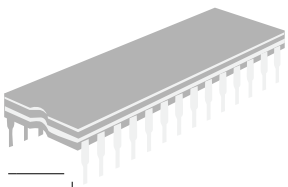


Таблица 1.6. *Команды логических операций (окончание)*

Мнемоника	Описание
ORL A, @Ri	(A) ← (A) | ((Ri)
ORL A, #direct8	(A) ← (A) | (#direct8)
ORL addr, A	(addr) ← (addr) | (A)
ORL addr, #direct8	(addr) ← (addr) | (#direct8)
XRL A, Rn	(A) ← (A) ^ (Rn)
XRL A, addr	(A) ← (A) ^ (addr)
XRL A, @Ri	(A) ← (A) ^ ((Ri))
XRL A, #direct8	(A) ← (A) ^ (#direct8)
XRL addr, A	(addr) ← (addr) ^ (A)
XRL addr, #direct8	(addr) ← (addr) ^ (#direct8)
CLR A	(A) ← 0
CPL A	(A) ← ~(A)
SWAP A	(A0–3) – (A4–7)
RL A	Циклический сдвиг влево
RLC A	Сдвиг влево через перенос
RR A	Циклический сдвиг вправо
RRC A	Сдвиг вправо через перенос

Команды логических операций манипулируют байтами и позволяют выполнить следующие операции:

- логическое И (&);
- логическое ИЛИ (|);
- исключающее ИЛИ (^);
- инверсию (~);
- очистку байта;
- обычные и циклические сдвиги.

Команды операций над битами микроконтроллера 8051 включают 12 команд, позволяющих выполнять операции над отдельными битами: сброс, установку, инверсию, а также «логическое И» (&) и «логическое ИЛИ» (|). В качестве операндов могут выступать 128 бит из внутренней памяти данных микроконтроллера, а также регистры специальных функций, допускающие адресацию отдельных битов. Перечень команд и их мнемоника показаны в табл. 1.7.

Таблица 1.7. *Команды битовых операций*

Мнемоника	Описание
CLR C	(C) ← 0
CLR bit	(bit) ← 0
SETB C	(C) ← 1
SETB bit	(bit) ← 1
CPL C	(C) ← ~(C)
CPL bit	(bit) ← ~ (bit)
ANL C, bit	(C) ← (C) & (bit)

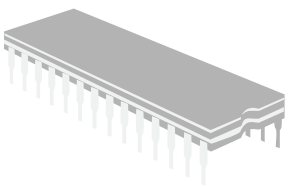


Таблица 1.7. *Команды битовых операций (окончание)*

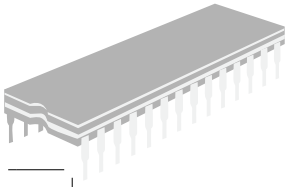
Мнемоника	Описание
ORL C, bit	(C) ← (C) ∣ (bit)
MOV C, bit	(C) ← (bit)
MOV bit, C	(bit) ← (C)

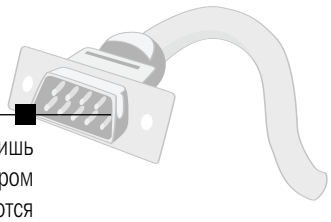
Последняя группа команд, которую мы рассмотрим, – это группа команд передачи управления микроконтроллера. В группе представлены команды безусловного и условного переходов, команды вызова подпрограмм и команды возврата из подпрограмм. Мнемоника команд и их описание представлены в табл. 1.8.

Таблица 1.8. *Команды передачи управления*

Мнемоника	Описание
LJMP addr16	Длинный безусловный переход по всем адресам памяти
AJMP addr11	Безусловный переход в пределах страницы 2 Кб
SJMP label	Безусловный переход на метку label в пределах страницы 256 байт
JMP @A+DPTR	Безусловный переход по косвенному адресу
JZ label	Переход на метку label, если нуль
JNZ label	Переход на метку label, если не нуль
JC label	Переход на метку label, если бит переноса установлен
JNC label	Переход на метку label, если бит переноса не установлен
JB bit, label	Переход на метку label, если бит установлен
JNB bit, label	Переход на метку label, если бит не установлен
JBC bit, label	Переход на метку label, если бит установлен с очисткой бита
DJNZ Rn, label	Переход на метку label, если содержимое Rn не равно
0DJNZ addr, label	Переход на метку label, если содержимое addr не равно
0CJNE A, addr, label	Сравнение аккумулятора с байтом и переход на метку label, если не равно
CJME A, #direct8, label	Сравнение аккумулятора с константой и переход на метку label, если не равно
CJNE Rn, #direct8, label	Сравнение регистра с константой и переход на метку label, если не равно
CJNE @Ri, #direct8, label	Сравнение байта памяти с константой и переход на метку label, если не равно
LCALL addr16	Длинный вызов подпрограммы по всем адресам памяти
ACALL addr11	Вызов подпрограммы в пределах страницы 2 Кб
RET	Возврат подпрограммы
RETI	Возврат подпрограммы обработки прерывания
NOP	Пустая операция

Рассмотрим, как работают команды передачи управления. Команда безусловного перехода LJMP осуществляет переход по абсолютному 16-битному адресу в пределах сегмента программ.





Действие команды AJMP аналогично команде LJMP, однако операндом являются лишь 11 младших разрядов адреса, что позволяет выполнить переход в пределах страницы размером 2 Кб. При этом содержимое счетчика команд вначале увеличивается на 2, затем заменяются 11 разрядов адреса.

В команде SJMP указан не абсолютный, а относительный адрес перехода. Величина смещения label рассматривается как число со знаком, поэтому переход возможен в пределах от –128 до +127 байт относительно адреса команды, следующей за командой SJMP.

Команда косвенного перехода JMP @A+DPTR позволяет вычислять адрес перехода в процессе выполнения самой программы. С помощью команд условного перехода можно проверять следующие условия:

- аккумулятор содержит нулевое значение (JZ);
- аккумулятор содержит ненулевое значение (JNZ);
- бит переноса C установлен (JC);
- бит переноса C не установлен (JNC);
- прямо адресуемый бит равен 1 (JB);
- прямо адресуемый бит равен 0 (JNB);
- прямо адресуемый бит равен 1 и очищается при выполнении команды (JBC).

Все команды условного перехода микроконтроллера 8051 оперируют с коротким относительным адресом из диапазона –128 – +127 относительно следующей команды.

Команда DJNZ предназначена для организации программных циклов. Указанные в команде регистр Rn или байт по адресу addr содержат счетчик повторений цикла, а смещение label – относительный адрес перехода к началу цикла. При выполнении команды содержимое счетчика уменьшается на 1 и проверяется на 0. Если значение содержимого счетчика не равно 0, то осуществляется переход на начало цикла, в противном случае выполняется следующая команда.

Команда CJNE удобна для реализации процедур ожидания событий. Операнды команды сравниваются между собой, после чего, в зависимости от результата сравнения, выполняется либо переход на метку label, либо выполняется следующая команда.

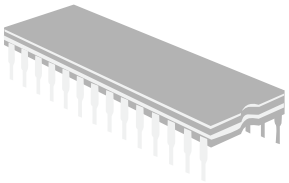
Действие команд вызова процедур полностью аналогично действию команд безусловного перехода – за исключением того, что они сохраняют в стеке адрес возврата.

Команда возврата из подпрограммы RET восстанавливает из стека значение содержимого счетчика команд, а команда возврата из процедуры обработки прерывания RETI, кроме того, разрешает прерывания. Следует отметить, что ассемблер 8051 допускает обобщенную мнемонику для команд безусловного перехода JMP и вызова подпрограмм CALL.

1.4. Система прерываний

Работа микроконтроллера 8051 в системах реального времени была бы невозможна без обработки событий, генерируемых внешними устройствами, и установки временных зависимостей между событиями в системе. Именно этим целям и служит логика обработки прерываний 8051, функциональная схема которой показана на рис. 1.8.

Классический микроконтроллер 8051 имеет 5 источников прерываний: два внешних прерывания, инициированных сигналами на входах, – INT0 (вывод P3.2) и INT1 (вывод P3.3); два прерывания таймеров – 0 и 1; прерывание последовательного порта (см. рис. 1.8). Очередность выполнения двух и более одновременно поступивших прерываний определяется их приоритетами.



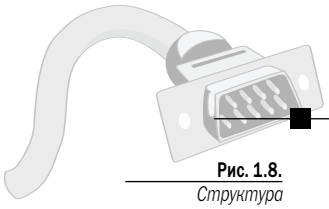
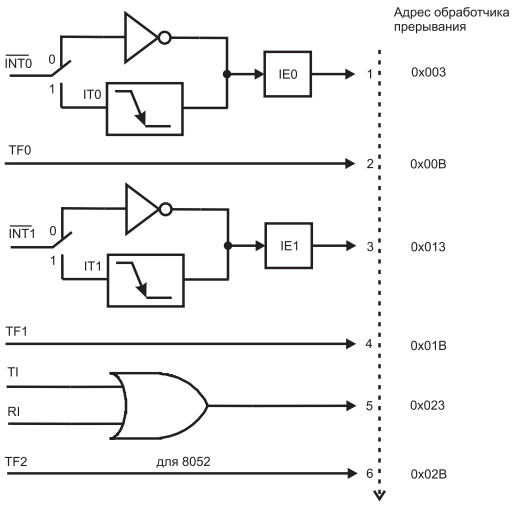


Рис. 1.8. *Структура прерываний 8051*



Если предположить, что все прерывания имеют одинаковые установки приоритета, то при одновременном возникновении нескольких прерываний они обрабатываются так, как показано на рис. 1.8. Например, прерывание INT0 имеет наивысший приоритет (ему соответствует условное обозначение 1 на рис. 1.8), прерывание таймера 0 имеет более низкий приоритет по сравнению с INT0 (обозначено цифрой 2) и т.д. Самый низкий приоритет в устройстве 8051 имеет прерывание последовательного порта (обозначено цифрой 5 на рисунке). Для микроконтроллера 8052 самым низким приоритетом обладает прерывание таймера 2 (обозначено цифрой 6). Для изменения приоритетов выполнения прерываний следует устанавливать специальные флаги в регистре приоритетов IP (табл. 1.9).

Таблица 1.9. *Регистр приоритетов*

Обозначение	Бит регистра IP	Описание
PX0	IP.0	Бит приоритета внешнего прерывания 0. Установка/сброс программой для назначения прерыванию INT0 высшего/низшего приоритета
PT0	IP.1	Бит приоритета таймера 0. Установка/сброс программой для назначения прерыванию от таймера 0 высшего/низшего приоритета
PX1	IP.2	Бит приоритета внешнего прерывания 1. Установка/сброс программой для назначения прерыванию INT1 высшего/низшего приоритета
PT1	IP.3	Бит приоритета таймера 1. Установка/сброс программой для назначения прерыванию от таймера 1 высшего/низшего приоритета
PS	IP.4	Бит приоритета последовательного порта. Установка/сброс программой для назначения прерыванию последовательного порта высшего/низшего приоритета

Посмотрим, как вызываются и обрабатываются прерывания. Микроконтроллер реагирует на прерывания, только если будет установлен флаг соответствующего прерывания в регистре TCON (табл. 1.10).

Любое из прерываний будет вызвано только в том случае, если его вызов разрешен путем установки соответствующих битов в регистре IE (табл. 1.11).

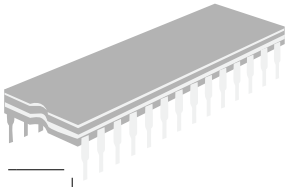


Таблица 1.10. *Биты прерываний в регистре TCON*

Обозначение	Бит регистра	Описание
IT0	TCON.0	Бит управления типом прерывания INT0, допускает программную установку. Бит IT0 = 1, если прерывание должно инициироваться перепадом сигнала 1–0 на выводе INT0/P3.2. Если IT0 = 0, то прерывание инициируется низким уровнем сигнала на выводе INT0
IE0	TCON.1	Флаг прерывания INT0. Устанавливается по перепаду 1–0 сигнала на выводе INT0. Сбрасывается аппаратно при обслуживании прерывания
IT1	TCON.2	Бит управления типом прерывания INT1, допускает программную установку. Бит IT1 = 1, если прерывание должно инициироваться перепадом сигнала 1–0 на выводе INT0/P3.3. Если IT1 = 0, то прерывание инициируется низким уровнем сигнала на выводе INT1
IE1	TCON.3	Флаг прерывания INT1. Устанавливается по перепаду 1–0 сигнала на выводе INT1. Сбрасывается аппаратно при обслуживании прерывания
TF0	TCON.5	Флаг переполнения таймера 0. Устанавливается аппаратно и сбрасывается при обслуживании прерывания
TF1	TCON.7	Флаг переполнения таймера 1. Устанавливается аппаратно и сбрасывается при обслуживании прерывания

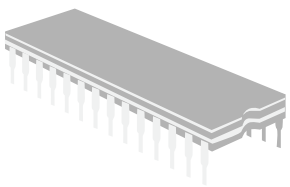
Таблица 1.11. *Биты разрешения прерываний*

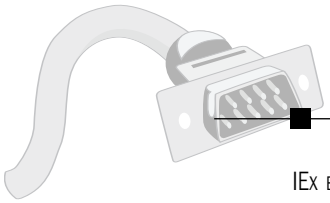
Обозначение	Бит регистра IE	Описание
EA	IE.7	Разрешение/запрет прерываний от всех источников
EX0	IE.0	Разрешение/запрет прерывания по входу INT0
ET0	IE.1	Разрешение/запрет прерывания таймера 0
EX1	IE.2	Разрешение/запрет прерывания по входу INT1
ET1	IE.3	Разрешение/запрет прерывания таймера 1
ES	IE.4	Разрешение/запрет прерывания последовательного порта
ET2	IE.5	Разрешение/запрет прерывания таймера 2 (только для 8052)

Здесь флаг EA разрешает работу всех прерываний, для каждого из которых должен быть установлен соответствующий бит. Например, в следующем фрагменте программного кода разрешается работа прерывания последовательного порта:

```
SETB    ES
SETB    EA
```

Рассмотрим, как вызываются и обрабатываются внешние прерывания INT0 и INT1 в микроконтроллере 8051. Внешние прерывания INT0 и INT1 могут быть вызваны уровнем или переходом сигнала из 1 в 0 на входах микроконтроллера в зависимости от значений управляющих битов IT0 и IT1 в регистре TCON. Если соответствующий бит ITx (x = 0,1) установлен, то прерывание вызывается по перепаду 1–0 на соответствующем входе INTx микроконтроллера. Если эти биты сброшены, то прерывание INTx вызывается при наличии низкого уровня на соответствующем входе.





При возникновении внешнего прерывания INTx устанавливается соответствующий флаг IEx в регистре TCON, инициирующий вызов соответствующей подпрограммы обслуживания или, по-другому, программы-обработчика прерывания. Сброс флага IEx выполняется аппаратно в случае, если прерывание было вызвано перепадом сигнала на входе INTx. Если же вход прерывания INTx настроен на срабатывание по низкому уровню, то о состоянии флага IEx должна заботиться программа-обработчик данного прерывания.

Внешние прерывания INT0 и INT1 могут (если разрешены) вызываться одним из трех способов:

- установкой низкого уровня сигнала на соответствующем выводе микроконтроллера;
- программной установкой битов P3.2 и/или P3.3;
- установкой флага прерывания IE0 или IE1.

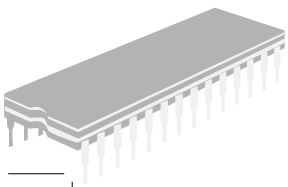
Вот пример программного вызова прерывания INT0 посредством сброса бита `P3.2`:

```
cseg    at 0
    jmp  start
int0Isr:
    org  03h
    setb P3.2
    nop
reti
start:
    mov  SCON,    #50h
    clr  RI
    mov  TH1, #0FDh
    orl  TMOD,    #20h
    setb TR1
    setb EX0
    setb EA
again:
    jnb  RI,  $
    mov  A,   SBUF
    clr  RI
    cjne A,#0dh,  skip
    clr  P3.2
skip:
    jmp  again
```

В этой ассемблерной программе внешнее прерывание INT0 вызывается путем очистки бита `P3.2` при получении символа с кодом 0x0D (перевод строки) с последовательного порта. В программе-обработчике прерывания INT0, расположенной по адресу 0x03 в памяти программ, выполняется единственное действие – установка бита `P3.2`. Если этого не сделать, мы получим рекурсивные вызовы прерываний, что приведет к краху программы.

Еще один пример вызова прерывания INT0 путем установки флага IE0 показан ниже:

```
cseg    at 0
    jmp  start
int0Isr:
    org  03h
cpl P1.7
```



```
    reti
start:
    mov  SCON, #50h
    clr  RI
    orl  PCON, #80h
    mov  TH1, #0F3h ;скорость обмена 9600 бод при тактовой частоте кристалла 24.0 МГц
    orl  TMOD, #20h
    setb TR1
    setb EX0
    setb EA
    setb IT0
again:
    jnb  RI, $
    mov  A, SBUF
    clr  RI
    cjne A, #0dh, skip
    setb TCON.1
skip:
    jmp  again
end
```

Программа-обработчик прерывания INT0 инвертирует бит 7 порта P1 каждый раз, когда вызывается данное прерывание. Вызов прерывания INT0 осуществляется при приеме символа с кодом 0x0D с последовательного порта, после чего выполняется команда

```
setb    TCON.1
```

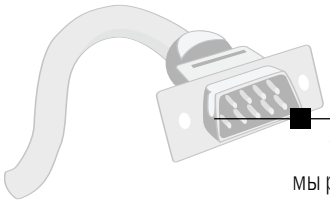
устанавливающая флаг прерывания IE0. Этот флаг сбрасывается аппаратно при обработке прерывания, поэтому очищать его в данном случае не нужно.

Подобным образом (устанавливая флаги прерываний) можно инициировать вызов любого из прерываний. Обратите внимание, что программа-обработчик прерывания должна всегда заканчиваться командой RETI (не RET!).

Прерывания от таймеров вызываются при переполнении регистров таймера (TH0 – TL0 для таймера 0, TH1 – TL1 для таймера 1 и TH2 – TL2 для таймера 2). Для всех трех таймеров при достижении значений 0xFF в THx – TLx (x = 1, 2, 3) следующий инкремент таймера вызовет установку флага TFx, что инициирует прерывание. Флаги прерывания таймеров 0 и 1 размещены в регистре TCON, при этом TF0 соответствует биту TCON.5, а TF1 – биту TCON.7. Что же касается таймера 2 микроконтроллера 8052, то для его управления и контроля служит отдельный регистр T2CON, при этом флаг прерывания TF2 соответствует биту T2CON.7.

Флаги запросов прерывания от таймеров TF0 и TF1 сбрасываются автоматически при передаче управления подпрограмме обслуживания, а флаг прерывания TF2 должен сбрасываться программой-обработчиком.

Прерывание последовательного порта вызывается установкой одного из флагов – TI или RI (см. рис. 1.8). Флаги запросов прерывания RI и TI устанавливаются логикой последовательного порта аппаратно, но сбрасываться должны программой. Прерывания могут быть вызваны или отменены программой, так как все перечисленные флаги программно доступны. Обмен данными через последовательный порт будет рассмотрен далее в этой книге, поэтому останавливаться более подробно на особенностях обработки прерываний последовательного порта мы пока не будем.



Таймер 2 как аппаратно-программный узел отличается от таймеров 0 и 1. Его работу мы рассмотрим более подробно далее в этой книге, сейчас же замечу, что в отличие от таймеров 0 и 1 флаг прерывания TF2 не сбрасывается автоматически при вызове прерывания, это необходимо сделать в самом обработчике прерывания. Кроме того, источником прерывания для таймера 2 может являться не только переполнение счетчика, но и внешнее событие.

Все операции по обработке прерываний выполняет логика прерываний микроконтроллера. Флаги прерываний опрашиваются в каждом машинном цикле, при этом анализ приоритетов прерываний выполняется в течение следующего машинного цикла. Логика прерываний аппаратно сформирует дальний вызов LCALL соответствующей программы-обработчика прерывания, если только этот вызов не блокируется по одной из следующих причин:

- в данный момент обслуживается запрос прерывания с приоритетом равным или более высоким, чем текущий;
- текущий машинный цикл не является последним в цикле выполняемой команды;
- в данный момент выполняется команда RETI или любая команда, которая обращается к одному из регистров – IE или IP.

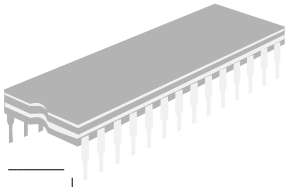
Если флаг прерывания был установлен, но по любой из перечисленных причин прерывание не было обслужено, а флаг к моменту окончания блокировки был сброшен, то запрос прерывания теряется.

Предположим, что ни одна из блокировок не работает. Тогда по аппаратно сформированному коду команды LCALL логика прерываний помещает в стек содержимое программного счетчика PC, загружая в него адрес вектора прерывания соответствующей подпрограммы обслуживания. По этому адресу должна быть расположена команда безусловного перехода JMP к начальному адресу программы-обработчика прерывания. При необходимости обработчик прерывания сохраняет в стеке при помощи команд PUSH регистры, содержимое которых может быть изменено. Перед окончанием обработки необходимо восстановить сохраненные регистры командами восстановления из стека POP. Программы-обработчики обслуживания прерывания обязательно завершаются командой RETI, которая загружает в программный счетчик сохраненный в стеке адрес возврата в основную программу.

Например, если при вызове обработчика прерывания таймера 0 будет изменено содержимое регистра-аккумулятора и слова состояния, то обработчик прерывания может включать такой программный код:

```
t0ISR:
    ORG  0Bh
    PUSH PSW
    PUSH ACC
    . . .
    POP  ACC
    POP  PSW
    RETI
```

Далее в этой книге мы часто будем использовать прерывания в программных проектах, поэтому практические аспекты реализации прерываний будут рассмотрены позже.



1.5. Параллельные порты ввода/вывода данных

Микроконтроллер 8051 имеет четыре 8-битовых порта ввода/вывода, обозначенных P0 – P3 и предназначенных для ввода или вывода информации как в форме байта, так и побитово. Каждый порт содержит управляемый регистр-защелку, входной буфер и выходной драйвер. Обобщенная функциональная схема порта ввода/вывода показана на рис. 1.9.

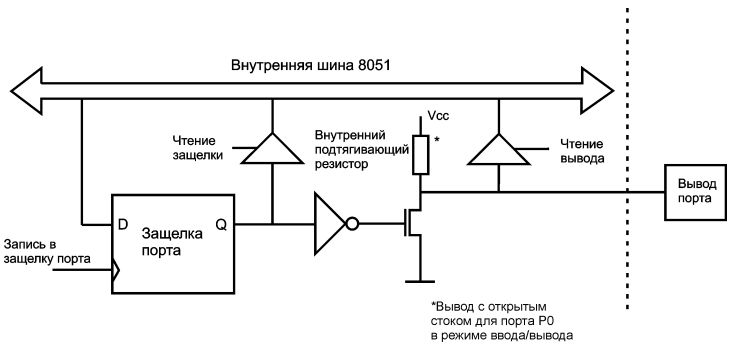


Рис. 1.9. *Обобщенная функциональная схема порта ввода/вывода*

Порт P0 является двунаправленным, а порты P1 – P3 – квазидвунаправленными, при этом каждая линия портов может быть использована независимо от остальных для ввода или вывода. По сигналу сброса в регистры-защелки всех портов автоматически записываются единицы, что переводит выходные линии портов в режим ввода. Все порты могут быть использованы для организации ввода/вывода информации по двунаправленным линиям передачи. Однако порты P0 и P2 не могут быть использованы для этой цели в случае, если система имеет внешнюю память, связь с которой организуется через общую разделяемую шину адреса/данных, работающую в режиме временного мультиплексирования.

Схема функционирования одной линии ввода/вывода порта P0 показана на рис. 1.10.

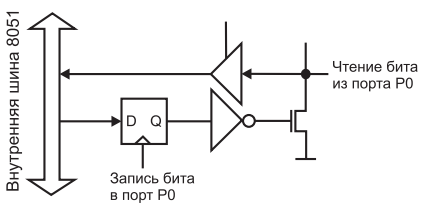


Рис. 1.10. *Функциональная схема одной линии порта P0*

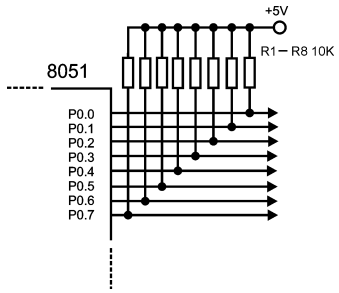
Выходные драйверы порта P0 реализованы с открытым стоком, как это видно из рис. 1.10, поэтому напрямую использовать этот порт для ввода/вывода данных нельзя. Для работы порта P0 в режиме ввода/вывода данных нужно к выводам порта присоединить резисторы номиналом 10 К (рис. 1.11).

Порт P0, сконфигурированный так, как показано на рис. 1.11, может использоваться для ввода/вывода данных. Вот фрагмент программного кода, в котором байт, находящийся в аккумуляторе, инвертируется и с определенной задержкой (функция DELAY) выводится в порт P0:

```
    MOV  A, #0FFH
AGAIN:
```

```
MOV  P0, A
ACALL    DELAY
CPL  A
SJMP AGAIN
```

Рис. 1.11. *Использование линий порта P0 для ввода/вывода*

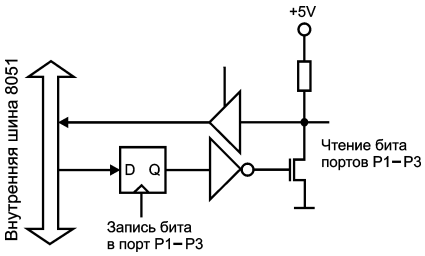


Для чтения порта P0 нужно вначале записать в него единицы, а затем прочитать входное значение. Следующий фрагмент программного кода демонстрирует, как это делается:

```
MOV A, #0FFH      ;запись в аккумулятор A значения FF
MOV P0, A         ;установить режим ввода с выводов порта P0
MOV A, P0         ;прочитать входные данные
```

Аппаратная архитектура портов P1 – P3 несколько отличается от той, что реализована для P0. Функциональная схема одной линии ввода/вывода портов P1 – P3 показана на рис. 1.12.

Рис. 1.12. *Функциональная схема одной линии портов P1 – P3*



Выходные каскады портов P1 – P3 используют внутренние подтягивающие резисторы, соединенные со стоком выходных транзисторов, поэтому необходимость во внешних резисторах, как в случае порта P0, отпадает. Выходные драйверы портов 0 и 2, а также входной буфер порта 0 используются при обращении к внешней памяти. При этом через порт 0 в режиме временного мультиплексирования сначала выводится младший байт адреса, а затем выдается или принимается байт данных. Через порт 2 выводится старший байт адреса в тех случаях, когда разрядность адреса равна 16 бит.

Порты P1, P2 и P3 могут использоваться, так же как и порт P0, для ввода/вывода данных, при этом не нужно устанавливать подтягивающие резисторы, как это делается для порта P0. Как и для порта P0, при чтении данных следует записать в соответствующие разряды портов P1, P2 и P3 единицы.

Все выводы порта P3 могут быть использованы для реализации альтернативных функций. Эти функции могут быть задействованы путем записи единицы в соответствующие биты регистра-защелки (P3.0 – P3.7) порта P3.

Перечень альтернативных функций порта P3 приведен в табл. 1.12.

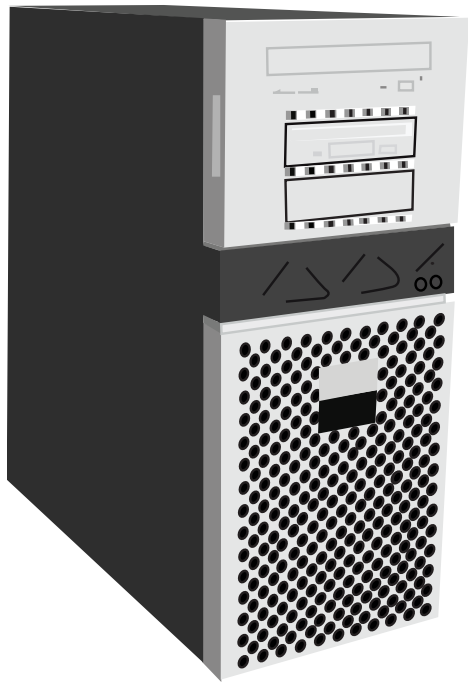
Таблица 1.12.
Альтернативные функции порта P3

Обозначение	Бит порта	Описание
RD	P3.7	Чтение. Активный сигнал низкого уровня формируется аппаратно при обращении к внешней памяти данных
WR	P3.6	Запись. Активный сигнал низкого уровня формируется аппаратно при обращении к внешней памяти данных
T1	P3.5	Вход таймера/счетчика 1
T0	P3.4	Вход таймера/счетчика 0
INT1	P3.3	Вход запроса внешнего прерывания INT1. Воспринимается сигнал низкого уровня или перепад 1–0
INT0	P3.2	Вход запроса внешнего прерывания INT0. Воспринимается сигнал низкого уровня или перепад 1–0
TXD	P3.1	Выход передатчика последовательного порта
RXD	P3.0	Вход приемника последовательного порта

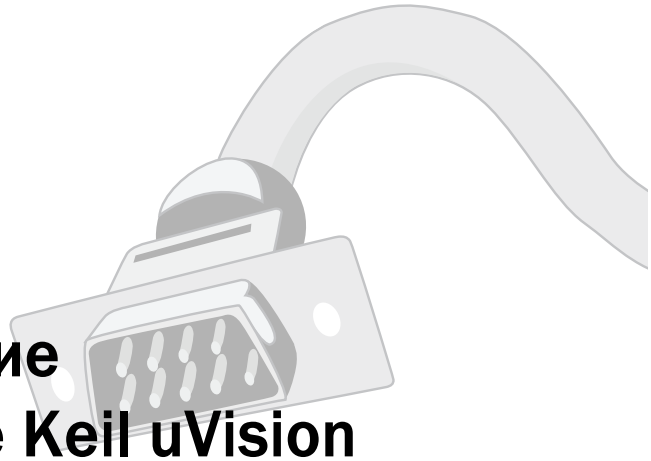
Обращение к портам ввода/вывода возможно с использованием команд, оперирующих с байтом, отдельным битом или с произвольной комбинацией битов. При этом в тех случаях, когда порт является одновременно и операндом и местом назначения результата, устройство управления автоматически реализует специальный режим, который называется «чтение-модификация-запись». Этот режим обращения предполагает ввод сигналов не с внешних выводов порта, а из его регистра-защелки, что позволяет исключить неправильное считывание ранее выведенной информации. Этот механизм обращения к портам реализован в командах:

- ANL – «логическое И», например: ANL P1, A;
- ORL – «логическое ИЛИ», например: ORL P2, A;
- XRL – «исключающее ИЛИ», например: XRL P3, A;
- JBC – переход с последующей очисткой бита, например: JBC P1.1, LABEL;
- CPL – инверсия бита, например: CPL P3.3;
- INC – инкремент порта, например: INC P2;
- DEC – декремент порта, например: DEC P2;
- DJNZ – декремент порта и переход, например: DJNZ P1, LABEL;
- MOV PX.Y, C – передача бита переноса C в бит Y порта X;
- SET PX.Y – установка бита Y порта X;
- CLR PX.Y – сброс бита Y порта X.

При работе с портами ввода/вывода микроконтроллеров 8051 необходимо учитывать, что выходные драйверы этих портов могут обеспечить максимальный ток в нагрузке порядка нескольких миллиампер, поэтому при подключении сильноточных выходных нагрузок следует использовать буферные схемы.



Программирование и отладка в среде Keil uVision



2 Программирование и отладка в среде Keil uVision

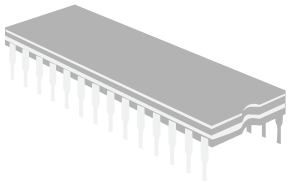
Для систем с микроконтроллерами 8051 разработаны мощные инструментальные средства, позволяющие снизить до минимума время разработки и отладки программного обеспечения. Среди прочих лидирующие позиции занимают программные средства, разработанные фирмой Keil. Инструментальные средства этой фирмы включают целый ряд мощных приложений, таких как компилятор языка C для микроконтроллеров 8051, известный под названием Keil C51, макроассемблер A51, совместимый с ASM-51, и наконец, удобная графическая оболочка для разработки и отладки программ Keil uVision.

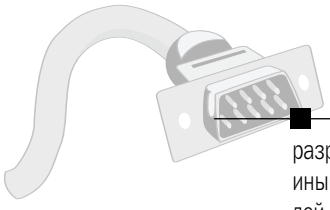
Среда разработки Keil uVision (в настоящее время используются версии 2 и 3) позволяет создавать сколь угодно сложные проекты, состоящие из разных модулей, написанных как на C, так и на языке ассемблера, подключать библиотеки функций и т.д. Кроме того, эта среда включает удобный отладчик (симулятор), позволяющий оценить и быстро проверить работоспособность программы. Хотя при разработке программ можно обойтись только командной строкой, откомпилировав и собрав программу вне среды Keil uVision, графическая среда разработки все же более удобна, поскольку позволяет автоматизировать сборку программ и проверить их работоспособность с помощью отладчика.

Для разработки и отладки программ в среде Keil желательно иметь установленную графическую среду Keil uVision версии 2 или 3 и компиляторы C51 и A51. Можно использовать демонстрационные версии этих программных инструментов, доступные для скачивания на сайте www.keil.com. Все примеры, приведенные в этой главе, будут работать в демонстрационной версии среды Keil. Более того, для разработки программ на языке C в среде Keil uVision можно использовать и свободно распространяемый компилятор SDCC, который разработан в рамках открытого лицензионного соглашения GNU и не имеет ограничений по размеру исполняемого программного кода. Несмотря на некоторые отличия в синтаксисе, многие (если не большинство) из принципов программирования систем с 8051, используемые в Keil C51, работают и в SDCC.

2.1. Преимущества и недостатки языков высокого уровня

При создании программного обеспечения для однокристальных микроконтроллеров в настоящее время наиболее широко используются языки высокого уровня C, Pascal и Basic. Использование этих средств позволяет в короткие сроки разрабатывать более-менее серьезные проекты, значительно улучшает алгоритмизацию программной части и дает возможность разработчику больше сосредоточиться на структуре самой программы, чем на инструменте





разработки. Языки высокого уровня скрывают от программиста детали операций с теми или иными аппаратными ресурсами, предлагая широкий набор встроенных и библиотечных модулей для работы с аппаратной частью разрабатываемой системы.

Язык С является наиболее мощным средством среди высокоуровневых средств разработки, поскольку, обладая очень хорошей структурируемостью, позволяет легко адаптировать алгоритм функционирования обобщенной системы с 8051 к конкретной аппаратной части. В настоящее время он является одним из наиболее популярных языков программирования для микроконтроллеров семейства 8051/8052, поскольку позволяет решать сложные задачи при максимальной эффективности и минимальном размере программного кода.

Являясь строго типизированным, язык С позволяет избегать в процессе разработки и отладки программ многих ошибок, связанных с неправильным применением типов данных. К тому же он включает в себя множество операторов, облегчающих процесс программирования.

Практически единственный недостаток языка С является одновременно и его преимуществом: абстрагирование (в известном смысле) от аппаратных средств, что может затруднить в целом ряде случаев процесс разработки программного обеспечения при применении в системе оригинальных аппаратных решений или специфичного оборудования. Тем не менее подобные проблемы очень успешно решаются, если использовать комбинацию программных модулей на языке С и ассемблере. Язык ассемблера более близок к аппаратуре и позволяет разрешить проблемы, которые сложно решить средствами С. Разумное сочетание программы на языке С и процедур, разработанных на языке ассемблера, позволяет решить любую задачу, связанную с разработкой программного обеспечения для 8051-совместимых систем.

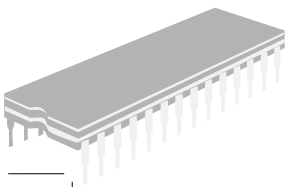
Реализация языка С фирмой Keil, более известная под названием С51, является полностью совместимой с ANSI-стандартом. При этом компилятор С51 не является инструментом общего применения, адаптированным для применения с микроконтроллерами 8051. Напротив, он специально разработан для создания эффективного и быстрого кода при работе с данным типом микроконтроллеров, позволяя разрабатывать очень быстрые программы, сравнимые по быстродействию с программами на ассемблере.

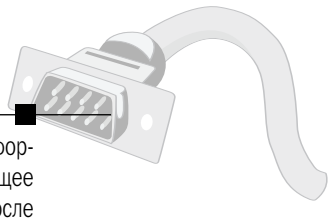
2.2. Создание программ в Keil C51

Перед тем как приступить к более глубокому изучению возможностей языка Keil C51, кратко рассмотрим все этапы создания программного кода для микроконтроллеров 8051 на примерах простых программ. Для компиляции исходных текстов в этой и последующих главах, мы будем использовать компилятор С51 версии 8, хотя все сказанное в равной мере будет относиться и к более ранним версиям данного компилятора, например 7. В качестве графической оболочки для разработки программных проектов мы будем использовать среду Keil uVision3, хотя все программы можно компилировать и в uVision2.

Здесь я хочу сделать одно важное уточнение: в большинстве случаев при упоминании о компиляторе С51 имеется в виду совокупность программ, выполняющих сборку приложения, куда обычно входят следующие программные средства:

- компилятор исходного текста программы (файл c51.exe);
- компоновщик объектных модулей или, по-другому, линкер (файл bl51.exe);
- программа-конвертер абсолютного модуля в формат загружаемого HEX-файла (файл oh51.exe);
- компоновщик библиотек (lib51.exe) и целый ряд других программ.





Для того чтобы микроконтроллер мог выполнить программу, нужно в специальном формате создать двоичный код этой программы и записать его в постоянное запоминающее устройство или, что в настоящее время более распространено, во флеш-память, откуда после включения устройства с микроконтроллером будут извлекаться машинные инструкции для выполнения.

Двоичный код программы записывается в постоянную память в формате так называемого HEX-файла, который создается после прохождения всех этапов трансляции и компоновки исходного текста программы. Чтобы лучше понять, как создается загружаемый двоичный образ программы, представим, что у нас имеется исходный текст программы, сохраненный в файле с расширением .c. В этом случае при использовании компилятора C51 последовательность всех этапов при создании одного загружаемого HEX-файла можно описать следующим образом:

- исходный текст программы компилируется в формат объектного модуля (файл с расширением .obj) с помощью программы-транслятора c51.exe;
- из полученного файла с расширением .obj компоновщик bl51.exe создает файл абсолютного объектного модуля, который можно использовать для загрузки в эмулятор или для работы с симулятором – отладчиком среды Keil uVision3. По умолчанию файл абсолютного объектного модуля не имеет расширения, хотя при желании его можно указать в командной строке;
- из абсолютного объектного модуля с помощью конвертера OBJ-HEX (исполняемый файл oh51.exe) создается загружаемый двоичный образ программы с расширением .hex, который может быть загружен для выполнения в постоянное запоминающее устройство или флеш-память системы с микроконтроллером 8051.

Замечу, что компоновщик создает и другие файлы, необходимые для отладки, но сейчас рассматривать их мы не будем. Читатели, желающие глубже изучить работу компилятора C51 и установки параметров командной строки, смогут найти интересующую их информацию или в справочном разделе среды Keil uVision2 (3), или на сайте www.keil.com.

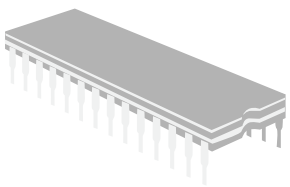
Рассмотрим на простом примере, как создается исполняемый код для 8051 с помощью компилятора C51.

Предположим, что имеется исходный текст программы, записанный в файл Program.c. Этот программный файл можно откомпилировать из командной строки, запустив исполняемый файл c51.exe, который размещается в каталоге \Keil\C51\Bin, и передав ему в качестве параметра путь к файлу Program.c. Если процесс компиляции прошел без ошибок, то будет создан объектный файл Program.obj (если не указано иное имя объектного модуля).

Созданный объектный файл не является обычным двоичным представлением исходного C-файла. Этот файл имеет специальный формат, позволяющий размещать информацию о символических именах, ссылки на библиотеки функций, отладочную информацию и т.д.

Двоичный код, записанный в объектном файле, не является абсолютным и может впоследствии располагаться компоновщиком по произвольному адресу, при этом ссылки на внешние переменные и функции устанавливаются компилятором по нулевому адресу. Объектный файл нельзя записать в память программ микроконтроллера и выполнить, для этого его нужно скомпоновать программой-компоновщиком bl51.exe. Тем не менее объектный файл можно использовать для отладочных целей в симуляторе программ среды Keil uVision3.

Полученный файл объектного модуля Program.obj далее преобразуется в формат исполняемого модуля при помощи компоновщика bl51.exe. Компоновщик создает исполняемый

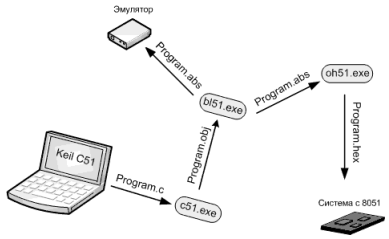


файл в формате Keil OMF51, который совместим со стандартным форматом OMF51 фирмы Intel. Полученный таким образом файл называют абсолютным объектным файлом, в котором сегменты данных и кода размещены по абсолютным адресам памяти. Такой файл готов к исполнению и не содержит информацию о перемещаемых модулях или внешних ссылках. Он может использоваться, например, симулятором Keil или аппаратным эмулятором для отладки программы. Абсолютный файл, генерируемый компоновщиком, по умолчанию не имеет расширения, поэтому для него можно принять расширение .abs. Таким образом, если не указывать дополнительных параметров, то компоновщик сгенерирует абсолютный объектный файл с именем Program.

Если абсолютный файл должен быть загружен в постоянную память целевой системы для выполнения, то предварительно нужно выполнить его преобразование в формат HEX, что можно выполнить при помощи программы-конвертера oh51.exe, передав ему в качестве параметра имя Program абсолютного объектного файла.

Если, например, имеется исходный текст программы, сохраненный в файле Program.c, то процесс компиляции, компоновки и отладки этой программы с помощью Keil C51 можно проиллюстрировать рис. 2.1.

Рис. 2.1. *Этапы создания программы*



Расширение абсолютного объектного файла (.abs) при вызове компоновщика указывать не обязательно, на рис. 2.1 это показано для большей наглядности. Компиляцию, компоновку и преобразование файла можно выполнить вне среды программирования Keil uVision, используя для этого только командную строку, но если вы работаете в оболочке Keil uVision3, то все этапы разработки, описанные ранее, можно выполнить, используя более удобный графический интерфейс пользователя и соответствующие опции меню.

При использовании командной строки облегчить процесс компиляции и сборку исполняемых файлов можно, добавив переменные окружения в операционные системы Windows (2000, XP или Vista), как показано в табл. 2.1.

Таблица 2.1. *Установки переменных окружения*

Переменная окружения	Путь	Описание переменной
PATH	\KEIL\C51\BIN	Путь к исполняемым файлам (c51.exe, bl51.exe, oh51.exe и т.д.)
TMP		Путь к временным файлам, генерируемым компилятором. Если указанный путь не существует, компилятор генерирует ошибку и останавливает процесс компиляции
C51INC	\KEIL\C51\INC	Путь к каталогу с файлами заголовков
C51LIB	\KEIL\C51\LIB	Путь к каталогу с библиотечными файлами

В начале пути следует указать через двоеточие букву диска, где размещаются файлы Keil (в таблице не указана).

Рассмотрим практический пример создания загружаемого HEX-файла из файла hello.c, содержащего исходный текст, показанный далее (его можно набрать в стандартном текстовом редакторе Notepad). Если вам непонятен смысл операторов программы, не следует беспокоиться – мы их проанализируем позже, когда будем рассматривать синтаксис языка.

```
#include <stdio.h>
#include <REG52.h>

void main(void)
{
    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 1;
    printf("Hello, World!\n");
    while (1);
}
```

Запустим командный интерпретатор cmd, перейдем в каталог, где содержится файл (в данном примере это каталог C:\HELLO) и выполним компиляцию файла hello.c (рис. 2.2).

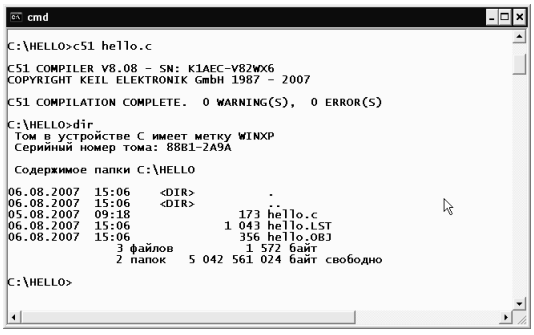


Рис. 2.2. *Результат работы компилятора*

Для компиляции файла hello.c следует ввести командную строку:

```
c51 hello.c
```

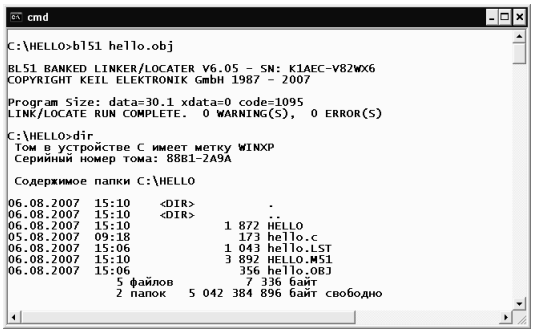
Поскольку никакие другие параметры не указаны, компилятор создает объектный файл с тем же именем, что и файл исходного текста, но с расширением .obj.

Как видно из рис. 2.2, кроме объектного файла hello.obj компилятор сгенерировал файл листинга hello.lst, в котором содержится информация об используемых сегментах программ и данных. Следующий этап – создание абсолютного объектного файла, для чего нужно выполнить команду:

```
bl51 hello.OBJ
```

Если компоновка выполнена успешно, то будет создан абсолютный объектный файл HELLO без расширения (рис. 2.3).

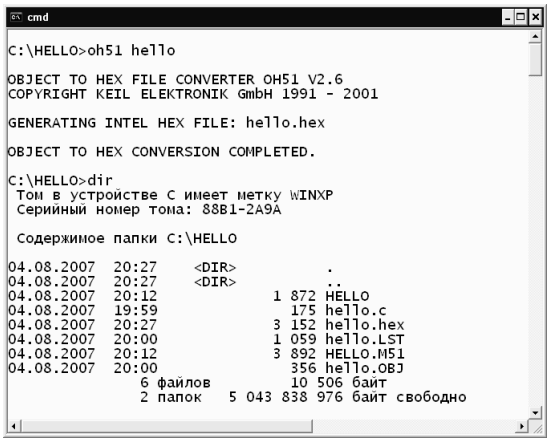
Рис. 2.3. *Результат работы компоновщика*



Из рис. 2.3 видно, что сгенерированный абсолютный объектный файл HELLO имеет размер 1872 байта. Кроме того, был создан файл HELLO.M51, в который помещается информация о размещении сегментов в памяти программ (абсолютные адреса) и адреса модулей, содержащих библиотечные функции, размещенные в этом объектном файле.

Для создания загружаемого HEX-файла нужно вызвать программу-конвертер oh51.exe, указав имя абсолютного объектного файла (рис. 2.4).

Рис. 2.4. *Результат преобразования абсолютного объектного файла*



Если преобразование выполнено успешно, в каталоге C:\HELLO появится файл hello.hex, который можно загрузить в память программ целевой системы с микроконтроллером 8051.

Этот простейший пример показывает, как можно работать с компилятором C51, используя для создания исходного текста программ обычный текстовый редактор и командную строку. Если на компьютере имеется установленная графическая оболочка Keil uVision 2 или 3, процесс создания всех модулей проекта упрощается.

Посмотрим, как выполнить цикл разработки программного кода с использованием среды Keil uVision3, используя исходный текст программы hello.c из предыдущего примера.

Запустим графическую оболочку Keil uVision3 и приступим к созданию нового проекта (рис. 2.5).

Мастер проекта попросит ввести имя файла проекта (файл с расширением .uv2 или .uv3) в диалоговом окне. Выберем каталог, в котором будет создан наш проект (в нашем случае это \HELLO) и укажем имя проекта (можно взять hello), после чего сохраним его (рис. 2.6).

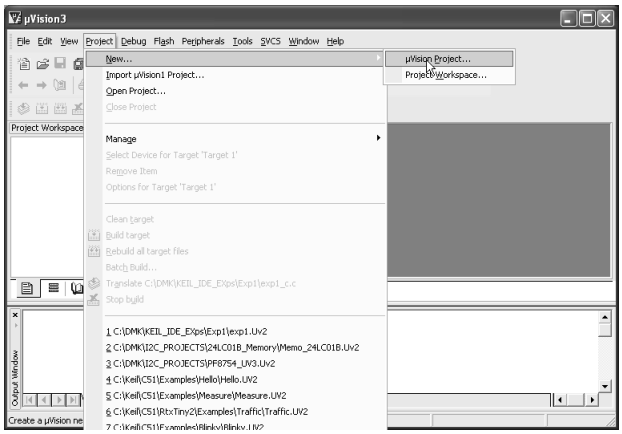


Рис. 2.5.
Создание нового проекта в Keil uVision3

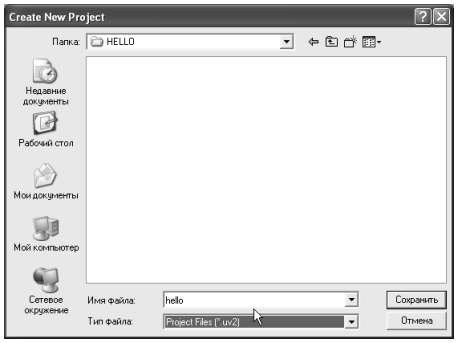


Рис. 2.6.
Окно выбора имени проекта

После того как проект сохранен, мастер проектов выводит диалоговое окно, в котором перечислены поддерживаемые Keil клоны 8051, один из которых можно выбрать для нашего проекта. Здесь можно указать либо конкретный тип микроконтроллера, если известно, с каким оборудованием будем иметь дело, либо выбрать общий тип. Для нашего проекта выберем строку **Generic** и опцию **8052 (all Variants)** – рис. 2.7.

После выбора типа устройства мастер проекта среды Keil uVision3 предложит сгенерировать код инициализации, выполняемый микроконтроллером при запуске программы (рис. 2.8).

Код инициализации в большинстве случаев пишут сами программисты для конкретного типа микроконтроллера, либо во многих случаях никакой инициализации не выполняют вообще. В нашем случае также нет необходимости выполнять дополнительные установки, поэтому откажемся от этого, нажав кнопку **Нет**.

В результате мастер проектов среды Keil uVision сгенерирует для нас пустой проект, не содержащий файлов исходных текстов (рис. 2.9).

В созданной мастером рабочей области проекта под названием **Target 1** отсутствуют какие-либо файлы исходных текстов программы, поэтому их нужно добавить вручную. В данном случае в проект нужно добавить единственный файл (назовем его hello.c) с исходным текстом программы. Вначале выберем опцию **New** в меню **File** (рис. 2.10).

В появившемся окне текстового редактора наберем исходный текст программы из предыдущего примера (рис. 2.11).

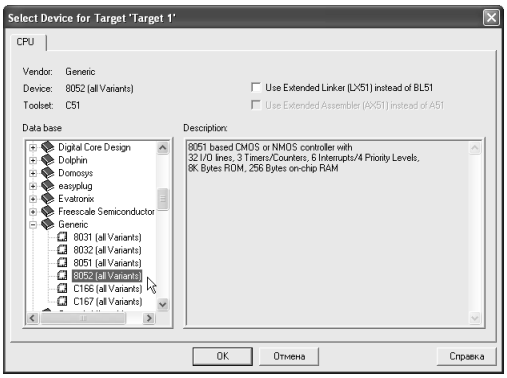


Рис. 2.7. *Выбор типа микроконтроллера*

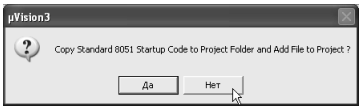


Рис. 2.8. *Окно выбора создания кода инициализации*

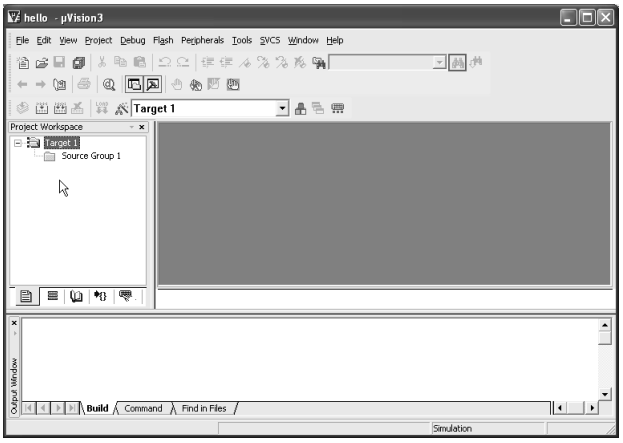


Рис. 2.9. *Окно созданного проекта*

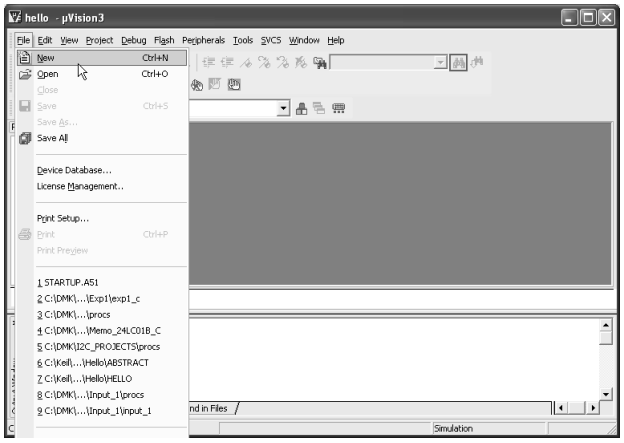


Рис. 2.10. *Добавление нового файла в рабочее пространство проекта*

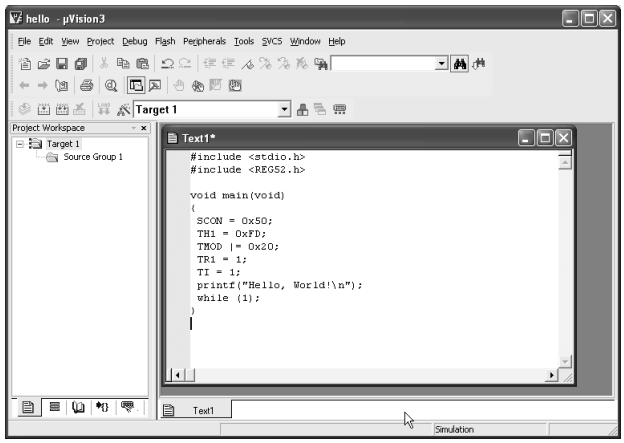


Рис. 2.11.
Создание файла исходного текста

Сохраним текст в файле hello.c в каталоге проекта. Теперь нужно включить созданный файл в проект, поскольку в проекте он еще не присутствует. Для этого выберем опцию **Source Group 1** и далее во всплывающем меню опцию **Add Files To Group 'Source Group 1'** (рис. 2.12).

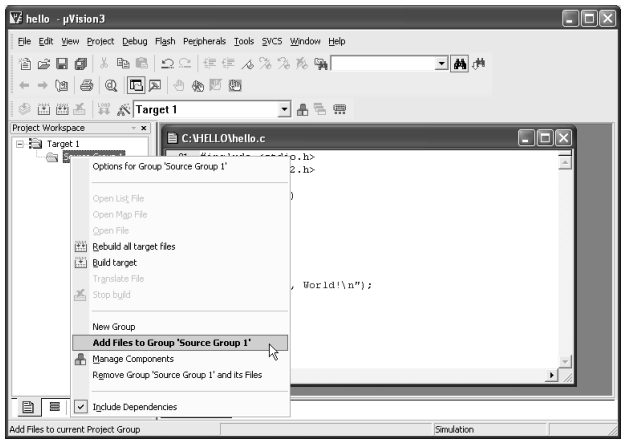


Рис. 2.12.
Включение файла в проект

После добавления файла hello.c в проект можно выполнить компиляцию и сборку программы, выбрав опцию **Build target** из меню **Project** (рис. 2.13).

Если сборка программы выполнена успешно, то ее работоспособность можно проверить, запустив на выполнение в симуляторе Keil. Несколько забегая вперед, сделаю одно замечание: все библиотечные функции Keil C51, такие как printf, scanf, getchar и т.д., работают с вводом/выводом через последовательный порт микроконтроллера. Это означает, например, что хорошо знакомая любому программисту функция prinf классического ANSI C в компиляторе Keil C51 (как, впрочем, и во всех остальных подобных инструментах разработки для микроконтроллеров) выводит данные в последовательный порт микроконтроллера 8051, а не на экран консоли. Для того чтобы увидеть вывод этой функции в форме текста на экране дисплея, в Keil uVision3 используется метод эмуляции ввода/вывода программ. Точно так же стандартная библиотечная функция C scanf из Keil C51 ожидает приема данных с последовательного порта, а не с консоли, как в обычных компиляторах C.

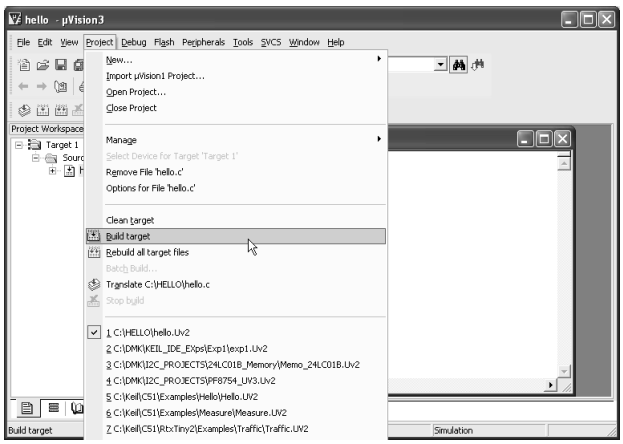


Рис. 2.13. *Компиляция и сборка исполняемого модуля*

Для отладки (симуляции) работы полученного абсолютного объектного файла программы следует установить режим отладки, для чего в меню **Debug** выберем опцию **Start/Stop Debug Session** (рис. 2.14).

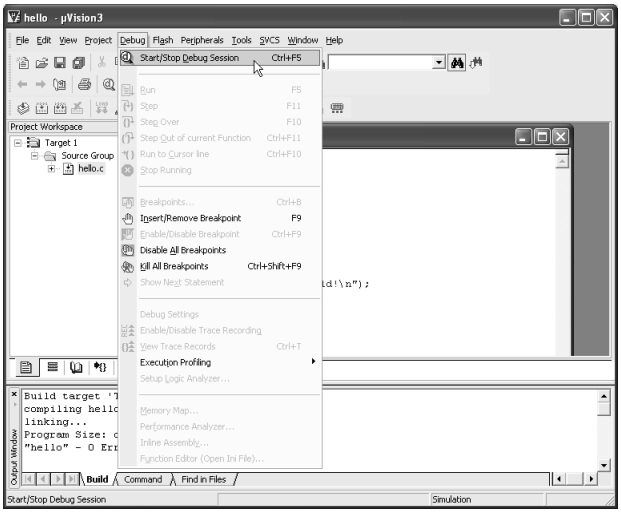


Рис. 2.14. *Запуск отладчика Keil*

После запуска отладчика курсор будет установлен на первый оператор программы. Здесь можно использовать либо пошаговый режим работы, либо выполнить программу в обычном режиме. Во всех примерах мы будем выполнять программы в обычном режиме.

Для запуска программы в отладчике нужно выбрать в меню **Debug** опцию **Run** или нажать клавишу **F5** быстрого вызова. Можно воспользоваться также и пиктограммой **Run**, размещенной в левом верхнем углу панели быстрых вызовов (рис. 2.15).

Для просмотра результата выполнения программы в меню **View** нужно последовательно выбрать опции **Serial Window** и **UART #0** (рис. 2.16).

В раскрывающемся окне виртуального терминала мы увидим результат работы программы – строку «Hello, world» (рис. 2.17).

Таким образом, мы прошли полный цикл разработки и симуляции (отладки) программы в среде Keil uVision3. Однако HEX-файл для загрузки в целевую систему с установленным

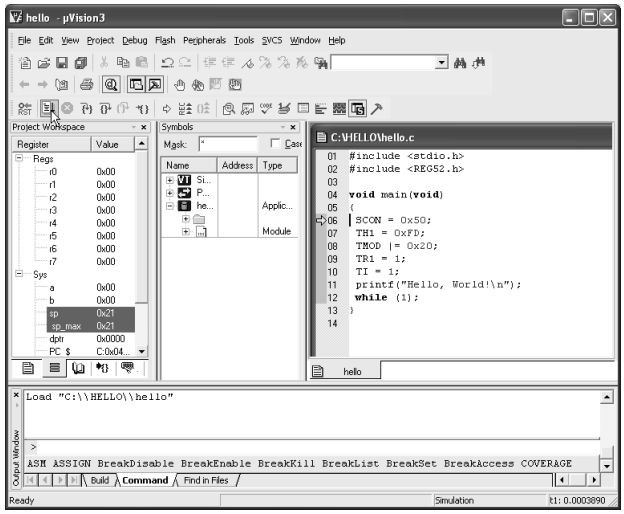


Рис. 2.15.
Запуск программы в отладчике

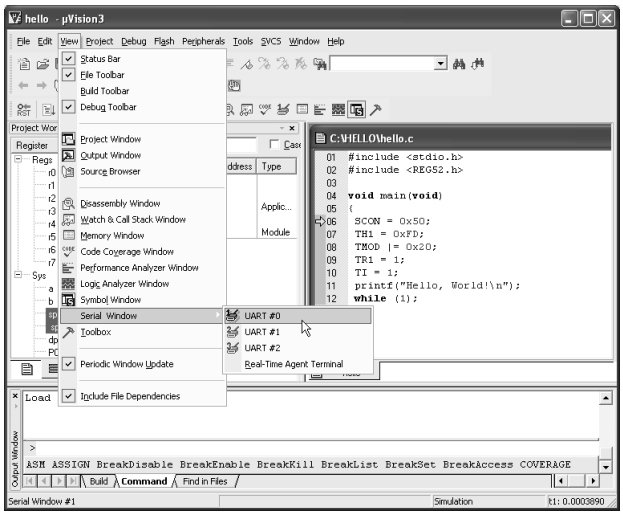


Рис. 2.16.
Выбор эмулятора последовательного порта

микроконтроллером 8051 пока еще не создан. Напомню, что для симуляции или эмуляции программы Keil uVision3 использует абсолютный объектный файл (в данном случае это файл hello без расширения), который загружается в симулятор, как показано в нижней части рис. 2.17.

В предыдущем примере для генерации HEX-файла из командной строки мы использовали программу-конвертер oh51.exe. Чтобы сделать то же самое в среде Keil uVision3, нужно установить соответствующую опцию в нашем проекте.

Выберем в меню **Project** опцию **Options for Target 'Target1'** (рис. 2.18).

В появившемся диалоговом окне перейдем на вкладку **Output** и отметим флажок **Create HEX File** (рис. 2.19).

Теперь при повторной сборке проекта в каталоге C:\HELLO появится файл hello.hex, который можно загрузить в память программ целевой системы для выполнения. Если такую систему соединить, например, с последовательным портом персонального компьютера, на котором

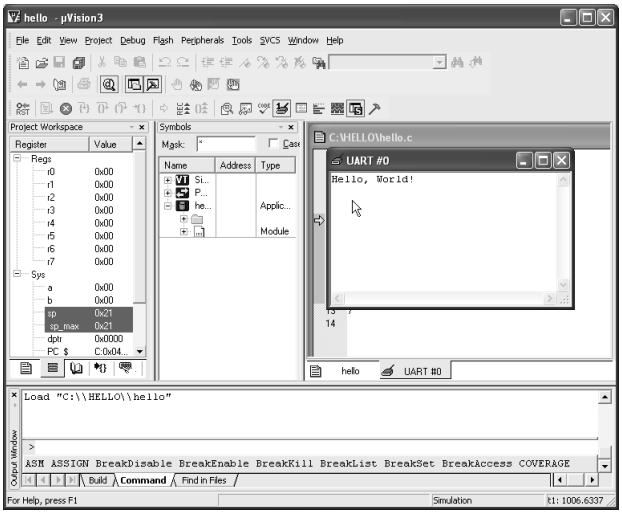


Рис. 2.17. *Вывод результата работы программы*

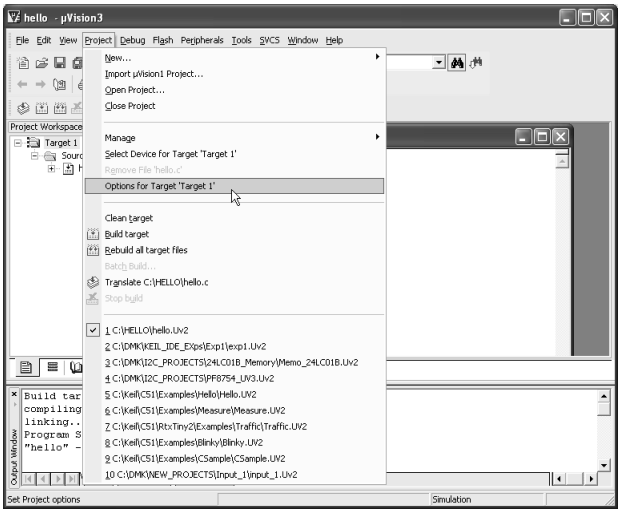


Рис. 2.18. *Выбор установки параметров проекта — шаг 1*

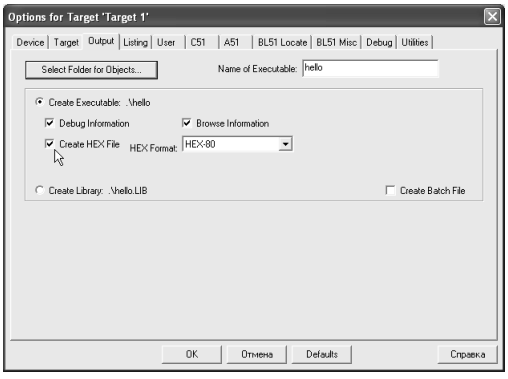
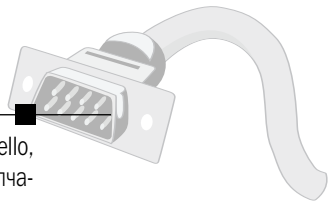


Рис. 2.19. *Выбор установки параметров проекта — шаг 2*



запустить программу-терминал, то на экране дисплея будет отображено сообщение «Hello, World!» (скорость обмена порта должна быть 9600 бод, что является параметром по умолчанию для Windows-систем).

На этом первое знакомство с компилятором Keil C51 и средой программирования Keil uVision3 можно считать законченным. Рассмотренный материал будет полезен при запуске и отладке примеров программного кода в этой и последующих главах.

Перейдем к более глубокому изучению языка Keil C51. Синтаксис языка мы рассмотрим в контексте практических примеров программирования, чтобы лучше понять и оценить возможности компилятора. Для разработки примеров программного кода будем использовать среду программирования Keil uVision3, основы работы с которой мы только что рассмотрели на простейших примерах.

2.3. Синтаксис Keil C51

Язык программирования Keil C51 разработан как инструмент структурно-модульного программирования. Любая программа, написанная на C51, состоит как минимум из одного модуля, хотя может включать их несколько. При этом исходный текст каждого модуля сохраняется в отдельном файле и компилируется отдельно.

Выполнение программы всегда начинается с подпрограммы, содержащей точку входа с именем main. Эту подпрограмму далее будем называть основной программой. Основная программа может вызывать другие подпрограммы, размещенные как в одном с ней, так и в других модулях. Программный проект может включать несколько объектных модулей, написанных, возможно, на разных языках программирования, но удовлетворяющих соглашениям о передаче параметров и возвращаемых значениях, принятых в Keil C51.

Все программы в Keil C51 должны разрабатываться с учетом требований синтаксиса этого языка, который мы вкратце рассмотрим далее.

2.3.1. Символы, ключевые слова и идентификаторы

Для создания ключевых слов и идентификаторов в Keil C51 используются прописные и строчные буквы английского алфавита, цифры и символ подчеркивания, при этом компилятор чувствителен к регистру букв. Например, идентификаторы bytes и Bytes в языке C51 считаются разными. Кроме того, в C51, как и в стандартном ANSI C, используются специальные символы, которые служат для формирования операторов языка и организации вычислений. Все эти символы перечислены в табл. 2.2.

Таблица 2.2. *Специальные символы языка C51*

Символ	Наименование	Символ	Наименование
,	Запятая	)	Круглая скобка правая
.	Точка	(	Круглая скобка левая
;	Точка с запятой	}	Фигурная скобка правая
:	Двоеточие	{	Фигурная скобка левая
?	Вопросительный знак	<	Меньше
'	Апостроф	>	Больше
!	Восклицательный знак	[	Квадратная скобка
|	Вертикальная черта	]	Квадратная скобка

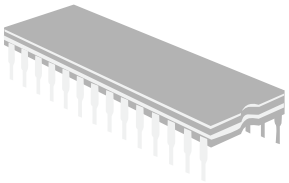


Таблица 2.2. *Специальные символы языка C51 (окончание)*

Символ	Наименование	Символ	Наименование
/	Дробная черта	#	Номер
\	Обратная черта	%	Процент
~	Тильда	&	Амперсанд
*	Звездочка	^	Исключающее ИЛИ
+	Плюс	=	Равно
–	Минус	”	Кавычки

При разработке программ очень часто применяются так называемые управляющие и разделительные символы, к которым относятся пробел, символы табуляции, перевода строки, возврата каретки, символы новой страницы и новой строки. Основное назначение этих символов состоит в разделении лексических единиц языка, к которым относятся ключевые слова, константы, идентификаторы и т.д. При этом последовательность разделительных символов (например, последовательность пробелов или символов табуляции) интерпретируется компилятором как один символ.

Особую группу символов языка C51 составляют так называемые управляющие последовательности, представляющие собой специальные символьные комбинации, используемые в функциях ввода и вывода информации. Первым символом управляющей последовательности является обратный слеш (\), за которым следует комбинация латинских букв и цифр.

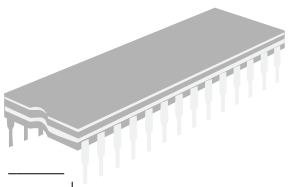
Список управляющих последовательностей приведен в табл. 2.3.

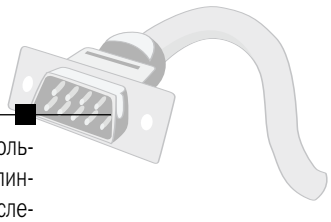
Таблица 2.3. *Управляющие последовательности языка C51*

Управляющая последовательность	Наименование	Шестнадцатеричный код
\a	Звонок	007
\b	Возврат на шаг	008
\t	Горизонтальная табуляция	009
\n	Переход на новую строку	00A
\v	Вертикальная табуляция	00B
\r	Возврат каретки	00D
\f	Новая страница	00C
\”	Кавычки	022
\’	Апостроф	027
\0	Ноль-символ	000
\\	Обратная дробная черта	05C
\000	Восьмеричный код ASCII- или ANSI-символа	–
\xHHH	Шестнадцатеричный код ASCII- или ANSI-символа	HHH

Например, управляющие последовательности \000 и \xHHH позволяют представить символ из кодовой таблицы ASCII или ANSI в виде последовательности восьмеричных или шестнадцатеричных цифр соответственно. Символ возврата каретки, например, можно представить одной из последовательностей \r, \015 или \x00D.

В том случае, если обратный слеш предшествует символу, не принадлежащему списку табл. 2.2 и не являющемуся цифрой, то слеш игнорируется, а сам символ представляется как





литеральный, например, символ «\c» представляется символом «c» в строковой или символьной константе. Обратный слеш часто используется как символ продолжения при записи длинных строк. Если за ним следует символ возврата каретки, то оба символа игнорируются, а следующая строка является продолжением предыдущей.

Идентификаторы языка C51 используются для определения имени переменной, подпрограммы, символической константы или метки оператора. Длина идентификатора может достигать 255 символов, но транслятор использует только первый 31 символ.

В качестве идентификатора может быть использована любая последовательность строчных или прописных букв латинского алфавита и цифр, а также символов подчеркивания. Идентификатор может начинаться с любой буквы или с символа «_» за исключением цифры, при этом компилятор различает регистр букв.

Идентификатор создается при объявлении переменной, функции, структуры, объединения и т.д., при этом он не должен совпадать с ключевыми словами, с зарезервированными словами и с именами функций из библиотеки компилятора языка C51. Нежелательно в качестве первого символа в идентификаторе использовать символ подчеркивания, поскольку такой идентификатор может совпасть с именами системных функций или переменных, в результате чего они станут недоступными.

В языке Keil C используются те же ключевые слова, что и в стандартном ANSI C. Кроме того, здесь определен целый ряд новых ключевых слов, которые используются исключительно при программировании устройств 8051. К ним относятся следующие:

```
_at_     alien     bdata  bit      code     compact  data     far      idata
interrupt     pdata       _priority_       reentrant        sbit     sfr
sfr16     small     _task_ using    xdata
```

Ключевые слова не могут использоваться в качестве идентификаторов. Смысл приведенных выше ключевых слов мы рассмотрим более подробно далее в этой главе.

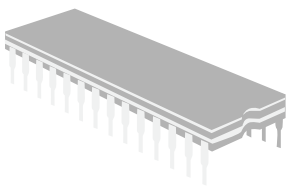
Язык Keil C51 позволяет определять константы. Как и в классическом ANSI C, в Keil C51 константы предназначены для введения чисел в состав выражений операторов языка программирования C. В отличие от идентификаторов, всегда начинающихся с буквы, константы всегда начинаются с цифры. В языке C51 используют следующие типы констант:

- целые знаковые и беззнаковые константы;
- константы с плавающей точкой;
- символьные константы и литеральные строки.

Целочисленные константы могут быть представлены в восьмеричной, десятичной или шестнадцатеричной форме записи. Десятичная константа состоит из одной или нескольких десятичных цифр, причем первая цифра не может быть нулем, иначе число будет интерпретировано как восьмеричное.

Восьмеричная константа состоит из обязательного нуля и одной или нескольких восьмеричных цифр из диапазона 0–7, а шестнадцатеричная начинается с последовательности символов 0x или 0X, после которых следуют шестнадцатеричные цифры 0–F. Примерами десятичных констант являются, например, числа 11, 127; примерами восьмеричных констант являются 013, 077; примерами шестнадцатеричных констант являются числа 0x2A, 0x1F. Целая константа, представляющая собой отрицательное значение, должна иметь впереди знак минус, например: –14, 0x2A, –057.

Любой целочисленной константе присваивается тип, определяющий преобразования, которые должны быть выполнены, если константа используется в выражениях. При этом десятичные



константы рассматриваются как знаковые числа, и им присваивается тип int (целая) или long (длинная целая) в соответствии со значением константы. Если значение константы меньше 32768, то ей присваивается тип int, если больше – тип long. Восьмеричным и шестнадцатеричным константам, в зависимости от значения, может быть присвоен тип int, unsigned int, long или unsigned long.

Константа с плавающей точкой представляется в виде действительного числа с десятичной точкой и порядком числа. Формат записи константы с плавающей точкой можно представить следующим образом:

```
[ цифры ].[ цифры ] [ E|e [+|-] цифры ]
```

Вот примеры записи констант с плавающей точкой: 75.19, 1.1E–3, –0.003, .015, –0.32E2.

Еще один тип констант – символьная константа – представляется ASCII- или ANSI-символом, заключенным в апострофы. Кроме того, управляющие последовательности также могут быть представлены в форме символьных констант, при этом они рассматриваются как один символ. Вот примеры символьных констант:

‘ ‘ – пробел;
‘S’ – буква S;
‘\n’ – символ новой строки;
‘\\’ – обратная дробная черта;
‘\v’ – вертикальная табуляция.

Символьные константы имеют тип int и при преобразовании типов могут дополняться знаком.

В языке Keil C для отображения сообщений широко используются строковые константы. Строковая константа (или, по-другому, литерал) представляет собой последовательность символов (буквы и цифры), заключенных в кавычки (“). Примерами строковых констант являются, например, записи “character input”, “Output value”. В строковых константах допускается использование пробелов.

Нужно отметить, что все управляющие символы, кавычка (“), обратный слеш (\) и символ новой строки в литеральной строке и в символьной константе представляются соответствующими управляющими последовательностями.

Символы литеральной строки обычно хранятся в памяти программ, но могут находиться и в памяти данных. В конец каждой литеральной строки компилятор добавляет нулевой символ (“\0”), который указывает на конец строки, поскольку компилятор C51, как и классический ANSI C, оперирует со строками с завершающим нулем (null-terminated string).

Компилятор интерпретирует литеральную строку как массив символов, причем общее количество элементов массива равно числу символов в строке плюс 1, поскольку учитывается завершающий символ 0.

2.3.2. Форматы данных в Keil C51

Компилятор Keil C позволяет работать с различными типами данных. Информация о типах данных и их форматах сведена в табл. 2.4.

Таблица 2.4. *Типы данных Keil C51*

Тип данных	Количество бит переменной	Количество байт	Диапазон значений
Бит (bit)	1		0 или 1
Символ со знаком (signed char)	8	1	–128 – +127

Таблица 2.4.
Типы данных Keil C51 (окончание)

Тип данных	Количество бит переменной	Количество байт	Диапазон значений
Символ без знака (unsigned char)	8	1	0 – 255
Короткое целое со знаком (signed short)	16	2	–32768 – +32767
Короткое целое без знака (unsigned short)	16	2	0 – 65535
Целое со знаком (signed int)	16	2	–32768 – +32767
Целое без знака (unsigned int)	16	2	0 – 65535
Длинное целое со знаком (signed long)	32	4	–2147483648 – +2147483647
Длинное целое без знака (unsigned long)	32	4	0 – 4294967295
Число с плавающей точкой (float)	32	4	±1.175494E–38 – ±3.402823E+38
Указатели data*, idata*, pdata*	8	1	0x00 – 0xFF
Указатели code*, xdata*	16	1	0x0000 – 0xFFFF
Указатель общего типа (generic pointer)	24	3	Тип памяти (1 байт), смещение (2 байта) 0 – 0xFFFF

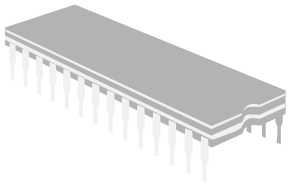
2.3.3. Специальные ключевые слова Keil C51

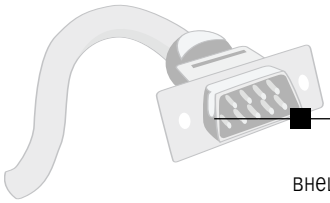
В данном разделе мы рассмотрим смысл наиболее часто используемых специальных ключевых слов Keil C51. Перечень этих ключевых слов был приведен ранее, а сейчас мы рассмотрим их более подробно, поскольку именно они определяют специфику разработки программ для 8051-совместимых систем. Рассмотрим ключевые слова в алфавитном порядке и начнем с ключевого слова `_at_`. При помощи `_at_` переменная может быть размещена по абсолютному адресу в памяти. Синтаксис `_at_` следующий:

```
тип имя_переменной константа,
```

где `тип` определяет тип переменной, `имя_переменной` указывает ее имя и `константа` указывает адрес, по которому размещается переменная. Например:

```
char xdata text[256] _at_ 0xE000;  /* массив символов по адресу xdata 0xE000 */
int xdata i1 _at_ 0x8000; /* целочисленная переменная i1 по адресу xdata 0x8000 */
volatile char xdata IO _at_ 0xFFE8; /* переменная порта ввода/вывода по адресу
xdata 0xFFE8 */
```





Здесь переменная `text`, определяющая массив из 256 символов, размещается во внешней памяти системы начиная с адреса 0xE000. Переменная `i1` размещается по адресу внешней памяти, равному 0x8000, и наконец, переменная `IO`, которая используется для хранения данных внешнего устройства ввода/вывода, должна размещаться по адресу 0xFFE8.

В последнем случае переменная объявлена с атрибутом `volatile`, что необходимо при работе с внешними портами ввода/вывода.

Ключевое слово `bit` определяет битовый тип переменной, смысл которой очевиден из названия. Битовая переменная может принимать одно из двух значений, 0 или 1. Вот примеры объявления битовых переменных:

```
bit myBit;
bit testBit (bit flag1, bit flag2);
```

Здесь определена простая битовая переменная `mybit` и функция `testBit`, принимающая в качестве аргументов битовые переменные `flag1` и `flag2` и возвращающая битовое значение. Для хранения битовых переменных используется внутренняя область памяти микроконтроллера 8051. Здесь имеются определенные ограничения: поскольку эта область памяти имеет размер в 16 байт, то максимальное количество переменных, которые можно разместить в этой области, не превышает 128.

Ключевое слово `bdata` используется только для объявления переменных. Его нельзя применять для объявления функций. Эта переменная адресует внутреннюю область памяти микроконтроллера 8051. Переменные, объявленные с этим ключевым словом, могут быть прочитаны или записаны с использованием битовых инструкций микроконтроллера 8051. Вот пример объявления переменной типа `bdata`:

```
unsigned char bdata bdata_var;
```

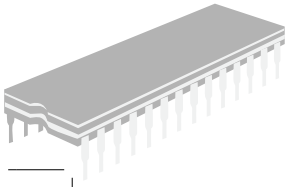
Ключевое слово `code` используется при объявлении функций и констант, размещаемых в памяти программ либо в самом микроконтроллере, либо во внешней памяти. Для доступа к данным и функциям, объявленным с этим атрибутом, применяется 16-битовая адресация, при этом следует учитывать ограничения для памяти программ (64 Кб максимум). Поскольку функции программы в подавляющем большинстве случаев размещаются в памяти программ, то очень редко возникает необходимость использовать этот атрибут. Что же касается констант, то, как правило, с этим атрибутом объявляются табличные данные. Вот пример объявления константы:

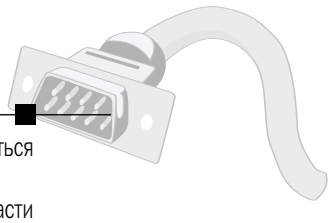
```
unsigned char code code_constant;
```

Ключевое слово `far` применяется для объявления переменных и констант, для доступа к которым следует использовать 24-битовую адресацию. Максимальное адресное пространство, доступное в этом случае, ограничивается 16 Мб. Вот примеры объявления переменных и констант, объявленных с этим атрибутом:

```
unsigned char far far_variable;
unsigned char const far far_const_variable;
```

Фактический диапазон адресов, доступный для адресации с использованием данного типа, в значительной степени зависит от архитектуры используемого кристалла, поэтому перед





использованием ключевого слова `far` в программах следует внимательно ознакомиться с аппаратными ресурсами данного типа микроконтроллера.

Ключевое слово `idata` определяет переменные, размещенные во внутренней области памяти микроконтроллера 8051, доступ к которым осуществляется с использованием 8-битовой непрямой адресации. Следует учитывать, что размер внутренней памяти программ кристалла 8051 не превышает 256 байт, при этом нижние адреса памяти типа `idata` могут перекрывать соответствующие адреса памяти, объявленной как `data`. Вот пример объявления переменной типа `idata`:

```
unsigned char idata variable;
```

Ключевое слово `interrupt` используется для объявления функции, с которой оно применяется, как обработчика прерывания. Напомню, что микроконтроллер 8051 позволяет использовать 5 типов прерываний (два внешних прерывания, два прерывания таймеров и прерывание последовательного порта). Перечень прерываний и адресов памяти программ, по которым размещаются программы обработки прерываний, показаны в табл. 2.5.

Для 8052-совместимых микроконтроллеров добавлено еще прерывание таймера 2, обработчик которого размещается по адресу 0x02B, а номер прерывания равен 5.

Таблица 2.5. *Номера прерываний для ключевого слова* `interrupt`

Номер прерывания	Описание	Адрес обработчика
0	Внешнее прерывание INT0	0x003
1	Прерывание таймера 0	0x00B
2	Внешнее прерывание INT1	0x013
3	Прерывание таймера 1	0x01B
4	Прерывание последовательного порта	0x023

Компилятор C51 поддерживает 32 прерывания с номерами от 0 до 31, которые могут размещаться в памяти программ, как показано в табл. 2.6.

Таблица 2.6. *Идентификация прерываний в Keil C51*

Номер прерывания	Адрес обработчика	Номер прерывания	Адрес обработчика
0	0x003	10	0x053
1	0x00B	11	0x05B
2	0x013	12	0x063
3	0x01B	13	0x06B
4	0x023	14	0x073
5	0x02B	15	0x07B
6	0x033	16	0x083
7	0x03B	17	0x08B
8	0x043	18	0x093
9	0x04B	19	0x09B

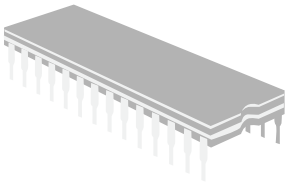


Таблица 2.6. *Идентификация прерываний в Keil C51 (окончание)*

Номер прерывания	Адрес обработчика	Номер прерывания	Адрес обработчика
20	0x0A3	26	0x0D3
21	0x0AB	27	0x0DB
22	0x0B3	28	0x0E3
23	0x0BB	29	0x0EB
24	0x0C3	30	0x0F3
25	0x0CB	31	0x0FB

Вот пример объявления программы-обработчика прерывания таймера 0:

```
void timer0ISR (void) interrupt 1 using 1  {
. . .
}
```

Атрибут `interrupt` вынуждает компилятор сгенерировать специальный программный код, в котором выполняются следующие действия:

- если в обработчике используются какие-либо регистры из ACC, B, DPH, DPL или PSW, то их содержимое сохраняется в стеке;
- если не указано ключевое слово `using`, определяющее банк регистров, используемый в обработчике, то рабочие регистры основной программы также сохраняются в стеке;
- перед выходом из обработчика прерывания содержимое предварительно сохраненных в стеке регистров восстанавливается к состоянию перед вызовом прерывания;
- программа-обработчик заканчивается командой RETI.

Рассмотрим несложный пример программы, в которой используется обработчик прерывания таймера 0. Исходный текст программы представлен ниже:

```
#include <REG52.H>

int cnt = 0;
void t0ISR (void) interrupt 1 using 1
{
    TF0 = 0;
    cnt++;
    if (cnt > 100)
    cnt = 0;
}

void main(void)
{
    TH0 = 0;
    TL0 = 0;
    TMOD |= 0x1;
    TR0 = 1;
    ET0 = 1;
    EA = 1;
```

```
    while(1);
}
```

В этой программе обработчик прерывания таймера 0 `t0ISR` каждый раз при возникновении прерывания инкрементирует переменную `cnt` до `100`, после чего сбрасывает ее в `0`. Дизассемблированный код обработчика прерывания показан ниже:

```
      6: void t0ISR (void) interrupt 1 using 1{
C:0x009A     C0E0      PUSH      ACC(0xE0)
C:0x009C     C0D0      PUSH      PSW(0xD0)
      7: TF0 = 0;
C:0x009E     C28D      CLR       TF0(0x88.5)
      8: cnt++;
C:0x00A0     0509      INC       0x09
C:0x00A2     E509      MOV       A,0x09
C:0x00A4     7002      JNZ       C:00A8
C:0x00A6     0508      INC       cnt(0x08)
      9: if (cnt > 100)
C:0x00A8     D3        SETB      C
C:0x00A9     9464      SUBB      A,#0x64
C:0x00AB     E508      MOV       A,cnt(0x08)
C:0x00AD     6480      XRL       A,#P0(0x80)
C:0x00AF     9480      SUBB      A,#P0(0x80)
C:0x00B1     4006      JC        C:00B9
     10:    cnt = 0;
C:0x00B3     750800    MOV       cnt(0x08),#0x00
C:0x00B6     750900    MOV       0x09,#0x00
     11: }
     12:
C:0x00B9     D0D0      POP       PSW(0xD0)
C:0x00BB     D0E0      POP       ACC(0xE0)
C:0x00BD     32        RETI
```

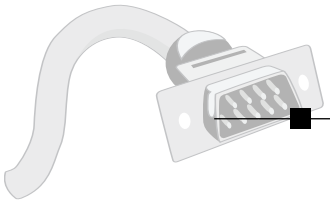
Из дизассемблированного фрагмента видно, что для организации вычислений в программе-обработчике используется регистр-аккумулятор, при этом в процессе вычислений возможны ситуации, когда биты слова состояния `PSW` микроконтроллера могут быть изменены. Чтобы предотвратить потерю данных в аккумуляторе и регистре `PSW`, которые, возможно, использовались в основной программе, первые две команды

```
PUSH     ACC
PUSH     PSW
```

программы-обработчика сохраняют содержимое этих регистров в стеке. Перед выполнением последней команды обработчика прерывания, которой всегда является `RETI`, содержимое стека восстанавливается командами

```
POP PSW
POP ACC
```

При написании программ-обработчиков прерываний следует учитывать следующие особенности:



- никакие параметры в программу-обработчик не передаются. Компилятор выдаст ошибку, если подобное будет обнаружено;
- функции-обработчики прерываний не возвращают значений, поэтому объявляются с атрибутом void;
- функцию-обработчик прерывания нельзя вызывать напрямую из программы, поскольку это может привести к краху системы из-за разрушения содержимого программных регистров, счетчика команд и стека.

Ключевое слово `pdata` используется для указания переменной, размещенной во внешней памяти данных микроконтроллера 8051 с размером, не превышающим 256 байт. Для доступа к переменным в такой области памяти используется непрямая 8-битовая адресация. Пример объявления переменной с атрибутом `pdata`:

```
unsigned char pdata variable;
```

Ключевое слово `sbit` используется очень часто в программах 8051 для доступа к отдельным битам специальных регистров (SFR). Этот атрибут применяется и в файлах заголовков Keil C51. Пример использования ключевого слова `sbit`:

```
sbit EA = 0xAF;
```

Этот вариант может использоваться и с другой формой записи. Например, для доступа к биту `0` порта `P1` можно создать переменную `bit0` следующим образом:

```
sbit bit0 = P1^0;
```

Подобный вариант объявления битовой переменной, ссылающейся на определенный бит специального регистра, мы будем часто использовать в примерах программного кода. Следует сказать, что переменные данного типа нельзя объявлять в теле функции, они должны быть объявлены вне функции, в которой используются.

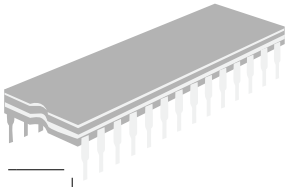
Последнее ключевое слово, которое мы рассмотрим, – это `xdata`. Оно используется только для объявления переменных, размещенных во внешней памяти системы 8051 с применением 16-битовой адресации. Внешняя память, доступная при такой адресации, не превышает 64 Кб. Вот пример объявления переменной типа `xdata`:

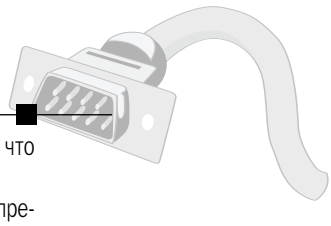
```
unsigned char xdata variable;
```

Более подробная информация об использовании ключевых слов содержится в документации фирмы Keil на компилятор C51. Кроме того, описание базовых возможностей C51 приводится во встроенной справочной документации компилятора C51.

2.3.4. Операторы и выражения в Keil C51

Выражение в языке C51 – это комбинация знаков операций и операндов, результатом которой является определенное значение, при этом знаки операций определяют действия, которые следует выполнить над операндами. Каждый операнд в выражении также может быть выражением. При вычислении значения выражения учитываются как приоритеты операций, так и изменения порядка выполнения операций при наличии разделителей в виде круглых





скобок. Все выражения в Keil C51 формируются и вычисляются по тем же правилам, что и в ANSI C, поэтому останавливаться на этом подробно мы не будем.

Единственный существенный момент, на который хотелось бы обратить внимание, – преобразование типов при вычислении выражений, поскольку во многих случаях именно неправильное выполнение такого преобразования является источником ошибок в программах, поэтому при выполнении операций над разными типами данных следует тщательно отслеживать ситуации возможного переполнения или потери значения.

В языке Keil C51 используются следующие арифметические операции:

- суммирование (оператор +);
- вычитание (оператор –);
- умножение (оператор *);
- деление (оператор /);
- вычисление остатка от целочисленного деления (оператор %).

Кроме того, в C51 также определено несколько унарных арифметических операций, которые могут выполняться над одним операндом:

- '–' – изменение знака операнда на противоположное;
- ++ – увеличение (инкремент) значения операнда на единицу;
- –– – уменьшение (декремент) значения операнда на единицу.

Над операндами можно осуществлять те же логические операции, что и в ANSI C:

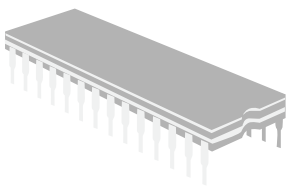
- '&&' – логическое «И»;
- '&' – побитовое логическое «И»;
- '||' – логическое «ИЛИ»;
- '|' – побитовое логическое «ИЛИ»;
- '^' – исключающее «ИЛИ».

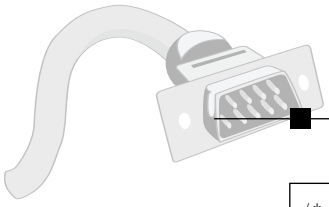
Учитывая специфику использования Keil C51, следует отметить, что побитовые операции очень широко используются при реализации ввода/вывода через порты данных микроконтроллера 8051. Логические операции в Keil C51 имеют ту же специфику выполнения, что и в обычном языке C.

Операторы присваивания, условный оператор if, switch…case и операторы цикла while и do работают в Keil C51 точно так же, как в стандартном ANSI C, поэтому останавливаться на них мы не будем. Наибольший интерес для разработчика программного обеспечения 8051-совместимых систем представляют расширения языка Keil C51, специально предназначенные для эффективного управления вводом/выводом, обработкой прерываний и манипуляциями с данными в памяти.

2.3.5. Файлы заголовков Keil C51

Исходный текст подавляющего большинства программ должен включать файл заголовка для типа микроконтроллера, используемого в разработке. В каждом таком файле определены аппаратные ресурсы микроконтроллера и даны их аббревиатуры, помогающие разработчику сделать программу более читабельной и наглядной. Многие программы используют стандартные аппаратные ресурсы, общие для всех типов микроконтроллеров, например, порты P1 – P3, регистры TCON, TMOD и т.д. В этих случаях можно использовать стандартные файлы заголовков, такие, например, как REG51.H или REG51. При работе в текстовом редакторе среды Keil uVision3 по нажатии правой кнопки мыши можно вставить файл заголовка в исходный текст программы.



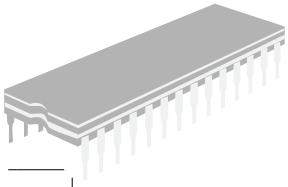


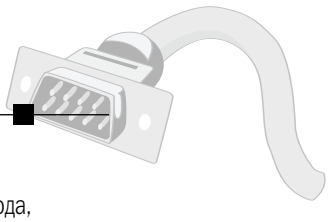
В качестве примера приведу содержимое файла заголовка REG51.H:

```
/*-------------------------------------------
REG51.H

Header file for generic 80C51 and 80C31 microcontroller.
Copyright (c) 1988-2002 Keil Elektronik GmbH and Keil Software, Inc.
All rights reserved.
-------------------------------------------*/

#ifndef __REG51_H__
#define __REG51_H__

/*  BYTE Register  */
sfr P0   = 0x80;
sfr P1   = 0x90;
sfr P2   = 0xA0;
sfr P3   = 0xB0;
sfr PSW  = 0xD0;
sfr ACC  = 0xE0;
sfr B    = 0xF0;
sfr SP   = 0x81;
sfr DPL  = 0x82;
sfr DPH  = 0x83;
sfr PCON = 0x87;
sfr TCON = 0x88;
sfr TMOD = 0x89;
sfr TL0  = 0x8A;
sfr TL1  = 0x8B;
sfr TH0  = 0x8C;
sfr TH1  = 0x8D;
sfr IE   = 0xA8;
sfr IP   = 0xB8;
sfr SCON = 0x98;
sfr SBUF = 0x99;
```

Данный файл содержит аббревиатуры аппаратных ресурсов микроконтроллера 8051, которые делают исходный текст программы более понятным. Теоретически можно обойтись и без таких заголовочных файлов, но тогда придется либо работать непосредственно с шестнадцатеричными адресами аппаратных ресурсов, что очень неудобно и нечитабельно, либо определять эти ресурсы как константы при помощи директивы define.

Может случиться так, что для микроконтроллера, который вы собираетесь программировать, файла заголовка не существует. В этом случае можно использовать исходный текст уже имеющегося файла заголовка, например REG52.H, внести в него необходимые изменения и сохранить под другим именем, но с расширением .h.





2.4. Управление вводом/выводом в Keil C51

Как известно, микроконтроллеры 8051/8052 имеют несколько портов ввода/вывода, позволяющих получать информацию от внешних устройств или отправлять данные внешним устройствам. В языке Keil C51 порты ввода/вывода интерпретируются как 8-битовые переменные, доступ к которым осуществляется стандартными способами. Это означает, например, что порту вывода можно присвоить значение или из порта ввода можно прочитать данные при помощи стандартных операторов присваивания. Кроме того, Keil C51 позволяет манипулировать с отдельными битами порта, для чего используется специальный тип переменной sbit. Рассмотрим несколько практических примеров программного кода для работы с портами ввода/вывода при использовании Keil C51.

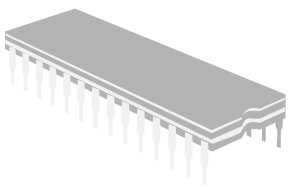
Пример 1. Инвертирование данных на выходных линиях порта `P1`. Исходный текст этой программы показан ниже:

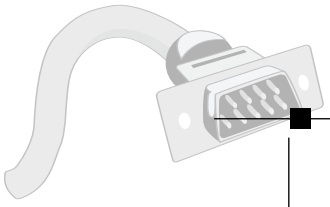
```
#include <stdio.h>
#include <REG52.H>

void main(void)
{
    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 1;

    while(1)
    {
         getchar();
         P1 = ~P1;
    }
}
```

Программа ожидает приема символа из последовательного порта, настроенного на обмен данными со скоростью 9600 бод, для чего используется таймер 1. Функция `getchar` блокирует выполнение программы, пока не будет принят символ из последовательного порта. Если это происходит, то сигналы на выводах порта `P1` микроконтроллера инвертируются (оператор `P1 = ~P1`).

Пример 2. Считывание данных с вывода `P1.7`.

```
#include <stdio.h>
#include <REG52.H>

sbit testBit = P1^7;

void main(void)
{
    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
```






```
    TR1 = 1;
    TI = 1;

    while(1)
    {
         if (testBit)
         printf("Bit P1.7 is set\n");
         getchar();
    }
}
```

Здесь для удобства работы биту `P1.7` присвоено имя `testBit`. Как и в предыдущем примере, основная программа ожидает прихода символа из последовательного порта, после чего выполняет очередное считывание бита `testBit` и вывод соответствующей строки сообщения в последовательный порт на скорости 9600 бод.

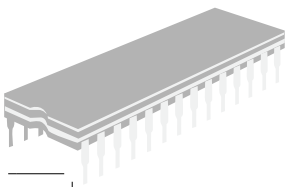
Пример 3. Использование встроенной функции `_testbit_` для считывания и очистки бита `P1.7`.

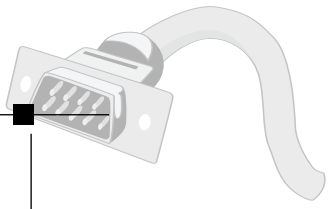
```
#include <stdio.h>
#include <intrins.h>
#include <REG52.H>

sbit tBit = P1^7;

void main(void)
{
    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 1;

    while(1)
    {
         if (_testbit_(tBit))
              printf("Bit P1.7 is set\n");
         else
              printf("Bit P1.7  is NOT set\n");
         getchar();
    }
}
```

Это модифицированный исходный текст предыдущей программы. В этом примере используется внутренняя функция `_testbit_` компилятора C51, определенная в файле заголовка `intrins.h`.

Пример 4. Анализ нескольких входных битов порта `P1`.

```
#include <stdio.h>
#include <REG52.H>
```





```
unsigned char c1;
void main(void)
{
    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 1;

    while(1)
    {
        c1 = P1 & 0xE0;
        if (c1 == 0xE0)
            printf("Bits P1.7-P1.5 are set\n");
        else
            printf("Bits P1.7-P1.5  are NOT set\n");
        getchar();
    }
}
```

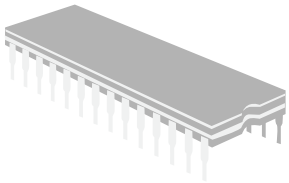
Программа анализирует биты P1.5-P1.7 и, в зависимости от их состояния, выводит соответствующее сообщение в последовательный порт, после чего ожидает прихода байта из последовательного порта для следующей итерации.

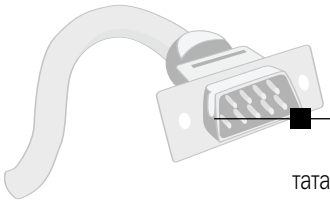
2.5. Операции с памятью

Здесь приводятся некоторые практические примеры программного кода для работы с различными типами памяти данных.

Пример 1. Вывод константной строки в последовательный порт.

```
#include <stdio.h>
#include <REG52.H>

unsigned char c1;

void main(void)
{
    const char *cstr = "Cstring";

    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 1;
    printf(cstr);
    while(1);
}
```





Пример 2. Копирование данных из области памяти типа `idata` в `xdata` и вывод результата в последовательный порт.

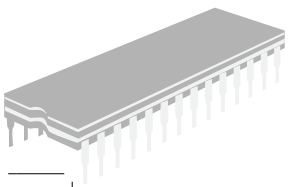
```
#include <stdio.h>
#include <string.h>
#include <REG52.H>

void main(void)
{
    idata char *src = "Data to move";
    char xdata buf[128];
    int len;

    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 1;

    len = strlen(src);
    memcpy(buf, src, len);
    printf(buf);
    while(1);
}
```

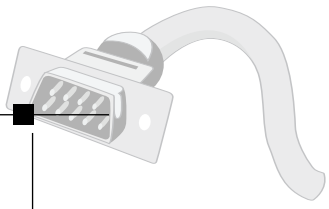
2.6. Программирование ввода/вывода через последовательный порт

Здесь показаны примеры программирования собственных функций ввода/вывода данных через последовательный порт.

Пример 1. Функция вывода символа `fputchar`, разработанная в этом примере, позволяет вывести символ в последовательный порт. Функция принимает в качестве параметра одиночный символ.

```
#include <stdio.h>
#include <string.h>
#include <REG52.H>

void fputchar (unsigned char c1)
{
    SBUF = c1;
    while (!TI);
    TI = 0;
}

void main(void)
{
    idata char src[] = "Output string";
    idata char *psrc = src;
    int len, cnt;

    SCON = 0x50;
```



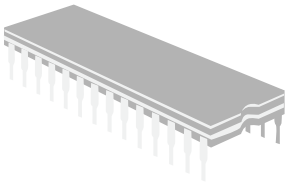


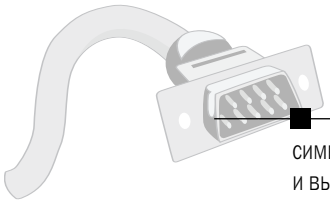
```
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 0;

    len = strlen(src);
    for (cnt = 0; cnt < len; cnt++)
    {
         fputchar(*psrc);
         psrc++;
    }
    while(1);
}
```

Пример 2. Функция вывода символа `fputs`, разработанная в этом примере, позволяет вывести строку символов в последовательный порт. Функция принимает в качестве параметра адрес строки.

```
#include <stdio.h>
#include <REG52.H>

void fputs (unsigned char *s1)
{
    while (*s1)
    {
         SBUF = *s1;
         while (!TI);
         TI = 0;
         s1++;
    }
}

void main(void)
{
    idata char src[] = "OUTPUT STRING !! !";
    idata char *psrc = src;

    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 0;

    fputs(psrc);
    while(1);
}
```

Пример 3. Функция ввода символа `getc`, разработанная в этом примере, позволяет принять символ через последовательный порт. Функция не имеет параметров и возвращает принятый



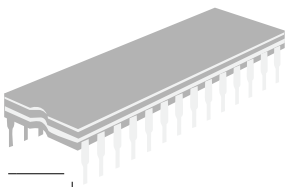


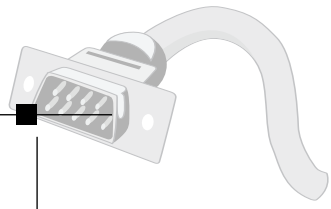
символ. Программа принимает символ из последовательного порта при помощи `getc` и выводит его обратно в порт функцией `putc`.

```
#include <stdio.h>
#include <REG52.H>

void putc (unsigned char s1)
{
    SBUF = s1;
    while (!TI);
    TI = 0;
}

unsigned char getc(void)
{
    unsigned char c1;
    while (!RI);
    RI = 0;
    c1 = SBUF;
    return c1;
}

void main(void)
{
    char src;

    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 0;
    RI = 0;

    while (1)
    {
         src = getc();
         if (src == 0x1B)
         break;
         putc(src);
    }
}
```

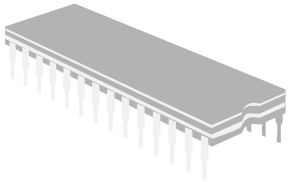
Пример 4. Ввод/вывод строки через последовательный порт с использованием пользовательских функций ввода/вывода.

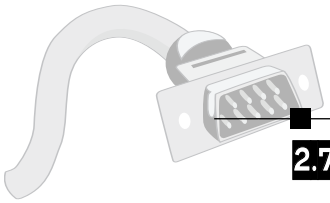
```
#include <stdio.h>
#include <REG52.H>

void putstr (unsigned char *s1)
{
```





```
    while (*s1)
    {
        SBUF = *s1;
        while (!TI);
        TI = 0;
        s1++;
    }
}

unsigned char * getstr(void)
{
    unsigned char buf[32];
    unsigned char *pbuf = buf;
    while (1)
    {
        while (!RI);
        RI = 0;
        *pbuf = SBUF;
        if (*pbuf == 0x0D)
        {
            *pbuf = '\0';
            break;
        }
        pbuf++;
    }
    return buf;
}

void main(void)
{
    unsigned char *pdat;

    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 0;
    RI = 0;

    while (1)
    {
        pdat = getstr();
        putstr(pdat);
    }
}
```





2.7. Интерфейс с языком ассемблера

В целом ряде случаев, особенно при работе с внешними устройствами, для повышения эффективности программного кода и упрощения разработки программы имеет смысл использовать язык ассемблера. Например, при работе с быстродействующими аналого-цифровыми преобразователями можно применить отдельные процедуры для считывания двоичного кода по интерфейсу SPI. В целом ряде случаев при программировании обмена данными с периферийным оборудованием программист сталкивается с необходимостью реализации сложных алгоритмов обмена данными, требующих использования многочисленных манипуляций с отдельными битами, что при применении только операторов языка высокого уровня может оказаться весьма затруднительным.

В таких случаях намного быстрее и проще решить задачу, применив в программе на С ассемблерный код. Единственное, что требуется от программиста при разработке ассемблерного кода, – достаточно хорошее понимание программной архитектуры микроконтроллера и системы команд.

Компилятор C51 позволяет работать как со встроенными (inline) ассемблерными вставками, так и с процедурами на ассемблере, которые можно хранить в отдельном файле с расширением .asm. В последнем случае можно оттранслировать исходный ASM-файл при помощи компилятора a51.exe, а затем скомпоновать программу с помощью компоновщика bl51.exe.

2.7.1. Встроенный ассемблерный код

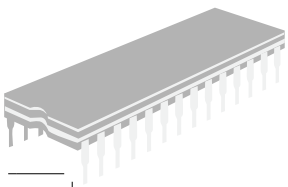
Использование встроенного ассемблерного кода позволяет оптимизировать критические участки программы, написанной на С. Для вставки ассемблерного кода используются директивы `asm` и `endasm`, как, например, в этом фрагменте программного кода:

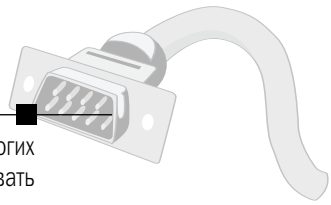
```
void main (void)
{
    . . .
    #pragma asm
    JMP  $
    #pragma endasm
}
```

Если в исходном тексте программы на С присутствуют ассемблерные вставки, то процесс компиляции и сборки программы будет отличаться от стандартного:

- на первом шаге нужно оттранслировать исходный текст программы компилятором c51, в командной строке для которого следует указать директиву src. При успешной компиляции создается файл с расширением .src;
- на втором шаге с помощью ассемблера a51 из SRC-файла создается файл объектного модуля с расширением .obj;
- на третьем шаге из объектного модуля создается абсолютный объектный модуль при помощи компоновщика bl51. После успешного выполнения этого шага полученный программный модуль можно проверить на работоспособность, загрузив его в симулятор Keil или в другой отладчик.

Здесь я хочу сделать одно замечание. Сгенерированный SRC-файл содержит исходный текст на языке ассемблера, т.е. фактически это ASM-файл. Однако если проанализировать





исходный текст такого файла, то он может оказаться неработоспособным. Поэтому во многих случаях перед тем, как компилировать SRC-файл при помощи a51, следует откорректировать исходный текст так, чтобы сделать его работоспособным.

Возможно, описанный процесс может показаться несколько запутанным, но я поясню его на практическом примере.

Пусть, например, файл inline_asm.c содержит следующий исходный текст:

```
#include <REG52.H>

int cnt = 0;
sbit bit0 = P1^0;

void t0ISR (void) interrupt 1 using 1{
    TF0 = 0;
    cnt++;
    if (cnt > 70)
    {
         bit0 = ~bit0;
         cnt = 0;
    }
}

void main(void)
{
    TH0 = 0;
    TL0 = 0;
    TMOD |= 0x1;
    TR0 = 1;

    #pragma asm
    SETB ET0
    SETB EA
    #pragma endasm

    while(1);
}
```

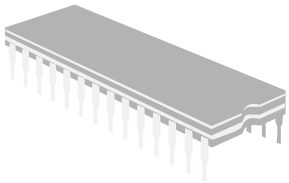
Работающая в соответствии с этим исходным текстом программа должна каждые 5 с инвертировать бит 0 порта P1. Для этого устанавливается прерывание таймера 0 (тактовая частота кристалла равна 11,059 МГц), причем инициализация прерывания осуществляется при помощи двух ассемблерных команд

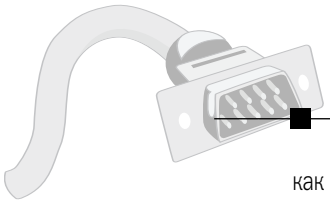
```
SETB ET0
SETB EA
```

ассемблерной вставки.

Откомпилируем исходный текст С-файла, выполнив команду:

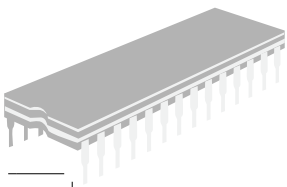
```
c51 inline_asm.c src
```

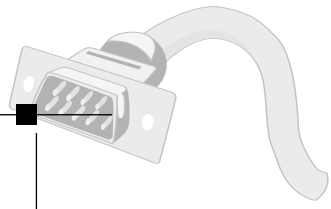




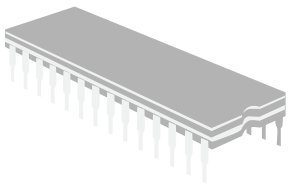
В результате получим файл с расширением .src, содержимое которого может быть таким, как показано в следующем листинге:

```
; inline_asm.SRC generated from: inline_asm.c
; COMPILER INVOKED BY:
;        C:\KEIL\C51\BIN\c51.exe inline_asm.c SRC

$NOMOD51

NAME     INLINE_ASM

P0  DATA 080H
P1  DATA 090H
P2  DATA 0A0H
P3  DATA 0B0H
T0  BIT  0B0H.4
AC  BIT  0D0H.6
T1  BIT  0B0H.5
T2  BIT  090H.0
EA  BIT  0A8H.7
IE  DATA 0A8H
EXF2     BIT  0C8H.6
RD  BIT  0B0H.7
ES  BIT  0A8H.4
IP  DATA 0B8H
RI  BIT  098H.0
INT0     BIT  0B0H.2
CY  BIT  0D0H.7
TI  BIT  098H.1
INT1     BIT  0B0H.3
RCAP2H   DATA 0CBH
PS  BIT  0B8H.4
SP  DATA 081H
T2EX     BIT  090H.1
OV  BIT  0D0H.2
RCAP2L   DATA 0CAH
C_T2     BIT  0C8H.1
WR  BIT  0B0H.6
RCLK     BIT  0C8H.5
TCLK     BIT  0C8H.4
SBUF     DATA 099H
PCON     DATA 087H
SCON     DATA 098H
TMOD     DATA 089H
TCON     DATA 088H
IE0 BIT  088H.1
IE1 BIT  088H.3
B   DATA 0F0H
CP_RL2   BIT  0C8H.0
ACC DATA 0E0H
```

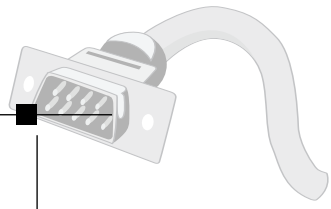




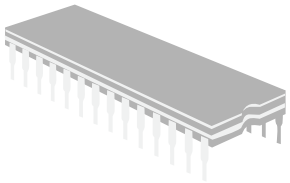
```
ET0 BIT  0A8H.1
ET1 BIT  0A8H.3
TF0 BIT  088H.5
ET2 BIT  0A8H.5
TF1 BIT  088H.7
TF2 BIT  0C8H.7
RB8 BIT  098H.2
TH0 DATA 08CH
EX0 BIT  0A8H.0
IT0 BIT  088H.0
TH1 DATA 08DH
TB8 BIT  098H.3
EX1 BIT  0A8H.2
IT1 BIT  088H.2
TH2 DATA 0CDH
P   BIT  0D0H.0
SM0 BIT  098H.7
TL0 DATA 08AH
SM1 BIT  098H.6
TL1 DATA 08BH
SM2 BIT  098H.5
TL2 DATA 0CCH
PT0 BIT  0B8H.1
PT1 BIT  0B8H.3
RS0 BIT  0D0H.3
PT2 BIT  0B8H.5
TR0 BIT  088H.4
RS1 BIT  0D0H.4
TR1 BIT  088H.6
TR2 BIT  0C8H.2
PX0 BIT  0B8H.0
PX1 BIT  0B8H.2
DPH DATA 083H
DPL DATA 082H
EXEN2    BIT  0C8H.3
REN BIT  098H.4
T2CON    DATA 0C8H
RXD BIT  0B0H.0
bit0     BIT  090H.0
TXD BIT  0B0H.1
F0  BIT  0D0H.5
PSW DATA 0D0H

?PR?t0ISR?INLINE_ASM SEGMENT CODE
?PR?main?INLINE_ASM  SEGMENT CODE
?C_INITSEG            SEGMENT CODE
?DT?INLINE_ASM       SEGMENT DATA
    EXTRN    CODE (?C_STARTUP)
    PUBLIC   cnt
    PUBLIC   main
```



```
    PUBLIC   t0ISR

    RSEG  ?DT?INLINE_ASM
cnt:    DS   2

    RSEG  ?C_INITSEG
    DB   002H
    DB   cnt
    DW   00000H

; #include <REG52.H>
;
; int cnt = 0;
; sbit bit0 = P1^0;
;
CSEG    AT   0000BH
    LJMP t0ISR

; void t0ISR (void) interrupt 1 using 1{

    RSEG  ?PR?t0ISR?INLINE_ASM
    USING    1
t0ISR:
    PUSH     ACC
    PUSH     PSW
             ; SOURCE LINE # 6
; TF0 = 0;
             ; SOURCE LINE # 7
    CLR      TF0
; cnt++;
             ; SOURCE LINE # 8
    INC      cnt+01H
    MOV      A,cnt+01H
    JNZ      ?C0006
    INC      cnt
?C0006:
; if (cnt > 70)
             ; SOURCE LINE # 9
    SETB     C
    SUBB     A,#046H
    MOV      A,cnt
    XRL      A,#080H
    SUBB     A,#080H
    JC       ?C0002
;  {
             ; SOURCE LINE # 10
;    bit0 = ~bit0;
             ; SOURCE LINE # 11
    CPL      bit0
;    cnt = 0;
```



```
                ; SOURCE LINE # 12
    MOV         cnt,#00H
    MOV         cnt+01H,#00H
;   }
                ; SOURCE LINE # 13
; }
                ; SOURCE LINE # 14
?C0002:
    POP         PSW
    POP         ACC
    RETI
; END OF t0ISR


;
; void main(void)

    RSEG  ?PR?main?INLINE_ASM
main:
                ; SOURCE LINE # 16
; {
                ; SOURCE LINE # 17
;   TH0 = 0;
                ; SOURCE LINE # 18
    CLR         A
    MOV         TH0,A
;   TL0 = 0;
                ; SOURCE LINE # 19
    MOV         TL0,A
;   TMOD |= 0x1;
                ; SOURCE LINE # 20
    ORL         TMOD,#01H
;   TR0 = 1;
                ; SOURCE LINE # 21
    SETB        TR0
;
;   #pragma asm
;    SETB ET0
      SETB ET0
;    SETB EA
       SETB EA
?C0003:
;   #pragma endasm
;
;   while(1);
                ; SOURCE LINE # 28
    SJMP        ?C0003
; END OF main


    END
```



Мы получили исходный текст нашей программы на ассемблере, при этом операторы и директивы исходного С-файла закомментированы и заменены соответствующими командами ассемблера. Для получения абсолютного объектного файла необходимо выполнить две операции: компиляцию SRC-файла с помощью a51.exe и сборку полученного OBJ-файла при помощи компоновщика bl51.exe. Здесь есть несколько нюансов. Просто откомпилировать и скомпоновать исходный текст SRC-файла, получив при этом работоспособную программу, нельзя. Если внимательно проанализировать листинг SRC-файла, то можно заметить, что здесь присутствуют внешние функции, определенные в других модулях. Если бы мы выполняли компиляцию и сборку программы без преобразования в промежуточный SRC-файл, как это и делается в большинстве случаев, то компилятор C51 автоматически подключил бы внешние библиотечные модули на этапе компоновки.

В данном случае ситуация иная. На первом этапе мы должны откомпилировать исходный текст SRC-файла с помощью макроассемблера Keil A51:

```
a51 inline_asm.SRC
```

Далее, для получения работоспособного абсолютного объектного файла нужно скомпоновать полученный OBJ-файл с библиотечными файлами. Для нашей программы нужно использовать библиотечный файл C51S.LIB. Командная строка для компоновщика bl51.exe будет такой:

```
bl51 inline_asm.obj, c51s.lib
```

Если компиляция и сборка прошли без ошибок, полученный абсолютный объектный файл программы можно запустить на отладку или конвертировать его в HEX-формат. В целом ряде случаев можно обойтись без подключения внешних библиотек функций, что позволяет оптимизировать размер программного кода и его быстродействие.

Вернемся к нашему SRC-файлу и посмотрим, как можно оптимизировать программный код. Первое, что нужно сделать, это закомментировать или удалить строку

```
EXTRN        CODE (?C_STARTUP)
```

которая является ссылкой на код инициализации, генерируемый компилятором. В ассемблерном модуле этот фрагмент кода не используется, а при дальнейшей компоновке объектного модуля будут выдаваться предупреждающие сообщения.

Далее, начальный код программы начинается не с абсолютного адреса 0 в сегменте программы, а с адреса обработчика прерывания таймера 0, о чем свидетельствует директива

```
CSEG     AT   0000BH
```

Для того чтобы программа работала, исходный текст следует изменить следующим образом:

- указать стартовый адрес в абсолютном сегменте команд равным 0 с помощью директивы `CSEG AT 0H`;
- первой командой программы должна быть команда `LJMP main`, передающая управление основной программе, размещенной в перемещаемом модуле `?PR?main?INLINE_ASM`;
- команда `LJMP t0ISR` должна быть удалена либо закомментирована.

Далее нужно немного откорректировать исходный текст обработчика прерывания таймера 0. Программа-обработчик прерывания должна находиться по фиксированному адресу 0x0B в абсолютном сегменте программы, поэтому закомментируем строки

```
?PR?t0ISR?INLINE_ASM SEGMENT CODE
RSEG  ?PR?t0ISR?INLINE_ASM
```

и после команды

```
LJMP     main
```

добавим директиву

```
org 0Bh
```

Сохраним все изменения. Теперь можно откомпилировать наш ассемблерный модуль, вызвав команду

```
a51 inline_asm.SRC
```

Далее нужно создать абсолютный объектный модуль при помощи компоновщика bl51, выполнив команду

```
bl51 inline_asm.OBJ
```

Наконец, создаем загружаемый модуль программы при помощи программы-конвертера форматов oh51:

```
oh51 inline_asm
```

Как видите, процесс создания кода программы из исходного текста на языке С с включенным ассемблерным кодом не слишком уж и сложен, хотя требует от программиста определенных знаний языка ассемблера и принципов построения программ.

2.7.2. Подпрограммы на ассемблере

Альтернативой встроенному ассемблерному коду являются написанные на ассемблере отдельные подпрограммы, оптимизирующие вычисления на отдельных участках программы. Такие программы могут быть написаны и оттранслированы как отдельные модули, которые затем можно использовать в других программах.

Поскольку код ассемблерных подпрограмм находится в другом модуле, то в основной программе ссылки на такие подпрограммы следует указывать с помощью директивы `external`.

Компилятор Keil C51 устанавливает определенные правила или, по-другому, соглашения о передаче параметров функциям и о возвращаемых значениях. Параметры могут передаваться функциям в регистрах и/или в памяти. В регистрах может передаваться максимум 3 параметра, остальные передаются через память, при этом имена функций, написанных на ассемблере и принимающих параметры в регистрах, должны начинаться с символа подчеркивания.

При передаче подпрограммам параметров в регистрах следует руководствоваться соглашениями, приведенными в табл. 2.7.

Передачу параметров функциям иллюстрируют следующие два примера.

Пусть имеется функция func1 (int a). Поскольку она принимает один 16-разрядный параметр, то он передается в регистрах R6 и R7.

Для такой функции, как func2 (int b, int c, int *d), первый параметр b передается в регистрах R6 и R7, второй параметр c передается в регистрах R4 и R5, а третий параметр d, являющийся указателем общего типа, передается в регистрах R1, R2 и R3.

Таблица 2.7. *Соглашения о передаче параметров функциям*

Номер параметра	8-разрядная переменная или указатель	16-разрядная переменная или указатель	32-разрядная переменная	Указатель общего типа
1	Регистр R7	R6 (старший байт) и R7 (младший байт)	R4 – R7	R1 – R3 (тип памяти в R3, старший байт в R2, младший в R1)
2	Регистр R5	Регистры R4 и R5 (старший байт в R4, младший в R5)	R4 – R7	R1 – R3 (тип памяти в R3, старший байт в R2, младший в R1)
3	Регистр R3	Регистры R2 и R3 (старший байт в R2, младший в R3)		R1 – R3 (тип памяти в R3, старший байт в R2, младший в R1)

Функции возвращают результат в регистрах в соответствии с соглашениями, приведенными в табл. 2.8.

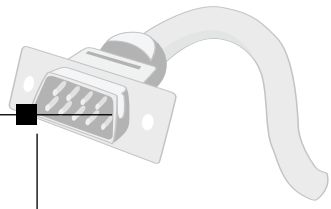
Таблица 2.8. *Соглашения о возвращаемых значениях*

Возвращаемое значение	Регистр(ы)	Замечания
Битовое	Флаг переноса (C)	
8-разрядное	R6	
16-разрядное	R6 и R7	Старший байт в регистре R6, младший в регистре R7
32-разрядное	R4 – R7	Старший байт в регистре R4, младший в регистре R7
Число с плавающей точкой	R4 – R7	32-разрядный формат IEEE
Указатель общего типа	R1 – R3	Тип памяти возвращается в регистре R3, старший байт в регистре R2, младший в регистре R1

Рассмотрим практические примеры интерфейса программ на C51 и ассемблере.

Пример 1. Основная программа выполняет операцию умножения при помощи внешней процедуры на ассемблере, передавая последней два 8-битовых числа и получая обратно 16-разрядный результат умножения. Исходный текст основной программы на C показан ниже:

```
#include <stdio.h>
#include <REG52.H>

extern unsigned int mul2(unsigned char c1, unsigned char c2);

void main(void)
{
    unsigned int res;

    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
```



```
    TR1 = 1;
    TI = 1;

    res = mul2(23, 56);
    printf("Result: %d\n", res);
    while(1);
}
```

Здесь функция `mul2`, выполняющая умножение двух 8-битовых переменных и возвращающая 16-разрядное значение, объявлена как внешняя с атрибутом `external`.

Исходный текст функции находится в отдельном файле с расширением .asm, который содержит следующий программный код:

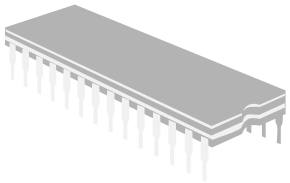
```
NAME PROCS
PUBLIC _mul2
PROG    SEGMENT CODE
RSEG    PROG
_mul2:
    MOV  A, R7
    MOV  B, R5
    MUL  AB

    MOV  R6, B
    MOV  R7, A
    RET
    END
```

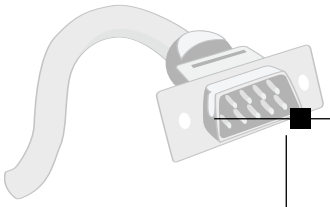
Этой функции передаются параметры в регистрах `R7` (первый параметр) и `R5` (второй параметр). Эти параметры запоминаются в регистрах `A` и `B`, после чего выполняется операция умножения. Полученный результат запоминается в двух регистрах: старший байт в регистре `R6`, младший байт в регистре `R7`. Основная программа извлекает результат из этих регистров и выводит его в последовательный порт.

Пример 2. В этом примере основная программа получает символ от функции `recvChar`, которая ожидает ввод с последовательного порта. Полученный символ затем выводится в последовательный порт в основной программе.

Вот исходный текст основной программы:

```
#include <stdio.h>
#include <REG52.H>

extern unsigned char recvChar(void);

void main(void)
{
    unsigned char res;

    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
```






```
    TR1 = 1;
    TI = 1;
    while (1)
    {
        res = recvChar();
        printf("%c", res);
    }
}
```

Программа на ассемблере представлена следующим исходным текстом:

```
    NAME    PROCS
    PUBLIC  recvChar
    PROG    SEGMENT CODE
    RSEG    PROG
recvChar:
    CLR     RI
    JNB     RI, $
    MOV     A, SBUF
    MOV     R7, A
    RET
    END
```

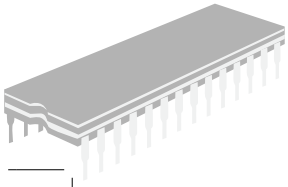
Здесь ассемблерная процедура не принимает никаких параметров, а возвращает полученный из последовательного порта символ основной программе в регистре `R7`.

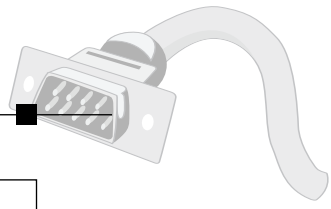
2.8. Программирование на языке ассемблера в среде Keil

До сих пор мы рассматривали язык ассемблера как вспомогательное средство при разработке программ на Keil C51. Тем не менее в целом ряде случаев, особенно когда не требуется создавать сложный интерфейс пользователя, а требуется лишь работа с оборудованием, программы на языке ассемблера работают очень эффективно. Слабым местом ассемблера традиционно считается трудность реализации математических функций и обработки сложных типов данных (длинных целых чисел, чисел с плавающей точкой), а также функции преобразования различных типов данных. Тем не менее в Интернете имеется масса исходных текстов программ, позволяющих решить эти и другие проблемы, значительно расширив при этом возможности программ на ассемблере. В принципе, настроив среду программирования соответствующим образом и подключив необходимые библиотеки функций, можно решать практически любые задачи по программированию 8051-систем.

В этом разделе мы рассмотрим основы программирования на языке ассемблера фирмы Keil, известного больше под названием A51. Синтаксис этого языка во многом похож на тот, что используется в «классическом» ASM-51. В программе на ассемблере A51 можно определять сегменты кода (абсолютные и перемещаемые), а также сегменты данных, указывая для них тип используемой памяти (DATA, IDATA, XDATA, CONST и т.д.). Описание языка ассемблера A51 довольно обширно и может занять несколько сотен страниц, поэтому мы не будем углубляться в детали функционирования A51, а на практических примерах посмотрим, как создаются исполняемые файлы программ на ассемблере.

Вначале создадим ассемблерный вариант программы «Hello, World» вне среды программирования Keil uVision.





Наберем в текстовом редакторе исходный текст программы:

```
NAME    PROCS
ESC EQU 1Bh
MAIN    SEGMENT CODE
myData  SEGMENT CODE
    CSEG AT  0
    USING   0
    JMP  start
    RSEG MAIN
start:
    MOV  SCON, #50h
    MOV  TH1, #0FDh
    ORL  TMOD, #20h
    SETB TR1
    MOV  DPTR, #text

next_byte:
    CLR  TI
    CLR  A
    MOVC A, @A+DPTR
    CJNE A, #ESC, write_char
    SJMP $
write_char:
    MOV  SBUF, A
    JNB  TI, $
    INC  DPTR
    SJMP next_byte

    RSEG myData
text: DB 'Hello, World', ESC
    END
```

Поскольку микроконтроллер после включения питания начинает выполнение программы с абсолютного адреса `0` в сегменте программ, то в начале нашего листинга указана директива

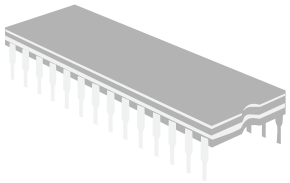
```
CSEG    AT  0
```

Первая команда

```
JMP start
```

передает управление на метку `start`, которая является точкой входа в основную программу. Основная часть программы размещена в перемещаемом сегменте программ `MAIN`, о чем свидетельствуют директивы

```
MAIN    SEGMENT CODE
RSEG    MAIN
```



Это означает, что при сборке программы компоновщик bl51.exe выполнит привязку этого сегмента к конкретному адресу. В выполняющейся автономно ассемблерной программе обязательно должен присутствовать как минимум абсолютный сегмент программного кода, откуда начинается выполнение программы, и один или несколько перемещаемых или абсолютных модулей кода и данных. Какие бы комбинации сегментов кода и данных ни использовались, всегда следует помнить, что программа начинает выполняться с абсолютного адреса 0, и строить соответственно с этим структуру программы.

В нашей программе присутствует константная (литеральная) строка text, размещенная в сегменте `myData`, которая выводится в последовательный порт. Для вывода данных в порт нужно предварительно его инициализировать, установив необходимые параметры обмена, что выполняют команды

```
MOV SCON, #50h
MOV TH1, #0FDh
ORL TMOD, #20h
SETB    TR1
```

Программирование последовательного порта рассмотрено далее в этой книге, сейчас же отмечу, что данный фрагмент кода настраивает порт на скорость обмена 9600 бод при тактовой частоте микроконтроллера 11,059 МГц.

Вывод строки `text` в последовательный порт выполняется построчно, а адрес первого элемента помещается в регистр `DPTR` командой

```
MOV DPTR, #text
```

Для вывода символа, на который указывает регистр `DPTR`, он помещается в регистр `A`, а затем отправляется в последовательный порт командой

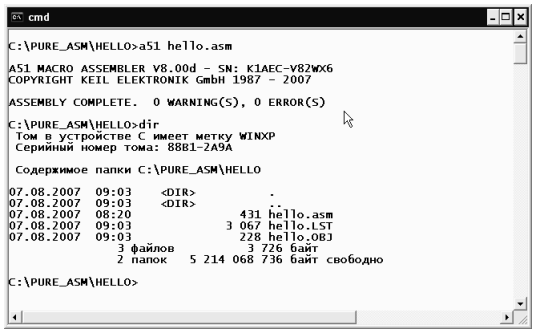
```
MOV SBUF, A
```

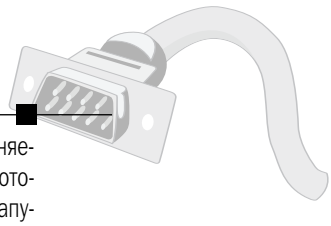
Вывод символов строки `text` продолжается до тех пор, пока не будет обнаружен символ ESCAPE, после чего программа входит в бесконечный цикл по команде

```
SJMP    $
```

Сохраним наш исходный текст в файле hello.asm. Теперь можно выполнить компиляцию нашей программы с помощью ассемблера a51.exe. При успешной компиляции в текущем каталоге программы появятся два файла: hello.obj и hello.lst (рис. 2.20).

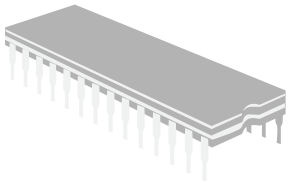
Рис. 2.20. *Результат компиляции ассемблерной программы*





Созданный файл объектного модуля hello.obj, как уже упоминалось, не является исполняемым файлом. Компилятор A51 транслирует исходный текст программы в двоичный код, который не привязан к каким-либо фиксированным адресам в памяти, как это требуется при запуске программы. Все сегменты (кода и данных) указаны со смещением 0, т.е. не привязаны к физическим адресам памяти системы, как это видно из содержимого файла hello.lst:

```
A51      MACRO    ASSEMBLER      HELLO
07/05/2007 09:03:21 PAGE     1


MACRO ASSEMBLER A51 V8.00d
OBJECT MODULE PLACED IN hello.OBJ
ASSEMBLER INVOKED BY: C:\KEIL\C51\BIN\A51.EXE hello.asm

LOC    OBJ               LINE             SOURCE

                         1                NAME      PROCS
  001B                   2                ESC       EQU 1Bh
                         3                MAIN      SEGMENT     CODE
                         4                myData    SEGMENT     CODE
—                        5                CSEG      AT          0
                         6                          USING       0
0000 020000    F         7                          JMP            start
—                        8                RSEG      MAIN
0000                     9      start:
0000 759850              10               MOV                   SCON, #50h
0003 758DFD              11               MOV                   TH1, #0FDh
0006 438920              12               ORL                   TMOD, #20h
0009 D28E                13               SETB      TR1
                         14
000B 900000    F         15               MOV                   DPTR,#text
                         16
000E                     17      next_byte:
000E C299                18               CLR                   TI
0010 E4                  19               CLR                   A
0011 93                  20               MOVC      A, @A+DPTR
0012 B41B02              21               CJNE      A, #ESC, write_char
0015 80FE                22               JMP                   $
0017                     23      write_char:
0017 F599                24               MOV                   SBUF, A
0019 3099FD              25               JNB                   TI, $
001C A3                  26               INC                   DPTR
001D 80EF                27               SJMP      next_byte
                         28
—                        29             RSEG      myData
0000 48656C6C            30      text: DB 'Hello, World', ESC
0004 6F2C2057
0008 6F726C64
000C 1B
                         31               END
```



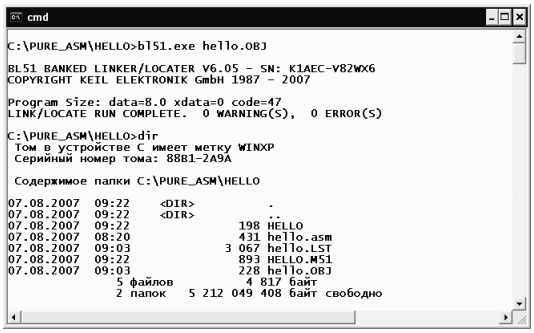
```
A51 MACRO ASSEMBLER  HELLO
07/05/2007 09:03:21 PAGE     2

SYMBOL                  TABLE            LISTING
------                  -----            -------

N A M E             T Y P E     V A L U E      ATTRIBUTES

ESC. . . . . .   N NUMB       001BH    A
MAIN . . . . . .   C SEG       001FH           REL=UNIT
MYDATA . . . . . .   C SEG       000DH           REL=UNIT
NEXT_BYTE. . . . .   C ADDR      000EH    R      SEG=MAIN
PROCS. . . . . .   N NUMB      ---
SBUF . . . . . .   D ADDR      0099H    A
SCON . . . . . .   D ADDR      0098H    A
START. . . . . .   C ADDR      0000H    R      SEG=MAIN
TEXT . . . . . .   C ADDR      0000H    R      SEG=MYDATA
TH1. . . . . .   D ADDR      008DH    A
TI . . . . . .   B ADDR      0098H.1  A
TMOD . . . . . .   D ADDR      0089H    A
TR1. . . . . .   B ADDR      0088H.6  A
WRITE_CHAR . . . .   C ADDR      0017H    R      SEG=MAIN


REGISTER BANK(S) USED: 0


ASSEMBLY COMPLETE.  0 WARNING(S), 0 ERROR(S)
```

Окончательную привязку (фиксацию адресов) выполняет компоновщик bl51.exe, который нужно запустить для создания абсолютного модуля с именем hello, набрав следующую команду:

```
bl51.exe hello.OBJ
```

При успешной компиляции будет создан файл абсолютного модуля без расширения (рис. 2.21).

Рис. 2.21. *Сборка абсолютного объектного файла*



Компоновщик bl51.exe, кроме абсолютного объектного модуля, который можно использовать для отладки, создает файл распределения памяти hello.M51 (так называемый MAP-файл),

содержащий информацию об абсолютных адресах памяти, по которым размещаются данные и код программы. Анализ содержимого этого файла позволяет четко представить структуру программы в памяти при ее загрузке для выполнения.

Проанализируем MAP-файл hello.M51, исходный текст которого показан ниже:

```
BL51 BANKED LINKER/LOCATER V6.05
07/05/2007  09:22:26  PAGE 1


BL51 BANKED LINKER/LOCATER V6.05, INVOKED BY:
BL51.EXE HELLO.OBJ


INPUT MODULES INCLUDED:
  HELLO.OBJ (PROCS)


LINK MAP OF MODULE:  HELLO (PROCS)


            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            ------------------------------

            * * * * * * *   D A T A   M E M O R Y   * * * * * * *
            REG     0000H     0008H     ABSOLUTE     "REG BANK 0"

            * * * * * * *   C O D E   M E M O R Y   * * * * * * *
            CODE    0000H     0003H     ABSOLUTE
            CODE    0003H     001FH     UNIT         MAIN
            CODE    0022H     000DH     UNIT         MYDATA

Program Size: data=8.0 xdata=0 code=47
LINK/LOCATE RUN COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

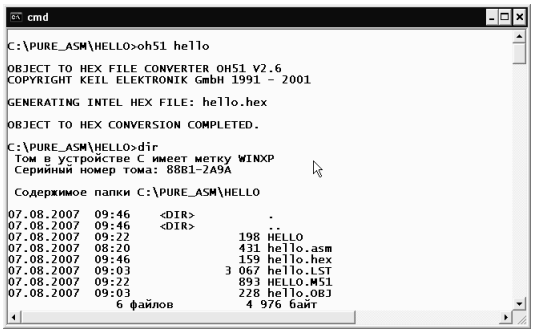
Проанализируем этот листинг, поскольку он предоставляет нам достаточно информации для размышления. Как видно из листинга, программа hello, содержащаяся в абсолютном модуле hello, имеет размер 47 байт. Именно это количество байт программного кода будет загружено в целевую систему после преобразования абсолютного модуля в HEX-файл. Для временного хранения данных используются регистры R0 – R7 (банк 0), занимающие 8 байт памяти на кристалле, которые, естественно, не учитываются при вычислении размера программного кода. Выполнение программы, как вы помните, начинается с команды

```
JMP start
```

размещенной по адресу 0 в абсолютном сегменте программы. Поскольку эта команда занимает 3 байта в памяти (с 0x0 по 0x2), то перемещаемый сегмент программы MAIN размещен, что вполне логично, начиная с адреса 0x3. Поскольку текстовый литерал размещен в сегменте программ, то его размер учитывается при подсчете объема памяти, занимаемого сегментом программ.

Последний шаг, который нужно выполнить для получения загружаемого HEX-файла, – преобразовать абсолютный объектный модуль в формат HEX, что можно выполнить с помощью программы-конвертера oh51.exe (рис. 2.22).

Рис. 2.22. *Создание HEX-файла*



Мы так подробно рассмотрели процесс ассемблирования файла hello.asm для того, чтобы читатели смогли понять внутренние механизмы этого процесса. Кроме того, такой анализ помогает и при поиске и обнаружении ошибок в программном коде.

Рассмотрим еще один пример программы на ассемблере, в которой используется прерывание INT0. При возникновении прерывания программа-обработчик прерывания выводит в последовательный порт строку сообщения и инкрементирует однобайтовую переменную во внутренней области памяти типа idata. Как и предыдущая, эта программа является демонстрационной.

Для разработки программы этого примера будем использовать среду программирования Keil uVision3. Вначале создадим пустой проект с именем int0_asm, не содержащий файла с исходным текстом, а затем включим в проект файл procs.asm, содержащий следующий исходный текст:

```
NAME    PROCS
    ESC  EQU 1Bh
    MAIN SEGMENT CODE
    myConst  SEGMENT CODE
    myVar    SEGMENT IDATA
;------------
    CSEG AT  0
    USING    0
    JMP  start
;------------
int0Isr:
    ORG  3h
    CLR  EX0
    CALL sendStr
    MOV  A, @R1
    CJNE A, #11, inc_cnt
    CLR  A
quit:
    MOV  @R1, A
```

```
    SETB EX0
    RETI
inc_cnt:
    INC  A
    SJMP quit
;------------
    RSEG MAIN
start:
    MOV  R1, #cnt
    MOV  @R1, #0
    MOV  SCON, #50h
    MOV  TH1, #0FDh
    ORL  TMOD, #20h
    SETB TR1

    SETB IT0
    SETB EX0
    SETB EA
    SJMP $
;----------------
sendStr:
    MOV  DPTR, #text
next_byte:
    CLR  TI
    CLR  A
    MOVC A, @A+DPTR
    CJNE A, #ESC, write_char
    RET
write_char:
    MOV  SBUF, A
    JNB  TI, $
    INC  DPTR
    SJMP next_byte
;-----------------
    RSEG myConst
text: DB 'Interrupt 0 occured', 0dh, 0ah, ESC
    RSEG myVar
cnt:  DS 1
    END
```

Программный код обработчика прерывания расположен в абсолютном сегменте программ начиная с адреса 0x03. Первая команда обработчика временно запрещает вызов прерываний по входу INT0. Затем в последовательный порт выводится сообщение, представляющее собой текстовую константу `text`. Далее из внутренней области памяти микроконтроллера 8051 (тип idata) считывается значение однобайтовой переменной `cnt`, и если оно равно `11`, то переменная обнуляется. Если значение `cnt` меньше `11`, то она инкрементируется.

Откомпилируем проект int0_asm в среде Keil uVision и посмотрим, каким образом компонуются сегменты программного кода и данных в этом примере. Вот интересующее нас содержимое файла int0_asm.MAP:

```
          BL51 BANKED LINKER/LOCATER V6.05
          07/05/2007  13:43:44  PAGE 1


BL51 BANKED LINKER/LOCATER V6.05, INVOKED BY:
C:\KEIL\C51\BIN\BL51.EXE procs.obj TO int0_Asm RAMSIZE (256)


INPUT MODULES INCLUDED:
  procs.obj (PROCS)


LINK MAP OF MODULE:  int0_Asm (PROCS)


            TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
            ------------------------------

            * * * * * * *   D A T A   M E M O R Y   * * * * * * *
            REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
            IDATA   0008H     0001H     UNIT         MYVAR

            * * * * * * *   C O D E   M E M O R Y   * * * * * * *
            CODE    0000H     0014H     ABSOLUTE
            CODE    0014H     002AH     UNIT         MAIN
            CODE    003EH     0016H     UNIT         MYCONST

    . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Program Size: data=9.0 xdata=0 code=84
LINK/LOCATE RUN COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

Код программы располагается в двух сегментах: абсолютном сегменте программы, начиная с адреса 0x0 и заканчивая адресом 0x13, и в перемещаемом MAIN, который начинается с адреса 0x14 и заканчивается адресом 0x3. В сегменте кода по адресу 0x03E располагается текстовый литерал MYCONST. Сегмент данных MYVAR располагается сразу же за регистровым банком 0, начиная с адреса 0x8, и занимает один байт памяти.

Как видим из приведенных примеров, программирование на языке ассемблера вовсе не кажется таким трудным, как иногда приходится слышать, но для этого нужно достаточно четко представлять себе архитектуру микроконтроллера и взаимодействие его функциональных узлов, а также знать синтаксис команд этого языка.

Далее мы рассмотрим еще одну возможность, предоставляемую разработчику программного обеспечения средой разработки Keil uVision3, а именно встроенные средства отладки программного обеспечения. В начале главы мы уже немного ознакомились с тем, как можно проверить работоспособность программ, выполняющих ввод/вывод в последовательный порт. Сейчас рассмотрим более детально, как можно проверить работу не только вывода, но и ввода данных через последовательный порт, а также работу обработчиков прерываний, таймеров и портов ввода/вывода. Естественно, что полностью сымитировать работу аппаратных узлов микроконтроллера нельзя, даже если использовать специальную среду моделирования наподобие Proteus VSM, особенно если дело касается работы в реальном времени. Тем не

менее, даже такой уровень отладки может принести несомненную пользу на начальных этапах разработки программного обеспечения в плане работоспособности логических алгоритмов программы и выявления ошибок в программировании.

2.9. Отладка программ в среде Keil uVision

Среда программирования Keil кроме инструментов создания, компиляции и сборки программ включает и средства отладки, позволяющие проверить работоспособность программы и исправить при необходимости ошибки в исходном тексте. Отладку в среде Keil uVision можно выполнить как с помощью внутреннего симулятора работы программ, так и подключив целевую систему с 8051. В этом разделе мы рассмотрим возможности внутреннего отладчика (симулятора) Keil.

Для того чтобы выполнить отладку программы в симуляторе, необходимо установить переключатель **Use Simulator** на вкладке **Debug** окна свойств проекта (рис. 2.23).

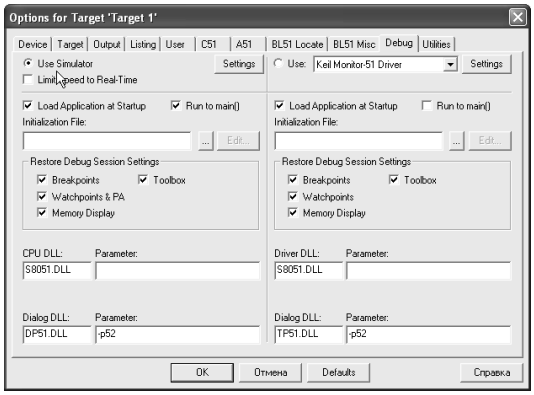


Рис. 2.23. *Установка настроек симулятора*

Отмечу, что при начальном запуске среды программирования Keil uVision3 эта опция установлена по умолчанию.

Напомню, что для тестирования программы в отладчике необходимо откомпилировать ее исходный текст, затем выполнить сборку программы с помощью компоновщика. Полученный таким образом абсолютный объектный модуль можно загрузить в отладчик и запустить на выполнение. Все функции ввода/вывода из классического ANSI C (printf, scanf, getchar и т.д.) в Keil C51 имеют несколько иной смысл: они оперируют с данными, проходящими через последовательный порт. Например, библиотечная функция printf в Keil C51 выводит данные в порт, а не на экран дисплея. Функция scanf ожидает ввода данных не с клавиатуры консоли, как в классическом ANSI C, а с последовательного порта. Если отладочный модуль с микроконтроллером 8051 соединен, например, с персональным компьютером по последовательному порту и на ПК работает какая-либо программа-терминал, то, действительно, данные с клавиатуры терминальной программы считываются в последовательный порт и создается иллюзия ввода данных с клавиатуры. То же самое касается и вывода данных в последовательный порт программой микроконтроллера, когда визуально данные появляются на экране, хотя фактически они выводятся в последовательный порт.

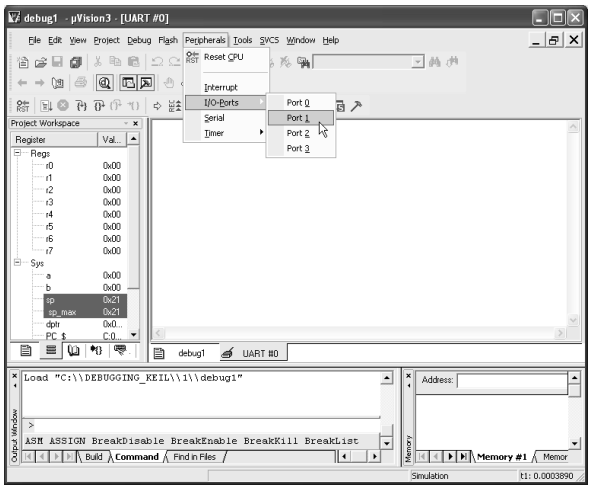
Для демонстрации возможностей отладчика среды Keil uVision разработаем несколько простых программных проектов, в каждом из которых продемонстрируем те или иные методы симуляции и отладки.

Пример 1. Необходимо проверить работоспособность следующей программы:

```
#include <stdio.h>
#include <REG52.H>

sbit Bit0 = P1^0;

void main(void)
{
    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 1;

    while (1)
    {
        printf("Press Enter to reverse a bit P1.0\n");
        getchar();
        Bit0 = ~Bit0;
    }
}
```

При вводе символа с последовательного порта инвертируется бит 0 порта P1. Выполним сборку программы и запустим ее на выполнение через меню **Debug → Start/Stop Debug Session → Run**. Мы хотим посмотреть, как работает ввод символов через последовательный порт, и проверить состояние бита 0 порта P1. Чтобы просмотреть прием символов с последовательного порта с помощью функции getchar, выберем меню **View → Serial Window → UART #0**. Желательно посмотреть, будет ли инвертироваться бит 0. Для этого нужно в меню **Peripherals → I/O–Ports** выбрать опцию **Port 1** (рис. 2.24).

Рис. 2.24. *Выбор порта ввода/вывода*



После этого появится небольшое диалоговое окно с обозначениями выводов порта P1, которые будут отображать состояние битов (рис. 2.25).

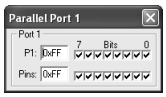


Рис. 2.25.
Индикация состояния выводов порта P1

Теперь, если перейти в окно **UART #0** и нажать несколько раз клавишу **Enter**, то можно наблюдать изменения в нулевом бите (рис. 2.26).

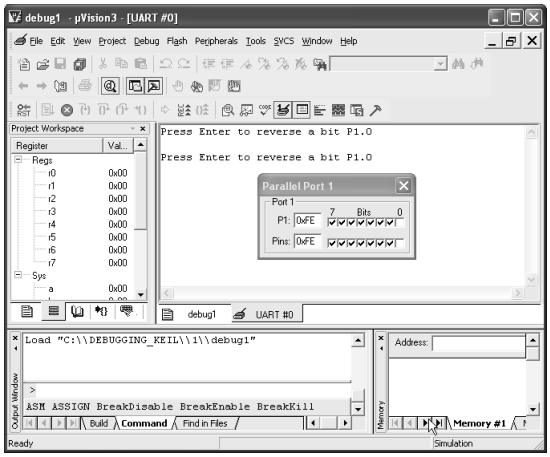


Рис. 2.26.
Индикация ввода/вывода программы и состояния порта

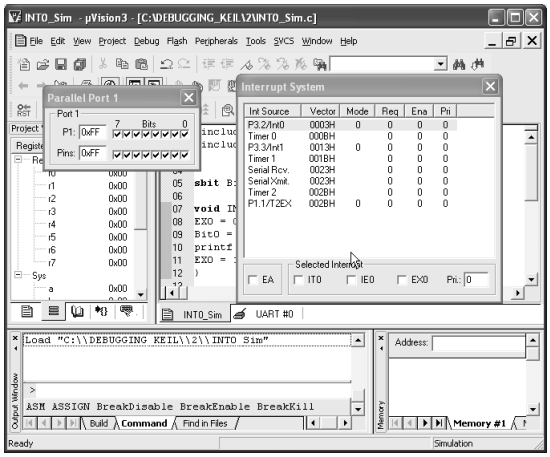
Пример 2. Здесь мы рассмотрим, как симулируется работа прерывания INT0 в Keil uVision. Откомпилируем и скомпонуем следующую программу:

```
#include <stdio.h>
#include <REG52.H>

sbit Bit0 = P1^0;

void INT0Isr (void) interrupt 0 using 1 {
    EX0 = 0;
    Bit0 = ~Bit0;
    printf("Interrupt 0 occured.\n");
    EX0 = 1;
}

void main(void)
{
    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 1;

    IT0 = 1;
    EX0 = 1;
    EA = 1;

    while (1);
}
```

В этой программе используется программа-обработчик внешнего прерывания 0 (INT0). Каждый раз при возникновении прерывания в последовательный порт выводится соответствующее сообщение и инвертируется бит `0` порта `P1`. Поскольку вывод данных в последовательный порт занимает относительно длительное время, в течение которого могут произойти новые прерывания, то на время передачи строки функцией `printf` внешнее прерывание 0 блокируется.

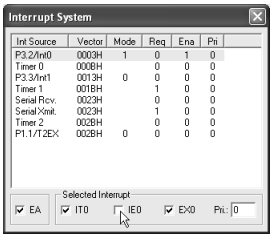
Для симуляции работы программы помимо двух окон отладки, используемых в предыдущем примере, нужно выбрать меню **Peripherals** → **Interrupt** (рис. 2.27).

Рис. 2.27. *Симуляция работы прерывания*



Обратите внимание на диалоговое окно **Interrupt System**. Если выбрать строку **P3.2/Int0**, то внизу появится строка устанавливаемых флагов для данного прерывания. Поскольку внешнее прерывание 0 может вызываться при установке флага **IE0**, то для моделирования возникновения/прекращения прерываний можно ставить или снимать отметку на флажке **IE0** (рис. 2.28).

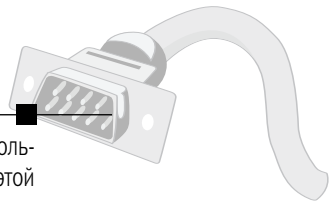
Рис. 2.28. *Имитация вызова прерывания*



При вызове прерывания будет изменяться и значение бита P1.0, что можно наблюдать в диалоговом окне **Parallel Port 1** (см. рис. 2.27).

При работе с портами ввода/вывода следует учитывать аппаратные особенности микроконтроллеров 8051, связанные с записью/чтением. Для записи данных в порт нужно просто выполнить оператор присваивания (в языке C), возможно, в комбинации с логическими побитовыми операциями (если требуется оперировать с отдельными битами). Можно оперировать с отдельными битами порта, для удобства присвоив им какое-либо имя. Например, для установки бита P1.7 в единичное состояние можно выполнить оператор

```
P1 |= 0x80;
```



При этом остальные биты порта P1 остаются нетронутыми. Для той же цели можно использовать и другой подход, назначив биту P1.7 переменную, после чего просто присваивать этой переменной нужное значение, например:

```
sbit bit0 = P1.7;
. . .
bit0 = 1;
```

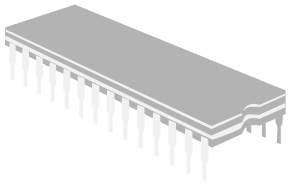
Этот фрагмент программного кода выполняет те же действия, что и предыдущий оператор, устанавливая бит P1.7 в единичное состояние.

Чтение данных с портов ввода/вывода отличается от записи тем, что предварительно в бит, значение которого нужно прочитать, записывается 1. В этом случае данные будут считаны не с внутреннего регистра-защелки данного бита, а с вывода порта микроконтроллера.

Пример 3. Для иллюстрации чтения/записи через порты ввода/вывода рассмотрим пример программы, исходный текст которой приведен ниже:

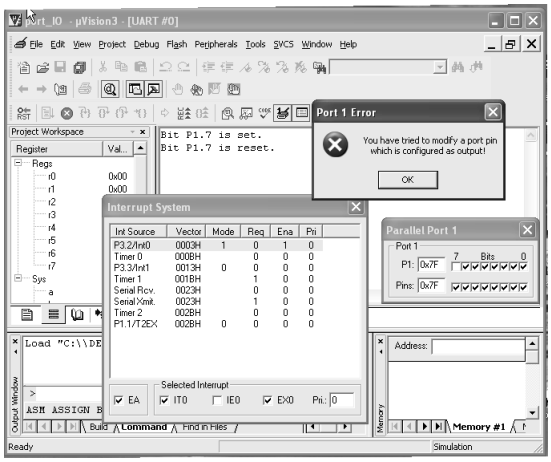
```
#include <stdio.h>
#include <REG52.H>

sbit Bit0 = P1^7;

void INT0Isr (void) interrupt 0 using 1 {
    EX0 = 0;
    if (Bit0)
    printf("Bit P1.7 is set.\n");
    else
    printf("Bit P1.7 is reset.\n");
    EX0 = 1;
}

void main(void)
{
    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 1;

    IT0 = 1;
    EX0 = 1;
    EA = 1;

    while (1);
}
```

Здесь при вызове прерывания INT0 считывается значение бита P1.7 с вывода микроконтроллера. Если скомпилировать эту программу и запустить ее в отладчике Keil, то при записанной в регистре-защелке P1.7 единице значение бита можно прочитать; если же в защелке



записан 0, бит порта работает как выходной, поэтому данные, считанные с вывода P1.7 (не с защелки!) могут быть некорректными. Симулятор Keil отслеживает подобные ситуации, выдавая сообщение, как это показано на рис. 2.29.

Рис. 2.29. *Моделирование ошибочной ситуации*



Из рис. 2.29 видно, что при установленном 0 в регистре-защелке бита P1.7 попытка симуляции 1 на этом же выводе приводит к появлению окна сообщения **Port 1 Error** с описанием ошибки.

В большинстве программ используются переменные различных типов, и требуется отслеживать корректность их значений при выполнении программного кода. В следующем примере мы посмотрим, как отслеживать значения переменных программы при отладке программы на симуляторе Keil.

Пример 4. Откомпилируем и запустим в симуляторе программу, исходный текст которой сохранен в файле debug_memo.c и имеет вид:

```
#include <stdio.h>
#include <REG52.H>

idata char COUNTER = 0;

void INT0Isr (void) interrupt 0 using 1 {
    COUNTER++;
    if (COUNTER == 11)
    COUNTER = 0;
}

void main(void)
{
    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 1;

    IT0 = 1;
    EX0 = 1;
```

```
    EA = 1;

    while (1);
}
```

Предположим, что нужно наблюдать за изменениями переменной COUNTER. Откомпилируем исходный текст программы и запустим на отладку абсолютный объектный модуль. В меню **View** выберем опции **Memory Window** и **Symbol Window** (рис. 2.30).

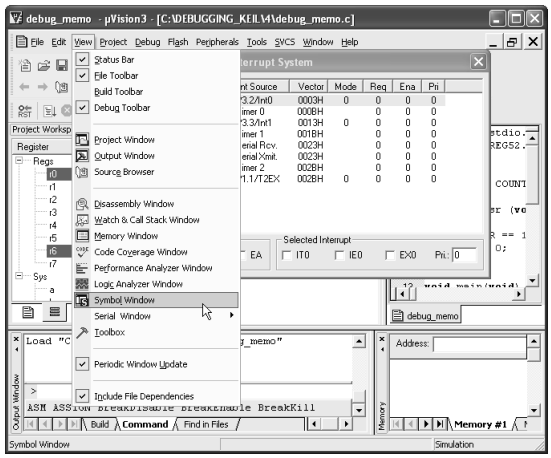


Рис. 2.30. *Симуляция работы с памятью*

В области **Symbols** найдем адрес переменной COUNTER, значение которой нужно отслеживать, и введем его в область **Memory** в правом нижнем углу (рис. 2.31).

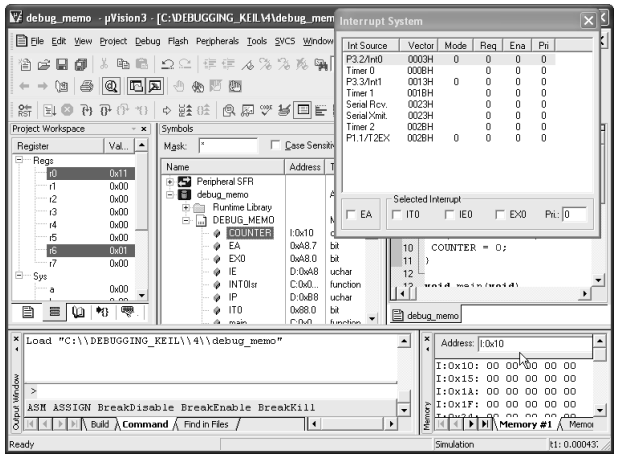
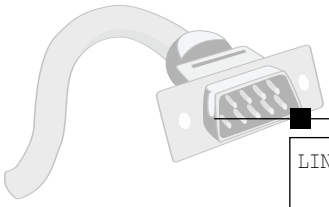


Рис. 2.31. *Анализ переменной COUNTER*

Переменная COUNTER размещается во внутренней памяти данных микроконтроллера по адресу 0x10, который в симуляторе обозначается как I:0x10. После запуска программы в симуляторе при моделировании прерывания INT0 можно наблюдать изменения значения переменной COUNTER в области **Memory**. Определить адрес переменной COUNTER, размещенной в сегменте памяти типа IDATA, можно и напрямую, проанализировав MAP-файл, созданный компоновщиком. Вот интересующий нас фрагмент листинга:

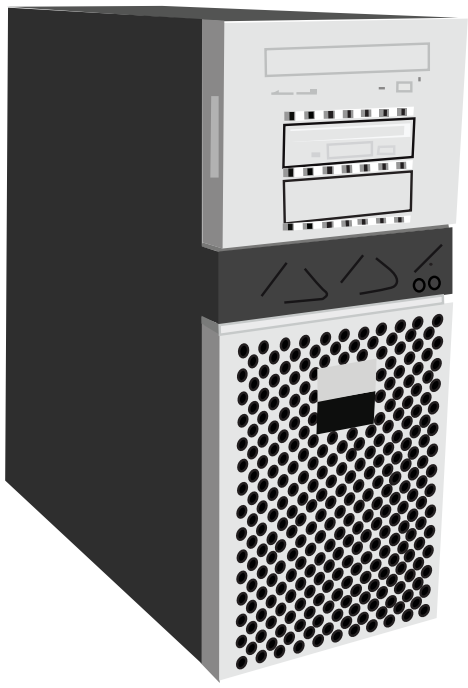



```
LINK MAP OF MODULE:  debug_memo (DEBUG_MEMO)

       TYPE     BASE      LENGTH    RELOCATION  SEGMENT NAME
       ------------------------------------------------------

       * * * * * * *      D A T A   M E M O R Y     * * * * * * *
       REG      0000H     0008H     ABSOLUTE    "REG BANK 0"
       REG      0008H     0008H     ABSOLUTE    "REG BANK 1"
       IDATA    0010H     0001H     UNIT        ?ID?DEBUG_MEMO
       IDATA    0011H     0001H     UNIT        ?STACK

       * * * * * * *      C O D E   M E M O R Y     * * * * * * *
       CODE     0000H     0003H     ABSOLUTE
       CODE     0003H     0003H     ABSOLUTE
       CODE     0006H     008CH     UNIT        ?C_C51STARTUP
       CODE     0092H     0015H     UNIT        ?PR?INT0ISR?DEBUG_MEMO
       CODE     00A7H     0015H     UNIT        ?PR?MAIN?DEBUG_MEMO
       CODE     00BCH     0004H     UNIT        ?C_INITSEG
```

Здесь сегмент данных, в котором размещена переменная COUNTER, обозначен как ?ID?DEBUG_MEMO (это стандартная форма записи, принципы формирования которой описаны в справке на компилятор Keil).

Как видно из листинга, этот сегмент занимает один байт памяти с адресом 0x10, т.е. наша переменная размещена именно по этому адресу.

Заканчивая анализ возможностей среды разработки Keil uVision3 и компиляторов C51 и A51, отмечу, что детальное изучение этих инструментальных средств поможет не только при создании программ, но и позволит глубже понять принципы функционирования микроконтроллера 8051, что само по себе очень полезно.



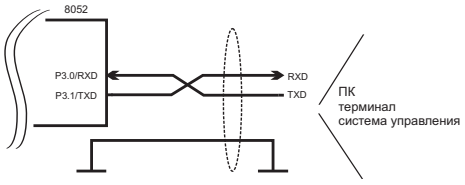
Использование последовательного порта

3 Использование последовательного порта

Микроконтроллеры серии 8051 имеют интегрированный последовательный порт обмена данными (Universal Asynchronous Receiver/Transmitter, UART), что позволяет системам на их базе обмениваться информацией с другими устройствами, используя стандартный протокол RS-232.

Микроконтроллеры 8051/8052 имеют два вывода для передачи-приема данных в последовательном формате. Эти выводы обозначаются как P3.1/TXD и P3.0/RXD. Фактически в микроконтроллерах 8052 используется минимально необходимая конфигурация для организации обмена данными по стандарту RS-232, показанная на рис. 3.1.

Рис. 3.1.
Схема обмена данными с внешними устройствами по RS-232

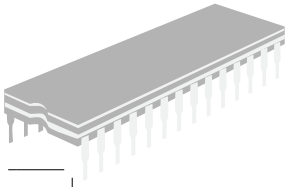


Последовательный порт 8051-совместимых устройств имеет несколько режимов работы и может быть легко запрограммирован всего несколькими командами ассемблера.

Первое, что нужно сделать перед использованием последовательного порта, – сконфигурировать его. Для этого следует установить определенные биты в специальном регистре SCON (Serial Control). Назначение битов этого порта приведено в табл. 3.1.

Таблица 3.1.
Регистр SCON

Номер бита	Наименование	Адрес бита (шест.)	Назначение
7	SM0	0x9F	Бит 0 установки режима
6	SM1	0x9E	Бит 1 установки режима
5	SM2	0x9D	Бит разрешения межпроцессорных коммуникаций
4	REN	0x9C	Бит разрешения приема. Должен быть установлен, если необходимо принимать данные
3	TB8	0x9B	Разрешение работы с 9-м битом в режиме передачи
2	RB8	0x9A	Разрешение работы с 9-м битом в режиме приема



Номер бита	Наименование	Адрес бита (шест.)	Назначение
1	TI	0x99	Бит завершения передачи. Устанавливается по завершению передачи байта данных
0	RI	0x98	Бит завершения приема. Устанавливается по завершению приема байта данных

Таблица 3.1. *Регистр SCON (окончание)*

Режимы работы последовательного порта определяются битами SM0 и SM1, как показано в табл. 3.2.

SM0	SM1	Режим работы	Наименование	Скорость обмена
0	0	0	8-разрядный регистр сдвига	Тактовая частота / 12
0	1	1	9-разрядный UART	Устанавливается таймером 1
1	0	2	8-разрядный UART	Тактовая частота / 64
1	1	3	9-разрядный UART	Устанавливается таймером 1

Таблица 3.2. *Установка режимов работы последовательного порта*

Как видно из таблицы, в зависимости от значений SM0 и SM1 последовательный порт может работать в двух режимах со скоростью обмена, которая определяется таймером 1, и в двух режимах, при которых скорость обмена определяется тактовой частотой кристалла микроконтроллера.

Бит SM2 устанавливается при организации мультипроцессорного обмена. В настоящее время в большинстве разработок этот бит не используется. Бит 4 (REN) разрешает прием данных по последовательному порту и должен быть обязательно установлен, если система будет принимать данные.

Установленный бит RI сигнализирует о завершении приема байта из последовательного порта. Если для обработки данных, проходящих через последовательный порт, используется прерывание, то установка этого бита вызывает прерывание (если установлены соответствующие биты разрешения прерываний).

Бит TI, будучи установлен, сигнализирует о завершении передачи байта через последовательный порт и, если прерывание от сериального порта разрешено, вызывает его.

Если последовательный порт сконфигурирован в один из режимов – 1 или 3, то нужно установить скорость обмена данными для таймера 1 (таймер 2 тоже можно использовать для установки скорости обмена, но об этом немного позже), поместив соответствующее значение в регистр TH1.

В общем случае, если не используется бит PCON.7 для удвоения скорости обмена, значение регистра TH1 вычисляется по формуле:

TH1 = 256 – (Fosc / 384 / скорость обмена),

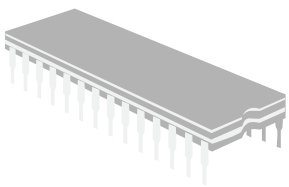
где *Fosc* – тактовая частота, на которой работает микроконтроллер.

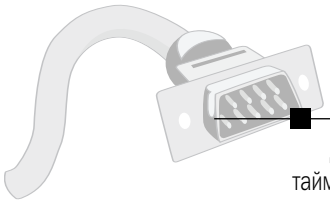
Например, при тактовой частоте 11,059 МГц и требуемой скорости обмена 9600 бод значение регистра TH1 должно быть равным

256 – (11059000 / 384 / 9600) = 256 – 2,999 = 253, или 0xFD в шестнадцатеричной нотации.

Если используется бит удвоения скорости PCON.7, то тактовая частота делится не на 384, а на 192. В этом случае для той же скорости обмена в 9600 бод значение TH1 должно равняться

256 – (11059000 / 192 / 9600) = 256 – 5.9999 = 250, или 0xFA в шестнадцатеричной нотации.





Для микроконтроллеров серии 8052 и совместимых с ними допускается использование таймера 2 в качестве генератора синхронизации скорости обмена для последовательного порта. В этом случае таймер 2, как и таймер 1, устанавливается в режим автоперезагрузки (auto-reload), но, в отличие от таймера 1, значение перезагрузки является 16-разрядным и помещается в пару регистров RCAP2H (старший байт) и RCAP2L (младший байт). В регистры RCAP2H:RCAP2L помещается значение, которое вычисляется по формуле:

65536 – (Fosc / 32 скорость обмена).

Так, например, для скорости обмена 9600 бод и тактовой частоты *Fosc* = 11059000 Гц регистры RCAP2H:RCAP2L должны содержать значение 65536 – 35,999 = 65500, или 0xFFDC в шестнадцатеричной нотации. При этом в регистр RCAP2H загружается старший байт, равный 0xFF, а в RCAP2L – значение 0xDC.

Если для таймера 2 установлен какой-нибудь или оба из флагов RCLK (T2CON.5) и/или TCLK (T2CON.4), то микроконтроллер для синхронизации последовательного обмена данными будет использовать только таймер 2.

Таким образом, для инициализации обмена данными через последовательный порт необходимо выполнить следующую последовательность шагов:

- задать режим работы последовательного интерфейса, установив соответствующие биты в регистре SCON;
- установить в качестве генератора синхронизации обмена данными либо таймер 1, либо таймер 2 с соответствующими настройками;
- начать обмен данными через последовательный порт.

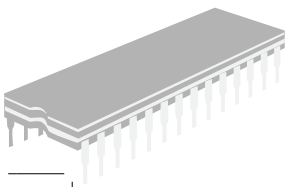
3.1. Запись данных в последовательный порт

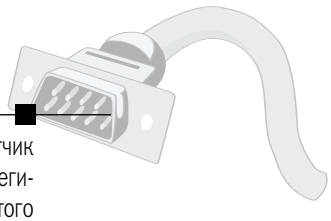
Перед тем как анализировать практические аспекты работы с последовательным портом, хочу сказать, что многие примеры программного кода будут разработаны полностью на ассемблере, несмотря на то что компилятор Keil C, который используется при разработке программного кода, позволяет относительно легко манипулировать с данными, проходящими через последовательный порт. Применение ассемблерного кода позволяет намного лучше понять работу аппаратной части микроконтроллера, особенно контроллера UART, и одновременно представить себе процесс передачи данных вживую. Кроме того, что намного более важно, для успешной отладки программ, в которых используется обмен данными по последовательному порту, знание процессов на уровне команд ассемблера в целом ряде случаев оказывается решающим.

Сказанное вовсе не означает, что во всех программах обязательно нужно использовать язык ассемблера, – многие относительно простые программы быстрее и проще написать, используя библиотечные функции C, такие, например, как printf, scanf и т.д. Тем не менее, зная, как работает протокол обмена на уровне команд ассемблера, не составит труда обнаружить ошибки в программах на C и во многих случаях улучшить качество таких программ.

Вернемся к программированию последовательного порта. Запись байта данных в последовательный порт можно выполнить, поместив передаваемый байт в специальный регистр, который обозначается как SBUF и имеет шестнадцатеричный адрес 0x99. Например, для передачи символа F можно использовать две команды:

```
MOV A, #'F'
MOV SBUF, A
```





Перед передачей байта в последовательный порт следует убедиться в том, что передатчик порта свободен и готов к отправке байта. Для этого нужно проверить состояние бита TI регистра SCON. Если он равен 0, следует подождать, пока порт будет свободен, и только после этого начать передачу.

Следующая короткая программа на ассемблере демонстрирует базовую технику передачи байта данных в последовательный порт. Здесь в последовательный порт непрерывно передается символ F. Перед передачей следующего символа делается небольшая задержка, а скорость обмена определяется таймером 1, который настроен на 9600 бод при тактовой частоте 11,059 МГц и работает в режиме автоперезагрузки (режим 2).

Вот исходный текст программы:

```
NAME    PROCS
    MAIN SEGMENT CODE
    CSEG AT 0
    USING   0
    JMP  start
    RSEG MAIN
start:
    MOV  SCON, #50h
    MOV  TH1, #0FDh
    ORL  TMOD, #20h
    SETB TR1
again:
    CLR  TI
    MOV  A, #'D'
    MOV  SBUF, A
    JNB  TI, $
    MOV  R0, #250
    DJNZ R0, $
    SJMP again
    END
```

Подробно проанализируем этот листинг, поскольку понимание того, как работает данный программный код, пригодится нам при анализе всех последующих программ. Первая команда

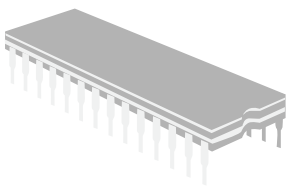
```
MOV SCON, #50h
```

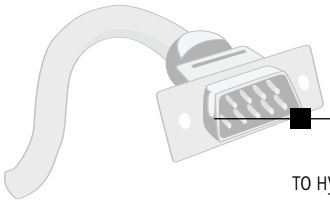
устанавливает 8-битовый режим обмена данными и скорость обмена, определяемую таймером 1. Кроме того, устанавливается бит разрешения приема данных через последовательный порт. Если порт работает только на передачу, этот бит можно не устанавливать, но в процессе разработки может понадобиться принимать данные из порта, поэтому лучше держать бит REN установленным и забыть про него.

Для скорости обмена данными 9600 бод при указанной тактовой частоте регистр TH1 должен содержать шестнадцатеричное значение 0xFD (ранее мы вычислили это значение). Команда

```
MOV TH1, #0FDh
```

помещает это значение в `TH1`.





Поскольку таймер 1 выбран в качестве генератора синхронизации при обмене данными, то нужно установить для него режим автоперезагрузки (auto-reload), что выполняет команда

```
ORL TMOD, #20h
```

Если остальные биты режима неважны, то вместо этой команды можно использовать MOV, например:

```
MOV TMOD, #20h
```

Далее выполняем запуск таймера 1 командой

```
SETB    TR1
```

Теперь все готово к передаче байта данных. Перед началом передачи данных нужно сбросить бит TI, что выполняется командой

```
CLR TI
```

Следующие две команды начинают передачу байта:

```
MOV R0, #'D'
MOV SBUF, R0
```

Далее программа анализирует состояние бита TI. Если передача байта в силу каких-то причин не закончена, этот бит сброшен. Никогда не следует начинать передачу следующего байта данных до тех пор, пока бит TI не установлен, поскольку этот байт будет потерян. Таким образом, команда

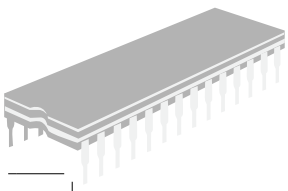
```
JNB TI, $
```

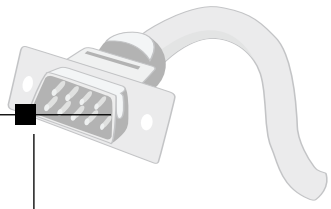
ожидает установки бита TI, чтобы можно было начать передачу следующего байта.

Далее, после некоторой временной задержки, программа возвращается на метку again и цикл начинается повторно. Не нужно забывать о том, что перед началом передачи байта в последовательный порт бит TI должен быть очищен.

Я уже упоминал о бите удвоения PCON.7 (он еще называется SMOD). При помощи этого бита можно установить частоту обмена данными, которая не может быть сконфигурирована только значением в TH1. Пример использования этого бита показан далее:

```
NAME    PROCS
    MAIN SEGMENT CODE
    CSEG AT 0
    USING   0
    JMP start
    RSEG MAIN
start:
    MOV SCON, #50h
    MOV TH1, #0FAh
    ORL PCON, #80h
```





```
    ORL  TMOD, #20h
    SETB TR1
again:
    CLR  TI
    MOV  A, #'D'
    MOV  SBUF, A
    JNB  TI, $
    MOV  R0, #250
    DJNZ R0, $
    SJMP again
    END
```

Здесь вместо команды

```
MOV TH1, #0FDh
```

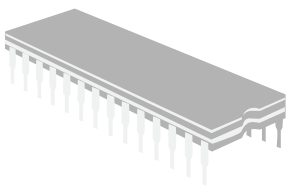
используются

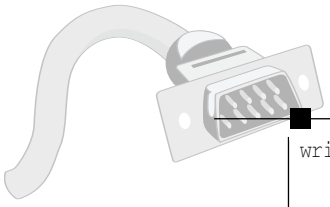
```
MOV TH1, #0FAh
ORL PCON, #80h
```

Как видно из этого фрагмента, содержимое регистра `TH1` в случае использования SMOD изменяется. Следует отметить, что бит SMOD не оказывает никакого влияния на скорость обмена, если в качестве генератора синхронизации используется таймер 2.

На основе алгоритма записи одного байта можно создать программу, позволяющую вывести текстовую строку в последовательный порт. В этом случае адрес первого элемента для записи является адресом строки. Следующая программа на ассемблере, исходный текст которой представлен далее, демонстрирует передачу строки в последовательный порт:

```
NAME     PROCS
    MAIN SEGMENT CODE
    myDATA    SEGMENT CODE
    CSEG AT 0
    USING    0
    JMP  start
    RSEG MAIN
start:
    MOV  SCON, #50h
    MOV  TH1, #0FDh
    ORL  TMOD, #20h
    SETB TR1

    MOV  DPTR, #txt
again:
    CLR  TI
    CLR  A
    MOVC A, @A+DPTR
    CJNE A, #1Bh, write_char
    JMP  $
```






```
write_char:
    MOV  SBUF, A
    JNB  TI, $
    INC  DPTR
    SJMP again
    RSEG myDATA
txt:    DB   'HELLO', 1Bh
    END
```

Проанализируем, как данная программа передает строку данных. Как и в предыдущей программе, устанавливаем режим автоперезагрузки таймера 1 на скорости 9600 бод и инициализируем регистр SCON соответствующим значением. В нашей программе определена константная строка `txt`, которая размещается в памяти программы и заканчивается символом ESC (`1Bh`). Символ ESC используется программой для определения конца строки и завершения вывода.

Для передачи строки `txt` загружаем ее адрес в регистр `DPTR` командой

```
MOV DPTR, #txt
```

Далее байт строки загружается в регистр `A`, для чего используются команды

```
CLR A
MOVC    A, @A+DPTR
```

Содержимое регистра-аккумулятора анализируется на равенство символу ESC, что будет свидетельствовать об окончании передачи. Это выполняется командой

```
CJNE    A, #1Bh, write_char
```

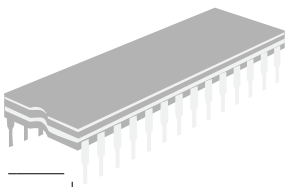
Если обнаружен символ ESC, программа входит в бесконечный цикл, если же это иной символ, то он передается в последовательный порт. После окончания передачи очередного символа регистр DPTR инкрементируется, указывая на следующий символ строки `txt`, и цикл передачи байта повторяется.

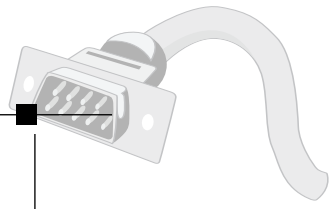
Эта программа является примером передачи строки символов в последовательный порт и демонстрирует некоторые программные приемы, используемые при передаче данных. При разработке подавляющего большинства программ данные в последовательный порт необходимо передавать из произвольной области памяти. В следующем примере показано, как можно передавать данные, размещенные во внутреннем ОЗУ микроконтроллера, в последовательный порт. Программа будет работать с 8052-совместимыми устройствами.

В этом примере строка-константа, размещенная в памяти программы, вначале копируется во внутреннюю область памяти данных, а затем строка из ОЗУ выводится в последовательный порт.

Вот исходный текст программы:

```
NAME    PROCS
    MAIN SEGMENT CODE
    myDATA   SEGMENT CODE
    RANDOM   SEGMENT IDATA
    CSEG AT  0
```

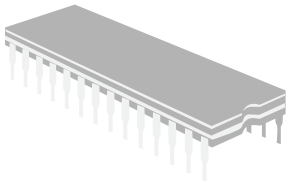




```
    USING    0
    JMP  start
    RSEG MAIN
start:
    MOV  SCON, #50h
    MOV  TH1, #0FDh
    ORL  TMOD, #20h
    SETB TR1

    MOV  DPTR, #txt
    MOV  R1, #vs
copy_byte:
CLR A
    MOVC A, @A+DPTR
    CJNE A, #1Bh, next
MOV @R1, #1Bh
    SJMP write_str
next:
    MOV  @R1, A
    INC  DPTR
    INC  R1
    SJMP copy_byte
write_str:
MOV R1, #vs
again:
    CLR  TI
    MOV  A, @R1
    CJNE A, #1Bh, write_char
    JMP  $

write_char:
    MOV  SBUF, A
    JNB  TI, $
    INC  R1
    SJMP again
    RSEG myDATA
txt:    DB   'HELLO, CRUEL WORLD!', 1Bh
RSEG RANDOM
vs: DS   20
    END
```

Программа использует сегмент данных RANDOM из области памяти 00–0xFF, для которого резервируется 20 байт. Вначале данные копируются в эту область памяти при помощи блока команд:

```
MOV DPTR, #txt
    MOV  R1, #vs
copy_byte:
CLR A
    MOVC A, @A+DPTR
```



```
    CJNE A, #1Bh, next
MOV @R1, #1Bh
    SJMP write_str
next:
    MOV  @R1, A
    INC  DPTR
    INC  R1
    SJMP copy_byte
```

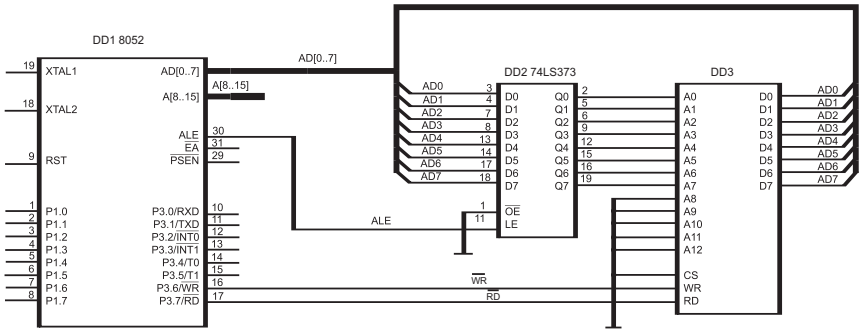
Здесь адрес исходной строки помещается в регистр DPTR, а адрес строки назначения vs в регистр R1, после чего выполняется побайтовое копирование данных до тех пор, пока не будет обнаружен символ ESC. После этого начинается вывод данных из области памяти vs в последовательный порт. Поскольку при копировании данных указатель в регистре R1 сместился от начала первого байта строки vs на число скопированных байтов, следует восстановить его значение:

```
write_str:
    MOV  R1, #vs
```

После этого строка, адресуемая регистром R1, выводится в последовательный порт.

При разработке многих систем для хранения данных используется внешняя статическая память, поэтому данные, которые требуется передавать по последовательному порту, необходимо вначале извлечь из оперативной памяти. Посмотрим, как можно вывести данные в последовательный порт из внешней памяти данных, для чего используем простое устройство на базе микроконтроллера 8052, принципиальная схема которого показана на рис. 3.2.

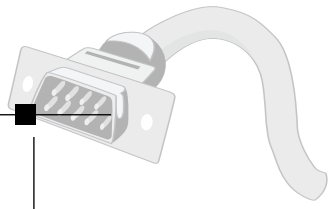
Рис. 3.2. *Подключение внешней памяти данных*



В устройстве на рис. 3.2 используется внешняя статическая память на микросхеме D3 (можно использовать любую подходящую микросхему). Схема подключения внешней памяти стандартная и особенностей не имеет. В цикле записи в память младшие 8 бит адреса фиксируются в регистре-защелке DD2 по спаду сигнала ALE, после чего по активному уровню одного из сигналов WR или RD байт данных записывается по указанному адресу памяти или считывается из указанного адреса. Для упрощения функционирования схемы используются только младшие 8 линий адреса, что позволяет адресовать ячейки памяти с адресами из диапазона 0–255.

Вывод данных из памяти в последовательный порт для данного устройства можно проиллюстрировать программой, исходный текст которой вы видите ниже:

```
NAME     PROCS
    MAIN SEGMENT CODE
    XSEG AT 1h
```



```
vs: DS   20
    CSEG AT 0
    USING    0
    JMP  start
    RSEG MAIN
start:
    MOV  SCON, #50h
    MOV  TH1, #0FDh
    ORL  TMOD, #20h
    SETB TR1

    MOV  DPTR, #vs
    MOV  R4, #10
    MOV  A, #'A'
copy_byte:
MOVX    @DPTR, A
    INC  A
    INC  DPTR
    DJNZ R4, copy_byte

MOV DPTR, #vs+9
    MOV  R4, #10
write_str:
    CLR  TI
    MOVX A, @DPTR
    MOV  SBUF, A
    JNB  TI, $
    DEC  DPL
    DJNZ R4, write_str
    SJMP $
END
```

Для работы с внешней памятью данных в программе объявлен сегмент внешних данных, в котором резервируется 20 байт адресного пространства:

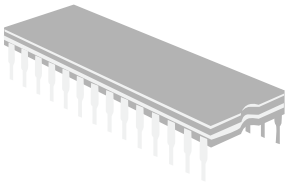
```
XSEG AT 1h
vs: DS   20
```

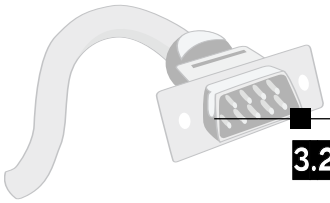
В качестве начального адреса здесь выбран 0x1 (это произвольный выбор, можно использовать и другие значения). Параметры последовательного порта устанавливаются, как и в предыдущих примерах (9600 бод при частоте 11,059 МГц). Программа последовательно записывает в адреса внешней памяти начиная с 0x1 10 символов, начиная с `'A'`, затем считывает их в обратном порядке из памяти и отправляет в последовательный порт.

Запись данных в память осуществляется в цикле `copy_byte`, а для адресации данных используется регистр `DPTR`, в который загружается начальный адрес области памяти, содержащийся в переменной `vs`. После того как данные сохранены в памяти, устанавливаем `DPTR` на адрес десятого байта памяти командой

```
MOV DPTR, #vs+9
```

и начинаем вывод байтов в последовательный порт в обратном порядке в цикле `write_str`.





3.2. Чтение данных из последовательного порта

Обратимся теперь к вопросам ввода данных через последовательный порт. Как уже упоминалось, ввод данных связан с флагами REN и RI. Для ввода данных необходима установка флага REN в регистре SCON, а установка флага RI свидетельствует об успешном приеме байта данных.

Разработаем простую программу на ассемблере, реализующую ввод данных с последовательного порта и обратную их передачу. Исходный текст такой программы представлен ниже:

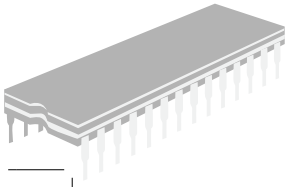
```
NAME    PROCS
    MAIN SEGMENT CODE
    CSEG AT 0
    USING   0
    JMP start
    RSEG MAIN
start:
    MOV SCON, #50h
    MOV TH1, #0FDh
    ORL TMOD, #20h
    SETB TR1

again:
    CLR TI
    CLR RI
    JNB RI, $
    MOV A, SBUF
    INC A
    MOV SBUF, A
    JNB TI, $
    SJMP again
    END
```

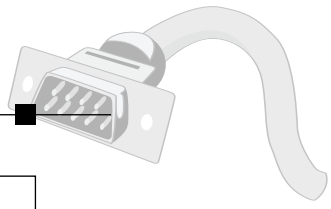
Программа ожидает приема символа через последовательный порт, для чего в начале цикла `again` бит `RI` сброшен в нулевое состояние, после чего программа ожидает приема символа, выполняя команду

```
JNB RI, $
```

Установка этого бита в 1 свидетельствует о приеме очередного символа. Далее полученный символ загружается в регистр-аккумулятор, где инкрементируется. Обработанный таким образом символ отправляется обратно в последовательный порт. Если, например, последовательный порт системы на базе 8051 соединен с последовательным портом персонального компьютера, то, запустив на ПК программу-терминал (она должна поддерживать работу с COM-портом), можно наблюдать результат работы программы на экране дисплея. Так, если пользователь вводит с клавиатуры ПК символ E, то на экране будет отображен символ F, т.е. символ, двоичный код которого на 1 больше вводимого.

Вот пример более сложной программы, в которой символ принимается с последовательного порта, декрементируется и записывается во внешнюю память данных, после чего считывается из памяти и выводится в последовательный порт. Эта программа работает с устройством на 8052, принципиальная схема которого изображена на рис. 3.2.





Исходный текст программы:

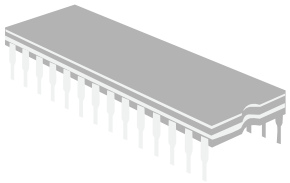
```
NAME    PROCS
    MAIN SEGMENT CODE
    XSEG AT 1h
vs: DS   1
    CSEG AT 0
    USING    0
    JMP  start
    RSEG MAIN
start:
    MOV  DPTR, #vs
    MOV  SCON, #50h
    MOV  TH1, #0FDh
    ORL  TMOD, #20h
    SETB TR1

again:
    CLR  TI
    CLR  RI
    JNB  RI, $
    MOV  A, SBUF
    DEC  A
    MOVX @DPTR, A

    MOVX A, @DPTR
    MOV  SBUF, A
    JNB  TI, $
    SJMP again
    END
```

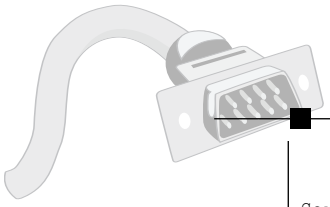
Исходный текст во многом напоминает предыдущий листинг, но здесь прочитанный символ декрементируется, затем записывается в память и считывается из нее. Эта программа демонстрирует программную технику при работе с внешней памятью и последовательным портом.

3.3. Прерывание последовательного порта

До сих пор мы рассматривали обработку данных от последовательного порта как часть основной программы управления. Тем не менее можно более эффективно работать с последовательным портом, используя специально выделенное для него прерывание, обработчик которого располагается по адресу 0x23. Прерывание последовательного порта вызывается при установке в 1 одного из флагов TI и RI, которые должны сбрасываться в программе-обработчике. Ввод символа через последовательный порт и его вывод через прерывание показан в следующей программе на ассемблере:

```
NAME    PROCS
MAIN    SEGMENT CODE
CSEG    AT 0
    USING    0
```






```
    JMP  start
    ORG  23h
SerialINT:
    JBC  TI, TRN
    JBC  RI, RCV
    RETI
TRN:
    RETI
RCV:
    MOV  A, SBUF
    ADD  A, #2h
    MOV  SBUF, A
    RETI
RSEG    MAIN
start:
    MOV  SCON, #50h
    MOV  TH1, #0FDh
    ORL  TMOD, #20h
    SETB TR1

    SETB ES
    SETB EA

    SJMP $
    END
```

Программа-обработчик прерывания размещается по фиксированному адресу 0x23, и ее первые две команды сбрасывают один из битов – `RI` или `TI`, в зависимости от того, какой из них вызвал прерывание:

```
JBC TI, TRN
JBC RI, RCV
```

В нашем обработчике прерывания основные действия выполняются при вводе символа через последовательный порт. К коду символа добавляется значение 2, после чего он передается обратно в последовательный порт. Команда

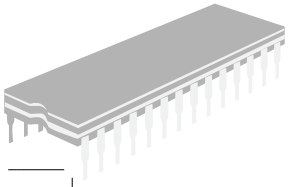
```
MOV SBUF, A
```

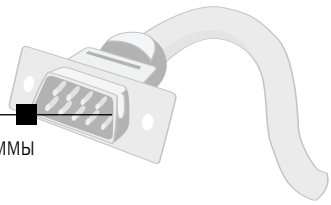
вызовет установку флага TI, но поскольку следующей за ней будет выполняться команда `RETI`, то новое прерывание не будет инициировано до завершения текущего.

В основной программе помимо конфигурирования последовательного порта требуется разрешить прерывание последовательного порта, что выполняется командами

```
SETB    ES
SETB    EA
```

Во всех примерах программного кода в качестве генератора синхронизации последовательного обмена использовался таймер 1. Для микроконтроллеров 8052 в целях установки скорости обмена можно применить таймер 2. Модифицируем наш предыдущий пример так,



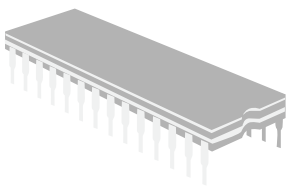


чтобы скорость обмена задавалась таймером 2. Исходный текст предыдущей программы после изменений будет выглядеть следующим образом:

```
NAME    PROCS
    T2CON    EQU 0C8h
    RCAP2H   EQU 0CBh
    RCAP2L   EQU 0CAh
MAIN    SEGMENT CODE
CSEG    AT 0
    USING    0
    JMP  start
    ORG  23h
SerialINT:
    JBC  TI, TRN
    JBC  RI, RCV
    RETI
TRN:
    RETI
RCV:
    MOV  A, SBUF
    ADD  A, #3h
    MOV  SBUF, A
    RETI
RSEG MAIN
start:
    MOV  SCON, #50h
    CLR  T2CON.0
    CLR  T2CON.1
    SETB T2CON.4
    SETB T2CON.5

    MOV  RCAP2H, #0FFh
    MOV  RCAP2L, #0DCh

    SETB T2CON.2

    SETB ES
    SETB EA

    SJMP $
    END
```

Прерывание ввода/вывода данных по последовательному порту не обязательно может вызываться при вводе символов с консоли удаленного терминала или их выводе. Установить биты RI и TI может и другое прерывание при возникновении определенной ситуации – например, при чтении входных сигналов датчиков, приходящих на выводы порта P1. Подобные программные конфигурации позволяют организовать асинхронную обработку событий, придавая системе дополнительную гибкость.

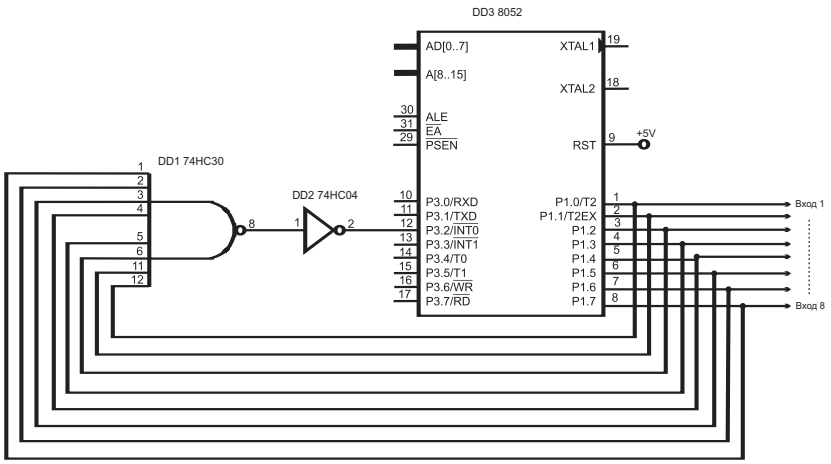
Рассмотрим пример разработки проекта системы обработки сигналов датчиков, подключенных к выводам порта P1. Поставим следующее условие: если хотя бы один из датчиков



сработал, он инициирует прерывание на входе P3.2/INT0, обработчик которого фиксирует код на входах порта P1 и инициирует его передачу по последовательному порту на удаленный терминал или ПК.

Аппаратная часть проекта представлена на рис. 3.3.

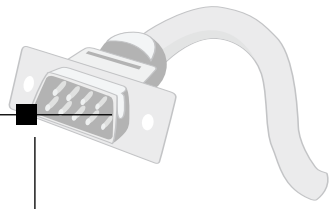
Рис. 3.3. *Схема обработки сигналов датчиков*



По этой схеме сигналы от дискретных датчиков приходят на выводы P1.0 – P1.7, причем активным уровнем сигналов считается низкий. Сигналы собираются элементом И–НЕ микросхемы DD1, который формирует сигнал, поступающий через инвертор DD2 на вход прерывания INT0. Активный уровень сигнала прерывания формируется в том случае, если сработал любой из датчиков, причем вход INT0 настроен (мы увидим это в программе) на срабатывание по спаду сигнала. При переходе на программу-обработчик прерывания формируется номер линии, на которой сработал датчик. Это число передается по последовательному порту на удаленный терминал или ПК.

Программное обеспечение этого проекта реализовано на языке ассемблера. Вот исходный текст программы:

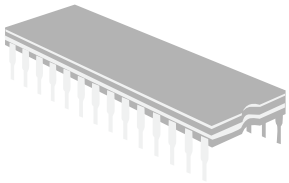
```
NAME PROCS
    T2CON    EQU 0C8h
    RCAP2H   EQU 0CBh
    RCAP2L   EQU 0CAh
    MAIN SEGMENT CODE
    tabl SEGMENT CODE
    CSEG AT 0
    USING    0
    JMP  start
Int0Isr:
    ORG  3h
    MOV  A, P1
    MOV  R4, #8
    SETB C
again:
    RLC  A
    JNC  next
```

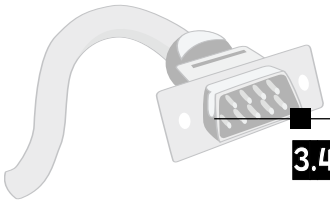


```
    DJNZ R4, again
    RETI
next:
    MOV  A, R4
    MOVC A, @A+DPTR
    MOV  SBUF, A
    RETI

SerialINT:
    ORG  23h
    CLR  TI
    RETI
    RSEG MAIN
start:
    MOV  DPTR, #num
    MOV  SCON, #50h
    CLR  T2CON.0
    CLR  T2CON.1
    SETB T2CON.4
    SETB T2CON.5
    MOV  RCAP2H, #0FFh
    MOV  RCAP2L, #0DCh
    SETB T2CON.2

    SETB IT0
    SETB EX0
    SETB ES
    SETB EA

    SJMP $
    RSEG tabl
num:    DB '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
    END
```

В этой программе определены два обработчика прерываний: прерывания INT0 и прерывания последовательного порта. Напомню, что прерывание INT0 возникает при перепаде 1–0 на одной из входных линий порта P1. Программа-обработчик прерывания INT0 сохраняет содержимое линий порта `P1` в аккумуляторе `A`, затем преобразует полученное значение в ASCII-представление с помощью таблицы `num`, после чего отправляет символ в последовательный порт. На этом обработчик прерывания INT0 заканчивает работу.

При помещении символа в последовательный порт устанавливается флаг TI, вызывая прерывание последовательного порта. Программа-обработчик этого прерывания выполняет единственную функцию – очищает бит TI, что дает возможность передаваться последующим данным. Теоретически можно было бы обойтись и без обработчика прерывания последовательного порта, для чего в обработчике прерывания INT0 нужно было бы дождаться окончания передачи символа, после чего сбросить флаг TI. Но при жестких временных требованиях такое ожидание может заблокировать работу обработчика прерывания INT0 на некоторое время, что недопустимо.





3.4. Работа с последовательным портом в Keil C51

Ввод/вывод данных через последовательный порт в среде разработки Keil C51 проще для программиста, поскольку имеется много библиотечных функций, позволяющих манипулировать с данными, не вникая в детали функционирования аппаратной части микроконтроллера. Для многих практических приложений вполне достаточно использования библиотечных функций, хотя для оптимизации программ по быстродействию лучше использовать отдельные процедуры, написанные на языке ассемблера.

Как и стандартный C, Keil C51 включает хорошо знакомые любому программисту функции, такие, например, как printf, scanf, gets, getchar и другие.

Основное отличие всех этих функций от стандартных (и это следует всегда помнить) в том, что ввод/вывод данных в Keil C подразумевает передачу или прием информации по последовательному порту. Так, например, оператор

```
printf("Hello, world\n");
```

будет выводить строку не на экран консоли, а в последовательный порт и, соответственно, в устройство, которое подключено к последовательному порту. Функция scanf, например, будет ожидать ввода не с клавиатуры, а с последовательного порта.

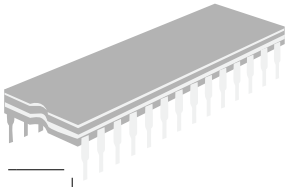
Таким образом, ввод/вывод информации в Keil C51 предполагает обмен данными с последовательным портом. Если устройство, в котором используется микроконтроллер, подсоединено, например, к последовательному порту персонального компьютера по интерфейсу RS-232, а на компьютере запущена программа-терминал, выполняющая обмен по COM-порту, то данные, посылаемые системой с 8051, будут отображаться на экране, а данные, вводимые с клавиатуры, будут приниматься через последовательный порт и обрабатываться, например, функцией gets программы на Keil C51.

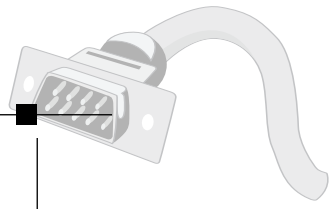
Поскольку функции ввода/вывода в Keil C51 ориентированы на работу с последовательным портом, то перед их использованием нужно сконфигурировать параметры порта, как мы это делали при запуске ассемблерных программ.

Вот исходный текст простейшей программы, выводящей текстовую строку в последовательный порт:

```
#include <stdio.h>
#include <REG52.H>

void main(void)
{
    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 1;
    printf("Hello, world\n");
}
```

Первые пять операторов программы выполняют установку параметров обмена данными по последовательному порту. Эту группу команд легко можно представить эквивалентными командами ассемблера:





```
MOV SCON, #50h
MOV TH1, #0FDh
ORL TMOD, #20h
SETB    TR1
SETB    TI
```

Эта группа операторов определяет стандартные для большинства микроконтроллеров настройки, соответствующие тактовой частоте 11,059 МГц и скорости обмена 9600 бод. Если микроконтроллер работает на другой тактовой частоте и/или используется другая скорость обмена, то значения параметров, естественно, должны быть пересчитаны. Отмечу, что стандартные функции ввода/вывода (printf, scanf, sprintf и др.) сами не устанавливают параметры обмена, а используют либо параметры по умолчанию, либо указанные разработчиком.

Если для указания корректной скорости обмена нужно использовать бит удвоения SMOD, то последовательность операторов, выполняющих настройку последовательного порта, может выглядеть так:

```
SCON = 0x50;
PCON |= 0x80;
TH1 = 0xFD;
TMOD |= 0x20;
TR1 = 1;
TI = 1;
```

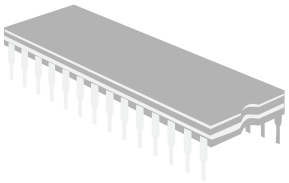
Для установки скорости обмена последовательного порта можно использовать (в системах с микроконтроллерами 8052) таймер 2. Например, для скорости обмена 9600 бод при тактовой частоте 11,059 МГц последовательность операторов перед `printf` будет такой:

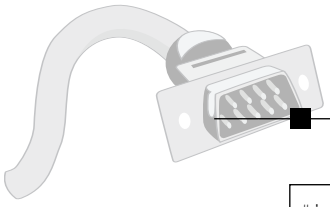
```
SCON = 0x50;
T2CON &= 0x0FC;
RCAP2H = 0x0FF;
RCAP2L = 0x0DC;
TR2 = 1;
TI = 1;
```

Обратите внимание, что для использования стандартных функций ввода/вывода следует предварительно указать файл заголовка, где они определены (в данном случае это стандартный файл `stdio.h`). Кроме того, нужно включить в исходный текст программы и файл заголовка для конкретного типа микроконтроллера (в нашем случае это `REG52.H`), в котором определены аппаратные адреса специальных регистров и портов ввода/вывода и соответствующие им аббревиатуры.

Рассмотрим пример использования библиотечных функций Keil C51 в программе ввода/вывода данных.

При этом используем аппаратную конфигурацию системы на базе 8052, показанную на рис. 3.2. Программа будет копировать строку данных из расширенной памяти микроконтроллера 8052 во внешнюю память данных, начиная с адреса 0x2, а затем выводить записанные данные из внешней памяти в последовательный порт. Это демонстрационная программа, и ее цель – показать различные способы вывода данных.





Исходный текст программы показан ниже:

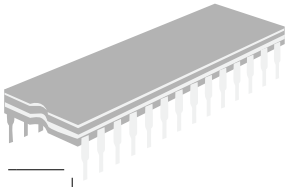
```
#include <stdio.h>
#include <string.h>
#include <REG52.H>

char xdata text[128] _at_ 0x2;
char idata mytext[] = "Data to be copied";

void main(void)
{
    char *p;
    int len;

    SCON = 0x50;
    TH1 = 0xFD;
    TMOD |= 0x20;
    TR1 = 1;
    TI = 1;

    len = strlen(mytext);
    p = memcpy(text, mytext, len);
    printf("%s\n", p);
    while (1);
}
```

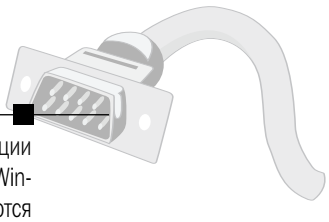
В программе исходная текстовая строка `mytext` размещается в расширенной памяти 8052, о чем свидетельствует атрибут `idata`. Эта строка должна быть скопирована в область внешней памяти `text` по абсолютному адресу `0x2`, где зарезервировано `128` байт (это значение выбирается произвольно, но следует учитывать размер исходной строки в байтах). Вначале исходная строка должна быть скопирована во внешнюю память, для чего используется функция `memcpy` (объявлена в файле заголовка `string.h`). Функция возвращает адрес области памяти, содержащей скопированные данные, который запоминается в переменной `p`. Функция `memcpy` в качестве третьего параметра принимает количество байтов, подлежащих копированию, поэтому это значение предварительно вычисляется с помощью библиотечной функции `strlen`.

После того как данные размещены во внешней памяти, они выводятся в последовательный порт функцией `printf`, принимающей в качестве единственного параметра указатель `p`, возвращенный `memcpy`.

3.5. Интерфейс систем на базе 8051 с персональным компьютером

Многие системы сбора и обработки информации на базе микроконтроллеров 8051 входят в состав других, более крупных систем. Очень часто 8051-совместимая система является звеном нижнего уровня в иерархической системе управления, передавая последней предварительно собранную и обработанную информацию, а также принимая от последней управляющие сигналы. В качестве системы верхнего уровня по отношению к модулю 8051 часто выступает либо персональный компьютер, либо другой модуль обработки информации, построенный, например, на базе 32-разрядного микроконтроллера с ядром ARM7.





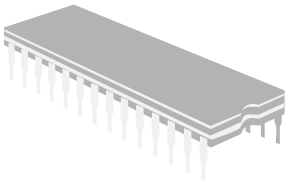
Здесь мы рассмотрим взаимодействие модуля 8051 сбора и обработки информации с персональным компьютером, на котором функционирует одна из операционных систем Windows 2000/XP/2003/Vista. Именно такие аппаратно-программные конфигурации используются во многих лабораторных и научных исследованиях, а также в целом ряде промышленных систем.

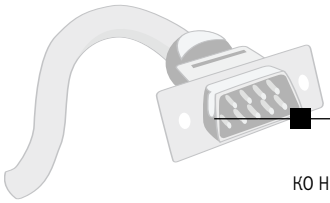
Взаимодействие системы на базе микроконтроллера 8051 с персональным компьютером может осуществляться по последовательному порту, по USB-порту или посредством сети. В настоящее время многие системы используют последовательный интерфейс для обмена информацией. Мы рассмотрим два простых примера взаимодействия программного обеспечения ПК и 8051-совместимой системы.

В первом примере будет показано, как управлять выходными сигналами порта P1 микроконтроллера 8051/8052 из программы, написанной на C++ и работающей под управлением операционной системы Windows XP. Модуль микроконтроллера соединен с ПК посредством последовательной линии связи с использованием выходов TxD и RxD. Для эксперимента был использован контроллер Rita51 фирмы Rigel Corp.

Вот исходный текст программы, написанной на языке ассемблера и зашитой во флеш-память микроконтроллера:

```
NAME    PROCS
    T2CON    EQU 0C8h
    RCAP2H   EQU 0CBh
    RCAP2L   EQU 0CAh
    CSEG AT 0
    USING    0
    JMP start
SerINT:
    ORG 23h
    JBC RI, RCV
    RETI
RCV:
    MOV P1, SBUF
RETI
;-------------
start:
    MOV P1, #0h
    MOV SCON, #50h
    CLR T2CON.0
    CLR T2CON.1
    SETB T2CON.4
    SETB T2CON.5

    MOV RCAP2H, #0FFh
    MOV RCAP2L, #0B2h;9600 бод при Fosc = 24.0МГц
    SETB T2CON.2

    SETB ES
    SETB EA
    SETB TI
    SJMP $
    END
```





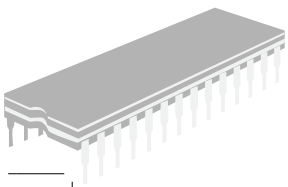
Как видно из исходного текста, последовательный порт микроконтроллера настроен только на прием байта данных в режиме прерывания. Программа-обработчик прерывания располагается в памяти программы по адресу 0x23, и единственное, что она делает, – посылает принятый байт в порт P1, устанавливая тем самым на его выводах полученную двоичную комбинацию.

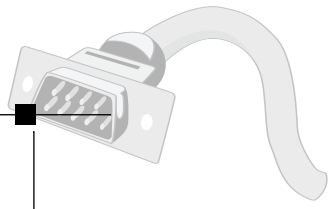
Скорость обмена данными по последовательному порту определяется установками таймера 2, который является генератором синхронизации при обмене данными. Таймер 2 устанавливается в режим автоперезагрузки и назначает скорость обмена данными 9600 бод при тактовой частоте 24,0 МГц.

Для разработки управляющей программы ПК можно воспользоваться одним из компиляторов C++ для создания Windows-приложений. В данной разработке использован компилятор Microsoft Visual C++ .NET, хотя приведенный далее программный код легко адаптируется для других компиляторов.

Ниже приведен исходный текст консольного приложения Windows, посылающего управляющий код для микроконтроллера 8052:

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

void main(void)
{
    HANDLE hCom;
    char *pcComPort = "COM1";
    DCB dcb;

    DWORD bytesWritten;
    __int8 i1;

hCom = CreateFile(      pcComPort,
                  GENERIC_READ | GENERIC_WRITE,
                  0,
                  NULL,
                  OPEN_EXISTING,
                  0,
                  NULL);
    if (hCom == INVALID_HANDLE_VALUE)
    {
         printf("COM1 opening error!\n");
         return;
    }

GetCommState(hCom, &dcb);
printf("COM1 baud rate is %d\n", dcb.BaudRate);
    {
while (true)
    {
         printf("\nEnter value to be outputted at COM1 port:");
         scanf("%du", &i1);
         WriteFile(      hCom,
```





```
                    &i1,
                    sizeof(i1),
                    &bytesWritten,
                    NULL);
    }
}
CloseHandle(hCom);
return;
}
```

Для читателей, знакомых с программированием на C++ в среде Windows, показанный исходный текст не представляет особой сложности. В этой программе вначале открывается файл последовательного порта `COM1` с помощью функции WINAPI `CreateFile`. При открытии коммуникационных портов в Windows необходимо указать режим монопольного доступа, установив третий параметр функции `CreateFile` равным `0` и указав атрибут `OPEN_EXISTING`.

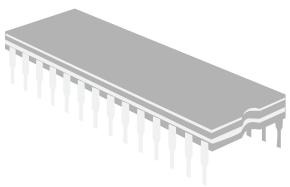
В случае успешного открытия порта функция `CreateFile` возвращает дескриптор hCom коммуникационного порта, куда и будет записываться введенный с клавиатуры консоли символ. Функция `GetCommState` позволяет получить параметры последовательного порта, такие как скорость обмена, наличие бита четности и т.д. Полученные данные записываются в структуру `dcb` типа `DCB`. В данном случае для контроля за скоростью обмена данными последовательного порта персонального компьютера на экран консоли выводится значение переменной `dcb.BaudRate`. Думаю, излишне напоминать, что параметры последовательного порта ПК должны совпадать с параметрами, установленными для порта микроконтроллера, иначе обмен данными будет невозможен.

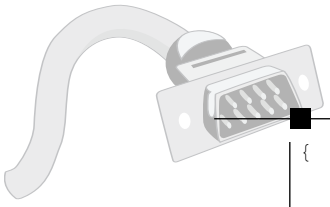
Далее, программа в цикле `while` выводит в последовательный порт ПК значение, введенное с помощью библиотечной функции `scanf`. Запись байта данных в последовательный порт осуществляется WINAPI-функцией `WriteFile`. Передаваемый в микроконтроллер байт затем выводится в порт P1.

Следующий программный проект, который мы разработаем, позволяет прочитать по последовательному порту ПК сообщение, выводимое микроконтроллером при возникновении прерывания INT0. Как и в предыдущем примере, микроконтроллер 8052 соединен с персональным компьютером по последовательному порту. Программа-обработчик прерывания INT0, выполняющаяся в 8052-совместимой системе, при появлении перепада 1–0 сигнала на входе P3.2/INT0 посылает сообщение в последовательный порт, которое затем будет принято и обработано программой, выполняющейся на ПК.

Вот исходный текст программы, написанной на C51 для микроконтроллера 8052:

```
#include <stdio.h>
#include <REG52.H>

idata char text[] = "Interrupt 0 occured!";

void EX0Isr (void) interrupt 0 using 0 {
    EX0 = 0;
    puts(text);
    EX0 = 1;
}

void main(void)
```



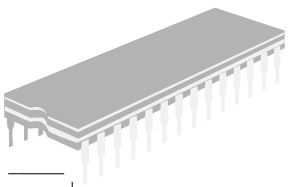


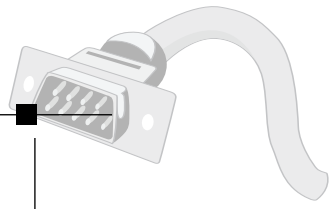

```
{
    SCON = 0x50;
    PCON |= 0x80;
    TH1 = 0xF3; //9600 бод при Fosc = 24.0МГц
    TMOD |= 0x20;
    TR1 = 1;
    TI = 1;

    IT0 = 1;
    EX0 = 1;
    EA = 1;

    while(1);
}
```

Собственно, основная работа выполняется в программе-обработчике прерывания INT0. Поскольку вывод байта данных в последовательный порт занимает определенное время, то на время этой операции запросы на прерывания, которые могут поступить на линию INT0, блокируются, а после вывода сообщения вновь разрешаются. Конечно, вывод строки символов в последовательный порт в обработчике прерывания в реальных системах вряд ли эффективен, поэтому данный исходный текст следует рассматривать лишь как демонстрацию возможностей.

Программа, принимающая текстовую строку от микроконтроллера и работающая на ПК под управлением Windows, представлена следующим исходным текстом:

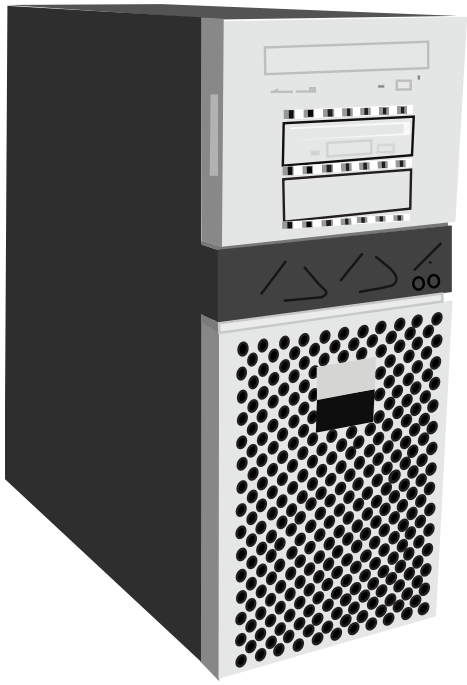
```
#include <windows.h>
#include <stdio.h>

void main(void)
{
    HANDLE hCom;
    char *pcComPort = "COM1";
    DCB dcb;

    DWORD bytesRead;
    char buf[128];

hCom = CreateFile(pcComPort,
GENERIC_READ | GENERIC_WRITE,
              0,
              NULL,
              OPEN_EXISTING,
              0,
              NULL);
if (hCom == INVALID_HANDLE_VALUE)
    {
    printf("COM1 opening error!\n");
    return;
    }
```





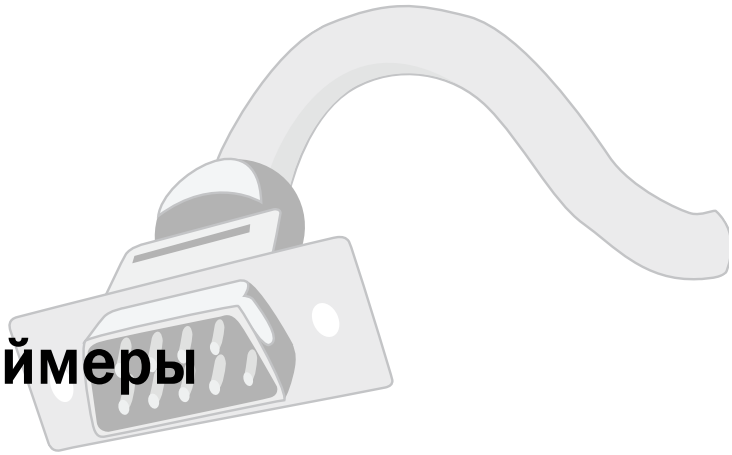
```
GetCommState(hCom, &dcb);
printf("COM1 baud rate is %d\n", dcb.BaudRate);
printf("Waiting for data from COM1...\n");
do {
    ReadFile(hCom,
buf,
         sizeof(buf),
         &bytesRead,
         NULL);
    if (bytesRead > 0)
    {
         buf[bytesRead] = '\0';
         printf(buf);
         printf("\n");
    }

    }while (true);
CloseHandle(hCom);
return;
}
```

Здесь, как и в предыдущем Windows-приложении, вначале открывается последовательный порт `COM1` (WINAPI-функция `CreateFile`), затем строка из последовательного порта считывается в буфер `buf` функцией `ReadFile`. Если количество `bytesRead` принятых байтов не равно `0`, то функция `printf` выводит принятый байт на экран консоли ПК.



Встроенные таймеры

4 Встроенные таймеры



Микроконтроллеры серии 8051, как мы уже знаем, имеют два, а серии 8052 – три встроенных таймера. Наличие на кристалле таких устройств позволяет строить системы, работающие в реальном времени. Встроенные таймеры чаще всего используются при измерениях различных временных параметров внешних по отношению к микроконтроллеру сигналов, для отработки временных задержек и для выполнения периодических операций.

Таймеры 8051-совместимых систем являются 16-разрядными и аппаратно реализованы как два 8-разрядных регистра, которые могут программироваться отдельно. Программный доступ к таймерам осуществляется посредством группы регистров специальных функций:

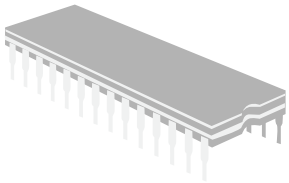
- TH0 содержит старший байт таймера 0;
- TL0 содержит младший байт таймера 0;
- TH1 содержит старший байт таймера 1;
- TL1 содержит младший байт таймера 1;
- TH2 содержит старший байт таймера 2 (только для 8052);
- TL2 содержит младший байт таймера 2 (только для 8052);
- TCON – регистр управления таймерами 0 и 1;
- T2CON – регистр управления таймером 2 (только для 8052);
- TMOD – регистр режимов работы таймеров.

Как работает таймер? Прежде всего следует отметить, что любой из таймеров может пребывать в одном из двух состояний: либо таймер работает, либо он остановлен. Когда таймер работает, он может выполнять одну из трех функций:

- таймера синхронизации;
- счетчика событий;
- генератора тактовой частоты для последовательного порта.

Рассмотрим более подробно каждый из режимов. При работе в режиме синхронизации 16-разрядное содержимое регистров таймера THx и TLx (x = 0, 1, 2) инкрементируется в каждом машинном цикле (рис. 4.1).

Подавляющее большинство микроконтроллеров линейки 8051 работает с машинными циклами, состоящими из 12 циклов тактовой частоты. В этом случае таймер будет инкрементироваться через каждые 12 циклов тактовой частоты. Например, если тактовая частота, с которой работает микроконтроллер, равна 11,059 МГц (стандартная частота для большинства микроконтроллеров), то таймер будет работать на частоте, равной 11,059 МГц / 12 = = 921583 Гц.



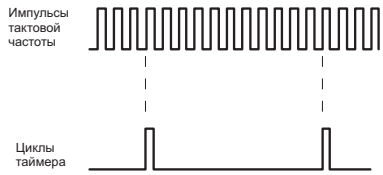


Рис. 4.1.
Режим синхронизации

Поскольку таймер считает вперед, то наступит момент, когда оба 8-разрядных регистра будут содержать шестнадцатеричные значения FF или 65535. В следующем машинном цикле значения регистров THx и TLx будут обнулены, после чего счет продолжится с 0. В этом случае говорят о переполнении таймера. Ситуация переполнения очень часто используется в программах, тем более что момент переполнения фиксируется путем установки определенных битов (они имеют обозначения TFx, x = 0, 1, 2), называемых битами переполнения, в регистре управления и контроля таймеров (регистр TCON).

Рассмотрим подробнее, как устанавливаются режимы работы таймеров и как они управляются. На базовом кристалле 8051 имеется два регистра – TMOD и TCON. Для клонов 8052 имеется и регистр T2CON, предназначенный для управления таймером 2. В отдельных битах регистра TMOD задаются режимы работы таймеров, а старшие 4 бита регистра TCON управляют работой таймеров 0 и 1.

Структура регистра TMOD показана на рис. 4.2.

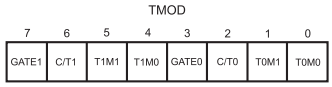


Рис. 4.2.
Регистр TMOD

В зависимости от содержимого битов TxM0 и TxM1 (x = 0, 1) соответствующий таймер может работать в одном из следующих режимов:

- 13-разрядного таймера (TxM1 = 0, TxM0 = 0);
- 16-разрядного таймера (TxM1 = 0, TxM0 = 1);
- автоперезагрузки – auto-reload (TxM1 = 1, TxM0 = 0);
- двух автономных 8-разрядных таймеров (TxM1 = 1, TxM0 = 1).

Биты 2–3 и 6–7 определяют дополнительные условия работы таймеров. Так, если установлен бит 2, то таймер 0 будет работать в качестве счетчика событий, считая перепады 1–0 импульсов на выводе P3.4 микроконтроллера. Аналогично, то же будет выполнять и таймер 1 на выводе P3.5, если бит 6 будет установлен. Если бит C/T0 или C/T1 сброшен, т.е. установлен в 0, то соответствующий таймер будет инкрементироваться в каждом машинном цикле.

Если установлен бит GATE0, то таймер 0 будет работать только при наличии высокого уровня сигнала на выводе P3.2. Если данный бит сброшен, то таймер 0 будет работать вне зависимости от сигнала на выводе P3.2. Бит GATE1 выполняет точно такие же функции, но для таймера 1.

Наиболее часто используются режим работы таймеров в качестве 16-разрядных таймеров и режим автоперезагрузки (синхронизация приема/передачи данных по последовательному порту). Для таймера 2, используемого в клонах 8052, существуют дополнительные возможности, которые мы рассмотрим отдельно.

Управление запуском и остановкой таймеров осуществляется посредством установки/сброса соответствующих битов в регистре управления TCON. Старшие 4 бита регистра TCON, используемые в программах, показаны на рис. 4.3.

Здесь биты TF1 и TF0 устанавливаются микроконтроллером при переполнении таймера 1 или таймера 0 соответственно. Биты TR1 и TR0, будучи установленными в 1, запускают таймер 1 и таймер 0 соответственно. Если бит TRx сбрасывается, то соответствующий ему таймер блокируется.

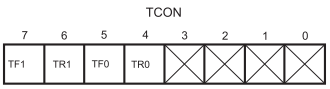


Рис. 4.3. *Биты запуска управления*

Оба регистра, TMOD и TCON, могут адресоваться побитово, поэтому, например, такие команды, как

```
ORL TCON, #80h
```

и

```
SETB TCON.7
```

являются эквивалентными. Побитовая адресация очень удобна при манипуляциях с отдельными битами, при этом легко избежать случайного изменения остальных битов, как это может случиться при выполнении обычных команд.

Для иллюстрации работы таймеров рассмотрим несколько примеров программирования этих устройств. Для разработки тестовых программ будем использовать среду Keil uVision3. Большинство примеров будет написано как на языке C, так и на ассемблере. Часть программного обеспечения будет использовать в качестве основной программу на C, из которой вызываются отдельные процедуры, написанные на ассемблере.

4.1. Режим работы таймера в качестве 16-разрядного таймера

Первый пример иллюстрирует работу таймера 0 в качестве 16-разрядного таймера, который используется для переключения светодиода, подсоединенного к биту 0 порта P1, каждые 15 с в противоположное состояние. Вначале рассчитаем требуемые временные характеристики, выбрав для этого тактовую частоту микроконтроллера, равную 11,059 МГц.

В этом случае таймер 0 инкрементируется 11059000 / 12 = 921583 раз в секунду. При этом интервал между тиками составляет 65536 / 921583 = 0,071 с (берем 3 знака после запятой). Следовательно, по прошествии 15 с таймер должен перегрузиться 15 / 0,071 = 211,26 раза. В этом случае можно принять без сколько-нибудь значительной потери точности число перегрузок равным 211. Таким образом, бит P1.0 нужно переключать после 211 перезагрузок таймера 0.

Приступим к разработке нашего первого проекта. Вот как будет выглядеть принципиальная электрическая схема устройства (рис. 4.4).

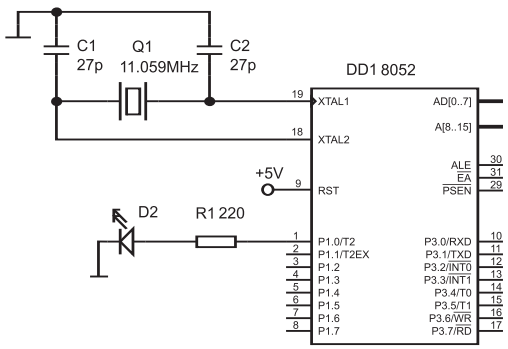


Рис. 4.4. *Схема переключения светодиода*

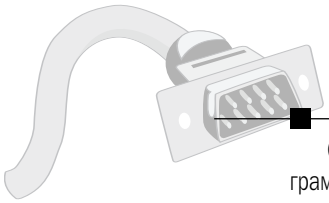


Схема устройства очень проста и не нуждается в дополнительном анализе. Тестовую программу разработаем в Keil uVision3. Для этого создадим новый проект и добавим в него файл с расширением .asm (имя можно выбрать любое). В файле должен содержаться следующий исходный текст:

```
    NAME      PROCEDURES
    MAIN      SEGMENT CODE
    CSEG      AT 0
    USING     0
    JMP       start
    RSEG      MAIN
start:
    MOV       P1, #0FEh
    MOV       TH0, #0h
    MOV       TL0, #0h
    MOV       TMOD, #1h
    MOV       R0, #211
    SETB      TCON.4
again:
    JNB       TCON.5, $
    CLR       TCON.5
    DJNZ      R0, skip
    CPL       P1.0
    MOV       R0, #211
skip:
    SJMP      again
    END
```

Проанализируем исходный текст программы. Поскольку бит 0 порта `P1` используется для вывода, то следует установить соответствующий режим, выполняемый командой

```
MOV P1, #0FEh
```

Далее инициализируем старший и младший байты таймера 0 нулевыми значениями с помощью команд

```
MOV TH0, #0h
MOV TL0, #0h
```

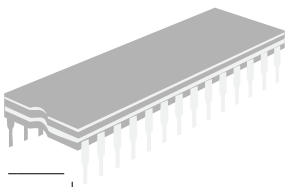
Выбираем режим 16-битового таймера, записав в регистр `TMOD` значение `1`. В качестве счетчика перезагрузок таймера будем использовать регистр `R0`, в который поместим значение `211`, соответствующее 15 с, при помощи команды

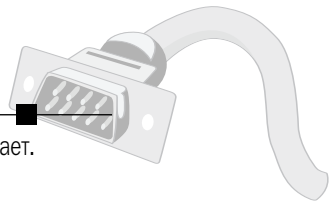
```
MOV R0, #211
```

Теперь можно запустить таймер командой

```
SETB    TCON.4
```

После этого таймер 0 будет инкрементироваться в каждом машинном цикле. Программа будет ожидать возникновения переполнения, о чем просигнализирует установленный бит TF0,





который следует сбросить программно, поскольку микроконтроллер этот бит не сбрасывает. Эта последовательность действий выполняется командами

```
JNB     TCON.5, $
CLR     TCON.5
```

Следующая команда декрементирует значение счетчика переполнений и проверяет его на равенство нулю:

```
DJNZ    R0, skip
```

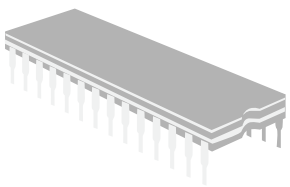
Если значение R0 достигло 0, т.е. прошло 15 с, то необходимо инвертировать бит P1.0 и, кроме того, загрузить R0 повторно тем же значением для организации нового 15-секундного цикла:

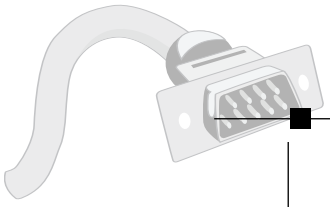
```
DJNZ    R0, skip
CPL P1.0
MOV R0, #211
```

Если значение R0 не равно 0, программа идет на новый цикл. После компиляции программы полученный HEX-файл можно использовать для тестирования на конкретном оборудовании. Необходимо учитывать, что если тактовая частота конкретного микроконтроллера отличается от 11,059 МГц, то следует пересчитать временные параметры, указанные в тестовом примере.

Разработаем версию тестовой программы на языке C, создав, как обычно, пустой проект в Keil uVision3 и добавив в него C-файл. Исходный текст такого файла может выглядеть следующим образом:

```
#include <stdio.h>
#include <REG52.H>

sbit LED_BLINK = P1^0;

void main(void)
{
unsigned int counter = 211;
P1 = 0x0FE;
TH0 = 0x0;
TL0 = 0x0;
TMOD |= 0x1;
TR0 = 1;

while (1)
{
    if (TF0 == 1)
    {
        TF0 = 0;
        counter—;
        if (counter == 0)
```





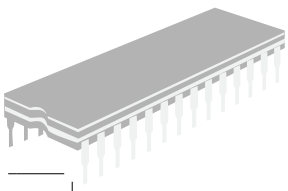

```
        {
            counter = 211;
            LED_BLINK = ~LED_BLINK;
        }
    }
}
}
```

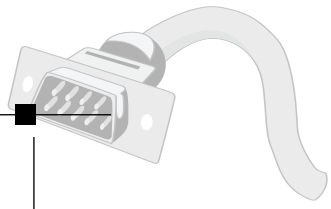
В наших примерах мы инициализировали старший и младший байты таймера 0 нулевыми значениями, хотя можно использовать произвольные значения. Это бывает удобно, когда в качестве интервала перезагрузки таймера необходимо выбрать значение, кратное какой либо величине, например 0,1 или 0,001 с. Посмотрим, например, какое значение следует занести в регистры таймера, если выбран интервал перезагрузки, равный 0,01 с.

Мы знаем, что при нулевых значениях в регистрах TH0 и TL0 таймер перезагружается каждые 0,071 с, проходя при этом 65536 циклов инкремента. В этом случае для определения необходимых значений для TH0 и TL0 можно воспользоваться выражением 65536 – – (65536 × 0,01 / 0,07). После округления получим значение 56306 или, в шестнадцатеричной форме, DBF2. Таким образом, в регистр TH0 нужно поместить шестнадцатеричное значение DB, а в TL0 – значение F2. Поскольку интервал перезагрузки изменился с 0,071 на 0,01 с, то значение счетчика перезагрузок в R0 теперь должно стать равным 1500 (для 15-секундного интервала!). Но регистр R0 может содержать максимальное значение 0xFF, поэтому для формирования счетчика перезагрузок таймера можно использовать еще один регистр, например R1, организовав при этом дополнительный цикл.

Исходный текст рассмотренной ранее ассемблерной программы изменится и будет выглядеть следующим образом:

```
    NAME    PROCEDURES
    MAIN    SEGMENT CODE
    CSEG    AT 0
    USING   0
    JMP     start
    RSEG    MAIN
start:
    MOV     P1, #0FEh
    MOV     TH0, #0DBh
    MOV     TL0, #0F2h
    MOV     TMOD, #1h
    MOV     R0, #10
    MOV     R1, #150

    SETB    TCON.4
again:
    JNB     TCON.5, $
    CLR     TCON.5
    DJNZ    R1, skip
    MOV     R1, #150
    DJNZ    R0, skip
    MOV     R0, #10
    CPL     P1.0
```





```
skip:
    MOV     TH0, #0DBh
    MOV     TL0, #0F2h
    SJMP    again
    END
```

Обратите внимание, что в счетчике перезагрузок используются регистры R0 и R1, при этом команда

```
DJNZ    R0, skip
```

образует внешний цикл, а команда

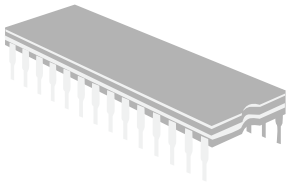
```
DJNZ    R1, skip
```

внутренний.

Общее значение счетчика перезагрузок при этом равно произведению значений R0 и R1, т.е. 10 × 150 = 1500. Таким образом, в ассемблерной программе мы фактически используем 16-разрядный счетчик перезагрузок, построенный на регистрах R0 и R1.

Проанализируем, как изменится исходный текст программы, написанной на C, при использовании интервала перезагрузки 0,01 с:

```
#include <stdio.h>
#include <REG52.H>

sbit LED_BLINK = P1^0;

void main(void)
{
unsigned int counter = 1500;

P1 = 0x0FE;
TH0 = 0xDB;
TL0 = 0xF2;
TMOD |= 0x1;
TR0 = 1;

while (1)
{
    if (TF0 == 1)
    {
    TF0 = 0;
    counter–;
    if (counter == 0)
    {
    counter = 1500;
    LED_BLINK = ~LED_BLINK;
    }
    TH0 = 0xDB;
```



```
    TL0 = 0xF2;
    }
    }
}
```

Основные отличия от предыдущего листинга на С состоят в том, что регистры `TH0` и `TL0` проинициализированы соответствующими значениями и изменено значение счетчика перезагрузок `counter`. Кроме того, в группу команд внешнего оператора `if` добавлены два оператора

```
TH0 = 0xDB;
TL0 = 0xF2;
```

В остальном программный код остался без изменений.

Рассмотренные примеры демонстрируют методику работы с устройствами, известную как опрос (polling). В этом случае микроконтроллер большую часть времени занят опросом состояния устройства (в наших примерах – таймера 0) и никакой другой полезной работы не выполняет. Такая методика используется в несложных программах, не слишком жестко завязанных с временными интервалами и выполняющих какую-либо одну функцию (алгоритм). Например, аппаратно-программное решение из предыдущего примера при небольшой доработке можно использовать для разработки несложного генератора импульсов.

Если аппаратно-программная конфигурация предусматривает выполнение многофункционального алгоритма, то следует использовать иные решения. Если, например, система должна выполнять опрос датчиков и периодически генерировать определенные сигналы на портах ввода/вывода, то в таких случаях алгоритм опроса лучше выполнять в основной программе, а для периодической генерации сигнала использовать прерывание таймера.

4.2. Прерывания таймеров

Для работы таймеров 0, 1 и 2 в режиме прерываний в микроконтроллерах 8051/8052 предусмотрены векторы прерываний, показанные в табл. 4.1.

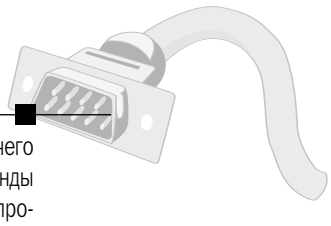
Таблица 4.1. *Прерывания таймеров*

Прерывание	Флаг прерывания	Адрес обработчика прерывания
Таймер 0	TF0 (TCON.5)	0x0B
Таймер 1	TF1 (TCON.7)	0x1B
Таймер 2	TF2 (T2CON.7)	0x2B

Так, например, при возникновении прерывания таймера 0 микроконтроллер автоматически переходит на выполнение программы обработки прерывания по шестнадцатеричному адресу 0B, при возникновении прерывания таймера 1 – на выполнение программы, находящейся по адресу 1B, и т.д.

Как возникает или, по-другому, инициируется прерывание таймера?

Всем источникам возможных прерываний поставлены в соответствие так называемые флаги прерывания, представляющие собой битовые значения и обозначаемые, как показано в табл. 4.1. Так, например, при установке флага TF0 в единичное состояние микроконтроллеру посылается сигнал прерывания. Микроконтроллер анализирует источник прерывания и определяет, что таковым является таймер 0. После этого выполнение текущей программы приостанавливается и управление передается программе-обработчику прерывания таймера 0 по адресу 0x0B.



Поскольку программе обработки прерывания по умолчанию выделены 8 байт памяти, чего во многих случаях явно недостаточно, то обработчик прерывания может начинаться с команды JMP, которая выполняет переход на основную область памяти, где размещается остальной программный код обработчика.

Напомню, что при переходе к обработке прерывания выполнение текущей программы прерывается и в стеке сохраняется адрес следующей за выполняемой в данный момент команды. По окончании обработки прерывания контекст основной программы восстанавливается с помощью команды RETI, которой должен заканчиваться обработчик прерывания.

Можно сказать, что прерывание в определенном смысле является «невидимым» для основной программы.

Для того чтобы программа обработки прерывания могла выполняться, необходимы три условия:

- наличие программы-обработчика прерывания, размещенной по определенному адресу в начальной области памяти;
- установка флага соответствующего источника прерывания;
- установка соответствующих битов в регистре разрешения прерываний и, при необходимости, в регистре приоритетов.

Регистр разрешения прерываний обозначается как IE и входит в группу специальных регистров (SFR). Любое из доступных прерываний может быть разрешено или запрещено посредством установки или сброса определенных битов в регистре разрешения прерываний. Биты прерываний таймеров регистра прерываний показаны в табл. 4.2.

Таблица 4.2. *Биты разрешения прерывания таймеров*

Бит	Наименование	Шестнадцатеричный адрес	Функция
7	EA	0xAF	Разрешение/запрет всех прерываний
5	ET	20xAD	Разрешение прерывания таймера 2 (8052)
3	ET	10xAB	Разрешение прерывания таймера 1
1	ET	00xA9	Разрешение прерывания таймера 0

Регистр IE допускает побитовую адресацию. Так, например, для разрешения прерывания от таймера 0 можно выполнить одну из команд:

```
MOV IE, #2h
SETB ET0
```

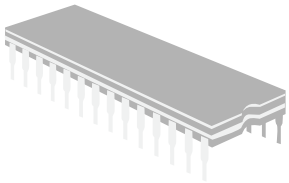
или указав номер бита:

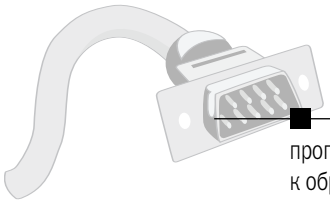
```
SETB IE.1
```

Таким образом, для выполнения, например, программы обработки прерывания от таймера 0 при возникновении прерывания необходимо предварительно выполнить последовательность команд:

```
SETB EA
SETB ET0
```

Несколько слов о флагах прерываний. За исключением флагов внешних прерываний 0 и 1 (IE0 и IE1 соответственно) программа-обработчик прерывания должна самостоятельно сбрасывать флаг прерывания, дабы избежать рекурсивных вызовов прерывания и краха





программы. Флаги внешних прерываний IE0 и IE1 сбрасываются аппаратно при переходе к обработке соответствующего прерывания. Флаги прерываний таймеров 0, 1 и 2 устанавливаются при возникновении переполнения в процессе счета. Кроме того, флаг прерывания таймера 2 может устанавливаться и при перепаде сигнала из 1 в 0 на выводе T2EX микроконтроллера (этот случай мы будем рассматривать отдельно).

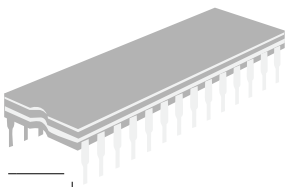
Далее мы рассмотрим на примере использование прерывания для обработки события от таймера 0. В качестве примера выполним модификацию предыдущих программ таким образом, чтобы переключение светодиода на P1.0 выполнялось при возникновении прерывания от таймера 0 по прошествии 5-секундного интервала времени.

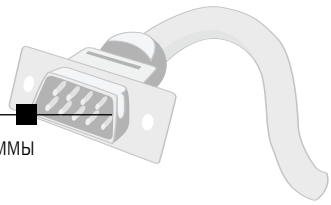
Принципиальная электрическая схема остается такой, как показана на рис. 4.4. Программа на ассемблере для этого случая будет содержать следующий исходный текст:

```
    NAME     PROCEDURES
    MAIN     SEGMENT CODE
    CSEG     AT 0
    USING    0
    JMP  start
t0ISR:
    ORG      0Bh
    CLR      TF0
    DJNZ     R1, skip
    MOV      R1, #50
    DJNZ     R0, skip
    MOV      R0, #10
    CPL      P1.0
skip:
    MOV      TH0, #0DBh
    MOV      TL0, #0F2h
    RETI
    RSEG     MAIN
start:
    MOV      P1, #0FEh
    MOV      R0, #10
    MOV      R1, #50
    MOV      TH0, #0DBh
    MOV      TL0, #0F2h
    MOV      TMOD, #1h

    SETB     ET0
    SETB     EA
    SETB     TCON.4
    JMP      $
    END
```

Проанализируем исходный текст программы. Основная программа начинается с метки `start`. Здесь выполняются необходимые действия по инициализации порта P1 и таймера 0. Обратите внимание на команды

```
SETB    ET0
SETB    EA
```





Они разрешают прерывание от таймера 0. Последней командой основной программы является

```
JMP $
```

которая выполняет бесконечный цикл ожидания.

Обработчик прерывания таймера 0 расположен по адресу 0x0B, что определяется директивой

```
ORG 0Bh
```

Первой командой обработчика является команда очистки флага прерывания:

```
CLR TF0
```

Далее выполняется проверка счетчика перезагрузок таймера, организованного с помощью регистров R0 и R1, и переход на соответствующую ветвь программы-обработчика. Перед выходом из обработчика следует восстановить значения старшего и младшего байтов таймера 0:

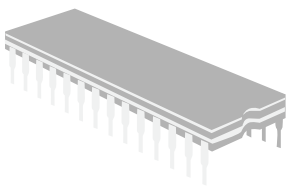
```
MOV      TH0, #0DBh
MOV      TL0, #0F2h
```

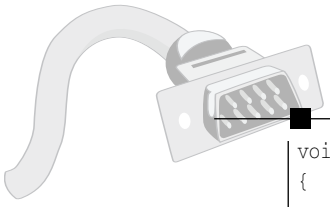
Наконец, завершающей командой программы обработки прерывания должна быть команда RETI.

Обратите внимание, что наш обработчик прерывания занимает явно больше 8 байт памяти, но, поскольку следующий за ним обработчик прерывания отсутствует, то можно использовать имеющийся резерв памяти. Если бы, например, по адресу 0x13 размещался обработчик внешнего прерывания 1, то обработчик таймера 0 нужно было бы размещать в свободной области памяти, а переход на него выполнять командой SJMP или LJMP.

Та же программа, реализованная на языке C, может быть представлена следующим программным кодом:

```
#include <stdio.h>
#include <REG52.H>

sbit LED_BLINK = P1^0;
unsigned int counter;

void T0Isr(void) interrupt 1 using 1 {
TF0 = 0;
counter—;
if (counter == 0)
{
    counter = 500;
    LED_BLINK = ~LED_BLINK;
}
TH0 = 0xDB;
TL0 = 0xF2;
}
```






```
void main(void)
{
    counter = 500;
    P1 = 0x0FE;
    TH0 = 0xDB;
    TL0 = 0xF2;
    TMOD |= 0x1;
    ET0 = 1;
    EA = 1;

    TR0 = 1;

    while (1);
}
```

В программах на Keil C51, впрочем, как и в большинстве других компиляторов C для 8051, программа-обработчик прерывания имеет специальное обозначение, в котором используется ключевое слово `interrupt`, после которого следует номер прерывания. В Keil C51 нумерация прерываний начинается с нуля, поэтому прерывание таймера 0 имеет индекс 1. В остальном программный код, я думаю, понятен читателю.

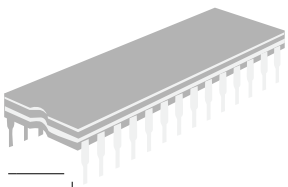
4.3. Режим автоперезагрузки

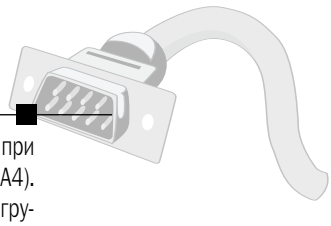
До сих пор мы рассматривали работу таймера в режиме 16-разрядного таймера. При работе в этом режиме обычно требуется каждый раз инициализировать таймер (если начальные значения не нулевые) после наступления переполнения. Облегчить себе жизнь можно, если установить для таймера режим автоперезагрузки. В этом режиме при переполнении таймера его регистры загружаются заранее определенными значениями, что исключает необходимость инициализации. Для таймеров 0 и 1 режим автоперезагрузки устанавливается следующим образом:

- в регистр THx помещается значение автоперезагрузки, при этом регистр TLx при переполнении перезагружается значением, указанным в THx. Значение THx при этом остается неизменным;
- в регистре TMOD указывается значение режима работы, равное 2.

Например, если в регистре TH0 указать значение 150, то регистр TL0 при перезагрузке каждый раз будет инкрементироваться, начиная со значения 150 и заканчивая 255. Таким образом, при установке режима автоперезагрузки для таймеров 0 и 1 значение THx остается неизменным, а сам таймер фактически работает в 8-битовом режиме.

Рассмотрим наш стандартный пример, в котором используется таймер 0, прерывание которого вызывает инверсию вывода P1.0 каждые 5 с. Посмотрим, как будет выглядеть программный код проекта, если таймер 0 использовать в режиме автоперезагрузки. Вначале определим, с какой частотой перезагрузки будет работать таймер 0. Предположим, таймер 0 должен работать с частотой 10 КГц. Если частота кристалла микроконтроллера равна 11,059 МГц при 12 тактах машинного цикла, то таймер 0 в режиме автоперезагрузки может работать на максимальной частоте 921583 / 256 = 3600 Гц (перегружается только 8-разрядный регистр TL0!). Для работы с частотой 10 КГц регистр TH0 должен содержать значение 256 – – (921583 / 10) = 256 – 92 = 164 или, в шестнадцатеричной нотации, 0xA4.





Таким образом, для получения требуемой частоты автоперезагрузки, равной 10 КГц, при частоте кристалла 11,059 МГц регистр TH0 таймера 0 должен содержать значение 164 (0xA4).

В этом случае для инверсии бита P1.0 каждые 5 с необходимо, чтобы счетчик перезагрузок декрементировался от начального значения, равного 50000 (период перезагрузки таймера 0 обратно пропорционален частоте и равен 0,0001 с, откуда и следует значение счетчика).

Программный код проекта на языке ассемблера представлен ниже:

```
    NAME     PROCEDURES
    MAIN     SEGMENT CODE
    CSEG     AT 0
    USING    0
    JMP      start
t0ISR:
    ORG      0Bh
    CLR      TF0
    DJNZ     R1, skip
    MOV      R1, #200
    DJNZ     R0, skip
    MOV      R0, #250
    CPL      P1.0
skip:
    RETI
    RSEG     MAIN
start:
    MOV      P1, #0FEh
    MOV      R0, #250
    MOV      R1, #200
    MOV      TH0, #0A4h
    MOV      TL0, #0A4h
    MOV      TMOD, #2h

    SETB     ET0
    SETB     EA
    SETB     TCON.4
    JMP      $
    END
```

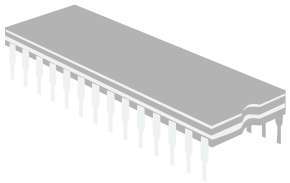
Как видно из листинга, инициализация таймера 0 выполняется в основной программе, которая начинается с метки `start`. Величина автоперезагрузки загружается в регистр `TH0`. Регистр `TL0` может, вообще говоря, содержать произвольное значение, но здесь в него загружается значение автоперезагрузки. Затем устанавливается режим работы таймера 0, равный 2.

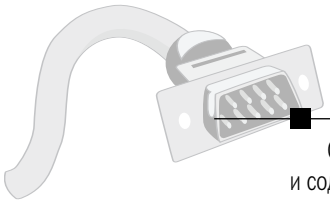
После установки флага разрешения прерывания от таймера 0 и глобального флага разрешения прерываний команда

```
SETB    TCON.4
```

запускает таймер 0, после чего программа уходит в бесконечный цикл по команде

```
JMP $
```





Обработчик прерывания таймера 0, находящийся по адресу 0x0B, особенностей не имеет и содержит уже знакомый нам из предыдущих примеров программный код. Обратите внимание на то, что после переполнения таймера больше не требуется инициализировать регистры TH0 и TL0 определенными значениями.

Далее представлен программный код, выполняющий те же действия, но написанный с использованием языка C:

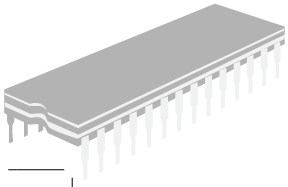
```
#include <stdio.h>
#include <REG52.H>

sbit LED_BLINK = P1^0;
unsigned int counter;

void T0Isr(void) interrupt 1 using 1 {
TF0 = 0;
counter--;
if (counter == 0)
{
counter = 50000;
LED_BLINK = ~LED_BLINK;
}
}

void main(void)
{
counter = 50000;
P1 = 0x0FE;
TH0 = 0x0A4;
TL0 = 0x0A4;
TMOD |= 0x2;
ET0 = 1;
EA = 1;

TR0 = 1;
while (1);
}
```

Следует сказать, что режим автоперезагрузки очень часто используется в случае установки таймера в качестве генератора тактовой частоты при последовательном обмене данными. Чаще всего в качестве генератора синхронизации при последовательном обмене данными используется таймер 1 или таймер 2.

4.4. Счетчики событий

Таймеры микроконтроллера 8051/8052 могут работать и как счетчики событий. Будучи установленным в этот режим, таймер больше не управляется машинными циклами микроконтроллера, а инкрементируется каждый раз при поступлении перепада сигнала 1–0 на один из входов P3.4 или P3.5 для таймеров 0 и 1 соответственно (рис. 4.5).

Этот режим работы можно использовать, например, при счете импульсов или при измерениях временных характеристик сигналов.



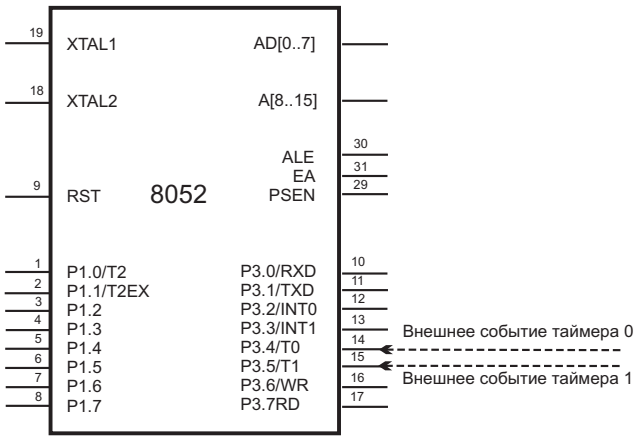


Рис. 4.5.
Схема подключения к источникам событий

Следующий пример программного кода демонстрирует принципы работы таймеров в этом режиме. Здесь таймер 0 считает перепады импульсов на выводе P3.4 и после прихода 4-го импульса генерирует прерывание с вектором 0xB, программа-обработчик которого инвертирует сигнал на выводе P1.0. Это не что иное, как алгоритм простейшего делителя частоты.

Вот ассемблерный код программы, которую легко скомпилировать в среде Keil uVision3:

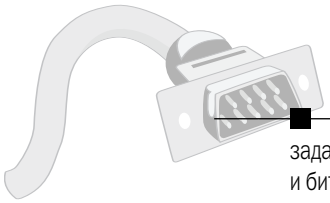
```
    NAME    PROCS
    MAIN    SEGMENT CODE
    CSEG    AT 0
    USING   0
    JMP     start
t0ISR:
    ORG     0Bh
    CLR     TF0
    CPL     P1.0
    RETI
    RSEG    MAIN
start:
    MOV     P1, #0FEh
    ORL     TMOD, #6h
    MOV     TH0, #0FDh
    MOV     TL0, #0FDh
    SETB    EA
    SETB    ET0
    SETB    TR0
    JMP     $
    END
```

Основная программа начинается с метки `start`. Вначале бит 0 порта устанавливается в режим вывода данных командой

```
MOV     P1, #0FEh
```

Команда

```
ORL     TMOD, #6h
```



задает режим работы таймера, который является логической комбинацией битов T0M0, T0M1 и бита C/T0. Комбинация битов T0M0 – T0M1 должна равняться 2 (режим автоперезагрузки), при этом в регистр TH0 мы помещаем шестнадцатеричное значение FD. Таким образом, перезагрузка таймера каждый раз будет выполняться после перепада 1–0 четвертого импульса на выводе P3.4. Для подсчета событий бит C/T0 должен быть установлен в 1.

Для работы прерывания таймера программа должна установить биты глобального разрешения прерываний EA и разрешения прерываний таймера 0 (ET0). Предпоследняя команда основной программы

```
SETB    TR0
```

запускает таймер 0, после чего программа входит в бесконечный цикл по команде

```
JMP $
```

Обработчик прерывания таймера 0 состоит всего из трех команд. Первая команда

```
CLR TF0
```

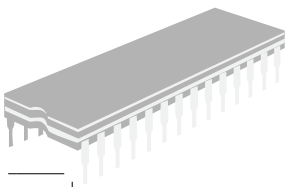
очищает флаг прерывания `TF0`, чтобы избежать повторных прерываний во время выполнения текущего обработчика. Следующая за ней команда

```
CPL P1.0
```

инвертирует бит `P1.0`, после чего программа-обработчик завершает работу командой `RETI`.

Далее представлен вариант программы, написанный на C:

```
#include <stdio.h>
#include <REG52.H>

sbit D1 = P1^0;

void T0Isr(void) interrupt 1 using 0 {
TF0 = 0;
D1 = ~D1;
}

void main(void)
{
P1 = 0x0FE;
TH0 = 0xFD;
TL0 = 0xFD;
TMOD |= 0x6;
EA = 1;
ET0 = 1;
TR0 = 1;

while(1);
}
```



Если на вход P3.4 подавать сигнал какой-либо частоты, то на выходе P1.0 сигнал будет появляться реже и определяться параметрами автоперезагрузки. Например, временная диаграмма для значения TH0, равного 6, будет выглядеть, как показано на рис. 4.6.

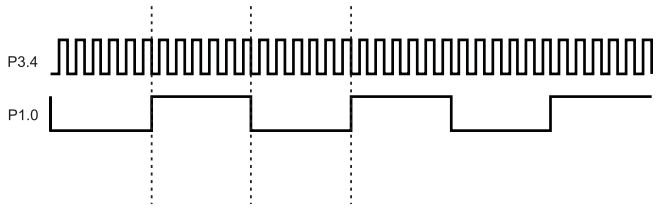


Рис. 4.6.
Схема работы делителя частоты

Подобный проект с необходимыми доработками может быть использован в качестве программного делителя частоты.

4.5. Таймер 2

Таймер 2 аппаратно представляет собой дополнительное устройство, которое включено в клоны микроконтроллера 8052. Таймер 2 имеет целый ряд особенностей и дополнительных возможностей по сравнению с таймерами 0 и 1 в классических микроконтроллерах 8051. Таймер 2 ассоциируется с группой специальных регистров:

- T2CON – регистр управления и контроля (0x0C8). Регистр является адресуемым побитово;
- RCAP2H – старший байт, содержимое которого используется при работе в режиме автоперезагрузки (0x0CB);
- RCAP2L – младший байт, содержимое которого используется при работе в режиме автоперезагрузки (0x0CA);
- TH1 – старший байт таймера 2 (0x0CD);
- TL1 – младший байт таймера 2 (0x0CC).

Таймер 2, как и рассмотренные нами ранее таймеры 0 и 1, может работать в качестве 16-разрядного таймера, в режиме автоперезагрузки, в режиме генератора синхронизации при последовательном обмене данными, а также имеет дополнительный режим, который называется режимом захвата (capture mode). Для управления таймером 2, как уже упоминалось, служит специальный регистр функций T2CON, назначение битов которого проиллюстрировано рис. 4.7.

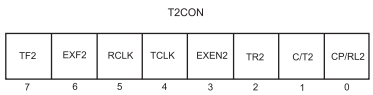
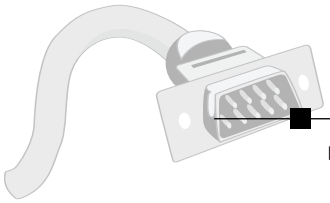


Рис. 4.7.
Регистр T2CON

Назначение битов регистра T2CON следующее:

- TF2 – установка этого бита свидетельствует о переполнении таймера 2. Если разрешены прерывания от таймера 2, то установка этого бита вызывает прерывание, программа-обработчик которого должна располагаться по адресу 0x2B;
- EXF2 – устанавливается при возникновении переполнения или если сигнал на выводе T2EX (P1.1) переходит из высокого в низкий уровень (перепад 1–0). Перепад сигнала фиксируется только при установленном бите EXEN2;



- RCLK – если данный бит установлен, то таймер 2 используется в качестве генератора синхронизации последовательного порта при приеме данных. Если бит равен 0, то для синхронизации используется таймер 1;
- TCLK – если данный бит установлен, то таймер 2 используется в качестве генератора синхронизации последовательного порта при передаче данных. Если бит равен 0, то для синхронизации используется таймер 1;
- EXEN2 – при установленном бите перепад сигнала из высокого в низкий уровень на выводе T2EX (P1.1) инициирует режим захвата или вызывает переполнение таймера 2;
- TR2 – установка этого бита в единичное состояние вызывает запуск таймера 2. Установка в 0 останавливает работу таймера 2;
- C/T2 – если данный бит сброшен, то таймер 2 функционирует в режиме интервального таймера. Если данный бит установлен, то таймер 2 инкрементируется каждый раз при перепаде 1–0 на выводе T2 (P1.0) микроконтроллера 8052;
- CP/RL2 – при сброшенном бите переполнение таймера 2 возникает при работе в режиме автоперезагрузки или при перепаде 1–0 на выводе T2EX (бит EXEN2 должен быть установлен). Если данный бит установлен, то таймер 2 работает в режиме захвата при возникновении перепада 1–0 на выводе T2EX (бит EXEN2 должен быть установлен).

Рассмотрим режимы работы таймера 2 более подробно.

4.5.1. Режим автоперезагрузки таймера 2

При работе в этом режиме таймер 2 при переполнении перезагружается значениями, находящимися в регистрах RCAP2H (старшая часть) и RCAP2L (младшая часть). Этот режим напоминает аналогичные режимы для таймеров 0 и 1 с той разницей, что там используется максимум 8 разрядов, что ограничивает диапазон 256 значениями, в то время как таймер 2 использует 16-разрядные значения.

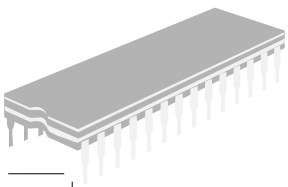
Работу таймера 2 в режиме автоперезагрузки продемонстрируем на уже знакомом нам примере проекта, установив время переключения светодиода, присоединенного к биту 0 порта P1, равным 4 с.

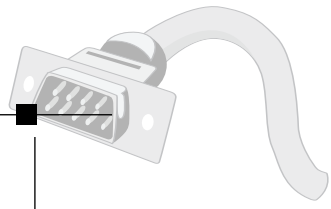
Напомню, что в нашем проекте мы используем схему устройства, показанную на рис. 4.4. Просчитаем значения, которые необходимо поместить в регистры RCAP2H и RCAP2L. Предположим, что время перезагрузки таймера 2 равно 0,05 с. В этом случае 16-разрядное значение, которое нужно поместить в регистры RCAP2H/RCAP2L, равно 18725 или, в шестнадцатеричной форме, 0x4925. Хочу напомнить, что все расчеты мы проводим в предположении, что тактовая частота генератора микроконтроллера равна 11,059 МГц. Для другой частоты значения будут, естественно, другими.

Таким образом, в регистр RCAP2H нужно поместить шестнадцатеричное значение 0x49, а в регистр RCAP2L – шестнадцатеричное значение 0x25. Для получения интервала переключения, равного 4 с, можем использовать регистр R0, загрузив в него значение 4 / 0,05 = 80 и проверяя его каждый раз при переполнении таймера 2.

Саму операцию переключения удобно выполнять в программе-обработчике прерывания таймера 2, которая должна находиться по адресу 0x2B.

Создадим проект в среде Keil uVision3 и добавим в него файл с расширением .asm, содержимое которого представляет исходный текст нашей программы:





```
    NAME PROCEDURES
    T2CON    EQU 0C8h
    RCAP2H   EQU 0CBh
    RCAP2L   EQU 0CAh
    MAIN     SEGMENT CODE
    CSEG     AT 0
    USING    0
    JMP      start
T2Isr:
    ORG      2Bh
    CLR      T2CON.7
    DJNZ     R0, skip
    MOV      R0, #80
    CPL      P1.0
skip:
    RETI
    RSEG     MAIN
start:
    MOV      P1, #0FEh
    MOV      R0, #80
    MOV      RCAP2H, #49h
    MOV      RCAP2L, #25h
    CLR      T2CON.0
    CLR      T2CON.1
    SETB     IE.5
    SETB     EA
    SETB     T2CON.2
    JMP      $
    END
```

Прежде всего посмотрим, как инициализируется таймер 2. Он должен работать в режиме автоперезагрузки, для чего следует выполнить команды:

```
CLR      T2CON.0
CLR      T2CON.1
```

Кроме того, нужно проинициализировать регистры RCAP2H/RCAP2L, что выполняет следующий фрагмент кода:

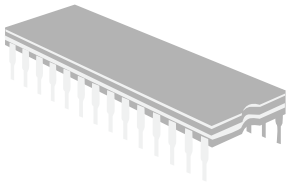
```
MOV      RCAP2H, #49h
MOV      RCAP2L, #25h
```

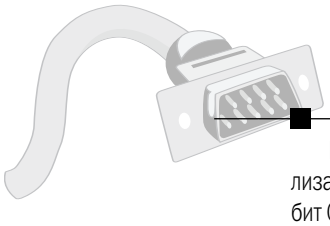
Далее нужно разрешить работу прерывания таймера с помощью команд:

```
SETB     IE.5
SETB     EA
```

Наконец, запускаем таймер 2 командой

```
SETB     T2CON.2
```





В основной программе, которая начинается с метки `start`, также выполняется инициализация регистра `R0`, в который помещается значение `80`. Кроме того, не забудем установить бит 0 порта P1 в режим записи.

Что же касается обработчика прерываний таймера 2, то его программный код большей частью нам уже знаком. В первой команде, которая выполняется при переходе к обработчику прерывания, обнуляется флаг прерывания таймера 2. Остальной код обработчика, думаю, понятен, и мы не будем на нем останавливаться.

Ниже приводится исходный текст программы, реализующей этот же алгоритм, но написанной на C:

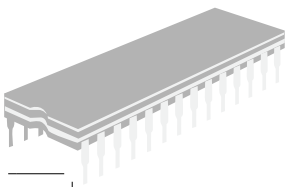
```
#include <stdio.h>
#include <REG52.H>

sbit LED_BLINK = P1^0;
unsigned int cnt;

void T2Isr(void) interrupt 5 using 1 {
TF2 = 0;
cnt–;
if (cnt == 0)
{
cnt = 80;
LED_BLINK = ~LED_BLINK;
}
}

void main(void)
{
cnt = 80;
RCAP2H = 0x49;
RCAP2L = 0x25;
T2CON &= 0x0FC;

ET2 = 1;
EA = 1;
T2CON |= 0x4;

while (1);
}
```

Здесь в основной программе выполняются необходимые действия по инициализации таймера 2 и переменных.

Операторы

```
RCAP2H = 0x49;
RCAP2L = 0x25;
```

устанавливают интервал перезагрузки таймера 2 равным 0,05 с при тактовой частоте 11,059 МГц.



Оператор

```
T2CON &= 0x0FC;
```

очищает 0-й и 1-й биты регистра T2CON, устанавливая тем самым режим автоперезагрузки.

Операторы

```
ET2 = 1;
EA = 1;
```

разрешают работу прерывания таймера 2. Наконец, оператор

```
T2CON |= 0x4;
```

устанавливает бит 2 регистра T2CON в единичное состояние, что вызывает запуск таймера 2.

Программный код обработчика прерывания таймера 2 особенностей не имеет. Обратите внимание, что прерывание таймера 2 указывается в заголовке обработчика с индексом 5.

Режим автоперезагрузки таймера 2 часто применяется при использовании его в качестве генератора синхронизации при обмене через последовательный порт.

4.5.2. Режим захвата таймера 2

Это новый режим работы, который может использоваться при функционировании таймера 2. Суть его в том, что при установке флага EXEN2 таймер может реагировать на перепад 1–0 на выводе T2EX (P1.1). В момент фиксации перепада текущие значения регистров TH2 и TL2 запоминаются в регистрах RCAP2H и RCAP2L соответственно. В этот же момент устанавливается флаг EXF2, вызывающий прерывание таймера 2. Следует иметь в виду, что даже при установленном режиме захвата установка флага TF2, сигнализирующего о переполнении таймера, также вызовет прерывание. В таких случаях программный код обработчика прерывания таймера 2 должен учитывать обе возможные причины возникновения прерывания и обрабатывать их соответствующим образом. Режим очень эффективен при измерениях временных параметров входящих сигналов. При этом флаг EXF2, так же как и TF2, должен сбрасываться программно. Работа в режиме захвата обладает определенной спецификой, поэтому для лучшего представления логики функционирования таймера 2 в этом режиме разработаем несколько простых проектов, в которых будут продемонстрированы различные аспекты работы.

Предположим, что на вывод T2EX подается перепад напряжения 1–0. Присоединим к выводу P1.0 один светодиод, который будет переключаться в обратное состояние каждый раз при перепаде сигнала 1–0 на T2EX. К выводу P1.7 присоединим светодиод, который будет переключаться всякий раз при возникновении переполнения таймера 2. Принципиальная схема устройства представлена на рис. 4.8.

Как видно из схемы, к выводам P1.0 и P1.7 микроконтроллера подсоединены светодиоды D1 и D2 соответственно. На вывод P1.1 (T2EX) в произвольные моменты времени при нажатии на кнопку подается перепад напряжения из высокого в низкий уровень, имитируя тем самым входной сигнал. Обработчик прерывания таймера 2 должен отслеживать два события:

- переполнение таймера;
- перепад 1–0 на выводе T2EX.

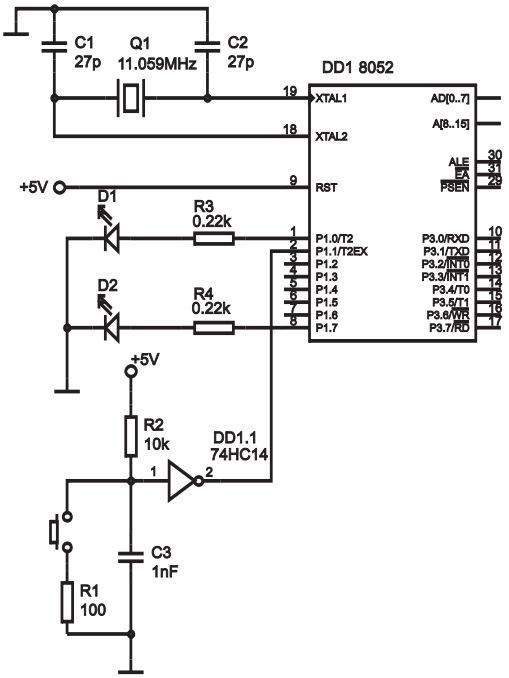
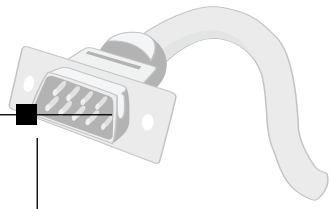


Рис. 4.8.
Схема демонстрации режима захвата

Если прерывание вызвано переполнением таймера 2, то переключается светодиод D2, если же прерывание инициировано посредством флага EXF2 (перепад 1–0 на T2EX), то переключается светодиод D1.

Вначале посмотрим, как такой алгоритм реализуется ассемблерным кодом. Исходный текст программы на ассемблере, которую нужно включить в проект Keil uVision3, показан ниже:

```
    NAME     MAIN
    T2CON    EQU 0C8h
    TH2      EQU 0CDh
    TL2      EQU 0CCh
    PROG     SEGMENT CODE
    CSEG     AT 0
    USING    0
    JMP      start
T2Isr:
    ORG      2Bh
    JBC      T2CON.6, captured
    CLR      T2CON.7
    DJNZ     R0, quit
    MOV      R0, #100
    CPL      P1.7
    JMP      quit
captured:
    CPL      P1.0
quit:
```



```
    RETI
    RSEG    PROG
start:
    MOV     P1, #7Eh
    MOV     R0, #100
    MOV     TH2, #0h
    MOV     TL2, #0h
    SETB    T2CON.0
    CLR     T2CON.1
    SETB    T2CON.3
    SETB    IE.5
    SETB    EA
    SETB    T2CON.2
    JMP     $
    END
```

Вначале проанализируем основную секцию программы, начиная с метки `start`. Биты P1.0 и P1.7 должны управлять светодиодами, поэтому их защелки должны быть настроены как выходы, что и делает команда

```
MOV     P1, #7Eh
```

Таймер 2 в нашем эксперименте должен работать как 16-разрядный таймер, поэтому выберем для удобства нулевые значения для регистров `TH2` и `TL2` следующими командами:

```
MOV     TH2, #0h
MOV     TL2, #0h
```

В этом случае перезагрузка таймера 2 будет случаться каждые 0,071 с при тактовой частоте 11,059 МГц. Выберем интервал переключения светодиода D2, равный 7 с, а в качестве счетчика перезагрузок выберем регистр R0. В этом случае для переключения светодиода с указанным интервалом регистр R0 должен содержать значение 7 / 0,071 ≈ 100. Примем значение R0 равным 100. Это же значение необходимо будет заносить в регистр R0 и в обработчике прерывания таймера 2, после того как будет достигнуто значение 0.

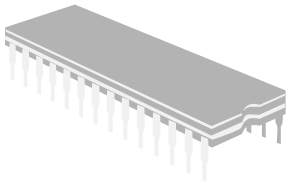
Команда

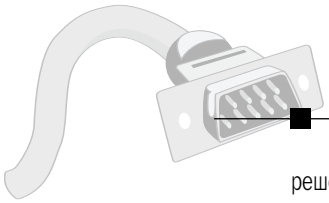
```
SETB    T2CON.0
```

устанавливает таймер в режим захвата (capture mode), а команда

```
CLR     T2CON.1
```

устанавливает режим работы таймера как 16-разрядного таймера. Здесь нужно понять важный момент: фактически таймер 2 будет работать одновременно в двух режимах. При этом он будет инкрементироваться в каждом машинном цикле независимо от того, есть ли перепад 1–0 на выводе T2EX или нет. Обработчик прерывания таймера 2, используемый в нашей программе, должен программными средствами отследить источник прерывания и выполнить соответствующий код. Можно сказать и так, что программный код обработчика прерывания таймера 2 состоит из двух относительно независимых друг от друга частей.





Как обычно, для работы прерывания необходимо установить соответствующие биты разрешения, что и выполняют следующие две команды:

```
SETB    IE.5
SETB    EA
```

Предпоследняя команда основной программы запускает таймер 2:

```
SETB    T2CON.2
```

Затем программа входит в бесконечный цикл по команде

```
JMP $
```

Перейдем теперь к обработчику прерывания таймера 2, который должен размещаться в младших адресах памяти программ начиная с адреса 0x2B.

Первая команда обработчика

```
JBC     T2CON.6, captured
```

анализирует, вызвано ли прерывание установкой бита EXF2 (`T2CON.6`), что свидетельствует о перепаде 1–0 на выводе T2EX микроконтроллера. Если это так, то выполняется очистка бита, далее выполняется переход на метку `captured`, после чего выполняется команда

```
CPL P1.0
```

инвертирующая выход `P1.0`. Затем происходит выход из обработчика прерывания по команде `RETI`, и управление передается следующей команде прерванной программы.

Если оказывается, что бит EXF2 не установлен, то понятно, что источником прерывания является бит TF2, который сразу же необходимо сбросить командой

```
CLR     T2CON.7
```

Затем проверяется, истек ли 7-секундный интервал, с помощью команды

```
DJNZ    R0, quit
```

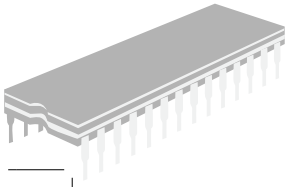
Если интервал не истек, то происходит выход из обработчика прерывания. Если же содержимое регистра `R0` достигло нуля, то регистр повторно перезагружается значением `100` для следующего цикла, а выход `P1.7` инвертируется:

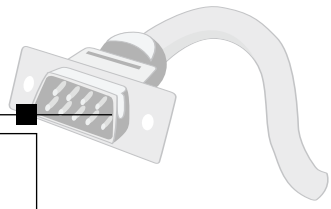
```
DJNZ    R0, quit
MOV     R0, #100
CPL     P1.7
```

Затем происходит выход из обработчика прерывания.

Как видим, таймер 2 предоставляет весьма гибкие возможности для работы с периферийным оборудованием.

Ниже представлен вариант программы, выполняющей тот же алгоритм, но написанной на C:



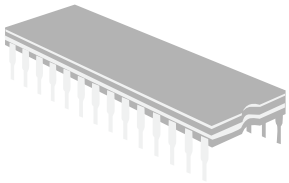


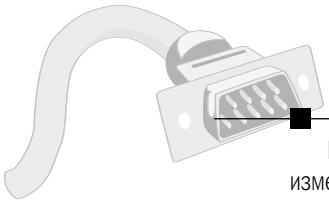
```
#include <stdio.h>
#include <REG52.H>

sbit D1 = P1^0;
sbit D2 = P1^7;

unsigned int cnt;

void T2Isr(void) interrupt 5 using 1 {
if (EXF2 == 1)
{
EXF2 = 0;
D1 = ~D1;
return;
}
if (TF2 == 1)
{
TF2 = 0;
cnt—;
if (cnt == 0)
{
cnt = 100;
D2 = ~D2;
}
return;
}
}

void main(void)
{
cnt = 100;
ET2 = 1;
EA = 1;

TH2 = 0;
TL2 = 0;
T2CON |= 0x9;
T2CON |= 0x4;

while(1);
}
```

Исходный текст программы, думаю, понятен, и анализировать его мы не станем.

В этом примере мы рассмотрели, как таймер 2 работает одновременно в режиме 16-разрядного таймера и в режиме захвата.

Вместо режима 16-разрядного таймера можно настроить таймер 2 на автоперезагрузку. Для этого модифицируем оба исходных текста программ (на ассемблере и С) из предыдущего примера так, чтобы таймер 2 мог работать в режимах автоперезагрузки и захвата. Зададим интервал автоперезагрузки таймера равным 0,03 с, установив интервал переключения светодиода D2 равным 6 с.





Вот как будет выглядеть исходный текст программы на ассемблере с учетом внесенных изменений:

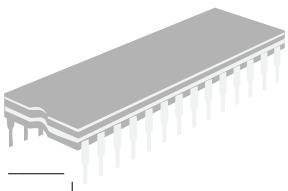
```
    NAME    MAIN
    T2CON   EQU 0C8h
    RCAP2H  EQU 0CBh
    RCAP2L  EQU 0CAh
    PROG    SEGMENT CODE
    CSEG    AT 0
    USING   0
    JMP     start
T2Isr:
    ORG     2Bh
    JBC     T2CON.6, captured
    CLR     T2CON.7
    DJNZ    R0, quit
    MOV     R0, #200
    CPL     P1.7
    JMP     quit
captured:
    CPL     P1.0
quit:
    RETI
    RSEG    PROG
start:
    MOV     P1, #7Eh
    MOV     R0, #200
    MOV     RCAP2H, #92h
    MOV     RCAP2L, #49h
    CLR     T2CON.0
    CLR     T2CON.1
    SETB    T2CON.3
    SETB    IE.5
    SETB    EA
    SETB    T2CON.2
    JMP     $
    END
```

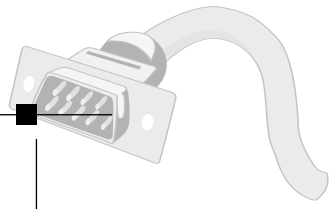
Основное отличие этой программы от предыдущей в том, что здесь используются регистры RCAP2H и RCAP2L для задания значений, которые будут присвоены регистрам таймера 2 при переполнении таймера. В данном случае 16-разрядное значение 0x9249 (или десятичное 37449) определяет период перезагрузки таймера, равный 0,03 с. Кроме того, бит 0 регистра T2CON должен быть сброшен – в этом случае таймер работает в комбинированном режиме (автоперезагрузка + захват).

Исходный текст обработчика прерывания, за исключением конкретной величины интервала переключения, практически не отличается от предыдущего примера.

Версия программы на языке C представлена ниже:

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

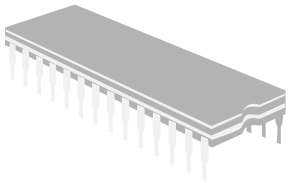




```
#include <REG52.H>

sbit D1 = P1^0;
sbit D2 = P1^7;

unsigned int cnt;

void T2Isr(void) interrupt 5 using 1 {
if (EXF2 == 1)
{
EXF2 = 0;
D1 = ~D1;
return;
}
if (TF2 == 1)
{
TF2 = 0;
cnt—;
if (cnt == 0)
{
cnt = 400;
D2 = ~D2;
}
return;
}
}

void main(void)
{
cnt = 400;
ET2 = 1;
EA = 1;

RCAP2H = 0x92;
RCAP2L = 0x49;
TL2 = 0;
T2CON |= 0x8;
T2CON |= 0x4;

while(1);
}
```

Перейдем к рассмотрению еще одного режима, в котором может работать таймер 2, а именно режима счетчика событий. Ранее мы анализировали подобный режим для таймеров 0 и 1. Для таймера 2 есть некоторые отличия, связанные с логической структурой этого устройства. Таймер 2 может работать в качестве счетчика событий, представляющих собой перепад 1–0 на выводе T2 (P1.0) микроконтроллера. Для работы в этом режиме бит C/T2 (T2CON.1) регистра управления T2CON должен быть установлен.

Далее приводится пример простой программы на ассемблере, выполняющей инверсию бита P1.7 после появления 5 перепадов 1–0 на выводе T2:



```
    NAME    PROCS
    T2CON   EQU 0C8h
    RCAP2H  EQU 0CBh
    RCAP2L  EQU 0CAh
    TH2     EQU 0CDh
    TL2     EQU 0CCh
    MAIN    SEGMENT CODE
    CSEG    AT 0
    USING   0
    JMP     start
t2ISR:
    ORG     2Bh
    CLR     T2CON.7
    CPL     P1.7
    MOV     TH2, #0FFh
    MOV     TL2, #0FBh
    RETI
    RSEG    MAIN
start:
    MOV     P1, #7Fh
    CLR     T2CON.0
    SETB    T2CON.1
    MOV     TH2, #0FFh
    MOV     TL2, #0FBh
    SETB    EA
    SETB    IE.5
    SETB    T2CON.2
    JMP     $
    END
```

Проанализируем программный код, который включает исходный текст основной программы и обработчик прерывания от таймера 2, располагающийся по адресу 0x2B. Вначале об основной программе. Первая команда устанавливает бит 7 порта P1 в качестве выхода. Затем команды

```
CLR     T2CON.0
SETB    T2CON.1
```

устанавливают режим работы 16-разрядного таймера с ожиданием внешних событий на выводе T2 (P1.0). При установленном бите C/T2 (T2CON.1) таймер 2 не будет работать в режиме автоперезагрузки, поэтому мы запишем соответствующие значения в регистры `TH2` и `TL2`, которые нужно будет восстанавливать всякий раз при переполнении таймера 2 в обработчике прерывания:

```
MOV     TH2, #0FFh
MOV     TL2, #0FBh
```

Следующие три команды разрешают прерывание от таймера 2 и запускают таймер 2.

Программа-обработчик прерывания таймера 2 вначале, как обычно, сбрасывает флаг прерывания TF2, после чего инвертирует бит `P1.7`:

```
CLR     T2CON.7
CPL     P1.7
```

Поскольку таймер 2 работает не в режиме автоперезагрузки, то необходимо переинициализировать регистры TH2 и TL2 требуемыми значениями, после чего программа-обработчик заканчивает работу командой RETI.

Как видим, таймеры микроконтроллеров семейства 8051/8052 предоставляют очень широкие возможности по синхронизации временных параметров систем управления. Перейдем к рассмотрению вариантов полезного практического применения таймеров.

4.6. Аппаратно-программные решения с использованием таймеров

В этом разделе мы рассмотрим некоторые примеры полезных разработок, в которых используются таймеры. Начнем с измерения частоты. Существует множество методов измерения частоты сигнала. Мы рассмотрим два из них и разработаем программные проекты их реализации в системах на базе микроконтроллеров 8051/8052.

4.6.1. Измерение частоты

Один из методов измерения частоты базируется на том, что берется один период исследуемого сигнала и подсчитывается количество импульсов другого сигнала, которые проходят за этот период. При этом период следования таких импульсов должен быть заранее известен и быть меньше периода измеряемого сигнала как минимум на два порядка. Сказанное иллюстрирует рис. 4.9.

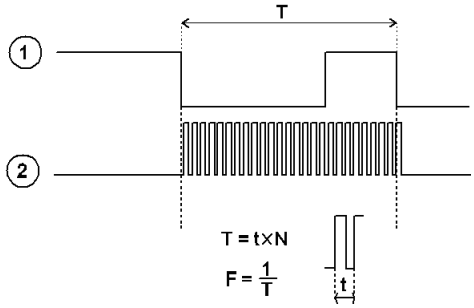


Рис. 4.9. *Измерение частоты методом подсчета количества импульсов*

Частота сигнала 1, измеренная таким способом, легко может быть вычислена по формуле:

$F = 1 / (t \times N)$, где

F – частота измеряемого сигнала;
t – период эталонного сигнала;
N – количество импульсов, посчитанное за один период измеряемого интервала T.

Попробуем реализовать изложенную выше методику измерения частоты сигнала, используя аппаратно-программные средства, предоставляемые 8051-совместимыми системами.

Для определения частоты сигнала требуется генератор импульсов с известным периодом их следования. Эту функцию легко может обеспечить, например, таймер 2 микроконтроллера

8052. Кроме того, нам нужно каким-то образом фиксировать начальный и конечный моменты измерения или, иными словами, начало и конец измерительного интервала T (см. рис. 4.9). Опять-таки, с этой функцией может справиться таймер 2, если его настроить на режим захвата.

Таким образом, мы решили использовать таймер 2 микроконтроллера 8052 для измерения частоты входного сигнала. Измеряемый сигнал будет подаваться на вход P1.1/T2EX микроконтроллера.

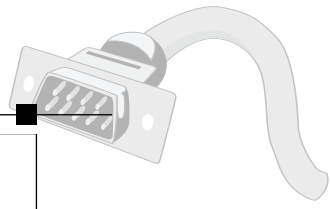
Хочу сделать важное замечание: во всех последующих экспериментах, касающихся обработки входных сигналов, мы предполагаем, что поступающие на определенные выводы микроконтроллера сигналы уже сформированы, т.е. имеют соответствующую амплитуду, например, 5 В для TTL-логики, а их фронты и спады имеют достаточную крутизну для срабатывания цифровой логики микроконтроллера.

На практике сигналы, особенно приходящие от удаленных источников, предварительно нормализуют, приводя их характеристики в соответствие с TTL-логикой. Обычно для этих целей используются буферные формирователи и/или триггеры Шмита.

Разработаем программу, измеряющую частоту приходящего на вывод T2 сигнала и выводящую результат измерения через последовательный порт на терминальное устройство либо персональный компьютер. Для этого создадим проект в Keil uVision3, содержащий два файла. Один из них будет содержать исходный текст основной программы, написанной на C. Второй файл будет содержать вспомогательные процедуры, написанные на ассемблере.

Основной файл содержит следующий исходный текст:

```
#include <stdio.h>
#include <REG52.H>

unsigned int bDone;

unsigned int startTime;
unsigned int stopTime;
unsigned int isrCounter;

extern int catch_timer2(void);
extern void start_timer2(void);

void Timer2ISR(void) interrupt 5 using 1

if (TF2 == 1)
{
TF2 = 0;
}
if (EXF2 == 1)
{
EXF2 = 0;
if (—isrCounter != 0)
    startTime = catch_timer2();
else
{
    TR2 = 0;
    stopTime = catch_timer2();
    bDone = 1;
```



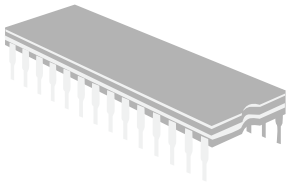
```
}
}
}

void main(void)
{
unsigned int cnt;
float timerFreq;

float FreqOfClock;

bDone = 0;
isrCounter = 2;

SCON = 0x50;
TH1 = 0xFD;
TMOD |= 0x20;
TR1 = 1;
TI = 1;

start_timer2();

while (bDone == 0);

EXEN2 = 0;
ET2 = 0;

cnt = stopTime - startTime;
timerFreq = 24000.0 / 12.0;
FreqOfClock = timerFreq * 1000 / cnt * 1.001;

printf("Captured difference: %d\n", cnt);
printf("Frequency measured = %8.1f Hz\n", FreqOfClock);
while (1);
}
```

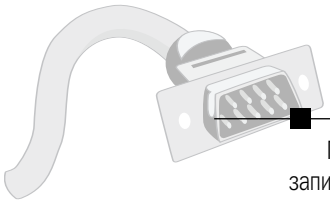
Проанализируем исходный текст основной программы. Группа команд

```
SCON = 0x50;
TH1 = 0xFD;
TMOD |= 0x20;
TR1 = 1;
TI = 1;
```

выполняет инициализацию последовательного порта микроконтроллера. В качестве генератора синхронизации мы будем использовать таймер 1, в старший байт которого заносится шестнадцатеричное значение 0xFD, которое определяет скорость обмена 9600 бод.

Функция `start_timer2` инициализирует таймер 2 для работы в режиме захвата с автоперезагрузкой и, кроме того, разрешает работу прерывания от таймера 2. Вычисление частоты поступающего на вход P1.1/T2EX сигнала выполняется в обработчике прерывания `Timer2ISR`, который вызывается два раза при двух последовательных перепадах 1–0 сигнала.





При первом вызове прерывания 16-разрядное содержимое таймера 2 автоматически записывается в регистры RCAP2H и RCAP2L и далее запоминается в переменной `startTime`, а при втором – в переменной `stopTime`. После двух вызовов прерывания таймер 2 останавливается, и переменная `bDone` устанавливается в `1`, позволяя операторам основной программы выполняться дальше.

Вычисленное значение частоты сигнала помещается в переменную `FreqOfClock` и выводится через последовательный порт в терминал.

В программе присутствует еще одна внешняя функция `catch_timer2`, основное назначение которой – поместить текущее содержимое таймера 2 (регистры `RCAP2H` и `RCAP2L`) в переменные `startTime` и `stopTime`.

Исходный текст процедур `start_timer2` и `catch_timer2` показан ниже:

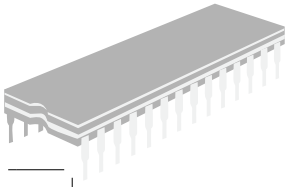
```
    NAME     PROGRAM
    PUBLIC   catch_timer2, start_timer2
    T2CON    EQU 0C8h
    RCAP2H   EQU 0CBh
    RCAP2L   EQU 0CAh
    TH2      EQU 0CDh
    TL2      EQU 0CCh
    PROGRAM  SEGMENT CODE
    RSEG     PROGRAM
catch_timer2:
    MOV      R6, RCAP2H
    MOV      R7, RCAP2L
    RET
start_timer2:
    SETB     T2CON.0
    CLR      T2CON.1
    SETB     T2CON.3
    MOV      RCAP2H, #0h
    MOV      RCAP2L, #0h
    MOV      TH2, #0h
    MOV      TL2, #0h
    SETB     IE.5
    SETB     EA
    SETB     T2CON.2
    RET
    END
```

Функция `start_timer2` устанавливает таймер 2 в режим автоперезагрузки и захвата с помощью команд

```
SETB    T2CON.0
CLR     T2CON.1
SETB    T2CON.3
```

последняя из которых устанавливает флаг разрешения EXEN2, который разрешает захват сигнала на порту P1.1/T2EX.

Содержимое регистров `RCAP2H` и `RCAP2L` установлено равным 0 из соображений удобства. Остальные команды разрешают прерывания таймера 2 и запускают сам таймер.



Измерение частоты очень широко используется в промышленности и в лабораторных исследованиях при анализе сигналов удаленных датчиков, передаваемых как последовательность импульсов; для измерения величин аналоговых сигналов, поступающих от удаленных объектов через преобразователи «напряжение – частота» (V-to-F conversion). На измерении частоты базируется методика измерения значений активных компонентов электрических цепей. В основном для этих целей используются генераторные цепи, в которых времязадающими элементами являются емкости и сопротивления (RC-генераторы, для низких частот) или индуктивности и емкости (LC-генераторы, для высоких частот).

Проиллюстрируем сказанное на примере. Если у нас имеется, например, схема генератора, управляемого напряжением (VCO, Voltage Controlled Oscillator), показанная на рис. 4.10, то с ее помощью можно легко выполнить как измерение величины неизвестного сопротивления R_t при известном значении емкости C_t, так и наоборот, по известному значению R_t можно найти неизвестную емкость C_t при фиксированных значениях входного напряжения V_{in}.

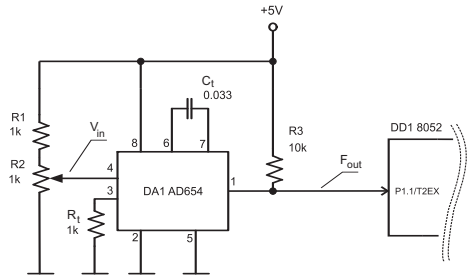


Рис. 4.10.
Возможная схема измерения значений компонентов цепи

$$F_{out} = \frac{V_{in}}{(10V) \times R_t \times C_t}$$

В общем случае для подобных измерений чаще всего используются генераторы, управляемые напряжением, подобные тому, схема которого изображена на рис. 4.10. Для такой и подобных ей схем, зафиксировав два из трех параметров (V_{in}, R_t, C_t), по измеренному значению частоты F_{out} схемы легко можно определить либо входное напряжение (для преобразователя V-to-F), либо сопротивление или емкость.

Важное замечание: сама электрическая схема подобного преобразователя должна обеспечивать высокую точность преобразования за счет применения специализированных чипов, как, например, AD654 фирмы Analog Devices или AD537 той же фирмы.

Для измерения частоты и связанных с ней параметров можно использовать и другой подход. Суть его состоит в том, что фиксируется интервал времени, равный, например, 1 с, и измеряется количество периодов сигнала, проходящих за это время. В этом случае это количество периодов и будет являться частотой сигнала. Сказанное иллюстрирует рис. 4.11.

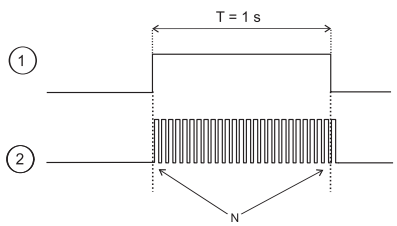
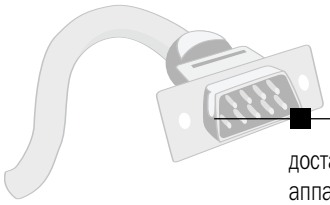


Рис. 4.11.
Измерение частоты за фиксированный интервал времени

Если, например, за 1 с насчитано N периодов частоты входного сигнала 2 (см. рис. 4.11), то частота входного сигнала принимается равной N Гц. Этот метод весьма прост и обеспечивает



достаточно высокую точность измерений, особенно если система на базе 8051 включает аппаратно независимый от 8051/8052 таймер реального времени, например, использующий микросхему DS1307 или подобную. Тем не менее 8051 позволяет довольно точно измерить частоту сигнала без привлечения дополнительных компонентов.

Для иллюстрации вышеизложенного метода разработаем демонстрационную программу в Keil uVision3, используя при этом таймер 0 и таймер 1.

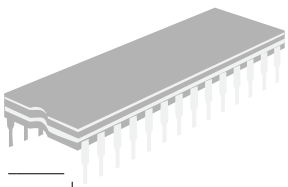
Таймер 0 в нашем проекте будет работать как счетчик событий, фиксирующий перепады напряжения 1–0 входного сигнала на входе P3.4/T0 микроконтроллера. Таймер 1 будет выполнять функцию интервального таймера, разрешающего прием сигнала таймером 0 в течение 1 с. По прошествии 1 с таймер 0 будет остановлен, а значения его старшего и младшего байтов, сохраненные в регистрах TH0 и TL0 соответственно, будут преобразованы в значение частоты сигнала и выведены через последовательный порт на терминал.

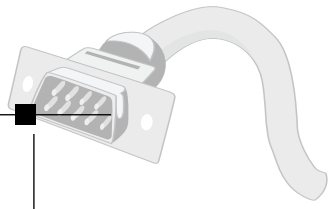
Программный код проекта состоит из основной программы, написанной на языке C, и нескольких процедур, написанных на ассемблере. Односекундный таймер программно реализован в виде обработчика прерывания, возникающего при переполнении таймера 0.

Хочу сделать важное замечание: все временные зависимости проекта рассчитаны для тактовой частоты микроконтроллера 11,059 МГц, поэтому при использовании других кристаллов нужно пересчитывать временные зависимости.

Исходный текст основной программы представлен ниже:

```
#include <stdio.h>
#include <REG52.H>

unsigned int cnt;
unsigned char bRunning;

unsigned int totalCounter0;

extern void t2init(void);
extern void t1init(void);
extern void t0init(void);
extern unsigned int readcounter0(void);

void T1Isr (void) interrupt 3 using 1 {
cnt–;
if (cnt != 0)
return;
TR0 = 0;
TR1 = 0;
totalCounter0 = readcounter0();
bRunning = 1;
}

void main(void)
{
SCON = 0x50;
TI = 1;
cnt = 14;

bRunning = 0;
```





```
t2init();
t0init();
t1init();

while (bRunning == 0);
printf("Timer 1 counted %d\n", totalCounter0);
while(1);
}
```

Вначале проанализируем смысл используемых переменных и функций, объявленных в основной программе.

Переменная `cnt` является счетчиком перезагрузок таймера 1. Она инициализируется значением `14` для получения временного интервала, равного 1 с. При тактовой частоте 11,059 МГц максимальное время перезагрузки таймера 1 равно 0,071 с, отсюда значение переменной `cnt` = 0,071 с × 14 ≈ 1 с.

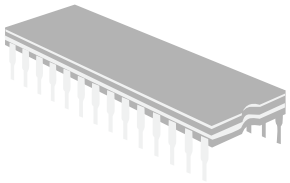
Переменная `totalCounter` по окончании интервала измерения будет содержать количество периодов, зафиксированное таймером 0.

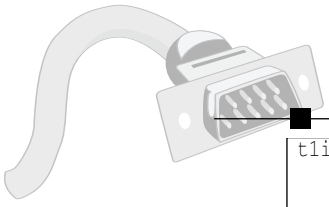
Переменная `bRunning` фиксирует момент завершения измерительного цикла, позволяя основной программе обработать полученные данные.

Функции `t0init`, `t1init` и `t2init` выполняют инициализацию таймера 0, 1 и 2 соответственно. Поскольку таймер 1, который обычно используется для синхронизации последовательного обмена, теперь задействован для других целей, то его роль берет на себя таймер 2. Функция `readcounter0` считывает 16-разрядное значение таймера 0 по прошествии 1-секундного интервала.

Все эти функции написаны на ассемблере и помещены в отдельный файл, исходный текст которого приводится ниже:

```
    NAME     PROCS
    PUBLIC   t0init, t1init, t2init, readcounter0
    T2CON    EQU 0C8h
    RCAP2H   EQU 0CBh
    RCAP2L   EQU 0CAh
    PROG     SEGMENT CODE
    RSEG     PROG
t2init:
    CLR      T2CON.0
    CLR      T2CON.1
    SETB     T2CON.4
    SETB     T2CON.5
    MOV      RCAP2H, #0FFH
    MOV      RCAP2L, #0DCh
    SETB     T2CON.2
    RET
;------------
t0init:
    MOV      TH0, #0h
    MOV      TL0, #0h
    ORL      TMOD, #5h
    RET
;------------
```






```
t1init:
    MOV     TH1, #0h
    MOV     TL1, #0h
    ORL     TMOD, #10h
    SETB    EA
    SETB    ET1
    SETB    TR1
    SETB    TR0
    RET
;————————————-
readcounter0:
    MOV     R6, TH0
    MOV     R7, TL0
    RET
;————————————-
    END
```

Функция `t0init` устанавливает таймер 0 в режим 16-разрядного таймера, который должен фиксировать перепады сигнала из высокого в низкий уровень на выводе P3.4/T0, для чего в регистр `TMOD` нужно записать значение 5. Это выполняет команда

```
ORL TMOD, #5h
```

Функция `t1init` устанавливает режим 16-разрядного таймера для таймера 1 с помощью команды

```
ORL TMOD, #10h
```

При тактовой частоте 11,059 МГц таймер 0 будет перезагружаться каждые 0,071 с. Перезагрузка таймера 0 должна обрабатываться посредством инициализации прерывания, для чего выполняются команды

```
SETB    EA
SETB    ET1
```

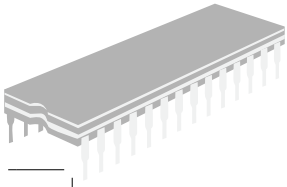
Поскольку начало счета импульсов на выводе P3.4/T0 должно быть синхронизировано с началом отсчета 1-секундного интервала, то необходимо выполнить синхронный запуск таймера 0 и таймера 1, что выполняется следующими двумя командами:

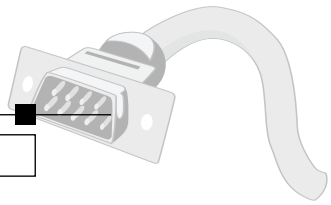
```
SETB    TR1
SETB    TR0
```

Функция `t2init` выполняет инициализацию таймера 2 для работы в качестве генератора синхронизации при последовательном обмене данными. Для этого таймер 2 устанавливается в режим автоперезагрузки по команде

```
CLR     T2CON.0
```

При этом следует обязательно указать, что таймер работает как интервальный таймер, выполнив команду





```
CLR     T2CON.1
```

Далее следует установить биты TCLK (T2CON.4) и RCLK (T2CON.5). Установка этих битов указывает микроконтроллеру, что вместо таймера 1 по умолчанию для генератора синхронизации последовательного обмена должен использоваться таймер 2. Кроме того, для работы таймера 2 в таком режиме следует установить соответствующие значения в регистрах RCAP2H и RCAP2L для скорости обмена 9600 бод при частоте 11,059 МГц:

```
MOV     RCAP2H, #0FFH
MOV     RCAP2L, #0DCh
```

Как обычно, после команд инициализации должна следовать команда запуска таймера:

```
SETB    T2CON.2
```

Функция readcounter0 передает основной программе 16-разрядное значение таймера 0, причем в соответствии с соглашениями, принятыми для компилятора Keil C, 16-разрядное значение передается в вызывающую программу следующим образом: в регистре R6 передается старший байт, а в регистре R7 – младший байт 16-битового значения.

Вернемся к основной программе. Собственно, программа после инициализации всех таймеров ожидает завершения операции измерения, о чем будет свидетельствовать установка переменной bRunning в 1. Затем выполняется вывод данных через последовательный порт ввода/вывода, и программа переходит в бесконечный цикл.

Обработчик прерывания таймера 1 отрабатывает 1-секундный интервал и по его завершении останавливает работу таймеров 0 и 1. Затем 16-разрядное содержимое таймера 0 записывается в переменную totalCounter0.

На этом обзор возможностей таймеров по измерению временных параметров сигналов мы закончим. Еще одним важным применением таймеров является их использование в качестве широтно-импульсных модуляторов.

4.6.2. Широтно-импульсная модуляция

Широтно-импульсная модуляция (ШИМ) или, по-английски, Pulse Width Modulation (PWM) используется главным образом для управления скоростью вращения электродвигателей, хотя может использоваться и для управления другими нагрузками, например, мощными нагревателями. ШИМ основана на следующем принципе: вместо подачи постоянного напряжения питания на двигатель (для микромощных двигателей это обычно 12 В) подается последовательность импульсов, имеющих определенную скважность. Изменяя скважность, можно менять среднее напряжение, подаваемое на двигатель, и, следовательно, скорость его вращения. Что такое скважность?

Чтобы лучше себе это представить, рассмотрим рис. 4.12.

Скважность импульсов представляет собой отношение длительности импульса к периоду его следования. На диаграмме 1 рис. 4.12 скважность, обозначенная литерой Q, равна 50%, поскольку отношение длительности импульса к периоду следования равно 1:2. Следовательно, половину времени двигатель включен, а половину выключен. Если, например, на двигатель подается напряжение питания 12 В, то при скважности 50% среднее значение напряжения на двигателе будет равно 6 В.

На диаграмме 2 скважность импульсов равна 75%. В практическом аспекте, если считать, что положительный импульс включает двигатель, а отрицательный его выключает, то $^3/_4$ времени

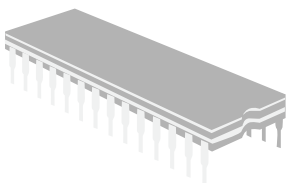
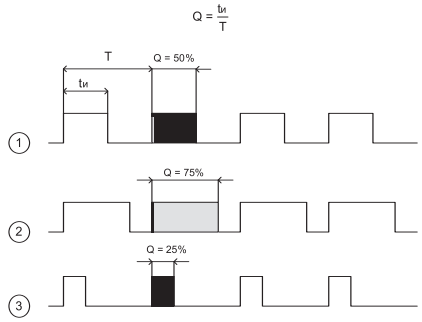


Рис. 4.12.
Иллюстрация различной скважности импульсов



двигатель остается включенным, следовательно, среднее напряжение на нем увеличивается по сравнению с 50-процентной скважностью и достигает 9 В (при питании 12 В). Поскольку среднее напряжение, приложенное к двигателю, возрастает, следовательно, скорость вращения тоже возрастает.

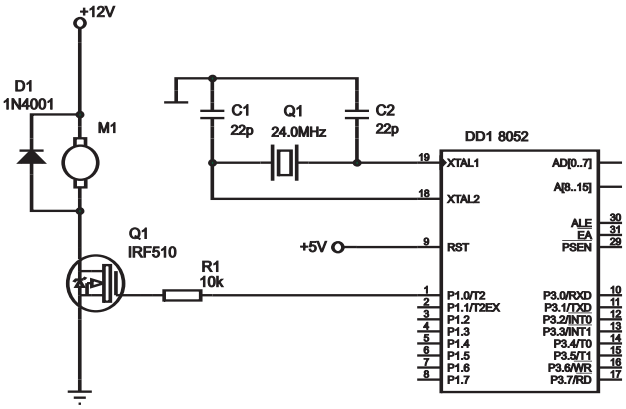
На диаграмме 3, наоборот, скважность импульсов уменьшается до 25%, в результате чего среднее значение напряжения за период уменьшается до 3 В, способствуя тем самым замедлению скорости вращения.

Микроконтроллеры 8051/8052 позволяют довольно легко реализовать ШИМ, причем это решение очевидно и основано на применении таймера. Многие современные 8051-совместимые устройства имеют встроенные функциональные узлы для реализации ШИМ. Тем не менее мы рассмотрим на практическом примере построение ШИМ, используя таймер 2 микроконтроллера.

Вот простая схема, иллюстрирующая применение ШИМ для управления электродвигателем (рис. 4.13).

В этой схеме электродвигатель M1 управляется через мощный полевой транзистор (Power MOSFET) типа IRF510, на затвор которого подаются импульсы управления с вывода P1.0

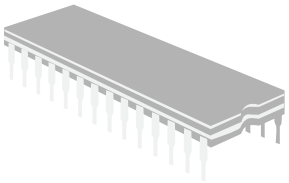
Рис. 4.13.
Реализация ШИМ в системе с 8051



микроконтроллера. Аппаратная часть этого проекта была разработана для тактовой частоты 24,0 МГц, поэтому для других значений частоты следует изменить параметры настройки таймера 2, который используется для генерации последовательности импульсов. Для желающих поэкспериментировать с этой схемой несколько советов: вместо транзистора Q1 типа IRF510 можно использовать любой другой N-канальный MOSFET с подходящими характеристиками, а в качестве электродвигателя M1 можно взять любой маломощный двигатель постоянного тока, например, электродвигатель от вентилятора неработающего блока питания ПК.

Программное обеспечение для данного проекта разработаем, как обычно, в среде Keil uVision3. Приведем исходный текст программы на языке С, позволяющей управлять частотой вращения электродвигателя:

```
#include <stdio.h>
#include <REG52.H>

sbit PWM_BIT = P1^0;
unsigned int cnt;
unsigned int tmpCnt;

void T2Isr(void) interrupt 5 using 1 {
TF2 = 0;
if (tmpCnt < 1000)
cnt = 1000;
else
cnt = tmpCnt;
PWM_BIT = 1;
while (cnt != 0)cnt–;
PWM_BIT = 0;
}

void main(void)
{
int argsread;

P1 = 0xFE;
SCON = 0x50;
PCON |= 0x80;
TH1 = 0xF3;
TMOD |= 0x20;
TR1 = 1;
TI = 1;

T2CON &= 0xFC;
RCAP2H = 0x0;
RCAP2L = 0x0;
ET2 = 1;
EA = 1;
T2CON |= 0x4;
while (1)
{
printf(“Enter the value in range 1000 - 10000 to set a motor speed:“);
argsread = scanf(“%d“, &tmpCnt);
if (argsread == EOF)
{
tmpCnt = 1000;
continue;
}
}
}
```



Для тестирования программы нужно подсоединить микроконтроллер через интерфейс RS-232 (серийный порт) к персональному компьютеру. Затем запустить какую-либо программу терминала и, установив для нее в качестве скорости обмена значение 9600 бод, по приглашению, выводимому на экран, ввести значение скважности. В данной программе скважность задается числом из диапазона 1000–10000, которое определяет длительность положительного уровня сигнала на выводе P1.0 и, соответственно, относительную длительность открытого состояния ключа на MOSFET, управляющего электродвигателем.

Основная работа программы выполняется в обработчике прерывания таймера 2, который используется как генератор импульсов с регулируемой скважностью. Программно генератор импульсов реализован путем установки таймера 2 в режим автоперезагрузки с нулевыми значениями кода перезагрузки. Все действия по инициализации таймера 2 выполнены с помощью следующих операторов:

```
T2CON &= 0xFC;
RCAP2H = 0x0;
RCAP2L = 0x0;
ET2 = 1;
EA = 1;
T2CON |= 0x4;
```

Программа-обработчик прерывания декрементирует значение переменной `cnt`, которая определяет длительность высокого уровня на выводе P1.0 и, соответственно, скважность импульса. Значение `cnt` присваивается посредством переменной `tmpCnt`, величина которой вводится с клавиатуры терминала. Таким образом можно регулировать частоту вращения электродвигателя с консоли.

Таймер 1 используется в программе как генератор синхронизации обмена данных по серийному порту. При тактовой частоте 24,0 МГц для обмена данными со скоростью 9500 бод значение регистра TH1 должно быть равным 0xF3 и, кроме того, должен быть установлен бит SMOD (команда `PCON |= 0x80;`).

Как выглядит окно терминальной программы, выполняющейся на персональном компьютере, показано на рис. 4.14.

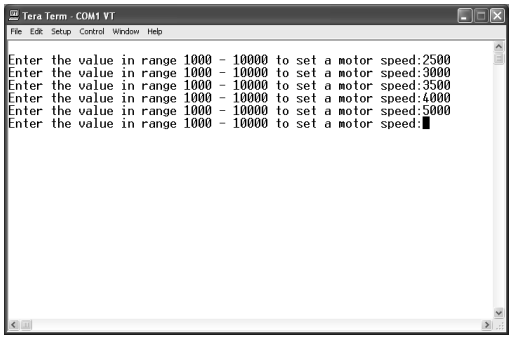
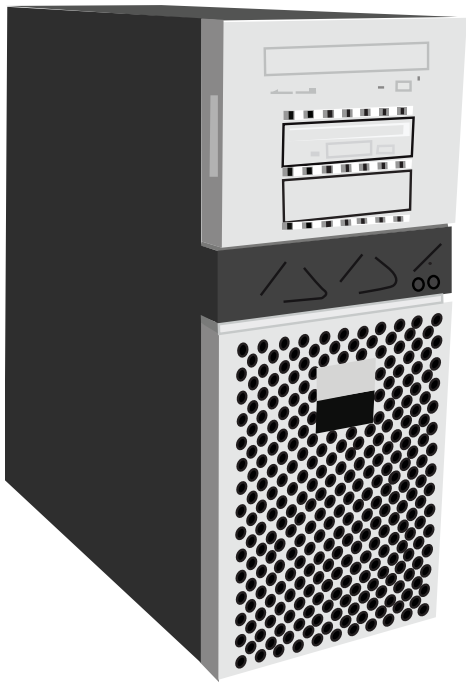


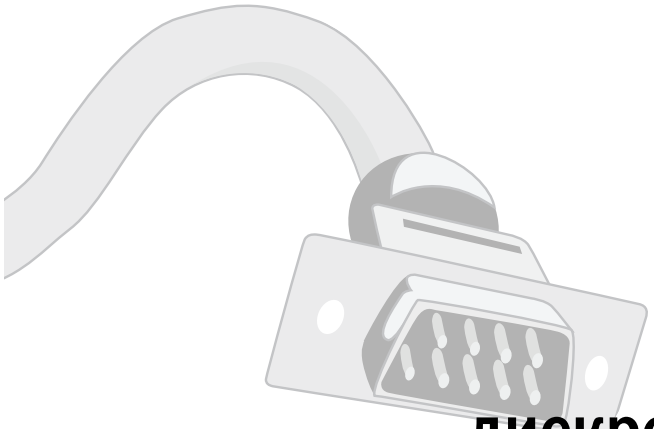
Рис. 4.14. *Вывод программы управления скважностью*

Оценить эффект от изменения скважности импульсов можно, присоединив вольтметр постоянного тока к стоку транзистора Q1 (см. рис. 4.13) и измерив с его помощью напряжение. С увеличением значения скважности (2500, 3000, 3500 и т.д.) среднее напряжение на стоке транзистора будет уменьшаться, что означает увеличение частоты вращения электродвигателя.

На этом мы закончим обзор возможностей таймеров микроконтроллера 8052 при разработке различных проектов.



Обработка дискретных сигналов



5 Обработка дискретных сигналов

Классический микроконтроллер 8051, как известно, имеет несколько портов ввода/вывода, позволяющих осуществлять обработку входных и генерировать выходные сигналы. Современные системы обработки данных манипулируют, как правило, с большим числом сигналов и требуют наличия намного большего количества портов ввода/вывода по сравнению с тем, что имеется в контроллере 8051. Есть несколько вариантов решения этой проблемы.

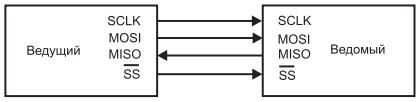
Один из вариантов – увеличить количество портов ввода/вывода за счет использования схемотехнических решений, применив, например, микросхемы расширителей интерфейсов или создав собственные аппаратные конфигурации, используя имеющуюся схемотехническую базу. В этих случаях для управления такими интерфейсами необходимо разработать соответствующее программное обеспечение (мы рассмотрим несколько вариантов таких решений).

Второй вариант предполагает использование специальных протоколов обмена данными, позволяющих подключать несколько устройств к ограниченному количеству сигнальных линий. В этом случае можно использовать, например, протоколы I²C, SMBus, SPI и др. Ведущими производителями электронных компонентов разработаны многочисленные устройства, совместимые с популярными протоколами, которые можно использовать в проектах на базе микроконтроллеров.

Рассмотрим оба варианта расширения интерфейсов. Вначале остановимся на реализации интерфейсов с устройствами на базе популярных протоколов SPI и I²C, а затем проанализируем возможности разработки собственных интерфейсов.

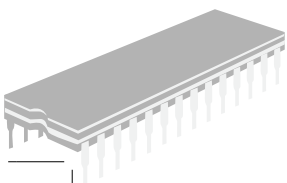
Интерфейс SPI (Serial Peripheral Interface) можно представить так, как показано на рис. 5.1.

Рис. 5.1. *Функциональная схема интерфейса SPI*



Интерфейс SPI представляет собой стандарт последовательной передачи данных в синхронном режиме, допускающий работу в полнодуплексном режиме. В большинстве случаев интерфейс функционирует между двумя устройствами, одно из которых является ведущим (master), а другое ведомым (slave). При этом ведущий инициирует обмен данными и синхронизирует их прием и/или передачу. Несколько устройств могут функционировать как ведомые, при этом ведущий должен выбирать ведомого, устанавливая сигнал выбора (slave select, chip select) для конкретного устройства.

Сигнал SCLK (Serial Clock) представляет собой последовательность синхронизирующих импульсов, генерируемых ведущим. Передача/прием данных обычно привязана к фронтам



или спадам 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 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, часто сигнал синхронизации SCLK обозначается как SCK, а 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, использует обычно упрощенную однонаправленную версию этого протокола. Например, большинство аналогово-цифровых преобразователей использует модель, в которой ведущим является устройство, получающее данные (микроконтроллер, компьютер и т.д.), которое соответственно синхронизирует передачу или прием данных. Практические примеры работы с устройствами по интерфейсу SPI мы рассмотрим далее.

Стандарт I^2C (читается как «I-square-C») реализован как двухпроводной последовательный интерфейс, разработанный Philips Corp. 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 I^2C могут быть подключены устройства с различными скоростями доступа, если скорость передачи данных будет удовлетворять требованиям самого низкоскоростного устройства.

Протокол передачи данных по шине I^2C разработан таким образом, чтобы гарантировать надежный и качественный прием/передачу данных. При передаче данных одно устройство является ведущим и инициирует передачу данных, а также формирует сигналы синхронизации. Другое устройство, ведомое, может начать передачу данных только по команде ведущего шины. Этот протокол обмена данными является более сложным в реализации, чем SPI, поэтому рассмотрим его подробно.

Протокол I^2C 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. Линии SCL и SDA могут соединять два и более устройств, как показано на рис. 5.2.

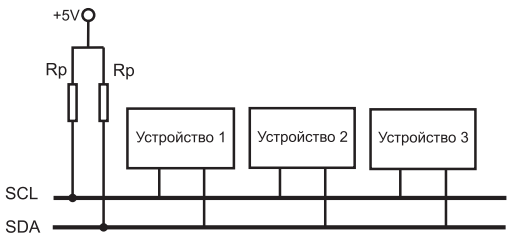
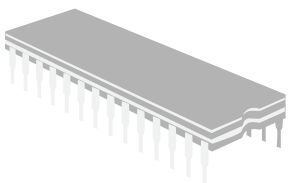


Рис. 5.2.
Схема взаимодействия устройств по шине I^2C

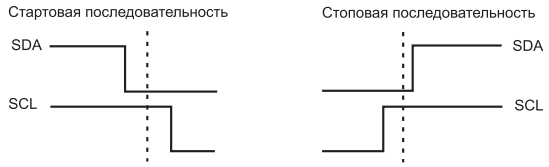
К сигнальным линиям необходимо подключить так называемые подтягивающие резисторы, обозначенные на схеме как Rp, присоединив их к источнику питания (для стандартной TTL-логики это +5 В). Подтягивающие резисторы (их значение может находиться в диапазоне 1–10 К) нужны для фиксации уровня сигналов, поскольку спецификация I^2C предусматривает использование устройств, имеющих выходные каскады с открытым коллектором (open collector) или стоком (open drain). Устройства могут быть или ведущими, или ведомыми. Ведущий



обеспечивает синхронизацию обмена данными, генерируя синхроимпульсы по линии SCL. На шине I²C может быть один или несколько ведущих и один или несколько ведомых устройств. В подавляющем большинстве случаев на шине находится только один ведущий и несколько ведомых устройств.

Обмен данными по шине I²C инициируется ведущим, который должен обеспечить выполнение стартовой (в начале обмена) и стоповой (в конце обмена) последовательности сигналов. Эти последовательности сигналов показаны на рис. 5.3.

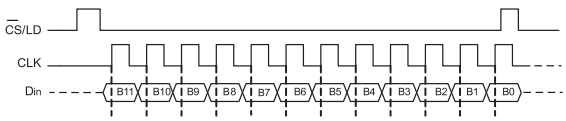
Рис. 5.3. *Стартовая и стоповая последовательности сигналов на шине I²C*



Протокол I²C требует, чтобы сигнал на линии данных SDA оставался неизменным при высоком уровне сигнала SCL. Перепад сигнала на линии SDA из высокого в низкий уровень при высоком уровне сигнала на линии SCL инициируется ведущим и указывает на начало обмена данными. Если при высоком уровне сигнала на линии SCL сигнал на линии SDA переходит из низкого в высокий уровень, то эта последовательность сигналов интерпретируется как завершение операции обмена данными на шине.

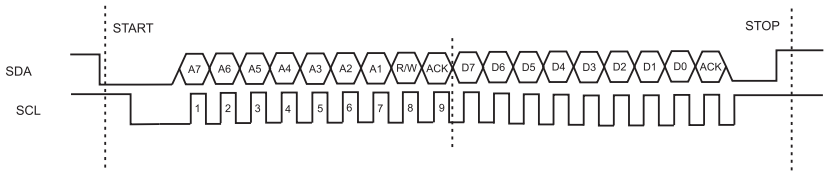
Данные передаются по шине в виде последовательности из 8 бит, причем первым в последовательности идет старший значащий бит (MSB). Каждый бит передается по нарастающему фронту сигнала SCL, причем, как уже упоминалось, при высоком уровне сигнала SCL сигнал на линии SDA должен оставаться неизменным, если 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). Бит ответа фиксируется по нарастающему фронту 9-го импульса SCL. Последовательность передачи данных показана на рис. 5.4.

Рис. 5.4. *Передача данных по шине I²C*



Схема, показанная на рис. 5.4, демонстрирует общие принципы обмена данными по шине I²C. На практике, чтобы записать, например, байт данных в устройство, необходимо знать как минимум его адрес. В этом случае запись байта в устройство можно изобразить схемой, показанной на рис. 5.5.

Рис. 5.5. *Запись байта 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/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.

5.1. Обработка входных данных с использованием SPI

Рассмотрим некоторые практические аспекты реализации обработки входных дискретных сигналов в микроконтроллере на примерах применения интерфейса SPI в пользовательских разработках.

Наш первый проект позволяет измерить температуру объекта с помощью термопары K-типа, причем диапазон измерения лежит в пределах от 0 до 1024 °C. Измерения температур с помощью термопар (thermocouples), так же как, например, с помощью резистивных датчиков температуры (RDT), наиболее часто используются в промышленности и научных исследованиях.

Роль измерительного преобразователя сигнала термопары выполняет микросхема MAX6675, цифровой сигнал с которой по интерфейсу SPI передается в микроконтроллер, как показано на рис. 5.6.

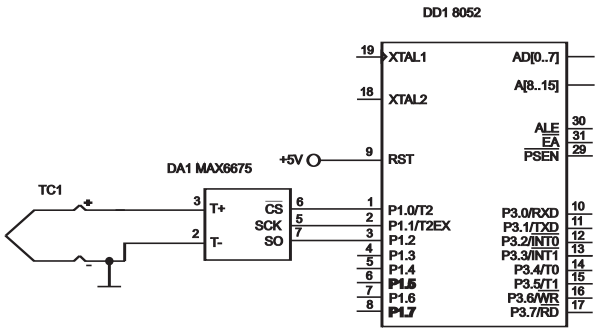


Рис. 5.6. *Измерение температуры по интерфейсу SPI*

Микросхема MAX6675 осуществляет полную первичную обработку сигнала термопары K-типа, выполняя расчет компенсирующей ЭДС для холодного спая и преобразование полученного сигнала в цифровую форму. Далее 12-разрядный цифровой сигнал передается по SPI-совместимому интерфейсу в микроконтроллер или компьютер. Данный преобразователь позволяет измерить температуру вплоть до 1024 °C с разрешением 0,25 °C. Детальное описание принципа функционирования данной микросхемы можно найти на сайте фирмы-производителя Maxim Inc., а описание функционирования термопар K-типа можно найти на просторах Интернета с помощью поисковых систем или на сайтах производителей соответствующего оборудования.

Микросхема подключается к термопаре следующим образом: вывод «T+» соединяется с хромелевым выводом термопары, а вывод «T–» с алюмелевым выводом термопары.

Вывод SCK микросхемы используется для приема синхронизирующих импульсов, которые инициируют прием данных по линии SO по нарастающему фронту. Сигнал CS, установленный в низкий уровень, разрешает прием данных по интерфейсу.

Сигналы SCK и CS являются входными для микросхемы, а SO – выходным. Временная диаграмма работы интерфейса показана на рис. 5.7.

Как видно из временной диаграммы, для считывания данных требуется 16 синхроимпульсов, причем значение температуры закодировано в 12 битах, начиная с D14 и заканчивая D3. Бит данных считывается в микроконтроллер по нарастающему фронту импульсов SCK. Считывание битов данных выполняется только при низком уровне сигнала CS.

Программная часть проекта разработана с использованием компилятора Keil C51. Вот исходный текст программы:

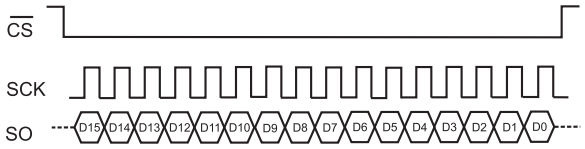


Рис. 5.7.
Передача данных по SPI в измерителе температуры

```
#include <stdio.h>
#include <REG52.H>

sbit CS = P1^0;
sbit SCK = P1^1;
sbit SO = P1^2;

unsigned int temp;
unsigned int cnt;

void delay(void)
{
unsigned int i1 = 5;
while(i1 != 0)i1—;
}

unsigned int getTemp(void)
{
temp = 0;
P1 = 0xFC;
CS = 1;
CS = 0;
delay();
for (cnt = 0; cnt < 16; cnt++)
{
    SCK = 0;
    delay();
    temp |= SO;
    temp = temp<<1;
    SCK = 1;
}
CS = 1;
temp = temp>>3;
temp &= 0xFFF;
return temp;
}

void main(void)
{
float ft1;

SCON = 0x50;
TH1 = 0xFD;
TMOD |= 0x20;
```

```
TR1 = 1;
TI = 1;

while (1)
{
ft1 = getTemp() * 1024.0 / 4096.0;
printf("Current temperature: %6.1f\n", ft1);
printf("Press Enter to continue...\n");
getchar();
}
}
```

Программа ожидает нажатия клавиши на консоли удаленного терминала, после чего выводит в последовательный порт значение температуры, измеренной с помощью термопары. Параметры обмена по последовательному порту рассчитаны для скорости 9600 бод при тактовой частоте 11,059 МГц.

Прием данных по интерфейсу SPI осуществляется в цикле `for` функции `getTemp`. Счетчик цикла `cnt` для упрощения принимает значения от 0 до 15, после чего 16-разрядное значение температуры, сохраненное в переменной `temp`, корректируется так, чтобы отсечь ненужные биты D15, D2 – D0. Это выполняется следующим фрагментом программного кода:

```
temp = temp>>3;
temp &= 0xFFF;
```

В данном проекте используется интерфейс SPI, в котором обмен данными осуществляется в одном направлении, причем инициатором обмена (ведущим) является микроконтроллер. Следует заметить, что в большинстве случаев обменом данными по интерфейсу SPI управляет именно микроконтроллер, обеспечивая требуемую последовательность сигналов. Ведомое устройство в таких случаях только выдает или принимает данные по фронту или спаду синхросигналов.

В следующем проекте интерфейс SPI используется как для записи, так и для считывания данных в/из устройства, причем ведущим, как и в рассмотренном измерителе температуры, является микроконтроллер 8052.

В этом проекте будет продемонстрирован обмен данными по протоколу SPI с микросхемой перепрограммируемой памяти (EEPROM) 25C320 с объемом памяти 32 Кбит. Принципиальная схема аппаратной части показана на рис. 5.8.

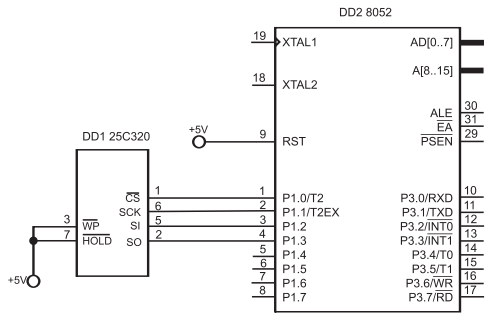


Рис. 5.8. *Схема обмена данными с памятью по протоколу SPI*

Несколько слов о принципах функционирования микросхемы 25C320. Микросхема памяти содержит 8-разрядный регистр команд. Считывание данных из устройства осуществляется

через вывод SO по фронту сигнала SCK, при этом сигнал CS должен иметь низкий уровень, а вывод HOLD – высокий. Вывод WP при выполнении операций записи должен иметь высокий уровень, при низком уровне микросхема защищена от записи.

Обмен данными по протоколу SPI осуществляется от старшего значащего бита (MSB), который идет первым, к младшему (LSB). При записи биты данных на линии SI записываются в микросхему по фронту сигнала SCK при низком уровне сигнала CS. При высоком уровне на линии CS прием/передача данных в/из EEPROM невозможны.

Таким образом, обмен данными с микросхемой памяти осуществляется в двух направлениях, при этом ведущим в данном случае является микроконтроллер.

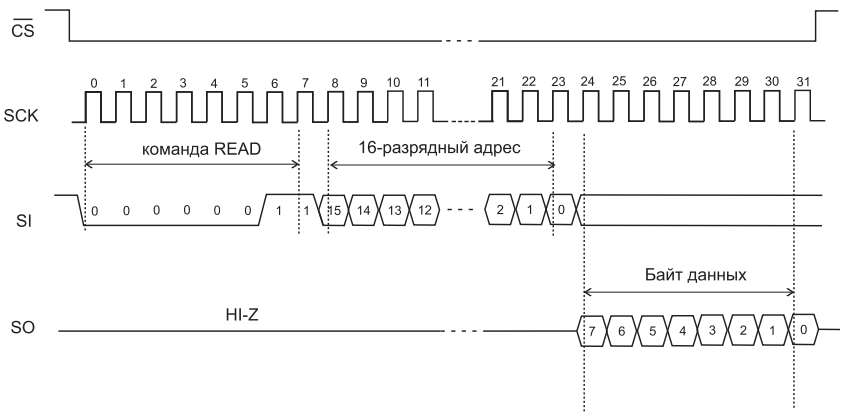
Все операции обмена данными начинаются с определенных команд, которые перечислены в табл. 5.1.

Таблица 5.1. *Команды микросхемы памяти*

Команда	Код	Выполняемая функция
READ	0000 0011	Чтение данных из памяти, начиная с определенного адреса
WRITE	0000 0010	Запись данных в память, начиная с определенного адреса
WRDI	0000 0100	Сброс регистра-защелки разрешения записи (запрет операций записи)
WREN	0000 0110	Установка регистра-защелки разрешения записи (разрешение записи данных)
RDSR	0000 0101	Чтение регистра состояния
WRSR	0000 0001	Запись в регистр состояния

Проанализируем, как выполняется чтение данных из микросхемы памяти, для чего обратимся к рис. 5.9.

Рис. 5.9. *Временная диаграмма чтения данных из микросхемы памяти 25C320*



Устройство выбирается при установке сигнала CS в низкий уровень. Вначале передается команда чтения READ (ее код равен 3), затем 16-разрядный адрес, старшие 4 бита которого игнорируются. Если команда чтения выполнена корректно и установлен корректный адрес, то считываемые данные бит за битом появляются на выводе SO. При этом данные, находящиеся по следующим адресам, могут считываться при подаче последующих синхроимпульсов, поскольку внутренний указатель адреса в микросхеме памяти автоматически инкрементируется после передачи байта данных в микроконтроллер. По достижении адреса 0xFFF следующее значение указателя станет 0x000, и цикл чтения продолжится. Операция чтения, как и другие, заканчивается при

установке сигнала CS в высокий уровень. Последовательность операций при записи данных несколько иная, чем при чтении. Прежде всего, перед выполнением операции записи данных в микросхему 25C320 необходимо разрешить запись данных, выполнив команду WREN (рис. 5.10).

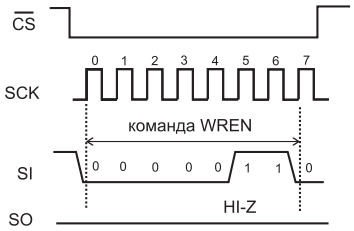


Рис. 5.10. *Разрешение записи данных*

Как обычно, вначале устанавливаем сигнал CS в низкий уровень, после чего бит за битом отправляем команду WREN (ее код равен 6) в EEPROM. По окончании записи команды обязательно следует установить сигнал CS в высокий уровень. Если этого не сделать, то следующая за данной командой запись данных не выполнится, поскольку регистр-защелка не будет установлен должным образом. Обратите внимание на то, что линия SO должна находиться в высокоимпедансном состоянии.

После установки регистра-защелки можно записывать байт данных в микросхему памяти. Последовательность операций, выполняемых при записи байта данных, показана на рис. 5.11.

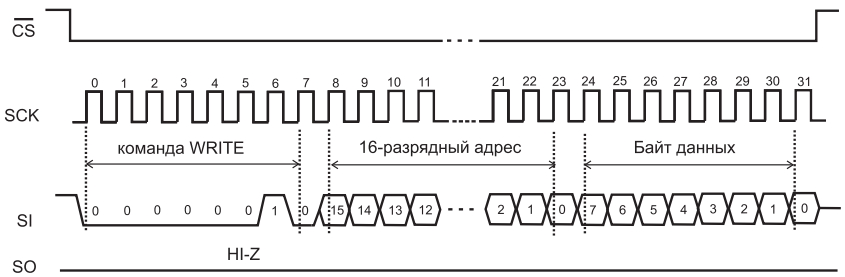


Рис. 5.11. *Запись байта данных в микросхему памяти*

Запись байта данных начинается с установки сигнала CS в низкий уровень, после чего на линию SI выталкиваются биты команды WRITE (код равен 2). Немедленно после команды микроконтроллер должен выставить 16-разрядный адрес ячейки памяти, затем байт данных для записи. Цикл записи байта в память заканчивается при установке сигнала CS в высокий уровень. Сигнал CS должен устанавливаться только после записи младшего бита данных, иначе операция записи будет неудачной.

Установка сигнала CS, тем не менее, не означает окончания внутренних операций микросхемы по записи данных. Вполне возможно, что операция записи фактически будет продолжаться еще некоторое время. В этом случае перед выполнением чтения байта следует проверить бит 0 регистра состояния EEPROM.

Регистр состояния имеет поля, изображенные на рис. 5.12.

Регистр состояния

7	6	5	4	3	2	1	0
WPEN	X	X	X	BP1	BP0	WEL	WIP

Рис. 5.12. *Регистр состояния микросхемы 25C320*

Бит WIP (Write-In-Process) показывает, выполняет ли устройство 25C320 операцию записи. Если бит установлен в 1, то операция записи выполняется, если в 0, то операция закончилась.

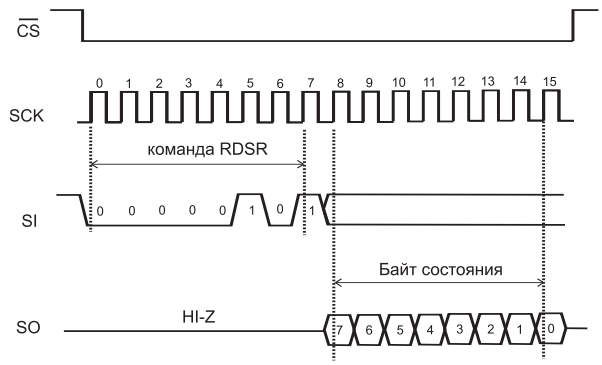
Этот бит устанавливается логикой микросхемы и доступен только для чтения. Прочитав этот бит, программа определяет, когда можно выполнить следующую операцию, например, чтение только что записанного байта.

Бит WEL (Write Enable Latch) показывает состояние регистра-защелки записи. Если бит установлен в 1, регистр-защелка разрешает запись данных, если установлен в 0, то запись запрещена. Состояние этого бита можно обновить, выполнив либо команду WREN (разрешение записи), либо команду WRDI (запрет записи).

По окончании цикла записи регистр-защелка сбрасывается. Биты BP0 и BP1 показывают, какие блоки памяти являются защищенными, и устанавливаются командой WRSR.

Программа может прочитать регистр состояния с помощью команды RDSR (ReaD Status Register). Временная диаграмма показана на рис. 5.13.

Рис. 5.13. *Временная диаграмма цикла чтения регистра состояния*



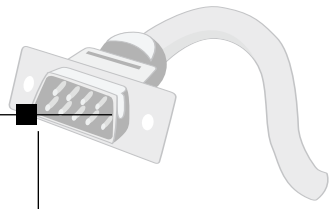
Цикл чтения регистра состояния начинается с установки сигнала CS в низкий уровень, после чего на линию SI побитово выводится команда RDSR (код равен 5). Вывод битов команды выполняется синхронно с импульсами 0–7 CLK, после чего на линию SO выводится бит за битом байт статуса, который считывается микроконтроллером. Завершается цикл чтения регистра состояния установкой сигнала CS в высокий уровень.

Рассмотрим программный код, демонстрирующий запись/чтение одного байта в/из EEPROM 25C320. Алгоритм работы программы таков:

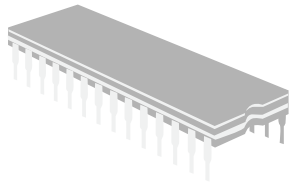
- по команде WREN разрешается запись данных в микросхему памяти;
- по команде WRITE байт данных записывается по указанному адресу в памяти;
- программа ожидает завершения операции записи, периодически анализируя бит 0 регистра состояния (используется команда RDSR);
- если операция записи завершена, программа читает только что записанный байт данных и выводит его значение в последовательный порт.

Программа написана на Keil C51, и ее исходный текст показан ниже:

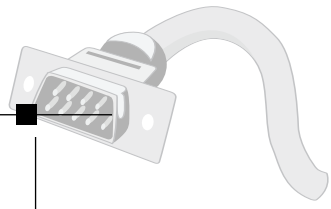
```
#include <stdio.h>
#include <intrins.h>
#include <REG52.H>

#define WREN  0x6
#define WRITE 0x2
#define READ  0x3
#define RDSR  0x5
```



```
#define ADDR  0x2

sbit CS = P1^0;
sbit SCK = P1^1;
sbit SI = P1^2;
sbit SO = P1^3;

unsigned char byteRead;
unsigned char Cmd;
unsigned int Addr;
unsigned int cnt;

void delay(unsigned int d1)
{
while(d1 != 0)d1–;
}

void shift_out8(unsigned char c1)
{
for (cnt = 0; cnt < 8; cnt++)
{
    SCK = 0;
    delay(20);
    c1 = _crol_(c1, 1);
    SI = c1 & 0x1;
    delay(2);
    SCK = 1;
    delay(20);
}
}

void shift_out16(unsigned int i1)
{
for (cnt = 0; cnt < 16; cnt++)
{
    SCK = 0;
    delay(20);
    i1 = _irol_(i1, 1);
    SI = i1 & 0x1;
    delay(2);
    SCK = 1;
    delay(20);
}
}

unsigned char shift_in8(void)
{
unsigned char t1;
t1 = 0;
for (cnt = 0; cnt < 9; cnt++)
```



```
{
    SCK = 0;
    delay(10000);
    t1 = t1<<1;
    t1 |= SO;
    SCK = 1;
    delay(10);
}
return t1;
}

unsigned char readByte(void)
{
CS = 1;
delay(1);
CS = 0;
Cmd = READ;
Addr = ADDR;

shift_out8(Cmd);
shift_out16(Addr);
byteRead = shift_in8();
delay(20);
CS = 1;
return byteRead;
}

bit readStatus(void)
{
bit stat;
unsigned char ctmp;

CS = 1;
CS = 0;
Cmd = RDSR;

shift_out8(Cmd);
ctmp = shift_in8();
CS = 1;
stat = ctmp & 0x1;
return stat;
}

void writeByte(unsigned char c1)
{
CS = 1;
delay(2);
CS = 0;

Cmd = WREN;
```



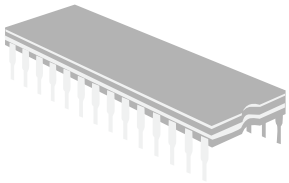
```
shift_out8(Cmd);
CS = 1;
delay(2);
CS = 0;

Cmd = WRITE;
Addr = ADDR;

shift_out8(Cmd);
shift_out16(Addr);
shift_out8(c1);
CS = 1;
}

void main(void)
{
unsigned char c2 = '9';
bit Busy;

P1 |= 0x8;

SCON = 0x50;
TH1 = 0xFD;
TMOD |= 0x20;
TR1 = 1;
TI = 1;

writeByte(c2);
delay(1000);
while ((Busy = readStatus()) == 1);
c2 = readByte();
printf("\nByte received: %c\n", c2);
while(1);
}
```

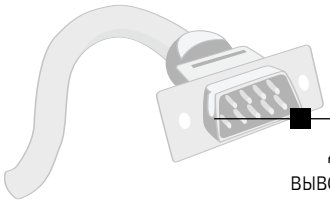
Перед анализом исходного текста хочу сделать одно важное замечание: при работе с EEPROM нужно очень тщательно просчитывать временные зависимости, особенно при чтении данных из микросхемы памяти, и задавать необходимые задержки между сигналами. В любом случае, перед использованием такой памяти следует изучить все временные зависимости данного чипа и рассчитать требуемые временные задержки.

Анализ исходного текста начнем с переменных и констант, определенных в нашей программе. Группа директив

```
#define  WREN   0x6
#define  WRITE  0x2
#define  READ   0x3
#define  RDSR   0x5
#define  ADDR   0x2
```

определяет константы, соответствующие кодам команд EEPROM, которые нам уже знакомы. Константа `ADDR` выбрана произвольно – она определяет адрес, по которому будет записан байт данных и из которого он позже будет считан.





Директивы `sbit` определяют биты порта P1, к которым присоединяются соответствующие выводы микросхемы памяти. Соединения соответствуют принципиальной схеме, показанной на рис. 5.8.

Переменная `byteRead` будет содержать прочитанный из EEPROM байт, в переменную `Cmd` помещаются коды выполняемых команд, переменная `Addr` содержит выбранный нами адрес памяти, а переменная `cnt` является счетчиком записываемых и читаемых битов.

Программа содержит несколько функций. В их число входят `shift_out8`, `shift_out16`, `shift_in8`, `delay`, `readByte`, `readStatus`, `writeByte`, `initWR`. Функции `readByte` и `writeByte` соответственно считывают и записывают байт данных в/из памяти. Функция `readStatus` выполняет считывание содержимого регистра состояния, а функция `initWR` устанавливает регистр-защелку перед записью байта данных в память.

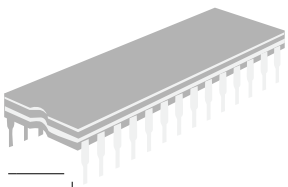
Функции `shift_out8`, `shift_out16` и `shift_in8` выполняют отдельные операции в циклах чтения/записи. Так, например, функция `shift_out8` записывает в EEPROM побитово байт по линии SI, функция `shift_out16` делает то же самое для 16-разрядного слова (при установке адреса в циклах чтения/записи), а функция `shift_in8` принимает байт из памяти по линии SO. Функция `delay` служит для формирования временных задержек при операциях чтения/записи.

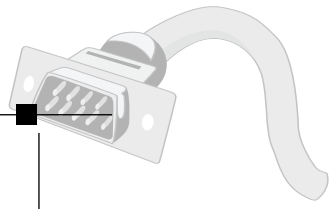
Для выполнения циклических сдвигов используются специальные встроенные функции Keil C `_crol_` (циклический сдвиг байта на определенное число позиций) и `_irol_` (циклический сдвиг 16-разрядного слова на определенное число позиций). Эти функции определены в файле заголовка intrins.h.

Для тех читателей, которые вместо компилятора Keil C51 используют другой, например, свободно распространяемый GNU SDCC, я приведу исходный текст программы, в котором не используются внутренние функции Keil C. В листинге, представленном далее, функции `shift_out8`, `shift_out16`, `shift_in8`, `delay` написаны на ассемблере, а основная программа на Keil C51. В этом случае перейти от Keil к другому компилятору, например, к тому же SDCC или IAR, не составит труда.

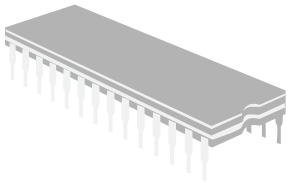
Вот исходный текст программы записи/чтения байта в/из EEPROM, в котором использованы функции, написанные на языке ассемблера:

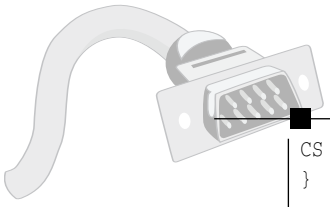
```
#include <stdio.h>
#include <REG52.H>

#define WREN  0x6
#define WRITE 0x2
#define READ  0x3
#define RDSR  0x5
#define ADDR  0x2

extern void delay(unsigned int i1);
extern void shift_out8(unsigned char c1);
extern void shift_out16(unsigned int i1);
extern unsigned char shift_in8(void);

sbit CS = P1^0;
unsigned char byteRead;
unsigned char Cmd;
unsigned int Addr;
unsigned int cnt;
```





```
unsigned char readByte(void)
{
CS = 1;
delay(1);
CS = 0;
Cmd = READ;
Addr = ADDR;

shift_out8(Cmd);
shift_out16(Addr);
byteRead = shift_in8();
delay(20);
CS = 1;
return byteRead;
}

bit readStatus(void)
{
bit stat;
unsigned char ctmp;

CS = 1;
CS = 0;
Cmd = RDSR;

shift_out8(Cmd);
ctmp = shift_in8();
CS = 1;
stat = ctmp & 0x1;
return stat;
}

void writeByte(unsigned char c1)
{
CS = 1;
delay(2);
CS = 0;

Cmd = WREN;
shift_out8(Cmd);
CS = 1;
delay(2);
CS = 0;

Cmd = WRITE;
Addr = ADDR;

shift_out8(Cmd);
shift_out16(Addr);
shift_out8(c1);
```



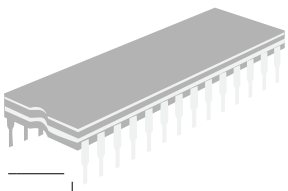


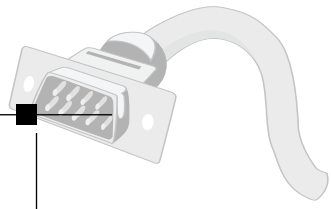

```
CS = 1;
}

void main(void)
{
unsigned char c2 = '7';
bit Busy;

P1 |= 0x8;

SCON = 0x50;
TH1 = 0xFD;
TMOD |= 0x20;
TR1 = 1;
TI = 1;

do {
    printf("Enter character to write to EEPROM:");
    c2 = getchar ();
    if (c2 == 0x1B)
    break;
    writeByte(c2);
    delay(1000);
    while ((Busy = readStatus()) == 1);
    c2 = readByte();
    printf("\nByte RECEIVED: %c\n", c2);
}while (1);
}
```

Как видно из исходного текста программы, мы несколько модифицировали программный код. После компиляции и запуска программа запрашивает ввод символа с консоли удаленного терминала, работающего с последовательным портом.

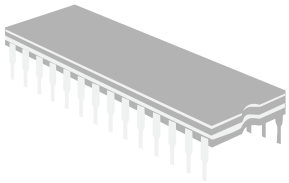
Функции, объявленные с директивой `extern`, содержатся в отдельном файле с расширением .asm. Вот содержимое этого файла:

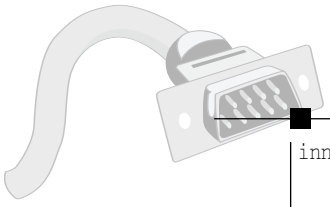
```
    NAME    PROCS
    PUBLIC      _shift_out8, _shift_out16, shift_in8, _delay
    SCK     EQU P1.1
    SI      EQU P1.2
    SO      EQU P1.3
    PROG    SEGMENT CODE
    USING   0
    RSEG    PROG
    ;---------------
_shift_out8:
    MOV     A, R7
    MOV     R4, #8

next_out8:
    CLR     SCK
```





```
    RLC     A
    MOV     SI, C
    SETB    SCK
    DJNZ    R4, next_out8
    RET
    ;————————————————
_shift_out16:
    MOV     A, R6
    MOV     R4, #8

next_high_out16:
    CLR     SCK
    RLC     A
    MOV     SI, C
    SETB    SCK
    DJNZ    R4, next_high_out16

    MOV     A, R7
    MOV     R4, #8

next_low_out16:
    CLR  SCK
    RLC     A
    MOV     SI, C
    SETB    SCK
    DJNZ    R4, next_low_out16
    RET
    ;————————————————
shift_in8:
    CLR     A
    MOV     R4, #8

next_in8:
    CLR     SCK
    MOV     R6, #250
    MOV     R7, #150
    CALL _delay
    MOV     C, SO
    RLC     A
    SETB    SCK
    MOV     R6, #5
    MOV     R7, #5
    CALL _delay
    DJNZ    R4, next_in8

    MOV     R7, A
    RET
    ;————————————————
_delay:
    DJNZ    R6, inner_loop
    RET
```






```
inner_loop:
    PUSH     7h
again:
    NOP
    DJNZ     R7, again
    POP      7h
    JMP  _delay

    END
```

Обратите внимание на то, что функция `_shift_out8` принимает однобайтовый параметр в регистре `R7`, а функции `_shift_out16` и `_delay` принимают двухбайтовые параметры в регистрах `R6` и `R7`, причем `R6` содержит старший байт, а `R7` – младший. Кроме того, в Keil C51 для функций, принимающих параметры, требуется ставить символ подчеркивания в начале имени. При этом имена функций, объявленные в основной программе на C, пишутся без подчеркивания. Функция `shift_in8` не принимает никаких параметров от вызывающей программы, поэтому символ подчеркивания в начале имени здесь не нужен. Эта функция возвращает прочитанный байт основной программе в регистре `R7`. Все функции, вызываемые из другой программы, должны быть объявлены как доступные из внешних модулей, для чего служит директива `PUBLIC` в начале листинга программы.

Хотел бы обратить внимание читателей на реализацию временной задержки в функции `_delay`. Поскольку используется 16-разрядное значение, то задержка организована в виде двух циклов с командами `DJNZ`. Здесь же используются команды `PUSH` и `POP`. Поскольку ассемблер Keil A51 (как и подавляющее большинство других ассемблеров) не использует команды наподобие

```
PUSH R7
POP R7
```

то, учитывая, что в программе используется банк 0 регистров общего назначения, имеющих адреса с 0x0 до 0x7, можно применить команды

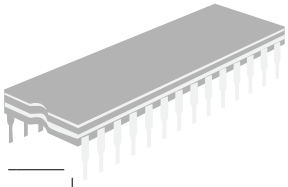
```
PUSH 7h
POP 7h
```

Особо хочу акцентировать внимание на временных задержках, используемых в программах, – их нужно просчитывать для конкретного типа микросхемы.

Использование протоколов SPI и I^2C хоть и решает в целом ряде случаев вопросы разработки интерфейсов, но, тем не менее, не всегда может быть применимо. В целом ряде случаев требуется разработать нестандартный интерфейс, который иногда трудно реализовать в рамках существующих протоколов. Далее мы рассмотрим некоторые практические примеры реализации таких интерфейсов.

5.2. Пользовательские интерфейсы ввода дискретных данных

Ограниченное количество портов ввода/вывода данных требует от разработчика использования аппаратно-программных решений, позволяющих устранить эту проблему. Рассмотрим некоторые практические проекты обработки входных данных.



Наш первый проект позволяет обрабатывать 8 сигналов от цифровых источников по одной линии, используя мультиплексирование. Аппаратная часть проекта схемотехнически может быть реализована так, как показано на рис. 5.14.

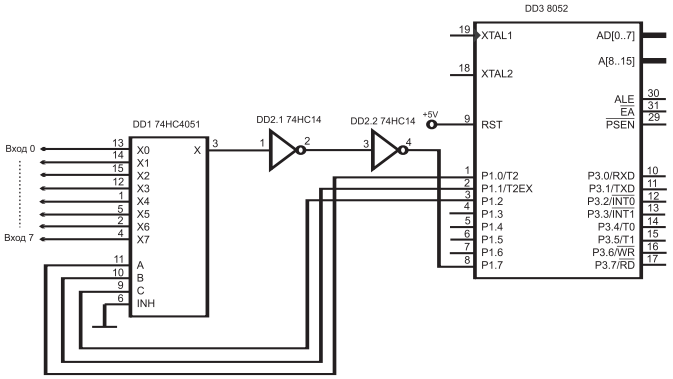
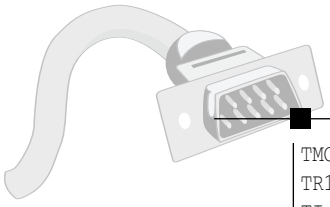


Рис. 5.14. *Мультиплексирование ввода дискретных данных*

Здесь для расширения количества обрабатываемых входных сигналов применена микросхема 74HC4051, представляющая собой 8-входовый мультиплексор сигналов. В зависимости от двоичного кода на входах A – C, на выходе X появляется один из входных сигналов. Вход INH микросхемы при низком уровне напряжения разрешает работу мультиплексора, поэтому он соединен с общим проводом схемы. Триггер Шмита используется для формирования сигнала, поступающего на вход P1.7 микроконтроллера.

Ниже представлен исходный текст программы, написанной на Keil C51 и реализующей обработку входных цифровых данных в соответствии с принципиальной схемой на рис. 5.14:

```
#include <stdio.h>
#include <intrins.h>
#include <REG52.H>

sbit A0 = P1^0;
sbit A1 = P1^1;
sbit A2 = P1^2;

sbit Input = P1^7;

extern unsigned int shift_in8(void);

void delay(unsigned int i1)
{
while(i1 != 0)i1–;
}
void main(void)
{
unsigned int cnt;
unsigned int dat;

SCON = 0x50;
PCON |= 0x80;
TH1 = 0xF3;
```




```
TMOD |= 0x20;
TR1 = 1;
TI = 1;

while(1)
{
P1 = 0x0F8;
dat = 0;
for (cnt = 0; cnt < 8; cnt++)
{
    delay(1);
    dat |= Input;dat = _iror_(dat, 1);
    P1++;
    delay(1);
}
dat = dat>>8;
printf("Input VALUE = %Xh\n", dat);
printf("Press Enter to continue...\n");
getchar();
}
}
```

Программа ожидает нажатия клавиши на консоли терминала или ПК, соединенного с микроконтроллером по последовательному порту, после чего выводит шестнадцатеричный код, показывающий состояние входов 0–7, обратно в последовательный порт. Параметры обмена данными рассчитаны для скорости 9600 бод при тактовой частоте 24,0 МГц. Для использования программы при других значениях частоты кристалла следует пересчитать значения для таймера 1, используемого в качестве генератора синхронизации.

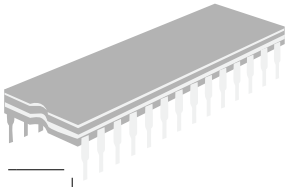
В этой программе, как и в предыдущих, в процессе вычислений мы используем одну из внутренних функций компилятора Keil C, а именно `_iror_`, с помощью которой осуществляется циклический сдвиг 16-разрядной переменной на 1 бит вправо. Считывание данных с 8 каналов осуществляется в цикле `for`, в котором принимаемый на выводе P1.7 бит данных последовательно сдвигается вправо так, чтобы младшему номеру канала мультиплексора соответствовал младший бит данных. Таким образом, после 8 сдвигов старший байт переменной `dat` будет содержать 8 бит данных, причем 15-й бит переменной `dat` будет соответствовать уровню сигнала на входе канала 7, а 8-й бит – уровню сигнала на входе канала 0.

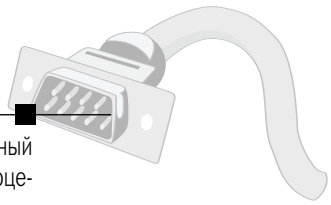
Далее программа выполняет сдвиг битов вправо на 8 позиций так, чтобы поместить данные в младший байт переменной `dat`. Это выполняет команда

```
dat = dat>>8;
```

Затем значение переменной `dat` выводится в последовательный порт, после чего программа ожидает нажатия клавиши на консоли для выполнения следующего цикла считывания данных.

В программе используется функция `delay`, позволяющая выполнить определенные временные задержки, необходимые для корректной работы микросхемы мультиплексора. Использование временной задержки является, вообще говоря, опциональным, и можно обойтись без этой функции, если в аппаратной части применяется достаточно быстродействующая микросхема мультиплексора. В любом случае желательно просмотреть документацию на используемые в устройстве комплектующие.



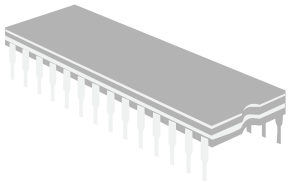


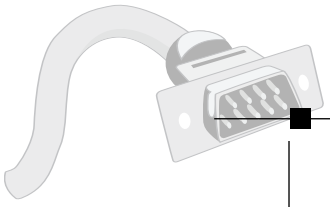
Для читателей, работающих с другими компиляторами С, приводится альтернативный листинг программы, в котором внутренняя функция Keil C51 заменена ассемблерной процедурой. Вот этот исходный текст:

```
#include <stdio.h>
#include <REG52.H>

sbit A0 = P1^0;
sbit A1 = P1^1;
sbit A2 = P1^2;

sbit Input = P1^7;

extern unsigned int shift_in8(void);

void delay(unsigned int i1)
{
while(i1 != 0)i1—;
}

void main(void)
{
unsigned int cnt;
unsigned int dat;

P1 = 0x0F8;
SCON = 0x50;
PCON |= 0x80;
TH1 = 0xF3;
TMOD |= 0x20;
TR1 = 1;
TI = 1;

while(1)
{
dat = 0;
dat = shift_in8();

printf(“Input value = %xh\n“, dat);
printf(“Press Enter to continue...\n“);
getchar();
}
}
```

Внешняя по отношению к основной программе процедура `shift_in8` содержится в файле с расширением .asm. Исходный текст ассемблерного модуля приводится ниже:

```
    NAME     PROCS
    PUBLIC   shift_in8
    dat      EQU P1.7
```






```
    PROG      SEGMENT CODE
    RSEG      PROG
;-----------
shift_in8:
    MOV       P1, #0F8h
    MOV       R4, #8
    CLR       A
next_bit:
    MOV       C, dat
    NOP
    NOP
    RRC       A
    INC       P1
    DJNZ      R4, next_bit
    MOV       R7, A
    MOV       R6, #0
    RET
    END
```

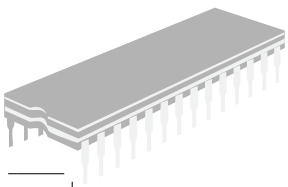
Процедура возвращает в основную программу беззнаковое целое в регистрах R6 и R7, причем в регистр R6 заносится нулевое значение, а младший байт помещается в регистр R7.

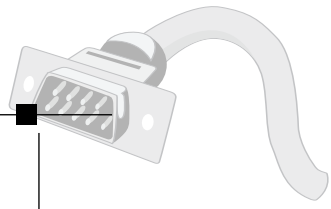
Как видите, даже такой простой интерфейс позволяет увеличить количество входных линий для приема дискретных сигналов, по крайней мере, на 5. Если в такой схеме использовать 16-разрядный мультиплексор, например 74HC4067, то количество дискретных входов можно еще более увеличить. При этом можно использовать только что приведенные исходные тексты программ, слегка их изменив. Естественно, что аппаратная часть также должна быть изменена соответствующим образом.

Недостатком такого алгоритма обработки входных данных является необходимость непрерывного считывания данных, что во многих случаях неприемлемо для программ, работающих в реальном времени. Выходом из этой ситуации может быть использование прерываний. При этом можно организовать циклический опрос входных данных, используя прерывания одного из таймеров микроконтроллера 8052, либо использовать одну из линий внешних прерываний.

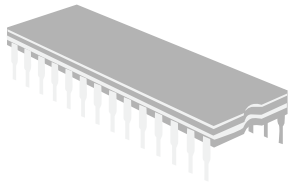
Рассмотрим пример программы обработки входных данных по схеме, приведенной на рис. 5.14, с использованием таймера 2 микроконтроллера 8052. Предполагаем, что считывание входов мультиплексора осуществляется каждую секунду, а результат выводится в последовательный порт. Исходный текст программы:

```
#include <stdio.h>
#include <intrins.h>
#include <REG52.H>

sbit A0 = P1^0;
sbit A1 = P1^1;
sbit A2 = P1^2;

sbit Input = P1^7;
bit bRunning = 1;

unsigned int cnt;
```





```
unsigned int dat;
unsigned int interval = 0;

void delay(unsigned int i1)
{
while(i1 != 0)i1—;
}

void T2Isr(void) interrupt 5 using 1{
TF2 = 0;
interval++;
if (interval > 330)
{
P1 = 0x0F8;
interval = 0;
dat = 0;
for (cnt = 0; cnt < 8; cnt++)
{
    delay(1);
    dat |= Input;
    dat = _iror_(dat, 1);
    P1++;
    delay(1);
}
dat = dat>>8;
bRunning = 0;
}
}
void main(void)
{

SCON = 0x50;
PCON |= 0x80;
TH1 = 0xF3;
TMOD |= 0x20;
TR1 = 1;
TI = 1;

T2CON &= 0xFC;
RCAP2H = 0x0;
RCAP2L = 0x0;
EA = 1;
ET2 = 1;
TR2 = 1;

while(1)
{
if (bRunning == 0){
    printf(“Input value total = %Xh\n“, dat);
    bRunning = 1;
```



```
    }
}
}
```

Здесь таймер 2 настроен на работу в режиме автоперезагрузки, для чего младшие два бита регистра T2CON сбрасываются в 0:

```
T2CON &= 0xFC;
```

Кроме того, биты разрешения всех прерываний и прерывания таймера 2 должны быть установлены:

```
EA = 1;
ET2 = 1;
```

Смысл команды

```
TR2 = 1;
```

думаю, понятен – она запускает таймер 2.

При указанных значениях в регистрах `RCAP2H` и `RCAP2L` таймер 2 будет перезагружаться приблизительно каждые 0,03 с, поэтому в обработчике прерывания таймера 2 значение счетчика перезагрузок `interval` выбрано равным 330 – в этом случае входные данные от мультиплексора будут обрабатываться каждые 10 с.

Программа-обработчик прерывания проверяет значение переменной `interval` и, если оно превысило `330`, считывает входные данные с 8 входов мультиплексора с помощью уже знакомого нам программного кода в операторе `if`. Кроме того, нужно как-то сообщить основной программе, что обработка данных закончена, для чего используется битовая переменная `bRunning`. Программный код остальной части программы достаточно понятен, поэтому останавливаться на нем я не буду.

Для многих практических задач считывания данных с определенной периодичностью вполне достаточно, тем не менее в целом ряде случаев необходимо зафиксировать появление одного или нескольких входных сигналов. Например, потребуется отследить появление низкого уровня любого из входных сигналов. Для критических по времени выполнения приложений использование прерывания таймера не сможет обеспечить немедленную фиксацию низкого уровня сигналов, а применять непрерывный опрос входов может показаться слишком накладным.

Применительно к нашему устройству, схема которого показана на рис. 5.14, можно использовать другую методику. Для этого потребуется изменить принципиальную схему устройства так, чтобы можно было зафиксировать перепад 1–0 одного из сигналов, а для обработки данных использовать одно из внешних прерываний, например EX0 (External 0 Interrupt).

Принципиальная схема модифицированного устройства обработки входных сигналов будет выглядеть так, как показано на рис. 5.15.

Как видно из принципиальной схемы, для фиксации перехода в низкий уровень любого из входных сигналов используется микросхема 8-входового элемента И–НЕ 74HC30 и триггер Шмита 74HC14. Сигнал с выхода 6 микросхемы DD3.3 поступает на вход внешнего прерывания P3.2/INT0 микроконтроллера. Программа-обработчик прерывания должна определить, какая из входных линий перешла в низкий уровень. Если, например, входными сигналами

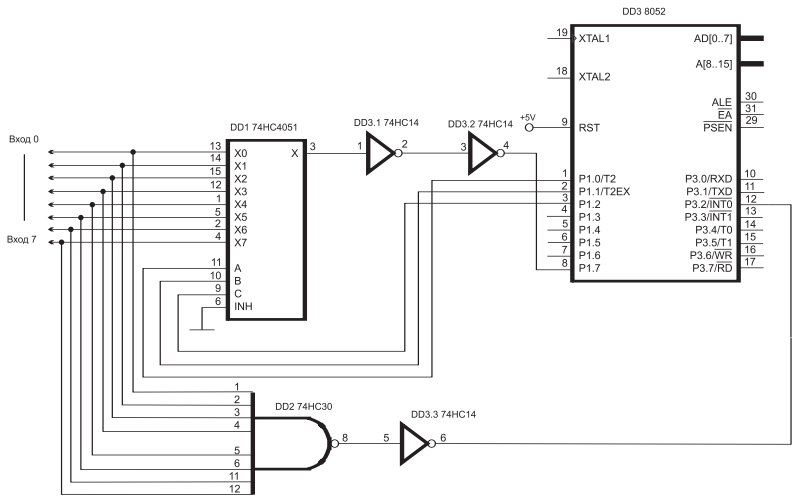
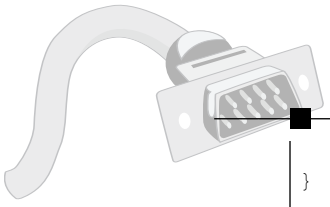


Рис. 5.15.
Схема обработки мультиплексированных данных по прерыванию

являются сигналы механических датчиков, то нужно предусмотреть защиту от дребезга (аппаратную или программную в самом обработчике прерывания). Замечу, что если сигналы на выходах изменяются очень быстро, то аппаратную часть нужно изменить так, чтобы можно было их зафиксировать. Это вызвано особенностями работы микроконтроллера в режиме прерывания, для фиксации которого требуется определенное время. Если это время сопоставимо со скоростью изменения сигналов, нужно использовать иные аппаратные решения, например, применить мультиплексоры с защелками данных и т.д. В данной схеме устройство, вызвавшее перепад сигнала 1–0, должно самостоятельно снимать его (устанавливать в 1), чтобы не блокировать работу остальных источников прерываний.

Программное обеспечение проекта разработано в Keil C51 и представлено следующим исходным текстом:

```
#include <stdio.h>
#include <intrins.h>
#include <REG52.H>

sbit A0 = P1^0;
sbit A1 = P1^1;
sbit A2 = P1^2;
sbit Input = P1^7;
bit bRunning = 1;

unsigned int dat;
unsigned int cnt;

void EX0Isr(void) interrupt 0 using 1{
dat = 0;
P1 = 0x0F8;
for (cnt = 0; cnt < 8; cnt++)
{
    dat |= Input;
    dat = _iror_(dat, 1);
```




```
    P1++;
}
    dat = dat>>8;
    bRunning = 0;
}

void main(void)
{
P1 = 0x0F8;
SCON = 0x50;
TH1 = 0xFD;
TMOD |= 0x20;
TR1 = 1;
TI = 1;

IT0 = 1;
EX0 = 1;
EA = 1;

while(1)
{
if (bRunning == 0)
{
    printf("INPUT Value Total = %Xh\n", dat);
    bRunning = 1;
    }
}
}
```

В этой программе считывание данных осуществляется в обработчике внешнего прерывания 0. Установка прерывания 0 выполняется операторами

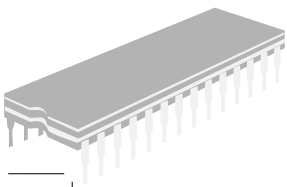
```
IT0 = 1;
EX0 = 1;
EA = 1;
```

Здесь бит `IT0` устанавливается в `1` для того, чтобы обеспечить вызов прерывания `EX0` по перепаду 1–0 на выводе P3.2/INT0, а не по уровню.

Мультиплексирование входных данных позволяет решить проблему недостаточного количества входных сигнальных линий, но это не единственный способ. Можно использовать преобразование параллельного кода входных сигналов в последовательный при помощи различных модификаций интерфейса SPI. Чаще всего для этого используют регистры сдвига.

Следующий проект демонстрирует эту возможность. На рис. 5.16 представлена аппаратная реализация обработки входных сигналов через преобразование параллельного кода в последовательный.

Для считывания входных данных в этой схеме используется регистр сдвига на микросхеме 74HC166, позволяющий преобразовать параллельный код на входах 0–7 в последовательный на выводе SO. Последовательные данные на этом выходе считываются микроконтроллером и преобразуются программой в параллельный формат. Используемый метод частично замедляет реакцию на входные сигналы по сравнению с параллельным считыванием данных в порт



микроконтроллера по 8 линиям, но при современной элементной базе для большинства приложений это работает вполне прилично.

Как работает схема? Данные с входов D0 – D7 загружаются в параллельном формате в регистр DD1 по низкому уровню сигнала на выводе PE микросхемы. Затем по фронту синхроимпульса, поступающего на вход CLK, данные побитово сдвигаются на выход SO, где считываются микроконтроллером.

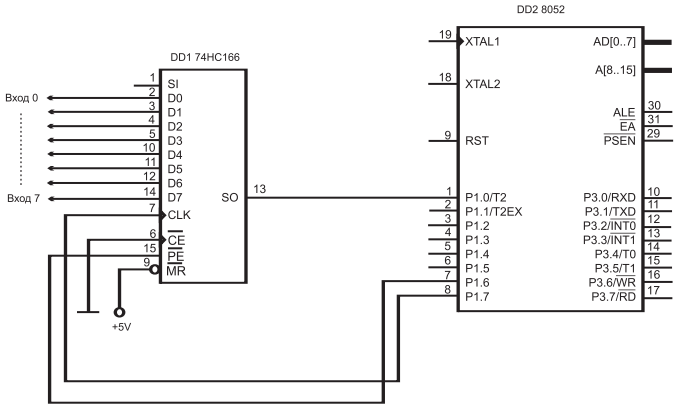
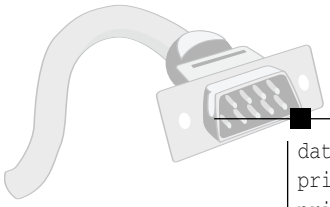


Рис. 5.16. *Преобразование параллельных входных данных в последовательный код*

Вместо микросхемы 74HC166 можно применить аналогичные чипы, при этом необходимо учитывать быстродействие конкретной микросхемы и использовать, где нужно, программные задержки.

Далее приводится программа на Keil C51, выводящая в последовательный порт считанные данные. Считывание данных по линии SO осуществляется при помощи вспомогательной процедуры, написанной на ассемблере и сохраненной в отдельном файле с расширением .asm. Исходный текст основной программы:

```
#include <stdio.h>
#include <REG52.H>

sbit SO = P1^0;
sbit CLK = P1^7;
sbit PE = P1^6;

unsigned int dat;

extern unsigned int shift_in8(void);
void main(void)
{
P1 = 0x3F;
SCON = 0x50;
TH1 = 0xFD;
TMOD |= 0x20;
TR1 = 1;
TI = 1;

while(1)
{
```

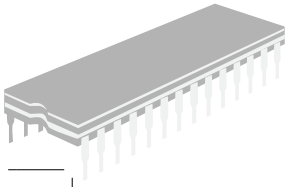


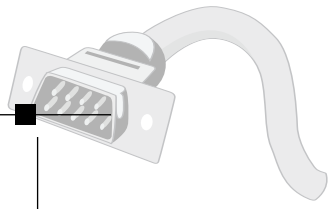

```
dat = shift_in8();
printf("Bin. value = %X\n", dat);
printf("Press Enter to get next value...\n");
getchar();
}
}
```

Вот содержимое ассемблерного модуля:

```
    NAME      PROCS
    PUBLIC        shift_in8
    SO        EQU P1.0
    CLK       EQU P1.7
    PE        EQU P1.6
    PROG      SEGMENT CODE
    RSEG      PROG
shift_in8:
    MOV       P1, #3Fh
    MOV       R4, #7
    CLR       A
    CLR       PE
    NOP
    CLR       CLK
    NOP
    NOP
    NOP
    SETB      CLK
    NOP
    SETB      PE
    MOV       C, SO
    RLC       A
    NOP
next_bit_in:
    CLR       CLK
    NOP
    NOP
    NOP
    SETB      CLK
    MOV       C, SO
    RLC       A
    DJNZ      R4, next_bit_in
    MOV       R7, A
    MOV       R6, #0
    RET
    END
```

В процедуре `shift_in8` 8-разрядный код с линии SO помещается в регистр аккумулятора. Перед началом цикла побитового считывания данных содержимое входов заносится в параллельном формате в регистр с помощью команд





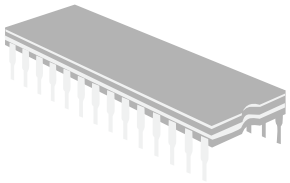
```
CLR     PE
NOP
CLR     CLK
NOP
NOP
NOP
SETB    CLK
NOP
SETB    PE
```

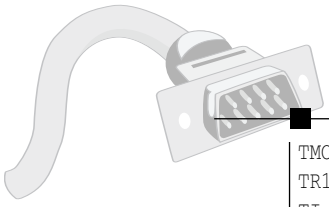
Команды NOP являются опциональными и могут вообще не понадобиться, если используемая микросхема является достаточно быстродействующей.

Считывание данных выполняется стандартным образом через флаг переноса и команды сдвига. Возвращаемый процедурой результат является 16-разрядным, при этом старший байт помещается в регистр R6 (в данном случае он равен 0), а младший в регистр R7.

Считывание входных данных можно сделать периодическим, используя, например, прерывание таймера 2. Вот исходный текст основной программы, в которой анализ входных данных осуществляется каждые 5 с:

```
#include <stdio.h>
#include <REG52.H>

sbit SO = P1^0;
sbit CLK = P1^7;
sbit PE = P1^6;

bit bRunning = 1;

unsigned int dat;
unsigned int interval = 0;

extern unsigned int shift_in8(void);

void T2Isr (void) interrupt 5 using 1 {
TF2 = 0;
interval++;
if (interval > 165)
{
interval = 0;
dat = shift_in8();
bRunning = 0;
}
}

void main(void)
{
SCON = 0x50;
PCON |= 0x80;
TH1 = 0xF3;
```






```
TMOD |= 0x20;
TR1 = 1;
TI = 1;

T2CON &= 0xFC;
RCAP2H = 0x0;
RCAP2L = 0x0;

EA = 1;
ET2 = 1;
TR2 = 1;

while(1)
{
if (bRunning == 0)
{
    printf("Bin. value = %X\n", dat);
    bRunning = 1;
}
}
}
```

При расчете интервала опроса входных линий, равно и остальных временных зависимостей, следует учитывать, как и во всех предыдущих случаях, тактовую частоту, на которой работает микроконтроллер (в данном примере частота кристалла равна 24,0 МГц).

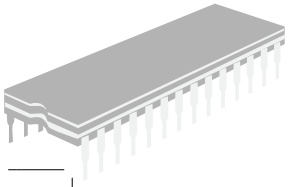
На этом анализ обработки входных дискретных сигналов мы закончим. Как видно из приведенных примеров, комбинируя различные аппаратно-программные решения, можно добиться весьма высокой эффективности обработки поступающих в микроконтроллер данных. Перейдем к анализу возможностей 8051-совместимых систем по выводу данных на внешние устройства и начнем с дискретных данных.

5.3. Пользовательские интерфейсы вывода дискретных данных

Для генерации выходных сигналов микроконтроллеры серии 8051, как известно, имеют несколько портов ввода/вывода. Для многих проектов стандартного количества выходных линий обычно бывает недостаточно, поэтому разработчикам приходится использовать схемотехнические методы для расширения количества линий вывода с соответствующей доработкой программного обеспечения.

Во многих случаях для расширения количества выходных линий используются однокристальные устройства так называемых расширителей интерфейса, которые работают по протоколу I²C либо SPI. В целом ряде случаев разработчик может обойтись и без таких расширителей, использовав довольно простое аппаратное решение (особенно если нужно добавить 5–6 дополнительных портов вывода), которое базируется на применении регистров сдвига, преобразующих последовательный код микроконтроллера в параллельный.

При этом передача последовательного кода осуществляется обычно с использованием модификации протокола SPI. В практическом плане такие расширители интерфейса очень просты и требуют минимума компонентов.



Рассмотрим простейший вариант расширения интерфейса вывода, принципиальная схема которого показана на рис. 5.17.

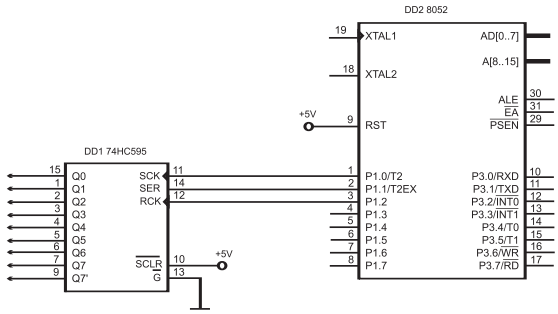


Рис. 5.17. *Схема расширителя интерфейса с использованием протокола SPI*

Регистр сдвига 74HC595 (микросхема DD1 на схеме) управляется с помощью трех сигнальных линий микроконтроллера P1.0 – P1.2. Последовательные данные передаются в регистр на вход SER, где каждый бит записывается по фронту сигнала синхронизации SCK. После передачи всех восьми бит они появляются на выходе регистра по нарастающему фронту сигнала RCK. Эта последовательность иллюстрируется временной диаграммой, показанной на рис. 5.18.

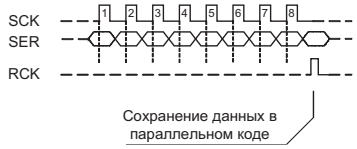
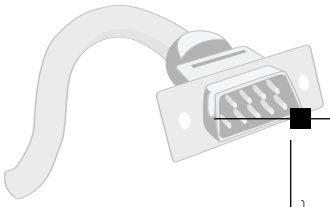


Рис. 5.18. *Временная диаграмма работы интерфейса с 74HC595*

Для схемы на рис. 5.17 разработана программа, выполняющая последовательный вывод логической «1» на выводы Q0 – Q7 регистра, причем смена данных осуществляется каждую секунду по прерыванию таймера 2, который по окончании вывода блокирует свою работу.

Программное обеспечение состоит из основной программы, написанной на Keil C51, и процедур, написанных на языке ассемблера и сохраненных в отдельном файле с расширением .asm. Вот исходный текст основной программы:

```
#include <stdio.h>
#include <REG52.H>

extern void t2init(void);
extern void shift_out8(unsigned char c1);

unsigned int cnt = 0;
unsigned char sVal;

void T2Isr (void) interrupt 5 using 1 {
TF2 = 0;
cnt++;
if (cnt > 33)
    {
    if (sVal > 128)
        TR2 = 0;
    shift_out8(sVal);
    cnt = 0;
```

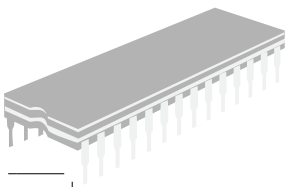


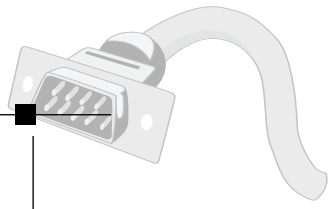

```
    sVal = sVal<<1;
    }
}

void main(void)
{

t2init();
sVal = 0x1;
while (1);
}
```

Как видно из листинга, исходный текст программы очень прост и затруднений не вызывает. Обработчик прерывания таймера 2 каждую секунду выводит байт данных в регистр сдвига с помощью внешней процедуры `shift_out8`, написанной на ассемблере, после чего сдвигает содержимое исходного байта данных, находящегося в переменной `sVal`, на один бит влево. Переменная `cnt` содержит счетчик перезагрузок таймера 2, соответствующий интервалу времени в 1 с.

Основная работа по передаче данных по последовательной линии выполняется в ассемблерной процедуре `shift_out8`, исходный текст которой приведен ниже:

```
    NAME      PROCS
    PUBLIC    t2init, _shift_out8
    T2CON     EQU 0C8h
    RCAP2H    EQU 0CBh
    RCAP2L    EQU 0CAh
    SCK       EQU P1.0
    SER       EQU P1.1
    RCK       EQU P1.2
    PROG      SEGMENT    CODE
    RSEG      PROG
t2init:
    CLR       T2CON.0
    CLR       T2CON.1
    MOV       RCAP2H, #0h
    MOV       RCAP2L, #0h
    SETB      IE.5
    SETB      EA
    SETB      T2CON.2
    RET
;-------------
_shift_out8:
    MOV       P1, #0F8h
    MOV       A, R7
    CLR       SCK
    CLR       RCK
    MOV       R4, #8
next_bit_out:
    RLC       A
    MOV       SER, C
    SETB      SCK
    CLR       SCK
```





```
    DJNZ    R4, next_bit_out
    SETB    RCK
    RET
;-------------------
    END
```

Обратите внимание, что процедура `_shift_out8` имеет символ подчеркивания в начале имени, что свидетельствует о том, что она принимает параметры из основной программы на С и должна подчиняться соглашениям, принятым для компилятора Keil C. Возможно, для других компиляторов С подобных требований не существует.

Процедура `t2init` выполняет инициализацию таймера 2, устанавливая для него режим автоперезагрузки. При указанных значениях в регистрах RCAP2H и RCAP2L и тактовой частоте 24,0 МГц таймер 2 будет перегружаться приблизительно каждые 0,03 с, поэтому для посекундного вывода данных счетчик перезагрузок должен содержать значение `33` (переменная `cnt` в основной программе). Кроме того, в этой процедуре разрешаются прерывания таймера 2.

Процедура `_shift_out8` осуществляет передачу 8-разрядного двоичного кода, передаваемого в регистре `R7`, по последовательной линии `SER`. Такая передача осуществляется путем помещения каждого двоичного бита из аккумулятора `A` в флаг переноса `C`, откуда бит передается в линию `SER` по нарастающему фронту сигнала синхронизации `SCK`.

Один из способов расширения интерфейса вывода может быть реализован, если использовать режим 0 последовательного порта микроконтроллера. В этом режиме вывод RxD используется как последовательная линия передачи, а на линию TxD подается последовательность синхронизирующих импульсов. Иными словами, последовательный порт в режиме 0 работает как 8-разрядный регистр сдвига. При этом биты данных последовательно выдвигаются из регистра SBUF и последовательно передаются с каждым синхроимпульсом на линии TxD.

Начало передачи инициируется при помещении в регистр SBUF байта данных, а об окончании передачи данных свидетельствует установка бита TI в состояние логической единицы. Синхронизирующая последовательность подается на линию TxD с частотой, равной 1/12 тактовой частоты микроконтроллера.

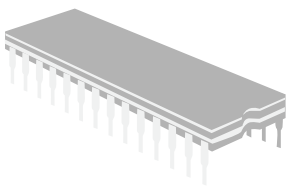
Так, например, для тактовой частоты 12 МГц частота передачи данных в режиме 0 будет равна 1 МГц. Этот режим очень удобен, поскольку упрощает программный код, необходимый для реализации преобразования последовательного кода в параллельный, но требует, чтобы устройство, принимающее данные, могло работать с такой частотой. Подавляющее большинство современных чипов, включая серию 74HC, легко справляется с такой задачей. Следует учитывать, что в этом режиме младший бит байта данных передается в линию первым, а старший – последним.

Рассмотрим практический пример использования режима 0 последовательного порта для организации расширения интерфейса. Для этого слегка модифицируем показанную на рис. 5.17 принципиальную схему устройства так, как на рис. 5.19.

В этой схеме вход приема данных SER регистра DD1 соединен с выводом RXD микроконтроллера, вход синхронизации SSK соединен с выходом TXD, а вход записи параллельных данных управляется посредством бита 0 порта P1 микроконтроллера.

Для иллюстрации принципа преобразования данных в таком интерфейсе разработана простая программа на языке ассемблера, исходный текст которой приведен ниже:

```
    NAME    PROCS
    RCK     EQU P1.0
    MAIN    SEGMENT CODE
    CSEG    AT 0
    USING   0
```



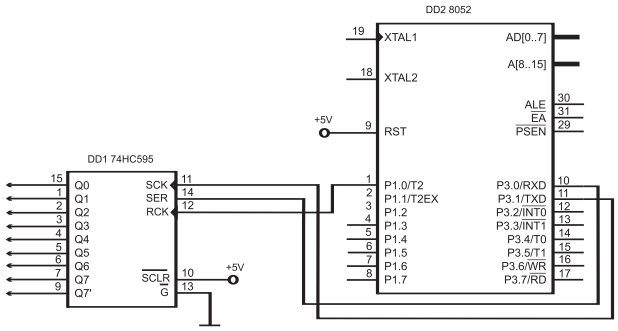


Рис. 5.19. *Расширение интерфейса с использованием последовательного порта*

```
        JMP  start
    RSEG     MAIN
start:
    MOV      P1, #0FEh
    MOV      SCON, #0h
    MOV      A, #37h
    CLR      TI
    CLR      RCK
    MOV      SBUF, A
    JNB      TI, $
    SETB     RCK
    END
```

Здесь мы используем бит 0 порта P1 как выходной для команды `RCK`, поэтому порт `P1` инициализируется соответствующим образом командой `MOV`.

Для работы последовательного порта в режиме 0 в регистр `SCON` следует поместить нулевое значение, что и выполняет команда

```
MOV      SCON, #0h
```

Затем в регистр-аккумулятор помещается значение `37h` (выбрано произвольно) и сбрасывается флаг `TI` окончания записи данных. Кроме того, блокируем параллельную запись в регистр DD1 до окончания передачи байта данных. Передача байта инициируется автоматически при помещении данных в регистр `SBUF` командой

```
MOV      SBUF, A
```

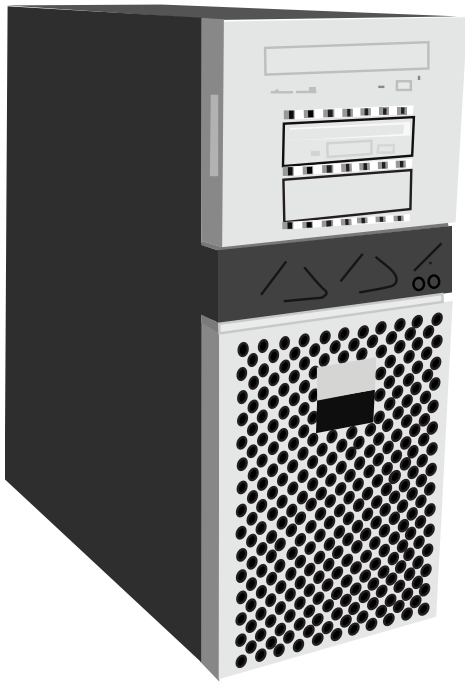
Далее программа ожидает установки бита `TI`, что будет свидетельствовать об окончании передачи данных:

```
JNB      TI, $
```

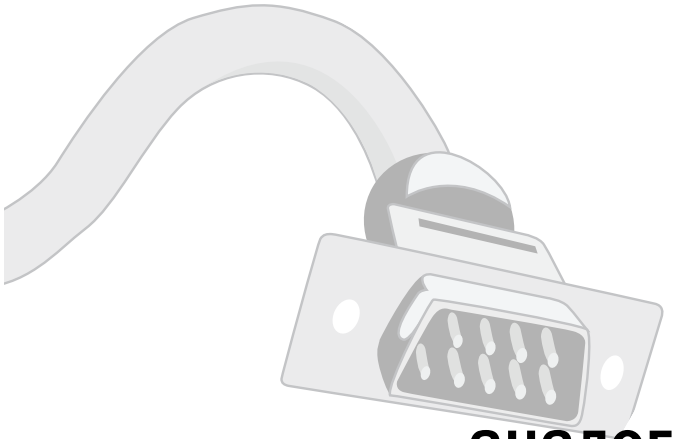
Наконец, выполняется запись и вывод данных в параллельной форме в регистре DD1 по команде

```
SETB     RCK
```

Как видите, программная реализация такого расширителя интерфейса достаточно проста; напомню лишь, что данные в регистре сдвига будут представлены битами, размещенными в обратном порядке по отношению к исходным данным.



Ввод/вывод аналоговых сигналов



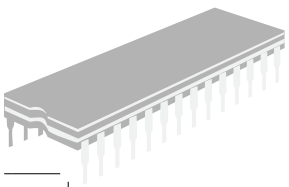
6 Ввод/вывод аналоговых сигналов

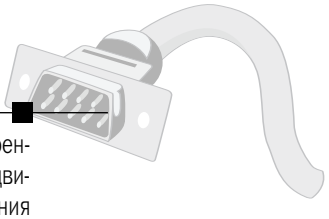
Микроконтроллеры по своей сути являются устройствами обработки цифровых данных. Тем не менее большинство задач научно-производственной деятельности связано с обработкой непрерывных или, по-другому, аналоговых сигналов. Первые микроконтроллеры серии 8051 не поддерживали аппаратно реализацию алгоритмов обработки аналоговых сигналов, но с развитием элементной базы и применением новых схемотехнических и архитектурных решений появились новые поколения микроконтроллеров, совместимых с 8051, позволяющие выполнять обработку аналоговых сигналов. В таких микроконтроллерах имеются встроенные аппаратные модули, реализующие функции преобразования непрерывных сигналов в цифровую форму представления (аналого-цифровое преобразование), а также преобразующие цифровые сигналы в непрерывные (цифро-аналоговое преобразование).

Обработка поступающих в микроконтроллер аналоговых сигналов осуществляется при помощи специальных устройств, называемых аналого-цифровыми преобразователями, или сокращенно АЦП.

Многие современные кристаллы 8051-совместимых устройств имеют встроенные АЦП, позволяющие минимизировать усилия, требуемые для обработки аналоговых входных сигналов устройств, и, как следствие, уменьшить общее время разработки проектов. Как правило, АЦП на кристалле микроконтроллера выполняются многоканальными, чаще всего 8-канальными. Подобные аппаратные решения имеют как свои преимущества, так и недостатки. К преимуществам следует отнести относительную легкость программирования встроенных аналого-цифровых преобразователей, поскольку разработчику не требуется знать особенности алгоритма преобразования. Для программирования встроенных АЦП достаточно выполнить несколько операций записи в специальные регистры, после чего спустя определенное время считать результат из других регистров или памяти. Детали преобразования при этом скрыты от программиста.

Недостатком встроенных преобразователей является то, что они жестко завязаны с архитектурой кристалла, что ограничивает гибкость настройки параметров для таких АЦП. Кроме того, устройства АЦП, выполненные на отдельном кристалле, обычно превосходят встроенные аналоги как по разрядности, так и по характеристикам преобразования. Современные АЦП, реализованные в виде отдельных устройств, работают со значительно более высоким быстродействием, чем встроенные аналоги, и имеют повышенную разрядность по сравнению с последними. В любом случае, решение о применении встроенного или отдельного АЦП должно приниматься в зависимости от поставленной задачи.





Для генерации выходных аналоговых сигналов многие микроконтроллеры имеют встроенные модули цифро-аналоговых преобразователей (ЦАП), позволяющие управлять электродвигателями постоянного тока или создавать, например, управляемые источники напряжения и тока. Цифро-аналоговое преобразование широко применяется и при создании генераторов сигналов, управляемых напряжением. Опять-таки, однокристальные цифро-аналоговые преобразователи обладают более высокими точностными параметрами по сравнению со встроенными в микроконтроллеры аналогами, имея к тому же более высокую разрядность и температурную стабильность.

В данной главе мы рассмотрим практические аспекты работы с аналого-цифровыми и цифро-аналоговыми преобразователями, представляющими собой отдельные устройства и подключаемые к портам ввода/вывода микроконтроллера 8051. Изучение методики работы с такими устройствами окажет неоценимую помощь и при работе со встроенными АЦП и ЦАП, позволив избежать ошибок при их программировании.

6.1. Обработка аналоговых входных сигналов

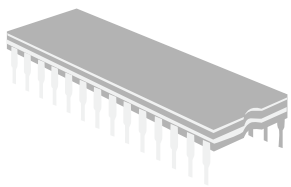
Кардинальные отличия между различными типами аналого-цифровых преобразователей состоят в способе обработки входного сигнала. Аналоговые входные сигналы могут обрабатываться по параллельной схеме, по методу последовательных приближений, с использованием сигма-дельта алгоритма и т.д. Мы рассмотрим применение АЦП, реализованных по алгоритму последовательных приближений, поскольку в настоящее время они наиболее широко используются в промышленности. Преимуществами этих преобразователей являются достаточно высокая разрядность (разрешение 12, 14 или 16 бит), приемлемая скорость преобразования (десятки и сотни киловыборок в секунду), а также невысокая цена и низкое энергопотребление. Данный тип АЦП чаще всего используется в разнообразных измерительных приборах и в системах сбора данных. В настоящий момент АЦП последовательного приближения позволяют измерять напряжение с точностью до 16 разрядов с частотой дискретизации более 100 киловыборок/с.

Рассмотрим несколько практических примеров. Предположим, что нужно выполнить измерение входного аналогового сигнала с использованием микросхемы аналого-цифрового преобразователя на микросхеме MCP3201 фирмы Microchip Technology Inc.

Микросхема MCP3201 представляет собой одноканальный 12-разрядный аналого-цифровой преобразователь последовательного приближения. Функционирование микросхемы включает два этапа: собственно преобразование аналоговой величины в двоичный код и передачу двоичных данных по интерфейсу внешнему устройству, совместимому с SPI.

Чтобы лучше понять логику работы преобразователя, рассмотрим его функциональную схему (рис. 6.1).

Входной сигнал поступает на устройство выборки и хранения (Sample-and-Hold) и заряжает конденсатор, величина заряда на котором будет пропорциональна величине входного напряжения. Работа устройства выборки и хранения синхронизируется устройством управления (Control Logic), которое запускает цикл выборки по перепаду 1–0 синхронизирующего сигнала CS/SHDN. Цифро-аналоговый преобразователь (DAC) и регистр последовательных приближений выполняют цикл сравнения напряжения на конденсаторе устройства выборки и хранения с выходным напряжением цифро-аналогового преобразователя при помощи компаратора напряжения. При превышении напряжения DAC над напряжением на конденсаторе алгоритм преобразования останавливается, а двоичное значение кода, соответствующего



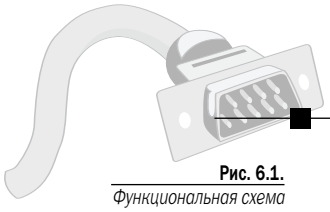
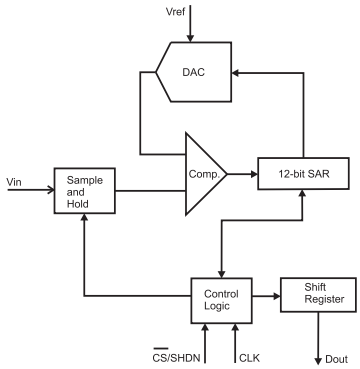


Рис. 6.1.
Функциональная схема преобразователя MCP3201



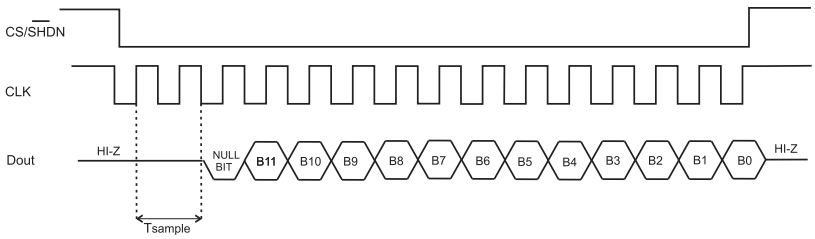
входному напряжению, фиксируется в регистре последовательных приближений. После этого двоичный код можно считывать во внешнее устройство, например микроконтроллер.

Каждый бит данных Dout считывается по фронту или спаду синхронизирующего сигнала CLK (для большинства АЦП по фронту). Обычно по спаду CLK бит данных помещается в регистр сдвига, а по фронту его можно считывать. Цикл обработки входного сигнала завершается при установке сигнала CS/SHDN в высокий уровень.

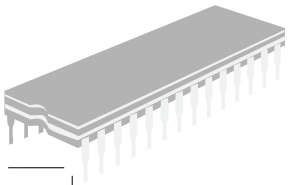
Следует отметить, что рассмотренная нами логика работы аналого-цифрового преобразователя, реализованного по принципу последовательных приближений, является типичной для этого класса устройств. На рынке присутствует много микросхем АЦП подобного типа, но принцип их работы аналогичен рассмотренному. Более того, их временные диаграммы и даже назначение внешних выводов весьма схожи, поэтому, изучив принцип функционирования и программирование для конкретной микросхемы (в данном случае MCP3201), очень легко можно использовать устройства другого производителя.

Временная диаграмма функционирования микросхемы MCP3201 показана на рис. 6.2.

Рис. 6.2.
Временная диаграмма работы MCP3201



Рассмотрим подробно временную диаграмму. Цикл преобразования начинается при установке сигнала CS/SHDN в низкий уровень. Изначально CS/SHDN должен находиться в высоком уровне, чтобы четко фиксировать начало преобразования. После установки низкого уровня CS/SHDN выполняется преобразование аналогового входного сигнала, что требует двух стробов CLK. Далее линия Dout переходит из высокоимпедансного состояния в низкое (нулевой бит) по фронту третьего синхроимпульса CLK. Интересующие нас данные выставляются на линию Dout, начиная с четвертого синхроимпульса CLK, причем первым идет старший значащий бит (MSB, Most Significant Bit). Каждый бит данных на линии Dout можно фиксировать в микроконтроллере по нарастающему фронту сигнала CLK. Цикл преобразования и считывания данных заканчивается при установке сигнала CS/SHDN в высокий уровень. Таким образом, для преобразования требуется 15 синхроимпульсов CLK.



Разработаем аппаратно-программный проект, в котором необходимо считать значение внешнего аналогового сигнала, находящегося в диапазоне 0–5 В, и вывести его значение через последовательный порт на терминальное устройство или дисплей персонального компьютера. Для этого следует подсоединить ПК через последовательный порт к микроконтроллеру, затем запустить на ПК любую программу-терминал, работающую с последовательным портом, и выполнить программу, записанную в микроконтроллере.

Принципиальная схема нашего устройства изображена на рис. 6.3.

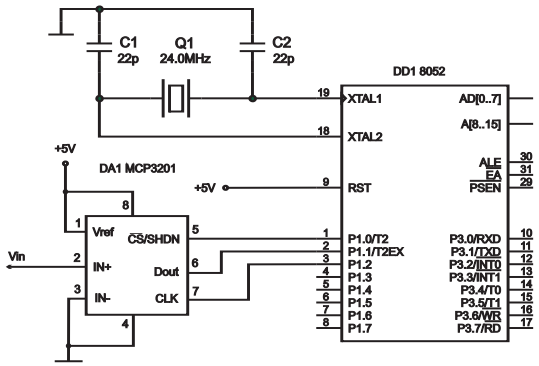


Рис. 6.3.
Схема обработки аналогового сигнала в микроконтроллере

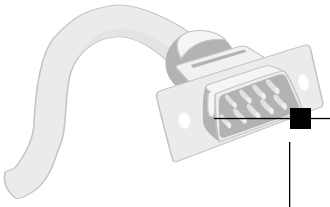
Наш АЦП обозначен как DA1 и подсоединен к микроконтроллеру следующим образом:

- вход разрешения преобразования CS/SHDN соединяется с выводом P1.0 микроконтроллера;
- вход синхронизации CLK соединяется с выводом P1.2 микроконтроллера;
- выход данных Dout соединяется с выводом P1.1 микроконтроллера.

Программную часть проекта разработаем в Keil uVision3. Наша программа должна считывать входной сигнал V_{in} на входе АЦП и отображать его значение на экране приблизительно каждые 10 с. Обратите внимание, что при разработке проекта использовался отладочный модуль с тактовой частотой 24,0 МГц. При использовании других отладочных модулей вам необходимо будет пересчитать временные характеристики таймера 2.

Для фиксации 10-секундных временных интервалов в состав программного проекта входит файл t2init.asm, написанный на ассемблере. Данный файл содержит исходный текст программы для инициализации таймера 2. Вот его содержимое:

```
    NAME     T2_INIT
    PUBLIC   t2init
    T2CON    EQU 0C8h
    RCAP2H   EQU 0CBh
    RCAP2L   EQU 0CAh
    PROG     SEGMENT CODE
    RSEG     PROG
t2init:
    CLR      T2CON.0
    CLR      T2CON.1
    MOV      RCAP2H, #0h
    MOV      RCAP2L, #0h
```

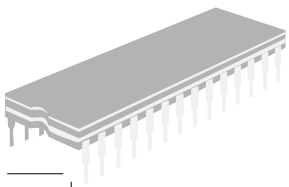


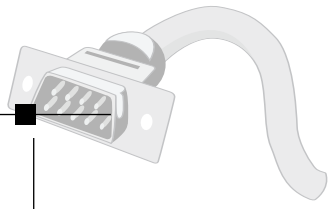

```
    SETB    T2CON.2
    SETB    IE.5
    SETB    EA
    RET
    END
```

Здесь таймер 2 устанавливается в режим автоперезагрузки, которая при нулевых установках в регистрах RCAP2H и RCAP2L и тактовой частоте 24,0 МГц будет происходить каждые 0,03 с. Кроме того, факт перезагрузки будет фиксироваться в обработчике прерывания таймера 2 (его мы рассмотрим далее).

Основная программа написана на языке С (файл mcp3201.c), и ее исходный текст представлен ниже:

```
#include <stdio.h>
#include <REG52.H>

sbit CS_SHDN = P1^0;
sbit DOUT = P1^1;
sbit CLK = P1^2;

unsigned int bRunning;
int cnt;

void delay(void)
{
unsigned char del = 10;
while(del != 0)del–;
}

float getConv(void)
{
    int countOfBits = 15;
    unsigned int binResult = 0;

    P1 = 0x0FA;
    CS_SHDN = 1;
    CLK = 1;
    delay();
    CS_SHDN = 0;

    do {
        CLK = 0;
        delay();
        CLK = 1;
        binResult = binResult << 1;
        binResult |= DOUT;
        countOfBits–;
        } while (countOfBits > 0);
    CS_SHDN = 1;
    binResult &= 0x0FFF;
```

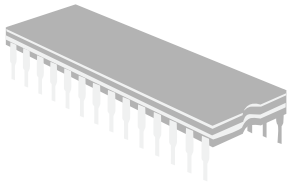


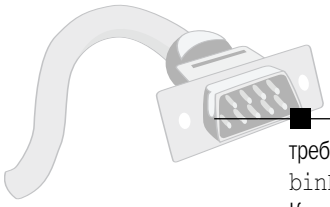


```
    return 4.78 / 4096 * binResult;
}

void T2ISR (void) interrupt 5 using 1 {
TF2 = 0;
cnt—;
if (cnt == 0)
{
cnt = 330;
bRunning = 1;
}
}

extern void t2init(void);

void main(void)
{
float total;
char buf[32];
int bytes;

bRunning = 0;

SCON = 0x50;
TH1 = 0xF3;
PCON |= 0x80;
TMOD |= 0x20;
TR1 = 1;
TI = 1;

cnt = 330;
t2init();
while (1)
{
if (bRunning == 1)
    {
    total = getConv();
    bytes = sprintf(buf, "%s", "Total result: ");
    bytes += sprintf(buf + bytes, "%5.3f\n", total);
    printf(buf);
    bRunning = 0;
    }
}
}
```

Собственно процесс преобразования аналогового сигнала в цифровую форму выполняется в программе `getConv`. Мы остановимся очень детально на анализе этой процедуры, поскольку она является ключевой для работы всей программы.

Вначале об используемых в данной процедуре переменных. Переменная `countOfBits` содержит счетчик синхроимпульсов `CLK`. Напомню, что для полного цикла преобразования





требуется 15 стробов, из них значащими для программы являются последние 12. Переменная `binResult` содержит текущее значение получаемых после каждого синхроимпульса данных. Кроме того, для удобства работы определим сигнальные биты CS/SHDN, CLK и Dout следующим образом:

```
sbit CS_SHDN = P1^0;
sbit DOUT = P1^1;
sbit CLK = P1^2;
```

В начале выполнения процедуры биты P1.0 – P1.2 должны быть установлены соответствующим образом (биты P1.0 и P1.2 работают как выходы, а бит P1.1 – как вход). Это выполняется при помощи оператора

```
P1 = 0x0FA;
```

Далее, перед началом каждого цикла преобразования следует соответствующим образом устанавливать сигнал CS/SHDN (сначала высокий, затем низкий уровень), что выполняют операторы

```
CS_SHDN = 1;
CLK = 1;
delay();
CS_SHDN = 0;
```

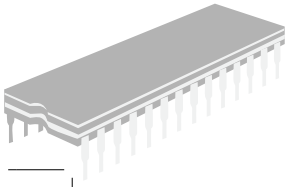
После установки сигнала CS/SHDN в высокий уровень можно сделать небольшую задержку, используя процедуру `delay`, хотя это и необязательно. Сигнал CLK также нужно установить в высокий уровень, причем сделать это нужно до того, как CS/SHDN примет значение низкого уровня.

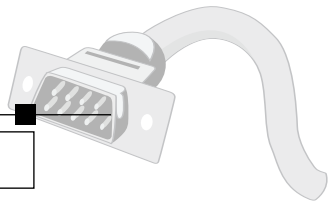
Инициировав начало преобразования, переходим к побитовому считыванию данных при нарастающем фронте сигнала `CLK`. Это реализовано в цикле `do`:

```
do {
    CLK = 0;
    delay();
    CLK = 1;
    binResult = binResult << 1;
    binResult |= DOUT;
    countOfBits–;
} while (countOfBits > 0);
```

Каждая итерация цикла начинается установкой сигнала `CLK` в низкий уровень, как это требуется в соответствии с временной диаграммой. Затем, спустя некоторое время `CLK` устанавливается в высокий уровень. После этого бит данных можно считывать в переменную `binResult`. Поскольку данные считываются от старшего бита к младшему, то в каждой итерации необходимо выполнять логический сдвиг битов, находящихся в `binResult`, на одну позицию влево. Затем выполняется операция «логическое ИЛИ» над содержимым `binResult`, в результате чего самый младший бит этой переменной будет равен `DOUT`.

Напомню, что в счетчик итераций `countOfBits` для удобства мы поместили значение `15`, поскольку при обработке окончательного результата, сохраненного в переменной `binResult`, лишние биты данных будут отсечены посредством оператора





```
binResult &= 0x0FFF;
```

По окончании цикла `do ... while` программа должна остановить процесс преобразования, установив линию CS/SHDN в высокий уровень, что выполняет оператор

```
CS_SHDN = 1;
```

Последний оператор процедуры `getConv` возвращает полученный результат, предварительно преобразовав его в число с плавающей точкой:

```
return 5.00 / 4096 * binResult;
```

Здесь нужно обратить внимание на один важный момент. Еще раз посмотрим на схему устройства, показанную на рис. 6.3. На вывод V_{ref} (1) микросхемы MCP3201 подается внешнее напряжение смещения от прецизионного источника опорного напряжения. В данном случае в качестве такого источника мы используем источник питания схемы, который, естественно, должен обладать высокой стабильностью. Напряжение V_{ref} определяет шаг разрядной сетки, т.е. значение младшего значащего бита в вольтах. Например, при V_{ref} = 5 В шаг преобразования равен 5/4096 = 0,00122, а значение входного напряжения преобразователя в этом случае получается как произведение шага преобразования на двоичное значение.

Значение 4096 определяется разрядностью преобразователя, которая в данном случае равна 12, отсюда и значение 4096 (2^{12}). Если, например, вывод V_{ref} подключить к источнику опорного напряжения, равного 4,096 В, то шаг преобразователя станет равным 1 мВ, т.е. точность преобразования станет выше, но одновременно диапазон входных напряжений будет ограничен верхним значением 4,096 В.

Это все, что касается работы процедуры `getConv`.

Исходный текст основной программы вряд ли вызовет затруднения, поэтому остановимся на нем вкратце. Вначале инициализируется таймер 1, который используется в качестве генератора синхронизации при последовательном обмене. Установки таймера 1 соответствуют скорости передачи данных 9600 бод при тактовой частоте 24,0 МГц. Далее программа выполняет цикл `while`, в котором анализирует значение переменной `bRunning`. Если обработчик прерывания таймера 2 устанавливает данную переменную в 1, то выполняется преобразование входного аналогового сигнала, после чего данные передаются в последовательный порт.

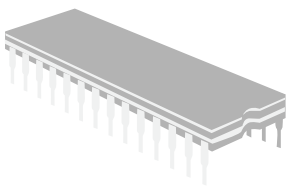
Рассмотренное нами программное обеспечение проекта – далеко не единственный вариант реализации алгоритма аналого-цифрового преобразования. Нередко для увеличения быстродействия работы программ и уменьшения объема используемой памяти имеет смысл написать большую часть программы на ассемблере.

Рассмотрим еще один пример обработки аналогового входного сигнала, причем выберем другую популярную микросхему – LTC1292 фирмы Linear Technology.

Временная диаграмма работы этой микросхемы показана на рис. 6.4.

Как видно из этой временной диаграммы, время преобразования аналогового сигнала занимает два синхроимпульса, а суммарное количество импульсов, необходимое для считывания данных, равно 14. Схема подключения микросхемы LTC1292 к микроконтроллеру в упрощенном виде показана на рис. 6.5.

В качестве программного обеспечения для этого преобразователя можно использовать модифицированный вариант программы из предыдущего примера, причем без ассемблерного кода.



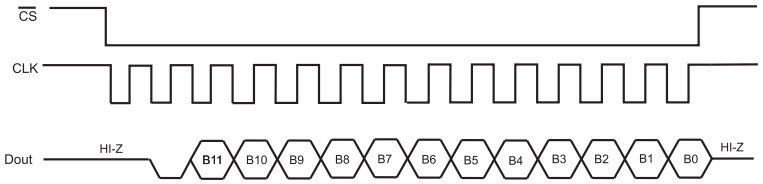


Рис. 6.4. *Временная диаграмма работы микросхемы LTC1292*

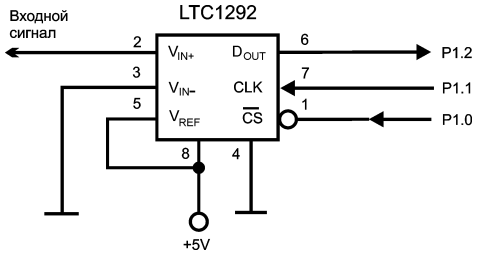
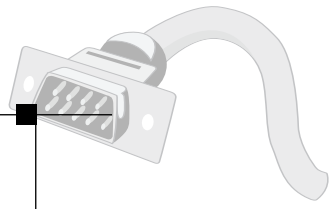


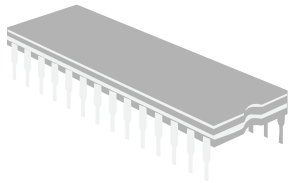
Рис. 6.5. *Подключение LTC1292 к микроконтроллеру*

Вот исходный текст программы обработки входного аналогового сигнала микросхемой LTC1292 на «чистом» C (интервал измерения для разнообразия здесь выбран равным 5 с при тактовой частоте 11,059МГц):

```
#include <stdio.h>
#include <REG52.H>

sbit CS = P1^0;
sbit DOUT = P1^2;
sbit CLK = P1^1;

unsigned int bRunning;
int cnt;

void delay(void)
{
unsigned char del = 10;
while(del != 0)del–;
}

float getConv(void)
{
    int countOfBits = 14;
    unsigned int binResult = 0;

    P1 = 0x0FC;
    CS = 1;
    delay();
    CS = 0;

    do {
         CLK = 0;
         delay();
         CLK = 1;
```



```
        binResult = binResult << 1;
        binResult |= DOUT;
        countOfBits—;
    } while (countOfBits > 0);
CS = 1;
binResult &= 0x0FFF;
return 5.00 / 4096 * binResult;
}

void T2ISR (void) interrupt 5 using 1 {
TF2 = 0;
cnt—;
if (cnt == 0)
{
cnt = 165;
bRunning = 1;
}
}

void main(void)
{
float total;
char buf[32];
int bytes;

bRunning = 0;

SCON = 0x50;
TH1 = 0xF3;
PCON |= 0x80;
TMOD |= 0x20;
TR1 = 1;
TI = 1;
cnt = 165;

ET2 = 1;
EA = 1;
T2CON &= 0x0FD;
RCAP2H = 0x0;
RCAP2L = 0x0;
T2CON |= 0x4;

while (1)
{
if (bRunning == 1)
{
    total = getConv();
    bytes = sprintf(buf, “%s“, “LTC1292 Total Result: “);
    bytes += sprintf(buf + bytes, “%5.3f\n“, total);
    printf(buf);
```

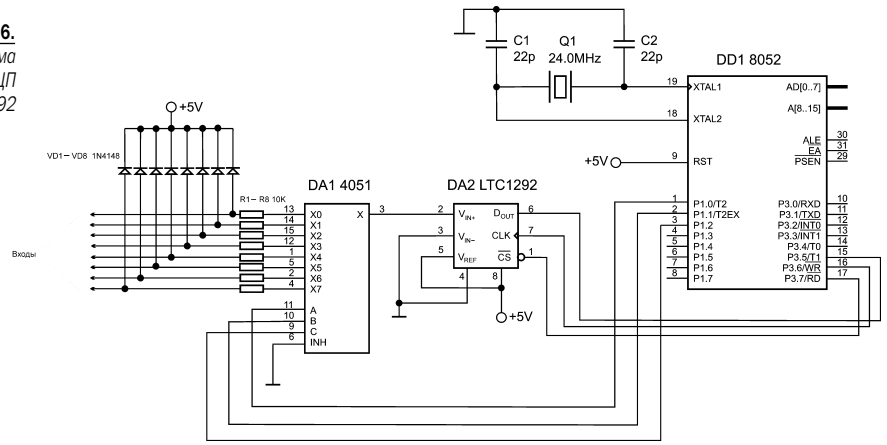


```
    bRunning = 0;
    }
}
}
```

В этой программе инициализация таймера 2 выполнена в основной программе. Кроме того, здесь изменены соединения выводов CS, CLK и Dout с микроконтроллером. Думаю, читатели без труда проанализируют исходный текст этой программы.

В практических разработках относительно редко приходится измерять аналоговые сигналы, приходящие от одного источника. Разработаем проект, который позволяет одновременно измерять аналоговые сигналы с 8 источников. Аппаратная часть этого проекта будет содержать микросхему LTC1292 и мультиплексор аналоговых сигналов 4051. Схема аппаратной части устройства представлена на рис. 6.6.

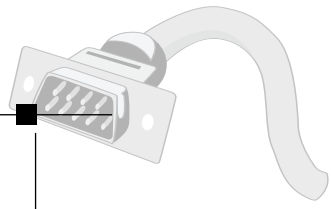
Рис. 6.6. *Схема 8-канального АЦП на LTC1292*



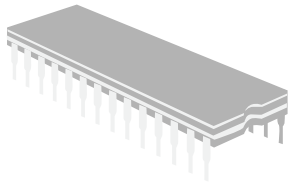
В этой схеме аналоговые входные сигналы подаются на 8 входов мультиплексора на микросхеме DA1. Микроконтроллер через определенные интервалы времени, задаваемые таймером 2, сканирует входы и после преобразования посылает данные на терминал. Для повышения точности преобразования можно уменьшить номиналы резисторов R1 – R8. На практике во многих случаях можно обойтись без ограничивающих цепей на резисторах и диодах, если заранее известно, что входные напряжения не превышают 5 В.

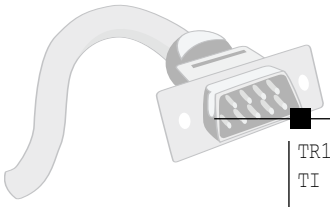
Программа сканирования написана на C, и ее исходный текст представлен ниже:

```
#include <stdio.h>
#include <REG52.H>

sbit CS = P3^7;
sbit DOUT = P3^5;
sbit CLK = P3^6;

unsigned int bRunning;
int cnt;

void delay(void)
{
```



```
unsigned char del = 10;
while(del != 0)del–;
}

float getConv(void)
{
    int countOfBits = 14;
    unsigned int binResult = 0;

    P3 &= 0x3F;
    CS = 1;
    delay();
    CS = 0;

    do {
         CLK = 0;
         delay();
         CLK = 1;
         binResult = binResult << 1;
         binResult |= DOUT;
         countOfBits–;
    } while (countOfBits > 0);
CS = 1;
binResult &= 0x0FFF;
return 5.00 / 4096 * binResult;
}

void T2ISR (void) interrupt 5 using 1 {
TF2 = 0;
cnt–;
if (cnt == 0)
{
cnt = 165;
bRunning = 1;
}
}

void main(void)
{
float total;
char idata buf[64];
int bytes;

int chNum;

bRunning = 0;

SCON = 0x50;
TH1 = 0xF3;
PCON |= 0x80;
TMOD |= 0x20;
```



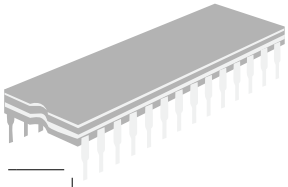


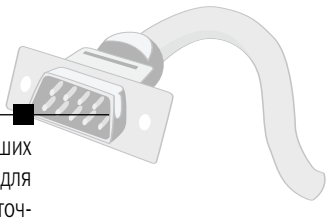

```
TR1 = 1;
TI = 1;

cnt = 165;

ET2 = 1;
EA = 1;
T2CON &= 0x0FD;
RCAP2H = 0x0;
RCAP2L = 0x0;
T2CON |= 0x4;

while (1)
{
if (bRunning == 1)
{
    for (chNum = 0; chNum < 8; chNum++)
    {
         P1 = chNum;
         total = getConv();
         bytes = sprintf(buf, "%s", "LTC1292 Channel ");
         bytes += sprintf(buf + bytes, "%d", chNum);
         bytes += sprintf(buf + bytes, "%s", " input: ");
         bytes += sprintf(buf + bytes, "%5.3f\n", total);
         printf(buf);
    }
    printf("----------\n\n");
    bRunning = 0;
}
}
}
```

Эта программа представляет собой модифицированный программный код из предыдущего проекта. Сигналы CS, CLK и Dout привязаны к битам порта P3, а адресация входов A, B, C (см. рис. 6.6) мультиплексора осуществляется через порт P1. Считывание данных с мультиплексора осуществляется в цикле `for` основной программы, в котором последовательно перебираются адреса 0–7 на линиях A, B и C. Переменная `chNum` в каждой итерации содержит адрес контролируемого входа. Оставшаяся часть программного кода была рассмотрена нами ранее, поэтому останавливаться на ней нет смысла.

Заканчивая анализ обработки аналоговых сигналов микроконтроллерами 8051, хочу остановиться на некоторых вопросах, касающихся функционирования аналого-цифровых преобразователей.

При разработке схем с аналого-цифровыми преобразователями особое внимание нужно уделить стабильности питающих напряжений схемы и развязке по питанию, поскольку именно эти факторы влияют на стабильность и точность преобразования. В наших проектах в качестве источника опорного напряжения использовалось питающее напряжение схемы, что, в принципе, допустимо для большинства проектов. Тем не менее, если вы хотите достичь очень высокой точности преобразования, ограниченной только погрешностями самого кристалла, то следует применить отдельный прецизионный источник опорного напряжения, желательно ведущих производителей, например Linear Technology, Analog Devices и др.





Другой важный момент касается нормализации входных сигналов. Опять таки, в наших проектах мы не учитывали выходное сопротивление и емкость источника сигнала, хотя для большей точности измерений, особенно малых сигналов, требуется применять промежуточные схемы буферизации, например, повторители сигналов на операционных усилителях.

Во многих случаях аналого-цифровые преобразователи используются для обработки сигналов от датчиков, в основном резистивного типа, в которых какой-либо контролируемый параметр связан с омическим сопротивлением материала определенной зависимостью. Примерами таких датчиков являются датчики температуры и давления, датчики ускорения и т.д. Изменение контролируемого параметра в таких датчиках приводит к изменениям сопротивления по определенному закону. В этих случаях для получения электрических сигналов используются преобразователи «сопротивление-ток» или «сопротивление-частота». Для выделения полезного сигнала при работе с такими датчиками применяют обычно мостовые схемы и дифференциальные усилители, в настоящее время реализованные как однокристальные измерительные усилители. В любом случае перед подачей сигнала на вход аналого-цифрового преобразователя нужно очень хорошо просчитать схемы предварительного усиления и нормализации сигналов. Как показывает практика, наиболее слабым звеном во всей цепочке преобразования аналогового сигнала является именно этап предварительной обработки.

Когда не следует применять аналого-цифровые преобразователи? Если вам нужно снимать сигнал с удаленного объекта, то непосредственная передача такого сигнала по линиям передачи приведет к заведомо отрицательным результатам. В таких случаях желательно использовать либо цифровые датчики, передающие сигнал в виде последовательности импульсов, которая в дальнейшем интерпретируется определенным образом, либо, если датчик является аналоговым, выполнить на месте преобразование «напряжение-частота» (V-to-F conversion) и передавать уже частотный сигнал.

В некоторых случаях можно сигнал от удаленного аналогового датчика преобразовать предварительно в ток, что обеспечит более высокое качество сигнала.

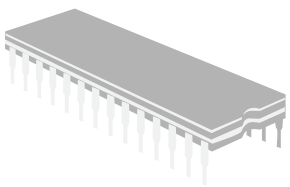
6.2. Использование цифро-аналоговых преобразователей

Цифро-аналоговые преобразователи используются для получения выходного напряжения непрерывной формы посредством подачи на их входы цифровых сигналов, управляющих по определенному закону величиной этого напряжения. К цифро-аналоговым преобразователям относят и устройства, выходной сигнал которых пропорционален частоте поступающих на их вход сигналов, – так называемые преобразователи «частота-напряжение» (F-to-V Converters), и цифровые потенциометры, эквивалентное сопротивление которых пропорционально подаваемому на их входы цифровому коду. Рассмотрим некоторые практические аспекты использования цифро-аналоговых преобразователей в системах на базе микроконтроллеров 8051.

Рассмотрим проект устройства, в котором выходной аналоговый сигнал генерируется при помощи микросхемы LTC1456, являющейся однокристальным ЦАП фирмы Linear Technology. Этот преобразователь управляется 12-разрядным двоичным кодом, поступающим через модифицированный вариант SPI интерфейса. Временная диаграмма работы микросхемы LTC1456 показана на рис. 6.7.

Выводы микросхемы имеют следующие назначения:

- CLK – строб синхронизации при приеме данных;
- Din – вывод, на который поступают данные. Данные на этом выводе запоминаются в регистре-защелке микросхемы по нарастающему фронту сигнала CLK;



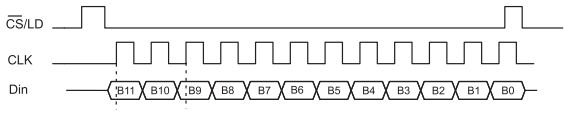


Рис. 6.7. *Временная диаграмма работы преобразователя LTC1456*

- CS/LD – сигнал разрешения/установки интерфейса. При низком уровне сигнала на входе разрешается прием данных по линии Din, при высоком уровне прием данных по линии Din запрещается, а принятые данные загружаются из регистра сдвига в регистр преобразования, изменяя значение выходного напряжения преобразователя.

Принципиальная схема цифро-аналогового преобразователя с управлением от микроконтроллера показана на рис. 6.8.

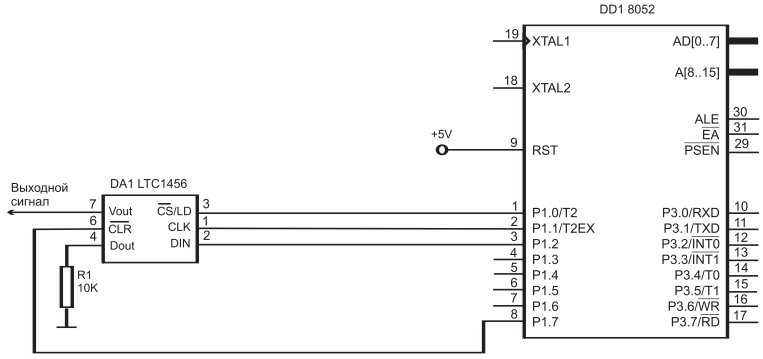
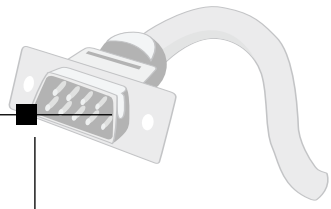


Рис. 6.8. *Генерация выходного аналогового сигнала с использованием LTC1456*

Программное обеспечение для проекта разработано с использованием Keil uVision3. Основная программа написана на C и позволяет ввести с клавиатуры терминала или ПК требуемое значение выходного напряжения. Собственно преобразование цифрового кода в аналоговое напряжение выполняет процедура `daconv`, написанная на ассемблере и содержащаяся в отдельном файле с расширением .asm.

Исходный текст основной программы приведен ниже:

```
#include <stdio.h>
#include <stdlib.h>
#include <REG52.H>

extern void daconv(unsigned int binData);

void main(void)
{
idata char buf[32];
float vout;
unsigned int i1;

SCON = 0x50;
TH1 = 0xFD;
TMOD |= 0x20;
TR1 = 1;
TI = 1;
do {
    printf("Enter OUTPUT Voltage: ");
```

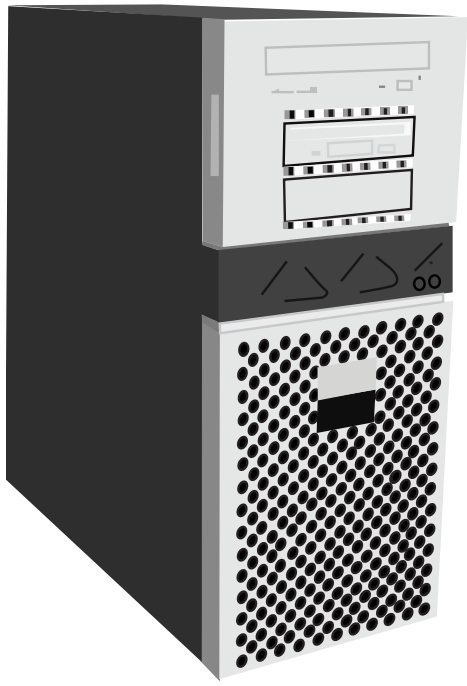


```
    gets(buf, sizeof(buf));
    vout = atof(buf) * 1000.0;
    i1 = vout;
    daconv(i1);
} while (buf [0] != '\0');
}
```

Содержимое ASM-файла приведено ниже:

```
    NAME CONVERTER
    PUBLIC   _daconv
    CS_LD    EQU P1.0
    CLK      EQU P1.1
    DIN      EQU P1.2
    CLEAR    EQU P1.7
    PROG     SEGMENT CODE
    RSEG     PROG
_daconv:
    MOV      P1, #7Ch
    CLR      CLEAR
    SETB     CLEAR
    CLR      CS_LD
    CLR      CLK

    MOV      R0, #4
    MOV      ACC, R6
    SWAP     A
high4:
    RLC      A
    MOV      DIN, C
    SETB     CLK
    CLR      CLK
    DJNZ     R0, high4

    MOV      R0, #8
    MOV      ACC, R7

low8:
    RLC      A
    MOV      DIN, C
    SETB     CLK
    CLR      CLK
    DJNZ     R0, low8
    SETB     CS_LD
    RET
    END
```

Процедура _daconv принимает 16-разрядный параметр, младший байт которого размещается в регистре R7, а старший в R6. Поскольку процедура принимает из основной программы параметр, ее имя должно начинаться с символа подчеркивания. Алгоритм работы процедуры, думаю, не вызовет затруднений при анализе, поэтому останавливаться на нем мы не будем.



Отображение информации в системах с микроконтроллерами 8051

7 Отображение информации в системах с микроконтроллерами 8051

Конечным результатом функционирования многих систем управления и контроля является визуальное отображение информации, позволяющее судить о состоянии текущих процессов в управляемой системе и принимать решения о тех или иных действиях. Такие системы управления и контроля должны отображать информацию в удобном для конечного пользователя формате, быть, как правило, компактными и потреблять минимум мощности от источников питания, что особенно важно для мобильных систем.

В настоящее время наибольшее распространение получили устройства отображения информации на базе полупроводниковых индикаторов (семисегментных и матричных), а также жидкокристаллические индикаторы. Особенно бурно развиваются технологии в сфере применения жидкокристаллических устройств – кроме обычных текстовых дисплеев к настоящему времени разработаны и используются системы отображения графической информации с настраиваемыми параметрами отображения.

Материал этой главы посвящен принципам разработки интерфейса семисегментных светодиодных индикаторов и текстовых жидкокристаллических дисплеев с системами на базе микроконтроллеров 8052.

7.1. Применение семисегментных индикаторов

Светодиодные полупроводниковые индикаторы в настоящее время продолжают очень широко использоваться в системах сбора и отображения информации. Наибольшее распространение среди этого класса устройств получили семисегментные индикаторы. Рассмотрим принципы управления выводом информации и методы построения интерфейсов с этим типом индикаторов. Но вначале немного теории.

Семисегментные светодиодные индикаторы могут подключаться к источнику питания по схеме с общим катодом или по схеме с общим анодом. Схема индикаторов обоих типов приведена на рис. 7.1.

Каждый элемент семисегментного индикатора представляет собой светодиод, на который от управляющей схемы может подаваться напряжение. Если все светодиоды соединяются своими катодами в общую точку, то говорят, что индикатор выполнен по схеме с общим катодом. Если аноды всех светодиодов соединены в общей точке, то индикатор выполнен по схеме с общим анодом. Промышленность выпускает индикаторы как с общим катодом, так и с общим анодом, поэтому перед подключением к схеме следует определить их тип. Для индикаторов

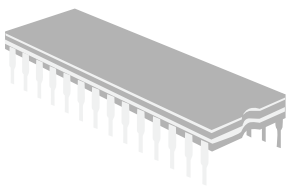
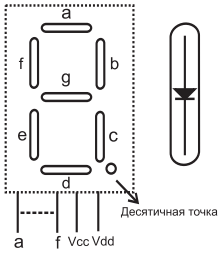


Рис. 7.1. *Схема и расположение выводов семисегментного индикатора*



с общим катодом засветка светодиодного элемента происходит при подаче на его анод положительного напряжения (высокий уровень), а для индикаторов с общим анодом – отрицательного на катод (низкий уровень). При использовании семисегментных индикаторов в цифровых схемах, совместимых с TTL-уровнями, элементы индикатора с общим катодом засвечиваются при подаче уровня логической 1 на анод, а элементы индикатора с общим анодом – уровня логического 0 на катод.

Двоичные комбинации, формирующие символы для индикаторов с общим катодом и общим анодом, естественно, будут различаться. Далее мы будем рассматривать разработку интерфейсов для семисегментных индикаторов обоих типов.

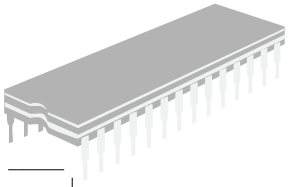
Двоичные комбинации, которые следует подавать на светодиодные элементы индикатора с общим катодом для получения представления десятичных цифр от 0 до 9, сведены в табл. 7.1.

Таблица 7.1. *Двоичные представления цифр для индикаторов с общим катодом*

Цифра	Двоичный код						
	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Как видно из таблицы, каждой цифре, отображаемой индикатором, соответствует определенная двоичная комбинация. Эту комбинацию можно сформировать как аппаратным способом (применив, например, микросхему CD4511), так и программным, используя для этого микроконтроллер. Можно комбинировать оба способа для экономии выводов 8051, подавая двоичный код цифры на входы микросхемы 4511 от микроконтроллера, а выходы 4511 подключив к соответствующим выводам индикатора. Какой вариант лучше, выбирать разработчику. Следует лишь учитывать то, что выходы микроконтроллера способны управлять очень незначительной нагрузкой, поэтому подключение элементов индикатора непосредственно к портам 8051 нежелательно. При выборе такого варианта нужно использовать либо буферные микросхемы с токовыми выходами, либо обычные биполярные транзисторы.

Рассмотрим практические варианты реализации вывода информации на семисегментные индикаторы. Вначале разработаем проект, в котором будет применена микросхема –



преобразователь кода CD4511. Принципиальная схема аппаратной части такого устройства показана на рис. 7.2.

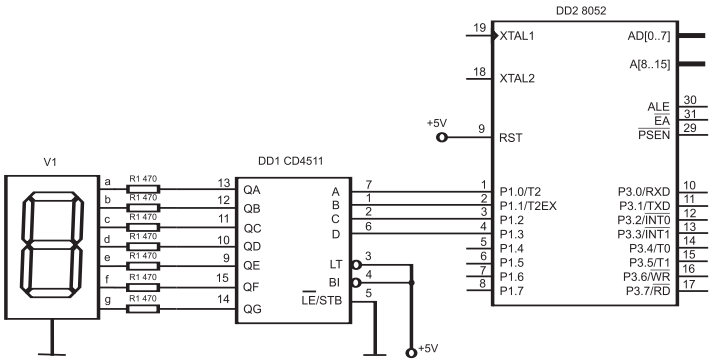


Рис. 7.2. *Интерфейс микроконтроллера с индикатором на микросхеме 4511*

В этой схеме двоичный код числа с выходов P1.0 – P1.3 микроконтроллера поступает на входы A – D преобразователя DD1. На выходах преобразователя появляется двоичный код управления сегментами индикатора V1, включенного по схеме с общим катодом. Тестовая программа написана на Keil C и работает так: каждые 2 с инкрементируется содержимое внутреннего счетчика, которое выводится на индикатор V1. По достижении значения 9 счет начинается снова, и т.д. Исходный текст программы приведен ниже:

```
#include <stdio.h>
#include <REG52.H>

int cnt = 0;
unsigned char c1 = 0;

void T0Isr (void) interrupt 1 using 0 {
cnt++;
if (cnt > 7200)
{
    cnt = 0;
    P1 = c1;
    if (c1++ > 9) c1 = 0x0;
    }
}

void main(void)
{

P1 = 0xF0;
ET0 = 1;
EA = 1;

TH0 = 0;
TL0 = 0;
TMOD |= 0x2;
TR0 = 1;

while(1);
}
```

Периодическое изменение цифр на индикаторе выполняется программой-обработчиком прерывания таймера 0. Таймер 0 настроен на работу в режиме автоперезагрузки. Для тактовой частоты 11,059 МГц частота возникновения прерывания таймера 0 равна 3600 Гц, т.е. для интервала в 2 с счетчик перезагрузок таймера 0 `cnt` должен принимать максимальное значение 7200. При превышении этого значения в порт `P1` выводится двоичное значение переменной `c1`, изменяющейся в диапазоне 0–9.

Можно обойтись и без преобразователя DD1, подключив индикатор к выводам порта P1 микроконтроллера. В этом случае программа должна сама формировать двоичный код для элементов индикатора. Создадим небольшой проект, в котором вывод на семисегментный индикатор выполняется непосредственно с выводов порта P1 микроконтроллера. Для разнообразия выберем в качестве индикатора семисегментный светодиод с общим анодом.

Принципиальная электрическая схема устройства показана на рис. 7.3.

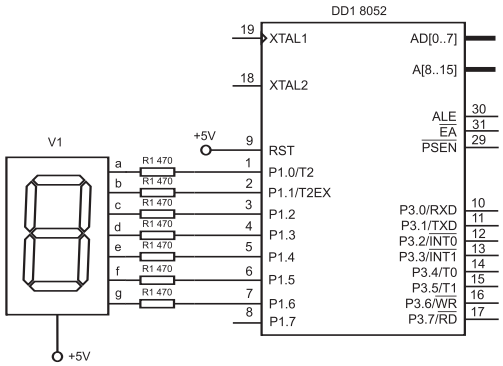
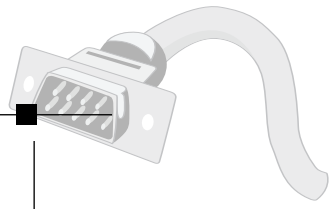


Рис. 7.3. *Прямое подключение семисегментного индикатора к 8051*

В этой схеме выводы элементов индикации подключены непосредственно к выводам порта P1 (желательно подключить порт P1 через буферные усилители во избежание перегрева и разрушения кристалла микроконтроллера во время длительной работы из-за повышенных выходных токов через соответствующие выводы!). Тестовая программа, написанная на C, непрерывно выводит цифры в обратном порядке на индикатор V1 с интервалом приблизительно в 2 с. Вот исходный текст:

```
#include <stdio.h>
#include <REG52.H>

#define ESC 0x1B

unsigned int cnt;
unsigned char codeTable[11] = { ESC, 0x0C0, 0x0F9, 0x0A4, 0x0B0, 0x99,
0x92, 0x82, 0x0F8, 0x80, 0x98};
unsigned char *pcodeTable;

void T0Isr(void) interrupt 1 using 1 {
cnt++;
if (cnt > 30)
{
    cnt = 0;
    P1 = *pcodeTable—;
    if (*pcodeTable == ESC)
```



```
    pcodeTable = codeTable+10;
    }
}

void main(void)
{
pcodeTable = codeTable+10;
cnt = 0;
P1 = 0x80;
P1 = *pcodeTable;

TH0 = 0x0;
TL0 = 0x0;
TMOD |= 0x1;
TR0 = 1;

ET0 = 1;
EA = 1;

while(1);
}
```

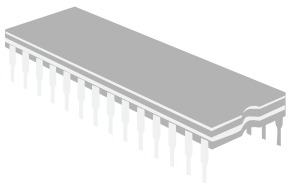
В этой программе для вывода нужной цифры на индикатор используется кодовая таблица `codeTable`, представляющая собой массив из 11 байт. Первым элементом таблицы (с индексом 0) является символ ESC, который используется в программе для фиксации окончания прохода по таблице. Остальные элементы таблицы представляют собой кодовые комбинации цифр для семисегментного индикатора с общим анодом. Вывод очередной цифры на индикатор осуществляется в обработчике прерывания таймера 0, который перезагружается приблизительно 14,1 раз в секунду, что при максимальном значении счетчика `cnt`, равном `30`, дает приблизительный интервал в 2 с. Цифры выводятся на индикатор в обратном порядке, от 9 к 0, а при достижении 0 счет опять начинается с 9. Читатели, знакомые со стандартным C, думаю, поймут смысл операций с указателями, используемыми в программе, поэтому останавливаться на этом мы не будем.

Зная, как работает одиночный семисегментный индикатор, можно разрабатывать аппаратно-программные конфигурации с несколькими индикаторами, позволяющими отображать как целочисленные значения, так и значения с плавающей точкой. Кроме одиночных промышленностью выпускаются матрицы, состоящие из нескольких семисегментных индикаторов, для управления которыми используются встроенные аппаратные контроллеры, что значительно упрощает жизнь разработчику.

7.2. Применение жидкокристаллических индикаторов

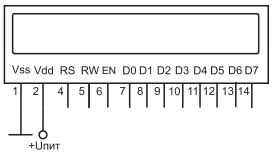
Многие встроенные системы в качестве устройства отображения информации используют жидкокристаллические дисплеи (Liquid Crystal Display, LCD). Наиболее часто используют дисплеи формата 16×2 и 20×2, которые позволяют вывести максимум по 16 и 20 символов в любой из двух строк.

Большинство выпускаемых в настоящее время жидкокристаллических индикаторов включает кроме самой матрицы стандартный интерфейс управления, основанный на применении



контроллера, совместимого с HD44780, который является стандартом де-факто для данного класса устройств отображения информации. Типичное обозначение LCD на схемах показано на рис. 7.4.

Рис. 7.4. *Схема расположения выводов жидкокристаллического индикатора*



Выводы устройства имеют следующие функциональные назначения:

- D0 – D7 – линии данных. По этим линиям в дисплей передаются команды и данные, а также считывается статус текущей операции (бит D7). Контроллер 44780 требует для управления 3 линии и для вывода данных и приема команд – 8 линий. Контроллер позволяет работать либо с 8 линиями данных, когда используются все сигналы на линиях D0 – D7, либо с 4;
- EN – расшифровывается как «Enable». Это линия управления, фиксирующая начало/окончание передачи команды/данных в дисплей или приема данных из устройства. Линия должна быть установлена в высокий уровень перед отправкой дисплею команды или данных, после чего она сбрасывается в низкий уровень. После установки EN в высокий уровень должны быть установлены соответствующим образом линии RW, RS и D0 – D7. Установка линии EN в низкий уровень завершает выполнение команды или посылки/приема данных. Если сигнал EN не сброшен в низкий уровень после отправки команды/данных, то дисплей никогда не сможет выполнить посланную ему команду/данные;
- RS (Register Select) устанавливается в 0, если данные на линиях D0 – D7 должны интерпретироваться как команда. Если RS устанавливается в 0, то данные на линиях D0 – D7 должны отображаться на экране;
- RW (Read/Write), установленная в низкий уровень, означает запись. Если RW = 1, то данные читаются с LCD. Единственный случай, когда нужно читать данные, – при выполнении команды Get LCD status.

Таким образом, последовательность операций при посылке команды или данных на дисплей следующая:

- установить сигнал на линии EN в высокий уровень;
- установить линию RS либо в 0 (если посылается команда), либо в 1 (данные);
- выставить на линиях D0 – D7 код команды или данных;
- установить сигнал на линии EN в низкий уровень.
- дождаться сброса бита D7, который свидетельствует о готовности дисплея принимать следующую команду/данные.

Типичная схема соединения микроконтроллера 8051/8052 с таким дисплеем показана на рис. 7.5.

По этой схеме выводы данных D0 – D7 жидкокристаллического индикатора LCD1 соединены с выводами порта P1, а линии сигналов управления RS, RW и EN – с выводами P3.5 – P3.7 микроконтроллера. Посмотрим, как на практике реализовать управление выводом информации на дисплей, для чего воспользуемся демонстрационной программой, написанной на ассемблере.

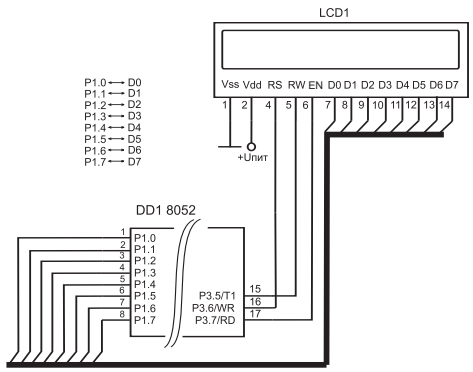
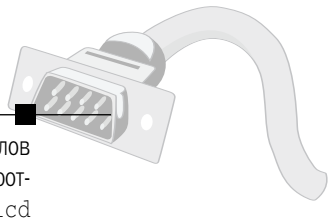


Рис. 7.5.
Интерфейс 8051 с жидкокристаллическим индикатором

Исходный текст программы, отображающей текстовую строку на экран дисплея, показан ниже:

```
    NAME      LCD_TEST
    DAT       EQU P1
    RW        EQU P3.5
    RS        EQU P3.6
    EN        EQU P3.7
    PROGRAM   SEGMENT CODE
    MYDATA    SEGMENT CODE

    CSEG      AT 0
    USING     0
    JMP  start

    RSEG      PROGRAM
start:
    CALL      init_lcd
    CALL      clear_lcd
    MOV       DPTR, #txt

next:
    CLR       A
    MOVC      A, @A+DPTR
    CJNE      A, #0dh, cont
    JMP       $
cont:
    CALL      write_char
    INC       DPTR
    JMP       next
;-----------------
wait_lcd:
    SETB      EN
    CLR       RS
    SETB      RW
    MOV       DAT, #0FFh
    MOV       A, DAT
```

```
    CLR     EN
    JB      ACC.7, wait_lcd
    CLR     RW
    RET
;-----------------
init_lcd:
    SETB    EN
    CLR     RS
    CLR     RW
    MOV     DAT, #38h
    CLR     EN
    CALL    wait_lcd

    SETB    EN
    CLR     RS
    CLR     RW
    MOV     DAT, #0Eh
    CLR     EN
    CALL    wait_lcd

    SETB    EN
    CLR     RS
    CLR     RW
    MOV     DAT, #06h
    CLR     EN
    CALL    wait_lcd
    RET
;-------------
clear_lcd:
    SETB    EN
    CLR     RS
    CLR     RW
    MOV     DAT, #01h
    CLR     EN
    CALL    wait_lcd
    RET
;-------------
write_char:
    SETB    EN
    SETB    RS
    CLR     RW
    MOV     DAT, A
    CLR     EN
    CALL    wait_lcd
    RET
;-------------
    RSEG    MYDATA
    txt: DB "Text on LCD", 0dh
    END
```



Проанализируем исходный текст программы более подробно. Перед выводом символов на дисплей необходимо выполнить инициализацию и очистку дисплея, послав устройству соответствующие команды. Инициализацию и очистку дисплея выполняют процедуры `init_lcd` и `clear_lcd`, а запись байта на дисплей – процедура `write_char`. Кроме того, после операции записи (команды или данных) обязательно следует проверить состояние дисплея, прочитав бит D7. При занятом устройстве бит 7 будет установлен, поэтому нужно дождаться, пока этот бит станет равным 0, чтобы послать следующую команду.

Как ранее упоминалось, передача команд или данных на жидкокристаллический дисплей происходит в строго определенной последовательности, в которой сигналы передаются устройству. Рассмотрим это на примере процедуры инициализации дисплея `init_lcd`.

Вначале линия `EN` устанавливается в состояние логической 1 командой

```
SETB    EN
```

Если передается команда (что и делается в данном случае), то на линии `RS` нужно установить низкий уровень сигнала командой

```
CLR RS
```

Для записи данных линия `RS` должна быть установлена в высокий уровень. Далее следует указать, что выполняется операция записи, очистив линию `RW` с помощью команды

```
CLR RW
```

Далее следует код команды для дисплея, который выставляется на линии D0 – D7 по команде

```
MOV DAT, #38h
```

Здесь код команды для дисплея равен 0x38. Наконец, последняя команда, которую нужно выполнить, – очистить линию `EN`:

```
CLR EN
```

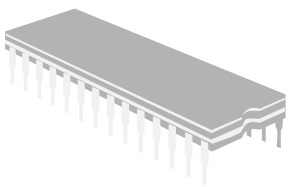
После этой ассемблерной команды следует прочитать статус операции, выполняемой дисплеем, с помощью процедуры

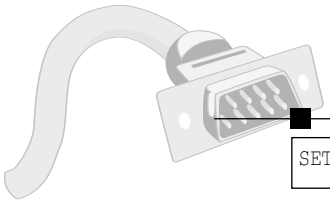
```
CALL    wait_lcd
```

Процедура `wait_lcd` выполняет команду с кодом 0xFF, означающую чтение статуса дисплея, а затем проверяет бит, установленный в разряде D7. Процедура `wait_lcd` должна выполняться после любой посылки команды/данных на дисплей.

Вернемся к `init_lcd`. После посылки на дисплей команды с кодом 0x38 необходимо отправить еще две команды с кодами 0x0E и 0x06, которые заканчивают процесс инициализации.

Процедура `clear_lcd` посылает дисплею команду очистки (код 0x00). После этого можно записывать байты данных на дисплей с помощью процедуры `write_char`. Здесь, как обычно, первая команда устанавливает линию EN в высокий уровень, затем команда






```
SETB RS
```

устанавливает линию RS в высокий уровень, указывая тем самым, что на линиях D0 – D7 будут присутствовать данные. Команды ассемблера

```
CLR RW
MOV DAT, A
```

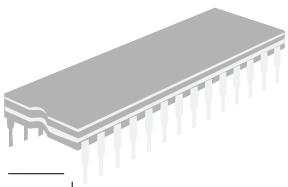
сбрасывают линию RW (первая команда), что свидетельствует о выполнении операции записи, а также помещают байт данных из регистра-аккумулятора на линии DAT. Для того чтобы дисплей выполнил команду, за указанными командами следует сброс линии EN:

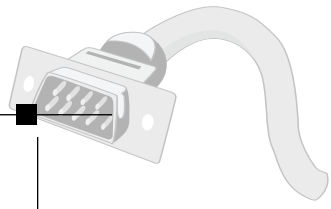
```
CLR EN
```

Для вывода строки символов функция write_char вызывается в цикле по метке next. При этом байт данных передается функции в регистре A из области программной памяти txt, на которую указывает регистр DPTR. Как видно из листинга программы, управление LCD не представляет собой слишком сложную задачу, требуется лишь четко следовать правилам обмена по интерфейсу 44780.

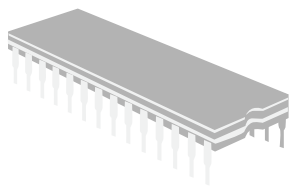
Более удобно для управления жидкокристаллическим дисплеем использовать язык C. Ниже приводится исходный текст программы, написанной на C, которая выводит на устройство LCD значение числа π:

```
#include <stdio.h>
#include <REG52.H>

sbit RW = P3^5;
sbit RS = P3^6;
sbit EN = P3^7;
sbit BUSY = P1^7;

unsigned char STATE;

void wait_lcd (void)
{
    do {
         EN = 1;
         RS = 0;
         RW = 1;
         P1 = 0xFF;
         STATE = BUSY;
         EN = 0;
    } while (STATE != 0);
RW = 0;
}

void init_lcd(void)
{
    EN = 1;
```





```
    RS = 0;
    RW = 0;
    P1 = 0x38;
    EN = 0;
    wait_lcd();
    EN = 1;
    RS = 0;
    RW = 0;
    P1 = 0x0E;
    EN = 0;
    wait_lcd();

    EN = 1;
    RS = 0;
    RW = 0;
    P1 = 0x06;
    EN = 0;
    wait_lcd();
}

void clear_lcd(void)
{
    EN = 1;
    RS = 0;
    RW = 0;
    P1 = 0x01;
    EN = 0;
    wait_lcd();
}

void write_char(unsigned char *c1)
{
    EN = 1;
    RS = 1;
    RW = 0;
    P1 = *c1;
    EN = 0;
    wait_lcd();
}

void main(void)
{
idata char buf[64];
char *pbuf = buf;

float f1 = 3.141592;

int bytes, cnt;
bytes = sprintf(buf, "%s", "PI ==  ");
bytes += sprintf(buf + bytes, "%8.6f\n", f1);
```



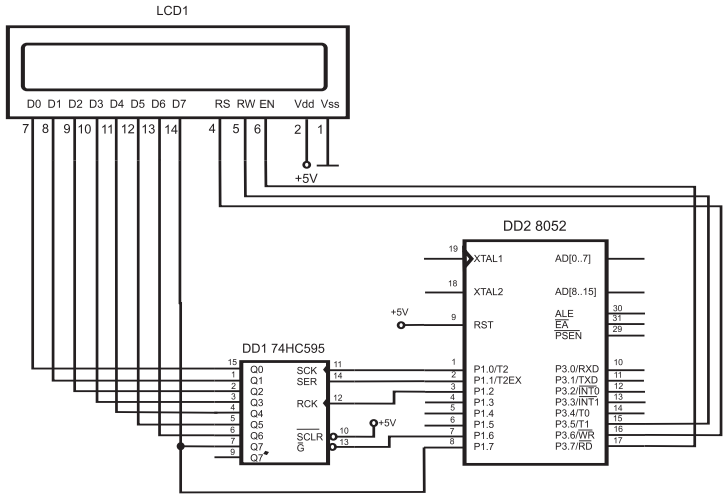
```
init_lcd();
clear_lcd();

for (cnt = 0; cnt < bytes; cnt++)
    write_char(pbuf++);
while(1);
}
```

В этой программе все вспомогательные функции реализованы с помощью операторов языка С. Программа выводит данные на индикатор в соответствии с рассмотренным в предыдущем листинге алгоритмом.

Как видно из принципиальной схемы, показанной на рис. 7.5, для управления жидкокристаллическим индикатором требуется как минимум 11 линий, что, в общем случае, очень накладно, учитывая ограниченное число линий ввода/вывода микроконтроллера. В некоторых случаях используют 4 линии данных для вывода информации, посылая на устройство последовательно две 4-битовые посылки, что требует усложнения программного алгоритма. Мы рассмотрим здесь другой метод, с использованием интерфейса SPI, в котором задействованы 3 линии для интерфейса с микроконтроллером. При этом получаем выигрыш в 5 линий, и, кроме того, данные на индикатор передаются по всем 8 линиям D0 – D7.

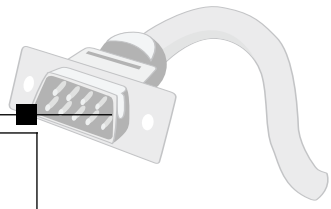
Для реализации такого метода разработана аппаратно-программная конфигурация, в которой для реализации SPI-совместимого интерфейса использован регистр сдвига 74HC595. Принципиальная схема аппаратной части проекта показана на рис. 7.6.

Рис. 7.6. *Схема SPI-интерфейса с жидкокристаллическим индикатором*

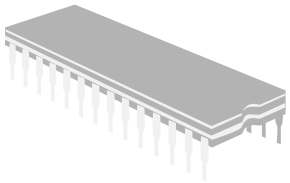


По этой схеме данные поступают по линии SER регистра сдвига DD1 74HC595 синхронно с тактовыми импульсами SCK, генерируемыми на выводе P1.0 микроконтроллера. По фронту сигнала RCK данные в параллельном формате выводятся на линии D0 – D7 индикатора. Вывод Q7 регистра DD1 соединен с линией D7 для считывания бита состояния дисплея в микроконтроллер через вывод P1.7. Управляющие сигналы RW, RS и EN подаются с выводов P3.5 – P3.7 микроконтроллера.

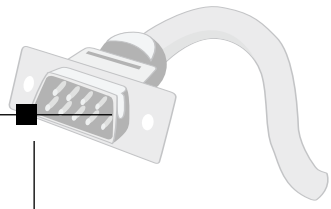
Исходный текст тестовой программы на ассемблере, демонстрирующей работу этого интерфейса, приведен ниже:



```
    NAME      LCD_TEST
    DAT       EQU P1
    RW        EQU P3.5
    RS        EQU P3.6
    EN        EQU P3.7

    SCK       EQU P1.0
    SER       EQU P1.1
    RCK       EQU P1.2
    OE        EQU P1.6
    PROGRAM   SEGMENT CODE
    MYDATA    SEGMENT CODE

    CSEG AT   0
    USING     0
    JMP       start

    RSEG      PROGRAM
start:
    MOV       P1, #0B8h
    CLR       P1.6
    CALL      init_lcd
    CALL      clear_lcd
    MOV       DPTR, #txt
next:
    CLR       A
    MOVC      A, @A+DPTR
    CJNE      A, #0dh, cont
    JMP       $
cont:
    CALL      write_char
    INC       DPTR
    JMP       next
;-----------------
wait_lcd:
    SETB      P1.6
still_wait:
    SETB      EN
    CLR       RS
    SETB      RW
    MOV       DAT, #0FFh
    MOV       C, P1.7
    CLR       EN
    JC        still_wait
    CLR       RW
    CLR       1.6
    RET
;-----------------
init_lcd:
```



```
    SETB    EN
    CLR     RS
    CLR     RW
    MOV     A, #38h
    CALL    shft
    CLR     EN
    CALL    wait_lcd
    SETB    EN
    CLR     RS
    CLR     RW
    MOV     A, #0Eh
    CALL    shft
    MOV     DAT, #0Eh
    CLR     EN
    CALL    wait_lcd
    SETB    EN
    CLR     RS
    CLR     RW
    MOV     A, #06h
    CALL    shft
    CLR     EN
    CALL    wait_lcd
    RET
;-------------
clear_lcd:
    SETB    EN
    CLR     RS
    CLR     RW
    MOV     A, #01h
    CALL    shft
    CLR     EN
    CALL    wait_lcd
    RET
;-------------
write_char:
    SETB    EN
    SETB    RS
    CLR     RW
    CALL    shft
    CLR     EN
    CALL    wait_lcd
    RET
;-------------
shft:
    CLR     SCK
    CLR     RCK
    MOV     R4, #8
again:
```



```
    RLC     A
    MOV     SER, C
    SETB    SCK
    CLR     SCK
    DJNZ    R4, again
    SETB    RCK
    RET
;-------------------
        RSEG MYDATA
txt:    DB "LCD interface works", 0dh
        END
```

Исходный текст программы во многом напоминает рассмотренный ранее пример, за исключением процедуры `write_char`. В этой процедуре выполняется побитовый прием байта данных по линии `SER`. После приема последнего бита байт данных по команде

```
SETB RCK
```

выставляется на линии D0 – D7, после чего выводится на индикатор.

Рассмотренные примеры аппаратно-программных реализаций вывода данных далеко не исчерпывают многообразие способов решения этой проблемы. Много информации по этим вопросам можно обнаружить на сайтах производителей соответствующих компонентов.

Заключение

В этой книге автор старался осветить как можно более широкий круг вопросов, касающихся программирования систем на базе микроконтроллеров 8051. В книгу, конечно же, не могли быть включены все аспекты проектирования таких систем, особенно с учетом бурного развития технологий. Тем не менее автор выражает уверенность, что анализ принципов создания и программирования 8051-совместимых систем принесет определенную пользу и при решении задач, которые не были рассмотрены на страницах данного издания. Автор надеется, что материал книги будет полезен заинтересованным читателям и станет незаменимым подспорьем 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.