

**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное бюджетное
образовательное учреждение
высшего профессионального образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

И. И. Кочегаров, В. А. Трусков

МИКРОКОНТРОЛЛЕРЫ СЕМЕЙСТВА AVR. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Учебное пособие

ПЕНЗА 2012

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное бюджетное
образовательное учреждение
высшего профессионального образования
«Пензенский государственный университет» (ПГУ)

И. И. Кочегаров, В. А. Трусков

Микроконтроллеры семейства AVR. Лабораторный практикум

Учебное пособие

Рекомендовано учебно-методическим объединением вузов
Российской Федерации по образованию в области радиотехники,
электроники, биомедицинской техники и автоматизации
в качестве учебного пособия для студентов высших учебных заведений
по направлению подготовки «Проектирование и технология
радиоэлектронных средств»

Пенза
Издательство ПГУ
2012

УДК 004.94

K75

Р е ц е н з е н т ы:

доктор технических наук, профессор,
заведующий кафедрой «Управление инновациями»
Пензенского регионального центра высшей школы (филиал) ФГБОУ ВПО
«Российский государственный университет инновационных технологий
и предпринимательства»

В. И. Чернецов;

кандидат технических наук,
заведующая лабораторией ОАО «Научно-исследовательский институт
электронно-механических приборов»

И. А. Кострикина

Кочегаров, И. И.

K75 Микроконтроллеры AVR. Лабораторный практикум : учеб.
пособие / И. И. Кочегаров, В. А. Трусов. – Пенза : Изд-во ПГУ,
2012. – 122 с.

ISBN 978-5-94170-509-2

Рассматриваются современное состояние микроконтроллеров, эволюция RISC-архитектуры. Приводится подробное описание AVR-микроконтроллеров, языка ассемблер. Дается описание среды разработки AVR Studio. Для практического освоения материала предназначены шесть лабораторных работ.

Учебное пособие подготовлено на кафедре «Конструирование и производство радиоаппаратуры» ПГУ и предназначено для использования студентами в учебных курсах бакалавров по направлению 211000, связанных с микропроцессорной и микроконтроллерной аппаратурой.

УДК 004.94

ISBN 978-5-94170-509-2

© Пензенский государственный
университет, 2012

Введение

Добавление интеллектуальных функций в современную радиоэлектронную аппаратуру базируется на использовании микропроцессоров (МП), программируемых логических интегральных схем (ПЛИС), приборов типа «система на кристалле» (System-On-Chip – SOC) и других современных цифровых устройств. Подобная интеграция позволяет автоматизировать процессы измерения, управления, контроля, регулирования и обработки информации, а также обеспечить такие свойства приборных комплексов, как многофункциональность, модифицируемость, адаптивность, обучаемость и ряд других.

В отличие от персональных компьютеров, вычислительным ядром которых служат универсальные высокопроизводительные микропроцессоры, для интеграции в различные устройства применяют микроконтроллеры. Микроконтроллер (МК, англ. Micro Controller Unit, MCU) – микросхема, которая сочетает на одном кристалле функции процессора и периферийных устройств, содержит ОЗУ или ПЗУ. По сути, это однокристалльный компьютер, способный выполнять простые задачи.

С появлением однокристалльных микроЭВМ связывают начало эры массового применения компьютерной автоматизации в области управления. В связи со спадом отечественного производства и возросшим импортом техники, в том числе вычислительной, термин «микроконтроллер» (МК) вытеснил из употребления ранее использовавшийся термин «однокристалльная микроЭВМ».

Первый патент на однокристалльную микроЭВМ был выдан в 1971 г. инженерам М. Кочрену и Г. Буну, сотрудникам американской Texas Instruments. Именно они предложили на одном кристалле разместить не только процессор, но и память с устройствами ввода-вывода.

В 1976 г. американская фирма Intel выпускает микроконтроллер i8048. В 1978 г. фирма Motorola выпустила свой первый микроконтроллер MC6801, совместимый по системе команд с выпущенным ранее микропроцессором MC6800. Через 4 года, в 1980 г., Intel выпускает следующий микроконтроллер i8051. Удачный набор периферийных устройств, возможность гибкого выбора внешней или внут-

ренной программной памяти и приемлемая цена обеспечили этому микроконтроллеру успех на рынке. С точки зрения технологии микроконтроллер i8051 являлся для своего времени очень сложным изделием – в кристалле было использовано 128 тыс. транзисторов, что в 4 раза превышало количество транзисторов в 16-разрядном микропроцессоре i8086.

На сегодняшний день существует более 200 модификаций микроконтроллеров, совместимых с i8051, выпускаемых двумя десятками компаний, и большое количество микроконтроллеров других типов. Популярностью у разработчиков пользуются 8-битные микроконтроллеры PIC фирмы Microchip Technology и AVR фирмы Atmel, 16-битные MSP430 фирмы TI, а также 32-битные микроконтроллеры, архитектуры ARM, которую разрабатывает фирма ARM Limited и продает лицензии другим фирмам для их производства. В СССР велись разработки оригинальных микроконтроллеров, также осваивался выпуск клонов наиболее удачных зарубежных образцов (например КР1816ВЕ51 – аналог популярного i8051). В 1979 г. в СССР в НИИ ТТ разработали однокристалльную 16-разрядную ЭВМ К1801ВЕ1, микроархитектура которой называлась «Электроника НЦ».

При проектировании микроконтроллеров приходится соблюдать баланс между размерами и стоимостью, с одной стороны, и гибкостью и производительностью – с другой. Для разных приложений оптимальное соотношение этих и других параметров может различаться очень сильно. Поэтому существует огромное количество типов микроконтроллеров, отличающихся архитектурой процессорного модуля, размером и типом встроенной памяти, набором периферийных устройств, типом корпуса и т.д. В отличие от обычных компьютерных микропроцессоров, в микроконтроллерах часто используется гарвардская архитектура памяти, т.е. раздельное хранение данных и команд в ОЗУ и ПЗУ соответственно.

Кроме ОЗУ, микроконтроллер может иметь встроенную энергонезависимую память для хранения программы и данных. Во многих контроллерах вообще нет шин для подключения внешней памяти. Наиболее дешевые типы памяти допускают лишь однократную запись. Такие устройства подходят для массового производства в тех случаях, когда программа контроллера не будет обновляться. Другие модификации контроллеров обладают возможностью многократной перезаписи энергонезависимой памяти.

Неполный список периферии, которая может присутствовать в микроконтроллерах, включает в себя:

- универсальные цифровые порты, которые можно настраивать как на ввод, так и на вывод;
- различные интерфейсы ввода-вывода, такие как UART, I²C, SPI, CAN, USB, IEEE 1394, Ethernet;
- аналого-цифровые и цифроаналоговые преобразователи;
- компараторы;
- широтно-импульсные модуляторы;
- таймеры;
- контроллеры бесколлекторных двигателей;
- контроллеры дисплеев и клавиатур;
- радиочастотные приемники и передатчики;
- массивы встроенной флеш-памяти;
- встроенный тактовый генератор и сторожевой таймер.

Ограничения по цене и энергопотреблению сдерживают также рост тактовой частоты контроллеров. Хотя производители стремятся обеспечить работу своих изделий на высоких частотах, они, в то же время, предоставляют заказчикам выбор, выпуская модификации, рассчитанные на разные частоты и напряжения питания. Во многих моделях микроконтроллеров используется статическая память для ОЗУ и внутренних регистров. Это дает контроллеру возможность работать на меньших частотах и даже не терять данные при полной остановке тактового генератора. Часто предусмотрены различные режимы энергосбережения, в которых отключается часть периферийных устройств и вычислительный модуль.

На сегодняшний день наиболее известные семейства микроконтроллеров:

- MCS 51 (Intel)
- MSP430 (TI)
- ARM (ARM Limited)
- AVR (Atmel) (ATmega, ATtiny, XMega)
- PIC (Microchip).

В то время как 8-разрядные процессоры общего назначения полностью вытеснены более производительными моделями, 8-разрядные микроконтроллеры продолжают широко использоваться. Это объясняется тем, что существует большое количество применений, в которых не требуется высокая производительность, но важна низкая стоимость. В то же время, есть микроконтроллеры, обладающие

большими вычислительными возможностями, например цифровые сигнальные процессоры.

В связи с широким распространением в современной инженерной практике англоязычной технической литературы и программного обеспечения в пособии приведены английские эквиваленты основных терминов.

Далее в учебном пособии дается краткая информация о гарвардской и RISC-архитектуре, более подробно рассматривается семейство микроконтроллеров AVR на примере ATmega. Дается описание ассемблера AVR, описание работы с программой AVR Studio 4. В практической части приводятся лабораторные задания для освоения работы с микроконтроллерами семейства AVR.

Глава 1. Гарвардская архитектура

Гарвардская архитектура – архитектура ЭВМ, отличительными признаками которой являются:

1. Хранилище инструкций и хранилище данных представляют собой разные физические устройства.
2. Канал инструкций и канал данных также физически разделены.

Архитектура была разработана Говардом Эйкеном в конце 1930-х гг. в Гарвардском университете.

Первым компьютером, в котором была использована идея гарвардской архитектуры, был «Марк I». Гарвардская архитектура используется в ПЛК и микроконтроллерах, таких как Microchip PIC, Atmel AVR, Intel 4004, Intel 8051.

Классическая гарвардская архитектура

Типичные операции (сложение и умножение) требуют от любого вычислительного устройства нескольких действий:

- выборку двух операндов;
- выбор инструкции и ее выполнение;
- и, наконец, сохранение результата.

Идея, реализованная Эйкеном, заключалась в физическом разделении линий передачи команд и данных. В первом компьютере Эйкена «Марк I» для хранения инструкций использовалась перфорированная лента, а для работы с данными – электромеханические регистры. Это позволяло одновременно пересылать и обрабатывать команды и данные, благодаря чему значительно повышалось общее быстродействие компьютера.

В гарвардской архитектуре характеристики устройств памяти для инструкций и памяти для данных не требуется иметь общими. В частности, ширина слова, тайминги, технология реализации и структура адресов памяти могут различаться. В некоторых системах инструкции могут храниться в памяти только для чтения, в то время как для сохранения данных обычно требуется память с возможностью чтения и записи. В некоторых системах требуется значительно больше памяти для инструкций, чем памяти для данных, поскольку данные обычно могут подгружаться с внешней или более медленной памяти. Такая потребность увеличивает битность (ширину) шины

адреса памяти инструкций по сравнению с шиной адреса памяти данных.

При альтернативном варианте, архитектуре фон Неймана процессор одномоментно может либо читать инструкцию, либо читать/записывать единицу данных из/в памяти. То и другое не может происходить одновременно, поскольку инструкции и данные используют одну и ту же системную шину. А в компьютере с использованием гарвардской архитектуры процессор может читать инструкции и выполнять доступ к памяти данных в то же самое время, даже без кэш-памяти. Таким образом, компьютер с гарвардской архитектурой может быть быстрее (при определенной сложности схемы), поскольку доставка инструкций и доступ к данным не претендуют на один и тот же канал памяти. Также машина гарвардской архитектуры имеет различные адресные пространства для команд и данных.

Модифицированная гарвардская архитектура

Соответствующая схема реализации доступа к памяти имеет один очевидный недостаток – высокую стоимость. При разделении каналов передачи команд и данных на кристалле процессора последний должен иметь почти вдвое больше выводов, так как шина адреса и шина данных составляют основную часть выводов микропроцессора. Способом решения этой проблемы стала идея использовать общие шину данных и шину адреса для всех внешних данных, а внутри процессора использовать шину данных, шину команд и две шины адреса. Такую концепцию стали называть модифицированной гарвардской архитектурой.

Такой подход применяется в современных сигнальных процессорах. Еще дальше по пути уменьшения стоимости пошли при создании однокристальных ЭВМ – микроконтроллеров. В них одна шина команд и данных применяется и внутри кристалла.

Разделение шин в модифицированной гарвардской структуре осуществляется при помощи отдельных управляющих сигналов: чтения, записи или выбора области памяти.

Гибридные модификации с архитектурой фон Неймана

Существуют гибридные архитектуры, сочетающие достоинства как гарвардской так и фон неймановской архитектур. Современные CISC-процессоры обладают отдельной кэш-памятью 1-го уровня для инструкций и данных, что позволяет им за один рабочий такт получать одновременно и команду, и данные для ее выполнения. То есть процессорное ядро, формально, является гарвардским, но

программно оно – фон неймановское, что упрощает написание программ. Обычно в данных процессорах одна шина используется и для передачи команд, и для передачи данных, что упрощает конструкцию системы. Современные варианты таких процессоров могут иногда содержать встроенные контроллеры сразу нескольких разнотипных шин для работы с различными типами памяти – например, DDR RAM и Flash. Тем не менее, и в этом случае шины, как правило, используются и для передачи команд, и для передачи данных без разделения, что делает данные процессоры еще более близкими к фон неймановской архитектуре при сохранении плюсов гарвардской архитектуры.

Глава 2. Архитектура RISC

RISC (от англ. *restricted (reduced) instruction set computer* – компьютер с сокращенным набором команд) – архитектура процессора, в которой быстродействие увеличивается за счет упрощения инструкций, чтобы их декодирование было более простым, а время выполнения – короче. Первые RISC-процессоры даже не имели инструкций умножения и деления. Это также облегчает повышение тактовой частоты и делает более эффективной суперскалярность (распараллеливание инструкций между несколькими исполнительными блоками).

Наборы инструкций в более ранних архитектурах для облегчения ручного написания программ на языке ассемблера или прямо в машинных кодах, а также для упрощения реализации компиляторов, выполняли как можно больше работы. Нередко в наборы включались инструкции для прямой поддержки конструкций языков высокого уровня. Другая особенность этих наборов – большинство инструкций, как правило, допускали все возможные методы адресации – к примеру, и операнды, и результат в арифметических операциях доступны не только в регистрах, но и через непосредственную адресацию, и прямо в памяти. Позднее такие архитектуры были названы CISC (англ. *Complex instruction set computer*).

Однако многие компиляторы не задействовали все возможности таких наборов инструкций, а на сложные методы адресации уходит много времени из-за дополнительных обращений к медленной памяти. Было показано, что такие функции лучше исполнять последовательностью более простых инструкций, если при этом процессор упрощается и в нем остается место для большего числа регистров, за счет которых можно сократить количество обращений к памяти. В первых архитектурах, причисляемых к RISC, большинство инструкций для упрощения декодирования имеют одинаковую длину и похожую структуру, арифметические операции работают только с регистрами, а работа с памятью идет через отдельные инструкции загрузки (*load*) и сохранения (*store*). Эти свойства и позволили лучше сбалансировать этапы конвейеризации, сделав конвейеры в RISC значительно более эффективными и позволив поднять тактовую частоту.

В середине 1970-х гг. разные исследователи (в частности, из IBM) показали, что большинство комбинаций инструкций и различных методов адресации не использовались в большинстве про-

грамм, порождаемых компиляторами того времени. Также было обнаружено, что в некоторых архитектурах сложные операции зачастую были медленнее последовательности более простых операций, выполняющих те же действия. Это было вызвано, в частности, тем, что многие архитектуры разрабатывались в спешке и хорошо оптимизировался микрокод только тех инструкций, которые использовались чаще.

Поскольку многие реальные программы тратят большинство своего времени на выполнение простых операций, многие исследователи решили сфокусироваться на том, чтобы сделать эти операции максимально быстрыми. Тактовая частота процессора ограничена временем, которое процессор тратит на выполнение наиболее медленных шагов в процессе обработки любой инструкции; уменьшение длительности таких шагов дает общее повышение частоты, а также зачастую ускоряет выполнение и других инструкций за счет более эффективной конвейеризации. Фокусирование на простых инструкциях и ведет к архитектуре RISC, цель которой – сделать инструкции настолько простыми, чтобы они легко конвейеризировались и тратили не более одного такта на каждом шаге конвейера на высоких частотах.

Позднее было отмечено, что наиболее значимая характеристика RISC в разделении инструкций для обработки данных и обращения к памяти – обращение к памяти идет только через инструкции `load` и `store`, а все прочие инструкции ограничены внутренними регистрами. Это упростило архитектуру процессоров: позволило инструкциям иметь фиксированную длину, упростило конвейеры и изолировало логику, имеющую дело с задержками при доступе к памяти, только в двух инструкциях.

Нередко слова «сокращенный набор команд» понимаются как минимизация количества инструкций в системе команд. На самом деле, термин «сокращенный» в названии описывает тот факт, что сокращен объем (и время) работы, выполняемый каждой отдельной инструкцией (как максимум один цикл доступа к памяти), тогда как сложные инструкции CISC-процессоров могут требовать сотен циклов доступа к памяти для своего выполнения.

Первая система, которая может быть названа RISC-системой, – суперкомпьютер CDC 6600, который был создан в 1964 г., за десять лет до появления соответствующего термина. CDC 6600 имел RISC-архитектуру всего с двумя режимами адресации («ре-

гистр+регистр» и «регистр+непосредственное значение») и 74 кодами команд (тогда как процессор 8086 имел 400 кодов команд).

Однако наиболее известные RISC-системы были разработаны в рамках университетских исследовательских программ, финансировавшихся программой DARPA VLSI.

Проект RISC в университете Беркли был начат в 1980 г. под руководством Дэвида Паттерсона и Карло Секвина. Исследования базировались на использовании конвейерной обработки и агрессивного использования техники регистрового окна. В обычном процессоре имеется небольшое количество регистров и программа может использовать любой регистр в любое время. В процессоре, использующем технологии регистрового окна, очень большое количество регистров (например 128), но программы могут использовать ограниченное количество (например только 8 в каждый момент времени).

Программа, ограниченная лишь восемью регистрами для каждой процедуры, может выполнять очень быстрые вызовы процедур: «окно» просто сдвигается к 8-регистровому блоку нужной процедуры, а при возврате из процедуры сдвигается обратно, к регистрам вызвавшей процедуры. (В обычном процессоре большинство процедур при вызове вынуждены сохранять значения некоторых регистров в стеке для того, чтобы пользоваться этими регистрами при исполнении процедуры. При возврате из процедуры значения регистров восстанавливаются из стека).

Проект RISC произвел на свет процессор RISC-I в 1982 г. В нем было 44 420 транзисторов (для сравнения: в CISC-процессорах того времени их было около 100 тыс.). RISC-I имел всего 32 инструкции, но превосходил по скорости работы любой однокиповый процессор того времени. Через год, в 1983 г., был выпущен RISC-II, который состоял из 40 760 транзисторов, использовал 39 инструкций и работал в три раза быстрее RISC-I.

Практически в то же время, в 1981 г., Джон Хеннесси начал аналогичный проект, названный «MIPS-архитектура» в Стэнфордском университете. Создатель MIPS практически полностью сфокусировался на конвейерной обработке, попытавшись «выжать все» из этой технологии. Конвейерная обработка использовалась и в других продуктах, некоторые идеи, реализованные в MIPS, позволили разработанному чипу работать значительно быстрее аналогов. Наиболее важным было требование выполнения любой из инструкций процессора за один такт. Это требование позволило конвейеру работать на гораздо больших скоростях передачи данных и привело к значитель-

ному ускорению работы процессора. С другой стороны, исполнение этого требования имело негативный побочный эффект в виде удаления из набора инструкций таких полезных операций, как умножение или деление.

В первые годы попытки развития RISC-архитектуры были хорошо известны, однако оставались в рамках породивших их университетских исследовательских лабораторий. Многие в компьютерной индустрии считали, что преимущества RISC-процессоров не проявятся при использовании в реальных продуктах из-за низкой эффективности использования памяти в составных инструкциях. Однако с 1986 г. исследовательские проекты RISC начали выпускать первые работающие продукты.

Как оказалось в начале 1990-х гг., RISC-архитектуры позволяют получить большую производительность, чем CISC, за счет распараллеливания, а также за счет возможности серьезного повышения тактовой частоты и упрощения кристалла с высвобождением площади под кэш-память, достигающий огромных емкостей. Также, RISC-архитектуры позволили сильно снизить энергопотребление процессора за счет уменьшения числа транзисторов.

Первое время RISC-архитектуры с трудом принимались рынком из-за отсутствия программного обеспечения для них. Эта проблема была решена переносом UNIX-подобных операционных систем (SunOS) на RISC-архитектуры.

В настоящее время многие архитектуры процессоров являются RISC-подобными, к примеру, ARM, DEC Alpha, SPARC, AVR, MIPS, POWER и PowerPC. Наиболее широко используемые в настольных компьютерах процессоры архитектуры x86 ранее являлись CISC-процессорами, однако новые процессоры, начиная с Intel 486DX, являются CISC-процессорами с RISC-ядром. Они непосредственно перед исполнением преобразуют CISC-инструкции x86-процессоров в более простой набор внутренних инструкций RISC.

После того, как процессоры архитектуры x86 были переведены на суперскалярную RISC-архитектуру, можно сказать, что большинство существующих ныне процессоров основаны на архитектуре RISC.

Глава 3. Общее описание микроконтроллеров AVR

AVR – семейство восьмибитных микроконтроллеров фирмы Atmel, впервые выпущенные в 1996 г. Они представляют собой мощный инструмент, универсальную основу для создания современных экономичных встраиваемых систем многоцелевого назначения.

Идея разработки нового RISC-ядра принадлежит двум студентам Норвежского университета наук и технологий (г. Тронхейм) – Альфу Богену (Alf-Egil Bogen) и Вегарду Воллену (Vegard Wollen). В 1995 г. Боген и Воллен решили предложить американской корпорации Atmel выпускать новый 8-битный RISC-микроконтроллер и снабдить его Flash-памятью для программ на одном кристалле с вычислительным ядром.

Идея была одобрена Atmel Corporation, и в конце 1996 г. был выпущен опытный микроконтроллер AT90S1200, а во второй половине 1997 г. корпорация Atmel приступила к серийному производству нового семейства микроконтроллеров.

Новое ядро было запатентовано и получило название AVR. Существует несколько трактовок данной аббревиатуры. Кто-то утверждает, что это Advanced Virtual RISC, другие полагают, что не обошлось здесь без инициалов разработчиков Alf Egil Bogen Vegard Wollan RISC.

Микроконтроллеры AVR имеют гарвардскую архитектуру (программа и данные находятся в разных адресных пространствах) и систему команд, близкую к идеологии RISC. Процессор AVR имеет 32 8-битных регистра общего назначения, объединенных в регистровый файл. В отличие от «идеального» RISC, регистры не абсолютно равноправны:

- три «сдвоенных» 16-битных регистра-указателя X (r26:r27), Y (r28:r29) и Z (r30:r31);
- некоторые команды работают только с регистрами r16...r31;
- результат умножения (в тех моделях, в которых есть модуль умножения) всегда помещается в r0:r1.

Система команд микроконтроллеров AVR

Система команд микроконтроллеров AVR весьма развита и насчитывает в различных моделях от 90 до 133 различных инструкций.

Большинство команд занимает только 1 ячейку памяти (16 бит).

Большинство команд выполняется за 1 такт.

Все множество команд микроконтроллеров AVR можно разбить на несколько групп:

- команды логических операций;
- команды арифметических операций и команды сдвига;
- команды операции с битами;
- команды пересылки данных;
- команды передачи управления;
- команды управления системой.

Управление периферийными устройствами осуществляется через адресное пространство данных. Для удобства существуют «сокращенные команды» IN/OUT.

Семейства и версии микроконтроллеров

Стандартные семейства:

- tinyAVR (ATtinyxxx):

Флеш-память до 16 Кб; SRAM до 512 б; EEPROM до 512 б;

Число линий ввод-вывода 4-18 (общее количество выводов 6-32);

Ограниченный набор периферийных устройств.

- megaAVR (ATmegaxxx):

Флеш-память до 256 Кб; SRAM до 8 Кб; EEPROM до 4 Кб;

Число линий ввода-вывода 23-86 (общее количество выводов 28-100);

Аппаратный умножитель;

Расширенная система команд и периферийных устройств.

- XMEGA AVR (ATxmegaxxx):

Флеш-память до 384 Кб; SRAM до 32 Кб; EEPROM до 4 Кб;

Четырехканальный DMA-контроллер;

Инновационная система обработки событий.

На основе стандартных семейств выпускаются микроконтроллеры, адаптированные под конкретные задачи:

- со встроенными интерфейсами USB, CAN, контроллером LCD;
- со встроенным радиоприемопередатчиком – серии ATAxxxx, ATAMxxx;
- для управления электродвигателями – серия AT90PWMxxxx;
- для автомобильной электроники;
- для осветительной техники.

Кроме указанных выше семейств, ATMEЛ выпускает 32-рядные микроконтроллеры семейства AVR32.

Версии контроллеров:

- AT(mega/tiny)xxx – базовая версия.
- ATxxxL – версии контроллеров, работающих на пониженном (Low) напряжении питания (2,7 В).
- ATxxxV – версии контроллеров, работающих на низком напряжении питания (1,8 В).
- ATxxxP – малопотребляющие версии (до 100 нА в режиме Power-down), применена технология рiсoPower (анонсированы в июле 2007)[1], повыводно и функционально совместимы с предыдущими версиями.
- ATxxxA – уменьшен ток потребления, перекрывается весь диапазон тактовых частот и напряжений питания двух предыдущих версий (также, в некоторых моделях, добавлены новые возможности и новые регистры, но сохранена полная совместимость с предыдущими версиями). Микроконтроллеры «А» и «не А» с точки зрения программатора ничем не отличаются.
- ATxxx-ууPI – корпус DIP
- ATxxx-ууPU – корпус DIP, бессвинцовый припой
- ATxxx-ууAI – корпус TQFP
- ATxxx-ууAU – корпус TQFP, бессвинцовый припой,

где уу (цифры 8/10/16/20) перед индексом означают максимальную частоту, на которой микроконтроллер может стабильно работать при нормальном для него напряжении питания.

Краткие характеристики встроенной периферии МК

МК AVR имеют развитую периферию, многофункциональные, двунаправленные порты ввода-вывода со встроенными подтягивающими резисторами. Конфигурация портов ввода-вывода задается программно.

В качестве источника тактовых импульсов может быть выбран:

- кварцевый резонатор;
- внешний тактовый сигнал;
- внутренний RC-генератор (частота 1, 2, 4, 8 МГц).

Внутренняя флеш-память команд до 256 Кб (не менее 10 000 циклов перезаписи).

Отладка программ осуществляется с помощью интерфейсов JTAG или debugWIRE.

Внутреннее EEPROM данных до 4 Кб (100 000 циклов).

Внутренняя SRAM до 8 Кб время доступа 1 такт.

Внешняя память объемом до 64 Кб (Mega8515 и Mega162).

Таймеры с разрядностью 8, 16 бит.

ШИМ-модулятор (PWM) 8-, 9-, 10-, 16-битный.

Аналоговые компараторы.

АЦП (ADC) с дифференциальными входами, разрядность 10 бит (12 для XMEGA AVR):

программируемый коэффициент усиления перед АЦП 1, 10 и 200;
опорное напряжение 2,56 В.

Различные последовательные интерфейсы, включая:

- двухпроводной интерфейс TWI, совместимый с I²C;
- универсальный синхронно/асинхронный приемопередатчик UART/USART;
- синхронный последовательный порт Serial Peripheral Interface (SPI).

Популярность микроконтроллеров AVR очень высока. С каждым годом они захватывают все новые и новые ниши на рынке. Не последнюю роль в этом играет соотношение показателей цена/быстродействие/энергопотребление, являющееся весьма привлекательным на рынке 8-битных микроконтроллеров. Кроме того, постоянно растет число выпускаемых сторонними производителями разнообразных программных и аппаратных средств поддержки разработок устройств на их основе.

Глава 4. Описание микроконтроллера ATmega

AVR-микроконтроллеры, как уже упоминалось, содержат на кристалле следующие аппаратные средства: 8-разрядное процессорное ядро, память программ, оперативную память данных, энергонезависимую память данных, регистры ввода-вывода, схему прерываний, схему программирования, а также периферийные устройства (рис. 1).

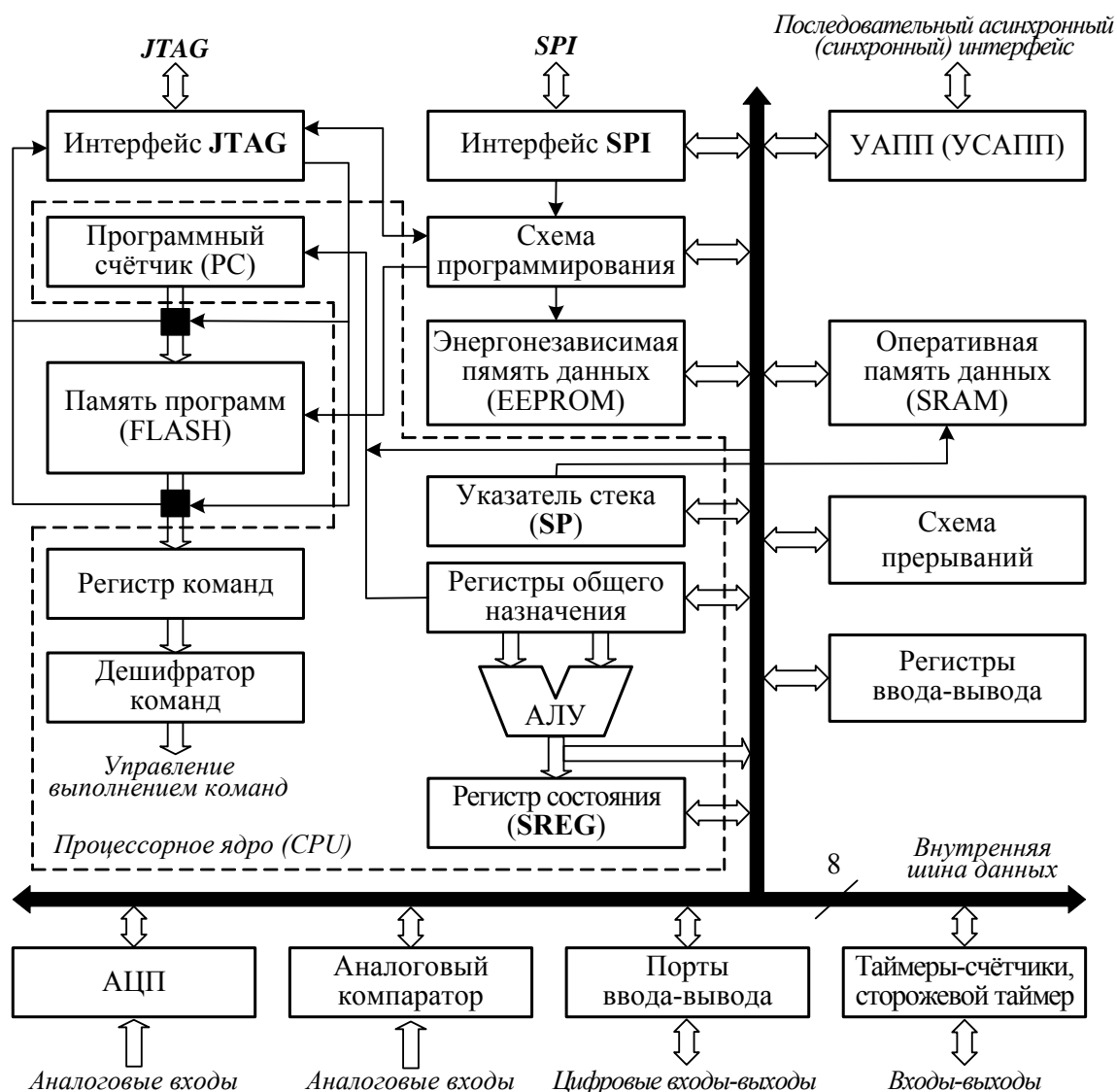


Рис. 1. Архитектура микроконтроллеров семейства AVR

Процессорное ядро (Central Processing Unit – CPU) AVR-микроконтроллеров содержит арифметико-логическое устройство (АЛУ), регистры общего назначения (РОН), программный счетчик, указатель стека, регистр состояния, регистр команд, дешифратор команд, схему управления выполнением команд.

В АЛУ выполняются все вычислительные операции. Операции производятся только над содержимым РОН. На выборку содержимого регистров, выполнение операции и запись результата обратно в РОН затрачивается один машинный такт (один период тактовой частоты).

Регистры общего назначения представляют собой 8-разрядные ячейки памяти с быстрым доступом, непосредственно доступные АЛУ. В AVR-микроконтроллерах имеется 32 РОН.

Программный счетчик (Program Counter – **PC**) содержит адрес следующей выполняемой команды.

Указатель стека (Stack Pointer – **SP**) служит для хранения адреса вершины стека.

Регистр состояния (Status Register – **SREG**) содержит слово состояния процессора.

Регистр команд, дешифратор команд и схема управления выполнением команд обеспечивают выборку из памяти программ команды, адрес которой содержится в программном счетчике, ее декодирование, определение способа доступа к указанным в команде аргументам и собственно выполнение команды. Для ускорения выполнения команд используется механизм конвейеризации, который заключается в том, что во время исполнения текущей команды программный код следующей выбирается из памяти и декодируется.

Память AVR-микроконтроллеров организована по схеме гарвардского типа – адресные пространства памяти программ и памяти данных разделены.

Память программ представляет собой перепрограммируемое ПЗУ типа FLASH и выполнена в виде последовательности 16-разрядных ячеек, так как большинство команд AVR-микроконтроллера являются 16-разрядными словами. Гарантируется не менее 10 000 циклов перезаписи. Память программ имеет размер от 2 до 256 Кбайт (от 1 до 128 К слов).

Оперативная память данных представляет собой статическое ОЗУ (SRAM – Static Random-Access Memory) и организована как последовательность 8-разрядных ячеек. Оперативная память данных может быть внутренней (до 16 Кбайт) и внешней (до 64 Кбайт).

Энергонезависимая (nonvolatile) память данных организована как последовательность 8-разрядных ячеек и представляет собой перепрограммируемое ПЗУ с электрическим стиранием (ППЗУ-ЭС, или EEPROM – Electrically Erasable Programmable Read-only Memory). Энергонезависимая память данных имеет размер до 64 Кбайт.

Регистры ввода-вывода предназначены для управления процессорным ядром и периферийными устройствами AVR-микроконтроллера.

Схема прерываний обеспечивает возможность асинхронного прерывания процесса выполнения программы при определенных условиях.

К *периферийным устройствам* AVR-микроконтроллера относятся порты ввода-вывода, таймеры, счетчики, сторожевой таймер, аналоговый компаратор, аналого-цифровой преобразователь, универсальный асинхронный (синхронно-асинхронный) приемопередатчик – УАПП (УСАПП), последовательный периферийный интерфейс **SPI**, интерфейс **JTAG** и др. Набором периферийных устройств определяются функциональные возможности микроконтроллера.

Обмен информацией между устройствами AVR-микроконтроллера осуществляется посредством внутренней 8-разрядной шины данных.

Программная модель AVR-микроконтроллеров

Программная модель микропроцессора представляет собой совокупность программно доступных ресурсов. В программную модель микроконтроллеров семейства AVR входят РОН, регистры ввода-вывода, память программ, оперативная память данных и энергонезависимая память данных (рис. 2).

РОН (**R0...R31**) могут использоваться в программе для хранения данных, адресов, констант и другой информации. Шесть старших регистров объединены попарно и составляют три 16-разрядных регистра **X** [**R27:R26**], **Y** [**R29:R28**] и **Z** [**R31:R30**] (рис. 3).

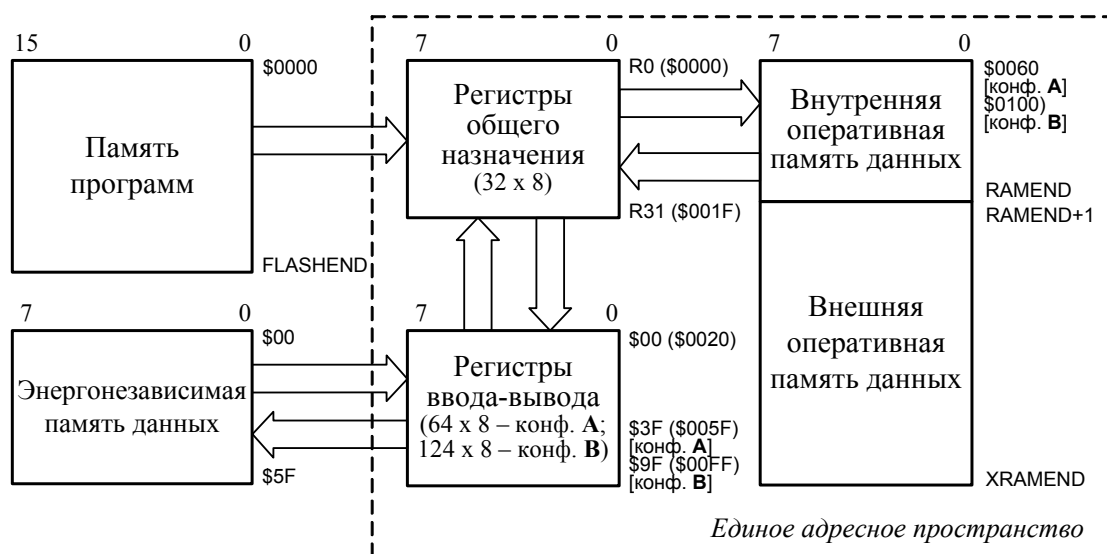


Рис. 2. Программная модель AVR-микроконтроллеров

РОН, регистры ввода-вывода и оперативная память данных образуют единое адресное пространство. Адресное пространство – это множество доступных ячеек памяти, различимых по адресам; адресом называется число, однозначно идентифицирующее ячейку памяти (регистр). Адреса ячеек памяти традиционно записываются в шестнадцатеричной системе счисления, на что указывает знак \$ в обозначении адреса.

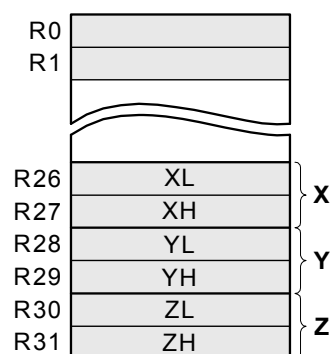


Рис. 3. Регистры общего назначения

Существует две конфигурации единого адресного пространства памяти AVR-микроконтроллеров. В конфигурации **A** младшие 32 адреса (\$0000...\$001F) соответствуют РОН, следующие 64 адреса (\$0020...\$005F) занимают регистры ввода-вывода, внутренняя оперативная память данных начинается с адреса \$0060. В конфигурации **B**, начиная с адреса \$0060, размещаются 160 дополнительных регистров ввода-вывода; внутренняя оперативная память данных начинается с адреса \$0100. Конфигурация **A** используется в младших моделях микроконтроллеров и в некоторых старших моделях в режиме совместимости с моделями, снятыми с производства; конфигурация **B** – в старших моделях.

В память программ, кроме собственно программы, могут быть записаны постоянные данные, которые не изменяются в процессе работы микропроцессорной системы (константы, таблицы линеаризации датчиков и т. п.). Выполнение программы при включении питания или после сброса микроконтроллера начинается с команды, находящейся по адресу \$0000 (т.е. в первой ячейке) памяти программ.

Энергонезависимая память данных предназначена для хранения информации, которая может изменяться непосредственно в процессе работы микропроцессорной системы (калибровочные коэффициенты, конфигурационные параметры и т.п.). Энергонезависимая память данных имеет отдельное адресное пространство и может быть считана и записана программным путем.

Система команд (instruction set) микропроцессора представляет собой совокупность выполняемых микропроцессором операций и правил их кодирования в программе. Система команд AVR-микроконтроллеров включает команды (инструкции) арифметических и логических операций, команды ветвления, управляющие последовательностью выполнения программы, команды передачи данных и

команды операций с битами. Всего в систему команд входит более 130 инструкций. Младшие модели микроконтроллеров не поддерживают некоторых из них.

Система команд AVR-микроконтроллеров приведена в гл. 5.

Периферия

Периферия микроконтроллеров AVR включает: порты (от 3 до 48 линий ввода и вывода), поддержку внешних прерываний, таймеры-счетчики, сторожевой таймер, аналоговые компараторы, 10-разрядный 8-канальный АЦП, интерфейсы UART, JTAG и SPI, устройство сброса по понижению питания, широтно-импульсные модуляторы.

Порты ввода-вывода (I/O). Порты ввода-вывода AVR имеют число независимых линий «вход-выход» от 3 до 53. Каждая линия порта может быть запрограммирована на вход или на выход. Мощные выходные драйверы обеспечивают токовую нагрузочную способность 20 мА на линию порта (втекающий ток) при максимальном значении 40 мА, что позволяет, например, непосредственно подключать к микроконтроллеру светодиоды и биполярные транзисторы. Общая токовая нагрузка на все линии одного порта не должна превышать 80 мА (все значения приведены для напряжения питания 5 В).

Архитектурная особенность построения портов ввода-вывода у AVR заключается в том, что для каждого физического вывода (пина) существует 3 бита контроля/управления, а не 2, как у распространенных 8-разрядных микроконтроллеров (Intel, Microchip, Motorola и т.д.). Это позволяет избежать необходимости иметь копию содержимого порта в памяти для безопасности и повышает скорость работы микроконтроллера при работе с внешними устройствами, особенно в условиях внешних электрических помех.

Прерывания (INTERRUPTS). Система прерываний – одна из важнейших частей микроконтроллера. Все микроконтроллеры AVR имеют многоуровневую систему прерываний. Прерывание прекращает нормальный ход программы для выполнения приоритетной задачи, определяемой внутренним или внешним событием. Для каждого такого события разрабатывается отдельная программа, которую называют подпрограммой обработки запроса на прерывание (для краткости – подпрограммой прерывания), и размещается в памяти программ. При возникновении события, вызывающего прерывание, микроконтроллер сохраняет содержимое счетчика команд, прерывает выполнение центральным процессором текущей программы и переходит к выполнению подпрограммы обработки прерывания. После

выполнения подпрограммы прерывания осуществляется восстановление предварительно сохраненного счетчика команд и процессор возвращается к выполнению прерванной программы. Для каждого события может быть установлен приоритет. Понятие приоритет означает, что выполняемая подпрограмма прерывания может быть прервана другим событием только при условии, что оно имеет более высокий приоритет, чем текущее. В противном случае центральный процессор перейдет к обработке нового события только после окончания обработки предыдущего.

Таймеры/счетчики (TIMER/COUNTERS). Микроконтроллеры AVR имеют в своем составе от 1 до 4 таймеров/счетчиков с разрядностью 8 или 16 бит, которые могут работать и как таймеры от внутреннего источника тактовой частоты, и как счетчики внешних событий.

Их можно использовать для точного формирования временных интервалов, подсчета импульсов на выводах микроконтроллера, формирования последовательности импульсов, тактирования приемопередатчика последовательного канала связи. В режиме ШИМ (PWM) таймер/счетчик может представлять собой широтно-импульсный модулятор и используется для генерирования сигнала с программируемыми частотой и скважностью. Таймеры/счетчики способны вырабатывать запросы прерываний, переключая процессор на их обслуживание по событиям и освобождая его от необходимости периодического опроса состояния таймеров. Поскольку основное применение микроконтроллеры находят в системах реального времени, таймеры/счетчики являются одним из наиболее важных элементов.

Сторожевой таймер (WDT). Сторожевой таймер (WatchDog Timer) предназначен для предотвращения катастрофических последствий от случайных сбоев программы. Он имеет свой собственный RC-генератор, работающий на частоте 1 МГц. Как и для основного внутреннего RC-генератора, значение 1 МГц является приближенным и зависит прежде всего от величины напряжения питания микроконтроллера и от температуры.

Идея использования сторожевого таймера предельно проста и состоит в регулярном его сбрасывании под управлением программы или внешнего воздействия до того, как закончится его выдержка времени и не произойдет сброс процессора. Если программа работает нормально, то команда сброса сторожевого таймера должна регулярно выполняться, предохраняя процессор от сброса. Если же микропроцессор случайно вышел за пределы программы (например,

от сильной помехи по цепи питания) либо заикнулся на каком-либо участке программы, команда сброса сторожевого таймера скорее всего не будет выполнена в течение достаточного времени и произойдет полный сброс процессора, инициализирующий все регистры и приводящий систему в рабочее состояние.

Аналоговый компаратор (АС). Аналоговый компаратор (Analog Comparator) сравнивает напряжения на двух выводах (пинах) микроконтроллера. Результатом сравнения будет логическое значение, которое может быть прочитано из программы. Выход аналогового компаратора можно включить на прерывание от аналогового компаратора. Пользователь может установить срабатывание прерывания по нарастающему или спадающему фронту или по переключению. Присутствует у всех современных AVR, кроме Mega8515.

Аналого-цифровой преобразователь (A/D CONVERTER). Аналого-цифровой преобразователь (АЦП) служит для получения числового значения напряжения, поданного на его вход. Этот результат сохраняется в регистре данных АЦП. Какой из выводов (пинов) микроконтроллера будет являться входом АЦП, определяется числом, занесенным в соответствующий регистр.

Универсальный последовательный приемопередатчик (UART или USART). Универсальный асинхронный или универсальный синхронно/асинхронный приемопередатчик (Universal Synchronous/Asynchronous Receiver and Transmitter – UART или USART) – удобный и простой последовательный интерфейс для организации информационного канала обмена микроконтроллера с внешним миром. Способен работать в дуплексном режиме (одновременная передача и прием данных). Он поддерживает протокол стандарта RS-232, что обеспечивает возможность организации связи с персональным компьютером. (Для стыковки МК и компьютера обязательно понадобится схема сопряжения уровней сигналов. Для этого существуют специальные микросхемы, например MAX232.)

Последовательный периферийный интерфейс SPI. Последовательный периферийный трехпроводный интерфейс SPI (Serial Peripheral Interface) предназначен для организации обмена данными между двумя устройствами. С его помощью может осуществляться обмен данными между микроконтроллером и различными устройствами, такими, как цифровые потенциометры, ЦАП/АЦП, FLASH-ПЗУ и др. С помощью этого интерфейса удобно производить обмен данными между несколькими микроконтроллерами AVR. Кроме того,

через интерфейс SPI может осуществляться программирование микроконтроллера.

Двухпроводной последовательный интерфейс TWI. Двухпроводной последовательный интерфейс TWI (Two-wire Serial Interface) является полным аналогом базовой версии интерфейса I2C (двухпроводная двунаправленная шина) фирмы Philips. Этот интерфейс позволяет объединить вместе до 128 различных устройств с помощью двунаправленной шины, состоящей из линии тактового сигнала (SCL) и линии данных (SDA).

Интерфейс JTAG. Интерфейс JTAG был разработан группой ведущих специалистов по проблемам тестирования электронных компонентов (Joint Test Action Group) и был зарегистрирован в качестве промышленного стандарта IEEE Std 1149.1–1990. Четырехпроводной интерфейс JTAG используется для тестирования печатных плат, внутрисхемной отладки, программирования микроконтроллеров. Многие микроконтроллеры семейства Mega имеют совместимый с IEEE Std 1149.1 интерфейс JTAG или debugWIRE для встроенной отладки. Кроме того, все микроконтроллеры Mega с флэш-памятью емкостью 16 Кбайт и более могут программироваться через интерфейс JTAG.

Тактовый генератор. Тактовый генератор вырабатывает импульсы для синхронизации работы всех узлов микроконтроллера. Внутренний тактовый генератор AVR может запускаться от нескольких источников опорной частоты (внешний генератор, внешний кварцевый резонатор, внутренняя или внешняя RC-цепочка). Минимальная допустимая частота ничем не ограничена (вплоть до пошагового режима). Максимальная рабочая частота определяется конкретным типом микроконтроллера и указывается Atmel в его характеристиках, хотя практически любой AVR-микроконтроллер с заявленной рабочей частотой, например, в 10 МГц при комнатной температуре легко может быть «разогнан» до 12 МГц и выше.

Система реального времени (RTC). RTC реализована во всех микроконтроллерах Mega и в двух кристаллах «classic» – AT90(L)S8535. Таймер/счетчик RTC имеет отдельный предделитель, который может быть программным способом подключен или к источнику основной тактовой частоты, или к дополнительному асинхронному источнику опорной частоты (кварцевый резонатор или внешний синхросигнал). Для этой цели зарезервированы два вывода микросхемы. Внутренний осциллятор оптимизирован для работы с внешним «часовым» кварцевым резонатором 32,768 кГц.

Питание

AVR функционируют при напряжениях питания от 1,8 до 6,0 В. Ток потребления в активном режиме зависит от величины напряжения питания и частоты, на которой работает микроконтроллер, и составляет менее 1 мА для 500 кГц, 5...6 мА для 5 МГц и 8...9 мА для частоты 12 МГц. AVR могут быть переведены программным путем в один из трех режимов пониженного энергопотребления.

Режим холостого хода (IDLE). Прекращает работу только процессор и фиксируется содержимое памяти данных, а внутренний генератор синхросигналов, таймеры, система прерываний и сторожевой таймер продолжают функционировать. Ток потребления не превышает 2,5 мА на частоте 12 МГц.

Стоповый режим (POWER DOWN). Сохраняется содержимое регистрового файла, но останавливается внутренний генератор синхросигналов, и, следовательно, останавливаются все функции, пока не поступит сигнал внешнего прерывания или аппаратного сброса. При включенном сторожевом таймере ток потребления в этом режиме составляет около 80 мкА, а при выключенном – менее 1 мкА. (Все приведенные значения справедливы для напряжения питания 5 В).

Экономичный режим (POWER SAVE). Продолжает работать только генератор таймера, что обеспечивает сохранность временной базы. Все остальные функции отключены.

В микроконтроллерах существует специальный режим – **сброс при снижении напряжения питания (BOD)**. Схема BOD (Brown-Out Detection) отслеживает напряжение источника питания. Если схема включена, то при снижении питания ниже некоторого значения она переводит микроконтроллер в состояние сброса. Когда напряжение питания вновь увеличится до порогового значения, запускается таймер задержки сброса. После формирования задержки внутренний сигнал сброса снимается и происходит запуск микроконтроллера.

Программирование микроконтроллеров

Процесс разработки прикладного программного обеспечения (ПО) устройств на основе однокристальных микроконтроллеров включает следующие этапы (рис. 4):

- разработки алгоритма и структуры программы;
- написания исходного текста программы;
- получения выполняемой программы;
- тестирования и отладки программы;
- получения загрузочной программы.



Рис. 4. Последовательность разработки ПО для микроконтроллеров

На этапе разработки алгоритма и структуры программы выбирается метод решения задачи и разрабатывается алгоритм его реализации. Алгоритм – это набор правил или описание последовательности операций для решения определенной задачи или достижения некоторой цели. Графическим изображением алгоритма является *схема алгоритма* (flowchart), выполняемая в соответствии с ГОСТ 19.701–90 «Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения».

На этапе написания исходного текста программы разработанный алгоритм записывается в виде программы на исходном языке (ассемблере или языке высокого уровня).

Языком ассемблера называется язык программирования, в котором каждой команде процессора или совокупности команд процессора соответствует сокращенная символическая запись (мнемоника). Использование символического обозначения команд, а также адресов регистров и ячеек памяти, переменных, констант и других элементов программы существенно облегчает процесс составления программ по сравнению с программированием на уровне машинных кодов. Символические обозначения элементов обычно отражают их содержательный смысл. Язык ассемблера обеспечивает возможность дос-

тупа ко всем ресурсам программируемого микропроцессора (микроконтроллера) и позволяет создавать программы, эффективные как по быстродействию, так и по объему занимаемой памяти. В этой связи программирование на языке ассемблера предполагает знание архитектуры и свойств микропроцессора, т.е. всего того, что входит в понятие «программная модель». Языки ассемблера различаются для разных типов микропроцессоров, т.е. являются машинно-ориентированными. В ряде ассемблеров допускается оформление повторяющейся последовательности команд как одной макрокоманды (макроса), такие ассемблеры называют *макроассемблерами*.

Языки высокого уровня (С, Паскаль, Бейсик и др.), как и ассемблер, обеспечивают доступ ко всем ресурсам микроконтроллера, но вместе с тем дают возможность создавать хорошо структурированные программы, снимают с программиста заботу о распределении памяти и содержат большой набор библиотечных функций для выполнения стандартных операций.

На этапе получения выполняемой программы исходный текст программы с помощью специальных средств (трансляторов, компиляторов, компоновщиков и др.) преобразуется в исполняемый код. *Транслятором* (translator) называют программу, служащую для перевода (трансляции) программ на языке ассемблера в машинный код, «понимаемый» процессором. *Компилятор* (compiler) представляет собой программу, преобразующую в эквивалентный машинный код текст программы на языке высокого уровня. Результатом работы транслятора или компилятора может быть как выполняемый загрузочный модуль, так и объектный модуль (программа, команды, переменные и константы которой не «привязаны» к конкретным адресам ячеек памяти). Для построения выполняемой программы из объектных модулей применяется *компоновщик* (редактор связей, linker). В процессе получения выполняемой программы из исходного текста программы устраняются синтаксические ошибки, состоящие в нарушении правил синтаксиса используемого языка программирования.

На этапе тестирования и отладки программы производится поиск, локализация и устранение в ней логических ошибок. *Тестирование* служит для обнаружения в программе ошибок и выполняется с использованием некоторого набора тестовых данных. Тестовые данные должны обеспечивать проверку всех ветвей алгоритма. При тестировании программы могут подвергаться проверке также некоторые показатели системы, связанные с программой (например, объем кодов и данных). После тестирования программа должна быть

подвергнута *отладке* (debug), задачей которой является локализация ошибки, т.е. нахождение места в программе, вызывающего ошибку.

Тестирование и отладка программы могут привести (и, как правило, приводят) к необходимости возврата к ранним этапам процесса разработки программы для устранения ошибок в постановке задачи, разработке алгоритма, написании исходного текста и т.д. Таким образом, процесс разработки программы, как и весь процесс проектирования, является итерационным.

На этапе получения загрузочной программы производится «освобождение» программы от лишних фрагментов, использовавшихся для тестирования и отладки. Эти фрагменты увеличивают объем программы и не нужны при нормальном функционировании микропроцессорной системы. Далее полученная загрузочная программа заносится в память микроконтроллера.

По завершении процесса разработки производится *документирование*, т.е. составление комплекта документов, необходимых для эксплуатации и сопровождения программы. Сопровождение программы (*program maintenance*) – это процесс внесения изменений, исправления оставшихся ошибок и проведения консультаций по программе, находящейся в эксплуатации. Виды программных документов регламентированы ГОСТ 19.101–77 «Единая система программной документации. Виды программ и программных документов».

Разработка ПО для встраиваемых микропроцессоров производится на персональном компьютере с использованием специальных программных и аппаратных средств. Такой способ создания ПО носит название *кросс-разработки*. Совокупность аппаратных и программных средств, применяемых для разработки и отладки ПО, объединяют общим наименованием *средства поддержки разработки*. В настоящем лабораторном практикуме процесс разработки ПО изучается на примере языка ассемблера AVR-микроконтроллеров. Создание исходного текста программы, трансляция и отладка выполняются в интегрированной среде разработки (Integrated Development Environment – IDE) **AVR Studio**. Более подробно работа с **AVR Studio** рассматривается в гл. 6.

Глава 5. Описание ассемблера AVR

Требования к исходному коду

Компилятор работает с исходными файлами, содержащими инструкции, метки и директивы. Инструкции и директивы, как правило, имеют один или несколько операндов.

Строка кода не должна быть длиннее 120 символов.

Любая строка может начинаться с метки, которая является набором символов заканчивающимся двоеточием. Метки используются для указания места, в которое передается управление при переходах, а также для задания имен переменных.

Входная строка может иметь одну из четырех форм:

[метка:] директива [операнды] [Комментарий]
[метка:] инструкция [операнды] [Комментарий]
[;Комментарий]
Пустая строка

Комментарий имеет следующую форму:

; [Текст]

Позиции в квадратных скобках необязательны. Текст после точки с запятой (;) и до конца строки игнорируется компилятором. Метки, инструкции и директивы более детально описываются ниже.

Примеры:

label: .EQU var1=10 ; Устанавливает var1 равным 10
(Это директива)

.EQU var2=200 ; Устанавливает var2 равным 200

test: rjmp test ; Бесконечный цикл (Это инструкция)
; Строка с одним только комментарием
; Еще одна строка с комментарием

Компилятор не требует нахождения меток, директив, комментариев или инструкций в определенной колонке строки.

Инструкции процессоров AVR

Ниже приведен набор команд процессоров AVR, более детальное описание их можно найти в описании ассемблера на официальном сайте (на английском языке, информация обновляется) [1].

Арифметические и логические инструкции

Мнемоника	Операнды	Описание	Операция	Флаги	Циклы
ADD	Rd,Rr	Суммирование без переноса	$Rd = Rd + Rr$	Z,C,N,V,H,S	1
ADC	Rd,Rr	Суммирование с переносом	$Rd = Rd + Rr + C$	Z,C,N,V,H,S	1
SUB	Rd,Rr	Вычитание без переноса	$Rd = Rd - Rr$	Z,C,N,V,H,S	1
SUBI	Rd,K8	Вычитание константы	$Rd = Rd - K8$	Z,C,N,V,H,S	1
SBC	Rd,Rr	Вычитание с переносом	$Rd = Rd - Rr - C$	Z,C,N,V,H,S	1
SBCI	Rd,K8	Вычитание константы с переносом	$Rd = Rd - K8 - C$	Z,C,N,V,H,S	1
AND	Rd,Rr	Логическое И	$Rd = Rd \wedge Rr$	Z,N,V,S	1
ANDI	Rd,K8	Логическое И с константой	$Rd = Rd \wedge K8$	Z,N,V,S	1
OR	Rd,Rr	Логическое ИЛИ	$Rd = Rd \vee Rr$	Z,N,V,S	1
ORI	Rd,K8	Логическое ИЛИ с константой	$Rd = Rd \vee K8$	Z,N,V,S	1
EOR	Rd,Rr	Логическое исключающее ИЛИ	$Rd = Rd \oplus EOR\ Rr$	Z,N,V,S	1
COM	Rd	Побитная инверсия	$Rd = \$FF - Rd$	Z,C,N,V,S	1
NEG	Rd	Изменение знака (Доп. код)	$Rd = \$00 - Rd$	Z,C,N,V,H,S	1
SBR	Rd,K8	Установить бит (биты) в регистре	$Rd = Rd \vee K8$	Z,C,N,V,S	1
CBR	Rd,K8	Сбросить бит (биты) в регистре	$Rd = Rd \wedge (\$FF - K8)$	Z,C,N,V,S	1
INC	Rd	Инкрементировать значение регистра	$Rd = Rd + 1$	Z,N,V,S	1
DEC	Rd	Декрементировать значение регистра	$Rd = Rd - 1$	Z,N,V,S	1
TST	Rd	Проверка на ноль либо отрицательность	$Rd = Rd \wedge Rd$	Z,C,N,V,S	1

Мнемоника	Операнды	Описание	Операция	Флаги	Циклы
CLR	Rd	Очистить регистр	$Rd = 0$	Z,C,N,V,S	1
SER	Rd	Установить регистр	$Rd = \$FF$	None	1
ADIW	Rdl,K6	Сложить константу и слово	$Rdh:Rdl = Rdh:Rdl + K6$	Z,C,N,V,S	2
SBIW	Rdl,K6	Вычесть константу из слова	$Rdh:Rdl = Rdh:Rdl - K6$	Z,C,N,V,S	2
MUL	Rd,Rr	Умножение чисел без знака	$R1:R0 = Rd * Rr$	Z,C	2
MULS	Rd,Rr	Умножение чисел со знаком	$R1:R0 = Rd * Rr$	Z,C	2
MULSU	Rd,Rr	Умножение числа со знаком с числом без знака	$R1:R0 = Rd * Rr$	Z,C	2
FMUL	Rd,Rr	Умножение дробных чисел без знака	$R1:R0 = (Rd * Rr) \ll 1$	Z,C	2
FMULS	Rd,Rr	Умножение дробных чисел со знаком	$R1:R0 = (Rd * Rr) \ll 1$	Z,C	2
FMULSU	Rd,Rr	Умножение дробного числа со знаком с числом без знака	$R1:R0 = (Rd * Rr) \ll 1$	Z,C	2

Инструкции ветвления

Мнемоника	Операнды	Описание	Операция	Флаги	Циклы
RJMP	k	Относительный переход	$PC = PC + k + 1$	None	2
IJMP	Нет	Косвенный переход на (Z)	$PC = Z$	None	2
EIJMP	Нет	Расширенный косвенный переход на (Z)	$STACK = PC + 1,$ $PC(15:0) = Z,$ $PC(21:16) = EIND$	None	2
JMP	k	Переход	$PC = k$	None	3

Мнемоника	Операнды	Описание	Операция	Флаги	Циклы
RCALL	k	Относительный вызов подпрограммы	$STACK = PC + 1$, $PC = PC + k + 1$	None	3/4*
ICALL	Нет	Косвенный вызов (Z)	$STACK = PC + 1$, $PC = Z$	None	3/4*
EICALL	Нет	Расширенный косвенный вызов (Z)	$STACK = PC + 1$, $PC(15:0) = Z$, $PC(21:16) = EIND$	None	4*
CALL	k	Вызов подпрограммы	$STACK = PC + 2$, $PC = k$	None	4/5*
RET	Нет	Возврат из подпрограммы	$PC = STACK$	None	4/5*
RETI	Нет	Возврат из прерывания	$PC = STACK$	I	4/5*
CPSE	Rd,Rr	Сравнить, пропустить, если равны	if (Rd == Rr) PC = PC + 2 or 3	None	1/2/3
CP	Rd,Rr	Сравнить	Rd - Rr	Z,C,N,V,H,S	1
CPC	Rd,Rr	Сравнить с переносом	Rd - Rr - C	Z,C,N,V,H,S	1
CPI	Rd,K8	Сравнить с константой	Rd - K	Z,C,N,V,H,S	1
SBRC	Rr,b	Пропустить, если бит в регистре очищен	if(Rr(b)==0) PC = PC + 2 or 3	None	1/2/3
SBRB	Rr,b	Пропустить, если бит в регистре установлен	if(Rr(b)==1) PC = PC + 2 or 3	None	1/2/3
SBIC	P,b	Пропустить, если бит в порту очищен	if(I/O(P,b)==0) PC = PC + 2 or 3	None	1/2/3
SBIS	P,b	Пропустить, если бит в порту установлен	if(I/O(P,b)==1) PC = PC + 2 or 3	None	1/2/3
BRBC	s,k	Перейти, если флаг в SREG очищен	if(SREG(s)==0) PC = PC + k + 1	None	1/2
BRBS	s,k	Перейти, если флаг в SREG установлен	if(SREG(s)==1) PC = PC + k + 1	None	1/2

Мнемоника	Операнды	Описание	Операция	Флаги	Циклы
BREQ	k	Перейти, если равно	if(Z==1) PC = PC + k + 1	None	1/2
BRNE	k	Перейти, если не равно	if(Z==0) PC = PC + k + 1	None	1/2
BRCS	k	Перейти, если перенос установлен	if(C==1) PC = PC + k + 1	None	1/2
BRCC	k	Перейти, если перенос очищен	if(C==0) PC = PC + k + 1	None	1/2
BRSH	k	Перейти, если равно или больше	if(C==0) PC = PC + k + 1	None	1/2
BRLO	k	Перейти, если меньше	if(C==1) PC = PC + k + 1	None	1/2
BRMI	k	Перейти, если минус	if(N==1) PC = PC + k + 1	None	1/2
BRPL	k	Перейти, если плюс	if(N==0) PC = PC + k + 1	None	1/2
BRGE	k	Перейти, если больше или равно (со знаком)	if(S==0) PC = PC + k + 1	None	1/2
BRLT	k	Перейти, если меньше (со знаком)	if(S==1) PC = PC + k + 1	None	1/2
BRHS	k	Перейти, если флаг внутреннего переноса установлен	if(H==1) PC = PC + k + 1	None	1/2
BRHC	k	Перейти, если флаг внутреннего переноса очищен	if(H==0) PC = PC + k + 1	None	1/2
BRTS	k	Перейти, если флаг T установлен	if(T==1) PC = PC + k + 1	None	1/2
BRTC	k	Перейти, если флаг T очищен	if(T==0) PC = PC + k + 1	None	1/2
BRVS	k	Перейти, если флаг переполнения установлен	if(V==1) PC = PC + k + 1	None	1/2
BRVC	k	Перейти, если флаг переполнения очищен	if(V==0) PC = PC + k + 1	None	1/2

Мнемоника	Операнды	Описание	Операция	Флаги	Циклы
BRIE	k	Перейти, если прерывания разрешены	if(I==1) PC = PC + k + 1	None	1/2
BRID	k	Перейти, если прерывания запрещены	if(I==0) PC = PC + k + 1	None	1/2

* Для операций доступа к данным количество циклов указано при условии доступа к внутренней памяти данных и не корректно при работе с внешним ОЗУ. Для инструкций CALL, ICALL, EICALL, RCALL, RET и RETI, необходимо добавить три цикла плюс по два цикла для каждого ожидания в контроллерах с PC меньшим 16 бит (128 Кб памяти программ). Для устройств с памятью программ свыше 128 Кб, добавьте пять циклов плюс по три цикла на каждое ожидание.

Инструкции передачи данных

Мнемоника	Операнды	Описание	Операция	Флаги	Циклы
MOV	Rd,Rr	Скопировать регистр	Rd = Rr	None	1
MOVW	Rd,Rr	Скопировать пару регистров	Rd+1:Rd = Rr+1:Rr, r,d even	None	1
LDI	Rd,K8	Загрузить константу	Rd = K	None	1
LDS	Rd,k	Прямая загрузка	Rd = (k)	None	2*
LD	Rd,X	Косвенная загрузка	Rd = (X)	None	2*
LD	Rd,X+	Косвенная загрузка с постинкрементом	Rd = (X), X=X+1	None	2*
LD	Rd,-X	Косвенная загрузка с предкрементом	X=X-1, Rd = (X)	None	2*
LD	Rd,Y	Косвенная загрузка	Rd = (Y)	None	2*
LD	Rd,Y+	Косвенная загрузка с постинкрементом	Rd = (Y), Y=Y+1	None	2*
LD	Rd,-Y	Косвенная загрузка с предкрементом	Y=Y-1, Rd = (Y)	None	2*
LDD	Rd,Y+q	Косвенная загрузка с замещением	Rd = (Y+q)	None	2*
LD	Rd,Z	Косвенная загрузка	Rd = (Z)	None	2*
LD	Rd,Z+	Косвенная загрузка с постинкрементом	Rd = (Z), Z=Z+1	None	2*
LD	Rd,-Z	Косвенная загрузка с предкрементом	Z=Z-1, Rd = (Z)	None	2*
LDD	Rd,Z+q	Косвенная загрузка с замещением	Rd = (Z+q)	None	2*

Мнемоника	Операнды	Описание	Операция	Флаги	Циклы
STS	k,Rr	Прямое сохранение	$(k) = Rr$	None	2*
ST	X,Rr	Косвенное сохранение	$(X) = Rr$	None	2*
ST	X+,Rr	Косвенное сохранение с постинкрементом	$(X) = Rr, X=X+1$	None	2*
ST	-X,Rr	Косвенное сохранение с предкрементом	$X=X-1, (X)=Rr$	None	2*
ST	Y,Rr	Косвенное сохранение	$(Y) = Rr$	None	2*
ST	Y+,Rr	Косвенное сохранение с постинкрементом	$(Y) = Rr, Y=Y+1$	None	2
ST	-Y,Rr	Косвенное сохранение с предкрементом	$Y=Y-1, (Y) = Rr$	None	2
ST	Y+q,Rr	Косвенное сохранение с замещением	$(Y+q) = Rr$	None	2
ST	Z,Rr	Косвенное сохранение	$(Z) = Rr$	None	2
ST	Z+,Rr	Косвенное сохранение с постинкрементом	$(Z) = Rr, Z=Z+1$	None	2
ST	-Z,Rr	Косвенное сохранение с предкрементом	$Z=Z-1, (Z) = Rr$	None	2
ST	Z+q,Rr	Косвенное сохранение с замещением	$(Z+q) = Rr$	None	2
LPM	Нет	Загрузка из программной памяти	$R0 = (Z)$	None	3
LPM	Rd,Z	Загрузка из программной памяти	$Rd = (Z)$	None	3
LPM	Rd,Z+	Загрузка из программной памяти с постинкрементом	$Rd = (Z), Z=Z+1$	None	3
ELPM	Нет	Расширенная загрузка из программной памяти	$R0 = (RAMPZ:Z)$	None	3
ELPM	Rd,Z	Расширенная загрузка из программной памяти	$Rd = (RAMPZ:Z)$	None	3
ELPM	Rd,Z+	Расширенная загрузка из программной памяти с постинкрементом	$Rd = (RAMPZ:Z), Z = Z+1$	None	3
SPM	Нет	Сохранение в программной памяти	$(Z) = R1:R0$	None	—
ESPM	Нет	Расширенное сохранение в программной памяти	$(RAMPZ:Z) = R1:R0$	None	—

Мнемоника	Операнды	Описание	Операция	Флаги	Циклы
IN	Rd,P	Чтение порта	$Rd = P$	None	1
OUT	P,Rr	Запись в порт	$P = Rr$	None	1
PUSH	Rr	Занесение регистра в стек	$STACK = Rr$	None	2
POP	Rd	Извлечение регистра из стека	$Rd = STACK$	None	2

* Для операций доступа к данным количество циклов указано при условии доступа к внутренней памяти данных и не корректно при работе с внешним ОЗУ. Для инструкций LD, ST, LDD, STD, LDS, STS, PUSH и POP необходимо добавить один цикл плюс по одному циклу для каждого ожидания.

Инструкции работы с битами

Мнемоника	Операнды	Описание	Операция	Флаги	Циклы
LSL	Rd	Логический сдвиг влево	$Rd(n+1)=Rd(n),$ $Rd(0)=0,$ $C=Rd(7)$	Z,C,N,V,H,S	1
LSR	Rd	Логический сдвиг вправо	$Rd(n)=Rd(n+1),$ $Rd(7)=0,$ $C=Rd(0)$	Z,C,N,V,S	1
ROL	Rd	Циклический сдвиг влево через C	$Rd(0)=C,$ $Rd(n+1)=Rd(n),$ $C=Rd(7)$	Z,C,N,V,H,S	1
ROR	Rd	Циклический сдвиг вправо через C	$Rd(7)=C,$ $Rd(n)=Rd(n+1),$ $C=Rd(0)$	Z,C,N,V,S	1
ASR	Rd	Арифметический сдвиг вправо	$Rd(n)=Rd(n+1),$ $n=0,...,6$	Z,C,N,V,S	1
SWAP	Rd	Перестановка тетрад	$Rd(3..0) =$ $Rd(7..4), Rd(7..4)$ $= Rd(3..0)$	None	1
BSET	s	Установка флага	$SREG(s) = 1$	$SREG(s)$	1
BCLR	s	Очистка флага	$SREG(s) = 0$	$SREG(s)$	1
SBI	P,b	Установить бит в порту	$I/O(P,b) = 1$	None	2
CBI	P,b	Очистить бит в порту	$I/O(P,b) = 0$	None	2
BST	Rr,b	Сохранить бит из регистра в T	$T = Rr(b)$	T	1
BLD	Rd,b	Загрузить бит из T в регистр	$Rd(b) = T$	None	1
SEC	Нет	Установить флаг переноса	$C = 1$	C	1

Мнемоника	Операнды	Описание	Операция	Флаги	Циклы
CLC	Нет	Очистить флаг переноса	$C = 0$	C	1
SEN	Нет	Установить флаг отрицательного числа	$N = 1$	N	1
CLN	Нет	Очистить флаг отрицательного числа	$N = 0$	N	1
SEZ	Нет	Установить флаг нуля	$Z = 1$	Z	1
CLZ	Нет	Очистить флаг нуля	$Z = 0$	Z	1
SEI	Нет	Установить флаг прерываний	$I = 1$	I	1
CLI	Нет	Очистить флаг прерываний	$I = 0$	I	1
SES	Нет	Установить флаг числа со знаком	$S = 1$	S	1
CLN	Нет	Очистить флаг числа со знаком	$S = 0$	S	1
SEV	Нет	Установить флаг переполнения	$V = 1$	V	1
CLV	Нет	Очистить флаг переполнения	$V = 0$	V	1
SET	Нет	Установить флаг T	$T = 1$	T	1
CLT	Нет	Очистить флаг T	$T = 0$	T	1
SEH	Нет	Установить флаг внутреннего переноса	$H = 1$	H	1
CLH	Нет	Очистить флаг внутреннего переноса	$H = 0$	H	1
NOP	Нет	Нет операции	Нет	None	1
SLEEP	Нет	Спать (уменьшить энергопотребление)	Смотрите описание инструкции	None	1
WDR	Нет	Сброс сторожевого таймера	Смотрите описание инструкции	None	1

Ассемблер не различает регистр символов.

Операнды могут быть следующих видов:

Rd: Результирующий (и исходный) регистр в регистровом файле

Rr: Исходный регистр в регистровом файле

b: Константа (3 бита), может быть константное выражение

s: Константа (3 бита), может быть константное выражение

P: Константа (5–6 бит), может быть константное выражение

K6: Константа (6 бит), может быть константное выражение

K8: Константа (8 бит), может быть константное выражение

k: Константа (размер зависит от инструкции), может быть константное выражение

q: Константа (6 бит), может быть константное выражение

Rdl: R24, R26, R28, R30. Для инструкций ADIW и SBIW

X,Y,Z: Регистры косвенной адресации (X=R27:R26, Y=R29:R28, Z=R31:R30)

Директивы ассемблера

Компилятор поддерживает ряд директив. Директивы не транслируются непосредственно в код. Вместо этого они используются для указания положения в программной памяти, определения макросов, инициализации памяти и т.д. Список директив приведен в табл. 1.

Таблица 1

Директива	Описание
BYTE	Зарезервировать байты в ОЗУ
CSEG	Программный сегмент
DB	Определить байты во флэш или EEPROM
DEF	Назначить регистру символическое имя
DEVICE	Определить устройство для которого компилируется программа
DSEG	Сегмент данных
DW	Определить слова во флэш или EEPROM
ENDM, ENDMACRO	Конец макроса
EQU	Установить постоянное выражение
ESEG	Сегмент EEPROM
EXIT	Выйти из файла
INCLUDE	Вложить другой файл
LIST	Включить генерацию листинга

Директива	Описание
LISTMAC	Включить разворачивание макросов в листинге
MACRO	Начало макроса
NOLIST	Выключить генерацию листинга
ORG	Установить положение в сегменте
SET	Установить переменный символический эквивалент выражения

Все директивы предваряются точкой.

BYTE – Зарезервировать байты в ОЗУ

Директива BYTE резервирует байты в ОЗУ. Если вы хотите иметь возможность ссылаться на выделенную область памяти, то директива BYTE должна быть предварена меткой. Директива принимает один обязательный параметр, который указывает количество выделяемых байт. Эта директива может использоваться только в сегменте данных (смотреть директивы CSEG и DSEG). Выделенные байты не инициализируются.

Синтаксис:

МЕТКА: .BYTE выражение

Пример:

```
.DSEG
var1: .BYTE 1          ; резервирует 1 байт для var1
table: .BYTE tab_size   ; резервирует tab_size байт

.CSEG
ldi r30,low(var1)      ; Загружает младший байт регистра Z
ldi r31,high(var1)     ; Загружает старший байт регистра Z
ld r1,Z                ; Загружает VAR1 в регистр 1
```

CSEG – Программный сегмент

Директива CSEG определяет начало программного сегмента. Исходный файл может состоять из нескольких программных сегментов, которые объединяются в один программный сегмент при компиляции. Программный сегмент является сегментом по умолчанию. Программные сегменты имеют свои собственные счетчики положения, которые считают не побайтно, а пословно. Директива ORG может быть использована для размещения кода и констант в необходимом месте сегмента. Директива CSEG не имеет параметров.

Синтаксис:

.CSEG

Пример:

.DSEG ; Начало сегмента данных
var1: .BYTE 4 ; Резервирует 4 байта в ОЗУ
.CSEG ; Начало кодового сегмента
const: .DW 2 ; Разместить константу 0x0002 в памяти программ
mov r1,r0 ; Выполнить действия

DB – Определить байты во флэш или EEPROM

Директива DB резервирует необходимое количество байт в памяти программ или в EEPROM. Если вы хотите иметь возможность ссылаться на выделенную область памяти, то директива DB должна быть предварена меткой. Директива DB должна иметь хотя бы один параметр. Данная директива может быть размещена только в сегменте программ (CSEG) или в сегменте EEPROM (ESEG).

Параметры, передаваемые директиве, – это последовательность выражений, разделенных запятыми. Каждое выражение должно быть или числом в диапазоне (–128...255), или в результате вычисления должно давать результат в этом же диапазоне, в противном случае число усекается до байта, причем БЕЗ выдачи предупреждений.

Если директива получает более одного параметра и текущим является программный сегмент, то параметры упаковываются в слова (первый параметр – младший байт), и если число параметров нечетно, то последнее выражение будет усечено до байта и записано как слово со старшим байтом, равным нулю, даже если далее идет еще одна директива DB.

Синтаксис:

МЕТКА: .DB список_выражений

Пример:

.CSEG
consts: .DB 0, 255, 0b01010101, -128, 0xaa
.ESEG
const2: .DB 1,2,3

DEF – Назначить регистру символическое имя

Директива DEF позволяет ссылаться на регистр через некоторое символическое имя. Назначенное имя может использоваться во всей

нижеследующей части программы для обращений к данному регистру. Регистр может иметь несколько различных имен. Символическое имя может быть переназначено позднее в программе.

Синтаксис:

.DEF Символическое_имя = Регистр

Пример:

.DEF temp=R16

.DEF ior=R0

.CSEG

ldi temp,0xf0 ; Загрузить 0xf0 в регистр temp (R16)

in ior,0x3f ; Прочитать SREG в регистр ior (R0)

eor temp,ior ; Регистры temp и ior складываются по искл. или

DEVICE – Определить устройство, для которого компилируется программа

Директива DEVICE позволяет указать, для какого устройства компилируется программа. При использовании данной директивы компилятор выдаст предупреждение, если будет найдена инструкция, которую не поддерживает данный микроконтроллер. Также будет выдано предупреждение, если программный сегмент, либо сегмент EEPROM превысят размер, допускаемый устройством. Если же директива не используется, то все инструкции считаются допустимыми, и отсутствуют ограничения на размер сегментов.

Синтаксис:

**.DEVICE AT90S1200 | AT90S2313 | AT90S2323 | AT90S2333 |
AT90S2343 | AT90S4414 | AT90S4433 | AT90S4434 | AT90S8515
| AT90S8534 | AT90S8535 | ATtiny11 | ATtiny12 | ATtiny22 | AT-
mega603 | ATmega103 | . . .**

Полный список поддерживаемых кристаллов в последней версии ассемблера доступен на сайте производителя[1]

Пример:

.DEVICE AT90S1200 ; Используется AT90S1200

.CSEG

**push r30 ; Эта инструкция вызовет предупреждение
; поскольку AT90S1200 ее не имеет**

DSEG – Сегмент данных

Директива DSEG определяет начало сегмента данных. Исходный файл может состоять из нескольких сегментов данных, которые объединяются в один сегмент при компиляции. Сегмент данных

обычно состоит только из директив BYTE и меток. Сегменты данных имеют свои собственные побайтные счетчики положения. Директива ORG может быть использована для размещения переменных в необходимом месте ОЗУ. Директива не имеет параметров.

Синтаксис:

.DSEG

Пример:

```
.DSEG                ; Начало сегмента данных
var1: .BYTE 1        ; зарезервировать 1 байт для var1
table: .BYTE tab_size ; зарезервировать tab_size байт.
.CSEG
    ldi r30,low(var1) ; Загрузить младший байт регистра Z
    ldi r31,high(var1); Загрузить старший байт регистра Z
    ld r1,Z           ; Загрузить var1 в регистр r1
```

DW – Определить слова во флэш или EEPROM

Директива DW резервирует необходимое количество слов в памяти программ или в EEPROM. Если вы хотите иметь возможность ссылаться на выделенную область памяти, то директива DW должна быть предварена меткой. Директива DW должна иметь хотя бы один параметр. Данная директива может быть размещена только в сегменте программ (CSEG) или в сегменте EEPROM (ESEG).

Параметры, передаваемые директиве, – это последовательность выражений, разделенных запятыми. Каждое выражение должно быть или числом в диапазоне (–32768...65535), или в результате вычисления должно давать результат в этом же диапазоне, в противном случае число усекается до слова, причем БЕЗ выдачи предупреждений.

Синтаксис:

МЕТКА: .DW expressionlist

Пример:

```
.CSEG
varlist: .DW 0, 0xffff, 0b1001110001010101, -32768, 65535
.ESEG
eevarlist: .DW 0,0xffff,10
```

ENDMACRO – Конец макроса

Директива определяет конец макроопределения, и не принимает никаких параметров. Для информации по определению макросов смотрите директиву MACRO.

Синтаксис:

.ENDMACRO

Пример:

```
.MACRO SUBI16          ; Начало определения макроса
    subi r16,low(@0) ; Вычесть младший байт первого
параметра
    sbci r17,high(@0) ; Вычесть старший байт первого
параметра
.ENDMACRO
```

EQU – Установить постоянное выражение

Директива EQU присваивает метке значение. Эта метка может позднее использоваться в выражениях. Метка, которой присвоено значение данной директивой, не может быть переназначена, и ее значение не может быть изменено.

Синтаксис:

.EQU метка = выражение

Пример:

```
.EQU io_offset = 0x23
.EQU porta    = io_offset + 2
.CSEG          ; Начало сегмента данных
    clr r2      ; Очистить регистр r2
    out porta,r2 ; Записать в порт A
```

ESEG – Сегмент EEPROM

Директива ESEG определяет начало сегмента EEPROM. Исходный файл может состоять из нескольких сегментов EEPROM, которые объединяются в один сегмент при компиляции. Сегмент EEPROM обычно состоит только из директив DB, DW и меток. Сегменты EEPROM имеют свои собственные побайтные счетчики положения. Директива ORG может быть использована для размещения переменных в необходимом месте EEPROM. Директива не имеет параметров.

Синтаксис:

.ESEG

Пример:

```
.DSEG          ; Начало сегмента данных
var1: .BYTE 1   ; зарезервировать 1 байт для var1
```

table: .BYTE tab_size ; зарезервировать tab_size байт.
.ESEG
eevar1: .DW 0xffff ; проинициализировать 1 слово в
EEPROM

EXIT – Выйти из файла

Встретив директиву EXIT компилятор прекращает компиляцию данного файла. Если директива использована во вложенном файле (см. директиву INCLUDE), то компиляция продолжается со строки следующей после директивы INCLUDE. Если же файл не является вложенным, то компиляция прекращается.

Синтаксис:

.EXIT

Пример:

.EXIT ; Выйти из данного файла

INCLUDE – Вложить другой файл

Встретив директиву INCLUDE компилятор открывает указанный в ней файл, компилирует его, пока файл не закончится или не встретится директива EXIT, после этого продолжает компиляцию начального файла со строки следующей за директивой INCLUDE. Вложенный файл может также содержать директивы INCLUDE.

Синтаксис:

.INCLUDE "имя_файла"

Пример:

; файл iodefs.asm:

.EQU sreg = 0x3f ; Регистр статуса

.EQU sphigh = 0x3e ; Старший байт указателя стека

.EQU splow = 0x3d ; Младший байт указателя стека

; файл incdemo.asm

.INCLUDE iodefs.asm ; Вложить определения портов
in r0,sreg ; Прочитать регистр статуса

LIST – Включить генерацию листинга

Директива LIST указывает компилятору на необходимость создания листинга. Листинг представляет из себя комбинацию ассемблерного кода, адресов и кодов операций. По умолчанию генерация листинга включена, однако данная директива используется совмест-

но с директивой NOLIST для получения листингов отдельных частей исходных файлов.

Синтаксис:

.LIST

Пример:

.NOLIST ; Отключить генерацию листинга
.INCLUDE "macro.inc" ; Вложенные файлы не будут
.INCLUDE "const.def" ; отображены в листинге
.LIST ; Включить генерацию листинга

LISTMAC – Включить разворачивание макросов в листинге

После директивы LISTMAC компилятор будет показывать в листинге содержимое макроса. По умолчанию в листинге показывается только вызов макроса и передаваемые параметры.

Синтаксис:

.LISTMAC

Пример:

.MACRO MACX ; Определение макроса
 add r0,@0 ; Тело макроса
 eor r1,@1
.ENDMACRO ; Конец макроопределения
.LISTMAC ; Включить разворачивание
макросов
 MACX r2,r1 ; Вызов макроса
 ;(в листинге будет показано
тело макроса)

MACRO – Начало макроса

С директивы MACRO начинается определение макроса. В качестве параметра директиве передается имя макроса. При встрече имени макроса позднее в тексте программы, компилятор заменяет это имя на тело макроса. Макрос может иметь до 10 параметров, к которым в его теле обращаются через @0-@9. При вызове параметры перечисляются через запятые. Определение макроса заканчивается директивой ENDMACRO.

По умолчанию в листинг включается только вызов макроса, для разворачивания макроса необходимо использовать директиву LISTMAC. Макрос в листинге показывается знаком +.

Синтаксис:

.MACRO макроимя

Пример:

```
.MACRO SUBI16          ; Начало макроопределения
    subi @1,low(@0) ; Вычесть мл. байт пар-ра 0 из пар-ра 1
    sbci @2,high(@0) ; Вычесть ст. байт пар-ра 0 из пар-ра 2
.ENDMACRO              ; Конец макроопределения
.CSEG                  ; Начало программного сегмента
    SUBI16 0x1234,r16,r17 ; Вычесть 0x1234 из r17:r16
```

NOLIST – Выключить генерацию листинга

Директива NOLIST указывает компилятору на необходимость прекращения генерации листинга. Листинг представляет из себя комбинацию ассемблерного кода, адресов и кодов операций. По умолчанию генерация листинга включена, однако может быть отключена данной директивой. Кроме того, данная директива может быть использована совместно с директивой LIST для получения листингов отдельных частей исходных файлов.

Синтаксис:

.NOLIST

Пример:

```
.NOLIST                ; Отключить генерацию листинга
.INCLUDE "macro.inc"   ; Вложенные файлы не будут
.INCLUDE "const.def"   ; отображены в листинге
.LIST                  ; Включить генерацию листинга
```

ORG – Установить положение в сегменте

Директива ORG устанавливает счетчик положения, равным заданной величине, которая передается как параметр. Для сегмента данных она устанавливает счетчик положения в SRAM (ОЗУ), для сегмента программ это программный счетчик, а для сегмента EEPROM это положение в EEPROM. Если директиве предшествует метка (в той же строке), то метка размещается по адресу, указанному в параметре директивы. Перед началом компиляции программный счетчик и счетчик EEPROM равны нулю, а счетчик ОЗУ равен 32 (поскольку адреса 0-31 заняты регистрами). Обратите внимание, что для ОЗУ и EEPROM используются побайтные счетчики, а для программного сегмента – пословный.

Синтаксис:

.ORG выражение

Пример:

```
.DSEG          ; Начало сегмента данных
.ORG 0x37      ; Установить адрес SRAM равным 0x37
variable: .BYTE 1 ; Зарезервировать байт по адресу 0x37H
.CSEG
.ORG 0x10      ; Установить программный счетчик равным
0x10
    mov r0,r1 ; Команда будет размещена по адресу 0x10
```

SET – Установить переменный символический эквивалент выражения

Директива SET присваивает имени некоторое значение. Это имя позднее может быть использовано в выражениях. Причем в отличие от директивы EQU значение имени может быть изменено другой директивой SET.

Синтаксис:

.SET имя = выражение

Пример:

```
.SET io_offset = 0x23
.SET porta    = io_offset + 2
.CSEG          ; Начало кодового сегмента
    clr r2     ; Очистить регистр 2
    out porta,r2 ; Записать в порт A
```

Выражения

Компилятор позволяет использовать в программе выражения, которые могут состоять из *операндов, знаков операций и функций*. Все выражения являются 32-битными.

Операнды

Могут быть использованы следующие операнды:

- Метки, определенные пользователем (дают значение своего положения).
- Переменные, определенные директивой SET.
- Константы, определенные директивой EQU.
- Числа, заданные в формате:
 - Десятичном (принят по умолчанию): 10, 255
 - Шестнадцатеричном (два варианта записи): 0x0a, \$0a, 0xff, \$ff

- Двоичном: 0b00001010, 0b11111111
- Восьмеричном (начинаются с нуля): 010, 077
- PC – текущее значение программного счетчика (Programm Counter).

Операции

Компилятор поддерживает ряд операций, которые перечислены в таблице (чем выше положение в таблице, тем выше приоритет операции). Выражения могут заключаться в круглые скобки, такие выражения вычисляются перед выражениями за скобками.

Приоритет	Символ	Описание
14	!	Логическое отрицание
14	~	Побитное отрицание
14	-	Минус
13	*	Умножение
13	/	Деление
12	+	Суммирование
12	-	Вычитание
11	<<	Сдвиг влево
11	>>	Сдвиг вправо
10	<	Меньше чем
10	<=	Меньше или равно
10	>	Больше чем
10	>=	Больше или равно
9	==	Равно
9	!=	Не равно
8	&	Побитное И
7	^	Побитное исключающее ИЛИ
6		Побитное ИЛИ
5	&&	Логическое И
4		Логическое ИЛИ

Логическое отрицание

Символ: !

Описание: Возвращает 1, если выражение равно 0, и наоборот

Приоритет: 14

Пример: ldi r16, !0xf0 ; В r16 загрузить 0x00

Побитное отрицание

Символ: ~

Описание: Возвращает выражение, в котором все биты проинвертированы

Приоритет: 14

Пример: ldi r16, ~0xf0 ; В r16 загрузить 0x0f

Минус

Символ: -

Описание: Возвращает арифметическое отрицание выражения

Приоритет: 14

Пример: ldi r16,-2 ; Загрузить -2(0xfe) в r16

Умножение

Символ: *

Описание: Возвращает результат умножения двух выражений

Приоритет: 13

Пример: ldi r30, label*2

Деление

Символ: /

Описание: Возвращает целую часть результата деления левого выражения на правое

Приоритет: 13

Пример: ldi r30, label/2

Суммирование

Символ: +

Описание: Возвращает сумму двух выражений

Приоритет: 12

Пример: ldi r30, c1+c2

Вычитание

Символ: -

Описание: Возвращает результат вычитания правого выражения из левого

Приоритет: 12

Пример: ldi r17, c1-c2

Сдвиг влево

Символ: <<

Описание: Возвращает левое выражение, сдвинутое влево на число бит, указанное справа

Приоритет: 11

Пример: `ldi r17, 1<<bitmask` ; В r17 загрузить 1, сдвинутую влево bitmask раз

Сдвиг вправо

Символ: >>

Описание: Возвращает левое выражение, сдвинутое вправо на число бит, указанное справа

Приоритет: 11

Пример: `ldi r17, c1>>c2` ; В r17 загрузить c1, сдвинутое вправо, c2 раз

Меньше чем

Символ: <

Описание: Возвращает 1, если левое выражение меньше, чем правое (учитывается знак), и 0 в противном случае

Приоритет: 10

Пример: `ori r18, bitmask*(c1<c2)+1`

Меньше или равно

Символ: <=

Описание: Возвращает 1, если левое выражение меньше или равно, чем правое (учитывается знак), и 0 в противном случае

Приоритет: 10

Пример: `ori r18, bitmask*(c1<=c2)+1`

Больше чем

Символ: >

Описание: Возвращает 1, если левое выражение больше, чем правое (учитывается знак), и 0 в противном случае

Приоритет: 10

Пример: `ori r18, bitmask*(c1>c2)+1`

Больше или равно

Символ: >=

Описание: Возвращает 1, если левое выражение больше или равно, чем правое (учитывается знак), и 0 в противном случае

Приоритет: 10

Пример: `ori r18, bitmask*(c1>=c2)+1`

Равно

Символ: ==

Описание: Возвращает 1, если левое выражение равно правому (учитывается знак), и 0 в противном случае

Приоритет: 9

Пример: `andi r19, bitmask*(c1==c2)+1`

Не равно

Символ: `!=`

Описание: Возвращает 1, если левое выражение не равно правому (учитывается знак), и 0 в противном случае

Приоритет: 9

Пример: `.SET flag = (c1!=c2) ;Установить flag, равным 1 или 0`

Побитное И

Символ: `&`

Описание: Возвращает результат побитового И выражений

Приоритет: 8

Пример: `ldi r18, High(c1&c2)`

Побитное исключающее ИЛИ

Символ: `^`

Описание: Возвращает результат побитового исключающего ИЛИ выражений

Приоритет: 7

Пример: `ldi r18, Low(c1^c2)`

Побитное ИЛИ

Символ: `|`

Описание: Возвращает результат побитового ИЛИ выражений

Приоритет: 6

Пример: `ldi r18, Low(c1|c2)`

Логическое И

Символ: `&&`

Описание: Возвращает 1, если оба выражения не равны нулю, и 0 в противном случае

Приоритет: 5

Пример: `ldi r18, Low(c1&& c2)`

Логическое ИЛИ

Символ: `||`

Описание: Возвращает 1, если хотя бы одно выражение не равно нулю, и 0 в противном случае

Приоритет: 4

Пример: `ldi r18, Low(c1||c2)`

Функции

Определены следующие функции:

- LOW(выражение) возвращает младший байт выражения
- HIGH(выражение) возвращает второй байт выражения
- BYTE2(выражение) то же что и функция HIGH
- BYTE3(выражение) возвращает третий байт выражения
- BYTE4(выражение) возвращает четвертый байт выражения
- LWRD(выражение) возвращает биты 0–15 выражения
- HWRD(выражение) возвращает биты 16–31 выражения
- PAGE(выражение) возвращает биты 16–21 выражения
- EXP2(выражение) возвращает 2 в степени (выражение)
- LOG2(выражение) возвращает целую часть $\log_2(\text{выражение})$

Глава 6. Работа с пакетом AVR Studio 4

Работа в среде AVR Studio. В состав среды **AVR Studio** входит редактор исходных текстов, транслятор с языка ассемблера, отладчик и симулятор.

Транслятор работает с исходными программами на языке ассемблера, содержащими метки, директивы, команды и комментарии. Метка представляет собой символическое обозначение адреса (последовательность символов, заканчивающаяся двоеточием). Метки используются для указания места в программе, в которое передается управление при переходах, а также для задания имен переменных. Директивы являются инструкциями для транслятора и не заносятся в исполняемый код программы. Директивы могут иметь один или несколько параметров. Команды записываются в программе в виде мнемонического обозначения выполняемой операции и могут иметь один или несколько *операндов*, т.е. аргументов, с которыми они вызываются. Транслятор позволяет указывать операнды в различных системах счисления: десятичной (по умолчанию, например, **15**, **154**), шестнадцатеричной (префикс **0x** или **\$**, например, **0x0f**, **\$0f**, **0x9a**, **\$9a**), восьмеричной (префикс – нуль, например, **017**, **0232**) и двоичной (префикс **0b**, например, **0b00001111**, **0b10011010**). Строка программы должна быть не длиннее 120 символов и может иметь одну из четырех форм:

```
[метка:] .директива [параметры] [;Комментарий]
[метка:] команда [операнды] [;Комментарий]
[;Комментарий]
[Пустая строка]
```

Позиции в квадратных скобках необязательны. Текст после точки с запятой и до конца строки является комментарием и транслятором игнорируется. Включение в текст программы комментариев является признаком хорошего стиля программирования и облегчает ее сопровождение. Кроме того, улучшению читаемости также способствует форматирование текста программы. При программировании на ассемблере выполнение этих правил особенно важно, так как программы на языке ассемблера неудобочитаемы.

Указать тип микроконтроллера, для которого транслируется программа, позволяет директива **.device**, например:
.device ATmega16; программа для микроконтроллера ATmega16

При наличии в программе команд, не поддерживаемых указанным в директиве микроконтроллером, транслятор выдает соответствующее предупреждение.

Входным для транслятора является файл **<имя_файла>.asm** с текстом программы на языке ассемблера. Транслятор создает четыре новых файла: файл листинга (**<имя_файла>.lst**), объектный файл (**<имя_файла>.obj**), файл-прошивку памяти программ (**<имя_файла>.hex**) и файл-прошивку энергонезависимой памяти данных (**<имя_файла>.eep**).

Файл листинга – это отчет транслятора о своей работе. На рис. 5 приведена часть листинга трансляции программы, в которой числа 2, 5 и 19 заносятся соответственно в регистры **R17**, **R18** и **R19**; вычисляются произведение и сумма содержимого регистров **R17** и **R18**; из суммы содержимого регистров **R17** и **R18** вычитается содержимое регистра **R19**. Листинг содержит исходный текст транслируемой программы, каждой команде которой поставлены в соответствие машинные коды (правый столбец чисел) и адреса ячеек памяти программ, в которых они будут размещены (левый столбец чисел). Машинные коды и адреса приводятся в шестнадцатеричной системе счисления. Например, строка листинга с командой **ADD** содержит следующую информацию: **0f12** – машинный код команды; **000004** – адрес размещения данной команды в памяти программ.

000000	e012	LDI R17, 2 ; загрузка числа 2 в регистр R17
000001	e025	LDI R18, 5 ; загрузка числа 5 в регистр R18
000002	e133	LDI R19, 19 ; загрузка числа 13 в регистр R19
000003	9f12	MUL R17, R18 ; умножение R17 на R18, результат в R1:R0
000004	0f12	ADD R17, R18 ; сложение R17 и R18, результат в R17
000005	1b31	SUB R19, R17 ; вычитание R17 из R19, результат в R19
000006	cfff	met: RJMP met ; бесконечный цикл (для отладки)

Рис. 5. Пример листинга трансляции

Объектный файл имеет специальный формат и используется для отладки программы с помощью симулятора-отладчика среды

AVR Studio. Файл прошивки памяти программ служит для занесения отлаженной программы в память программ микроконтроллера. Файл прошивки EEPROM-памяти данных предназначен для загрузки информации в энергонезависимую память данных. Операции загрузки памяти программ и энергонезависимой памяти данных выполняются с помощью специальных аппаратных средств (программаторов).

AVR Studio позволяет отслеживать выполнение программы в режиме симуляции или эмуляции, поддерживает программирование низкого уровня на ассемблерах Atmel Corporation's AVR и IAR Systems, а также программ, написанных на языке Си, в том числе и с ассемблерными вставками IAR Systems и поддерживает ICCA90 – Си-компилятор для AVR-микроконтроллеров. AVR Studio также поддерживает COFF как выходной формат для символьной отладки и программные пакеты от производителей Imagecraft C и E-lab pascal. AVR Studio работает под управлением Windows XP и более новых операционных систем корпорации Microsoft.

Вся система команд AVR поддерживается AVR Studio, в обоих режимах просмотра (дизассемблера и исходного программного кода). Можно выполнять программу пошагово или до достижения определенного условия. Кроме того, можно определять практически неограниченное число контрольных точек останова, причем каждая точка может быть установлена или отключена. Состояние контрольных точек можно сохранять между прогонами.

В процессе выполнения программы пользователь выбирает необходимое окно и режим просмотра процесса выполнения программы, тем самым привязывая режим просмотра к проекту. Каждый раз при загрузке проекта этот режим автоматически восстанавливается.

Процесс построения, ассемблирования и компилирования проекта отражается в окне сообщений (рис. 6). При обнаружении ошибки двойной щелчок на соответствующем сообщении установит курсор в позицию ошибки в текущем окне редактора.

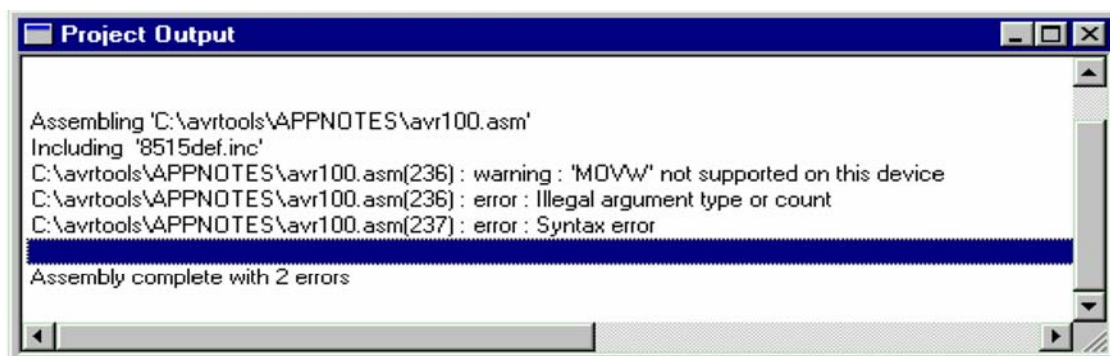


Рис. 6. Окно сообщений проекта

Окна и режимы AVR Studio

Окно Source. Одно из основных окон диалога AVR Studio – окно Source (рис. 7). При открытии файла автоматически создается окно Source. По умолчанию считается, что программа представлена на языке ассемблера и происходит запуск симуляции именно в этом формате. Можно переключать вид представления программы из формата дизассемблера в формат ассемблера и обратно при условии, что программа в настоящий момент не выполняется. Окно Source отражает программу, выполняемую в настоящий момент на микроконтроллере. Маркер команды всегда указывает на следующую за выполняемой командой. При этом панель Status отображает работу эмулятора или симулятора.

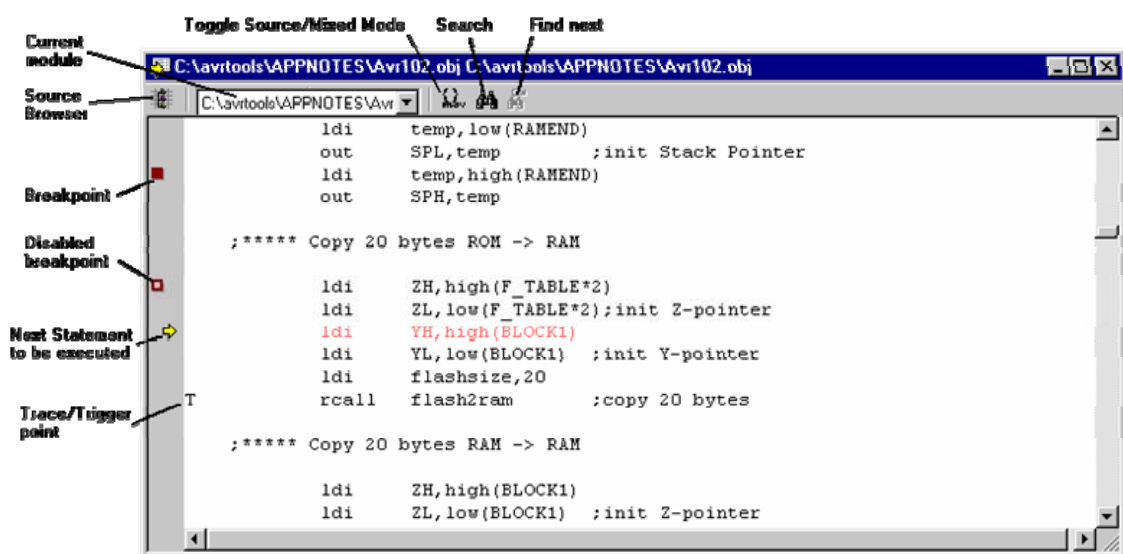


Рис. 7. Основное окно отладки (Source)

Окно Source создается при загрузке объектного файла на выполнение и присутствует на протяжении всего процесса отладки. Если закрыть окно Source, процесс отладки будет прерван. Отметки на левом поле указывают позицию программного счетчика, программных меток, точек останова и трассировку выполняющихся операторов.

Желтая стрелка указывает на текущее положение **счетчика команд (Program Counter)** и отображает следующую за выполняемой командой. Коричневый квадрат обозначает **точку останова (Breakpoint)**. Выполнение программы прекращается при достижении этой точки. Незакрашенный коричневый квадрат обозначает **отключенную точку останова (Disabled Breakpoint)**. Выполнение программы в этом месте не будет прервано, и точка используется как метка.

Зеленая буква **T** – включение **буфера трассировки (Trace On)**. При достижении этой метки будет включен буфер трассировки. Запись в буфер будет продолжаться до тех пор, пока не встретится коричневая буква **T** – точка окончания записи **Trace Off**.

Окно Source поддерживает буфер обмена Windows. Можно выбрать часть или весь текст в окне Source и скопировать его в буфер обмена Windows. Также поддерживаются функции поиска текста в текущем модуле или во всех модулях. Первый случай совпадения выделяется. Выберите find next (или Ctrl-N) для дальнейшего поиска по тексту. Точки остановки, точки трассировки и переключения, текущие позиции выполнения и функция копирования также доступны через нажатие правой кнопки мыши в окне Source.

Режимы отладки. Объектный файл может состоять из нескольких модулей. Одновременно можно просматривать не более одного модуля, но можно выбирать другие модули в окне, расположенном в верхней левой части окна Source, что очень удобно для просмотра и установки контрольных точек в различных модулях одной программы.

Исходный отлаживаемый текст можно просматривать различными способами. В зависимости от того, какой из способов просмотра выбран, отладка будет производиться в выбранном режиме.

Отображение в исходном коде (Source Mode). По умолчанию окно открывается в данном режиме при загрузке объектного файла на отладку, при этом доступны все опции отладки.

П р и м е ч а н и е. При открытии объектного файла, не содержащего информации для отладки, опции отладки недоступны. Для возврата на первоначальную позицию отладки в этом режиме выполните операцию Step Into.

Режим смешанного отображения (Mixed Mode). Находящаяся в памяти программа дизассемблируется и отображается вместе с текстовой информацией в исходном коде. Все операции отладки происходят через дизассемблер.

Окно Processor. Окно процессора содержит текущую информацию о ядре контроллера и имеет следующие поля.

Программный счетчик (Program Counter). Программный счетчик всегда указывает на следующий адрес команды относительно выполняющейся в данный момент. Содержимое программного счетчика отображается в шестнадцатеричной системе счисления и может быть изменено в режиме остановки программы. Если изменить содержимое программного счетчика, то будет отменена та команда, на

которую он прежде указывал. После того как программный счетчик изменен, необходимо нажать кнопку F11 режима пошагового выполнения для перехода по новому адресу.

Указатель стека (Stack Pointer). Указатель стека содержит адрес вершины стека. Если в микроконтроллере присутствует аппаратный стек, а не стек, расположенный в SRAM, то он отражается в области указателя стека. Содержимое указателя стека можно изменять в режиме останова программы.

Счетчик циклов (Cycle Counter). Счетчик циклов содержит информацию о количестве пройденных тактовых циклов с момента последнего сброса. Содержимое счетчика циклов отображается в десятичной форме и не может быть изменено в процессе выполнения программы.

Частота (Frequency). В области «частота» отображена текущая эмулируемая опорная частота процессора.

Флаги (Flags). В окне «флаги» отображается содержимое регистра флагов (Status register). В режиме остановки программы эти биты могут быть изменены, если щелкнуть мышкой на флаге. Тестирование регистра статуса устанавливает соответствующие флаги в 1.

Окно сообщений Message. В окне сообщений отображается текущая информация пользователю от AVR Studio. После команды сброса окно сообщений очищается. При свертывании окна сообщений его содержимое не удаляется.

Окно устройств ввода-вывода I/O. Окно I/O используется для тестирования и изменения содержимого регистров ввода-вывода контроллера (рис. 8). Одновременно могут быть открыты несколько окон регистров ввода-вывода. Новое окно I/O открывается через меню *View > New IO Window* или нажатием кнопки *New IO Window* на панели инструментов.

Окно ввода-вывода автоматически конфигурируется AVR Studio на применяемый тип микроконтроллера. Это происходит при открытии нового окна или открытии существующего проекта. Конфигурация окон загружается с диска из файла конфигурации (*configuration file*). После открытия окна ввода-вывода используйте программное меню или кнопки на панели инструментов для конфигурирования окна. AVR Studio автоматически загрузит стандартную конфигурацию для текущего устройства. При закрытии текущего проекта AVR Studio автоматически сохранит конфигурацию окон ввода-вывода в директории проекта. При последующих открытиях проекта AVR Studio автоматически загрузит сохраненную конфигурацию.

Окна ввода-вывода организовываются иерархически. Такой подход позволяет одновременно просматривать содержимое регистров ввода-вывода и быстро переключаться между окнами. В стандартной конфигурации окон пользователь имеет возможность довольно быстро наблюдать и изменять текущее состояние устройства в целом.

Окно ввода-вывода разбито на три колонки. Левая колонка отображает регистры и группы (*register or group names*). Двойным нажатием правой кнопки мыши в этой колонке можно вывести всю информацию, если она не помещается в окне. То же самое можно сделать, выделив необходимое окно или регистр. В средней колонке отображается формат регистра (*register value*), здесь данные могут быть представлены в необходимом формате. В правой колонке отображаются байтовый адрес регистра (*byte address*) и номера бит (*bit numbers*), если некоторые биты в регистре замаскированы.

Двойное нажатие левой кнопки мыши в соответствующей колонке вызывает окно диалога.

Содержимое регистров в окне I/O может быть отображено в различных форматах. Нажатием правой кнопки мыши можно переключать форматы представления данных между шестнадцатеричным, десятичным или двоичными форматами. Допускается изменение содержимого регистров и ячеек памяти.

П р и м е ч а н и е. Переменные вводятся с префиксом *0x* для шестнадцатеричных чисел, *0d* или без префикса для десятичных чисел, или *0b* для двоичных. Регистры и память, претерпевшие изменение данных, изображаются красным цветом.

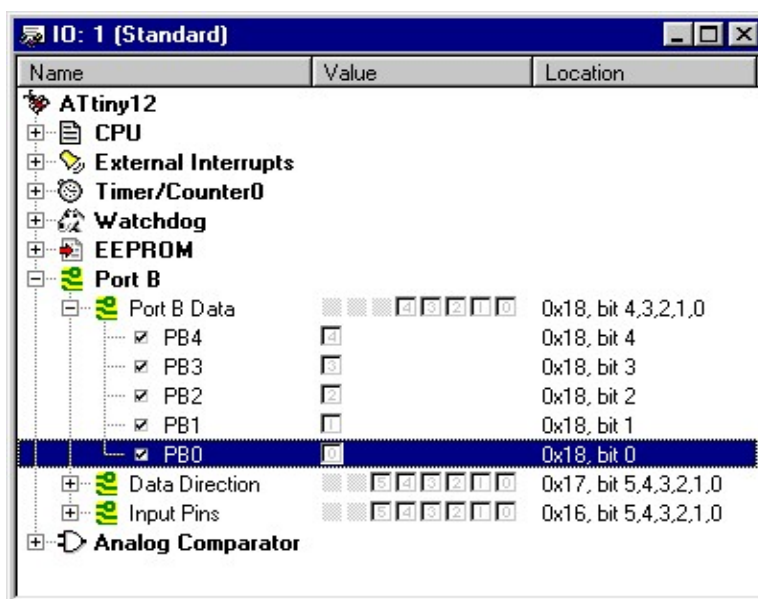


Рис. 8. Окно I/O

Допускается управление окнами ввода-вывода с клавиатуры:
Up, down, Page Up, Page Down – перемещение курсора вверх-вниз в окне;
 +(плюс), -(минус) – увеличение или уменьшение размера;
Insert – вставка;
Delete – удаление;
H – свернуть текущее и все остальные окна;
J – развернуть свернутые окна.

Окно дампа памяти (Memory window). Окно дампа памяти позволяет просматривать и модифицировать содержимое всех видов памяти контроллера (рис. 9). Одновременно может быть открыто несколько окон дампа памяти.

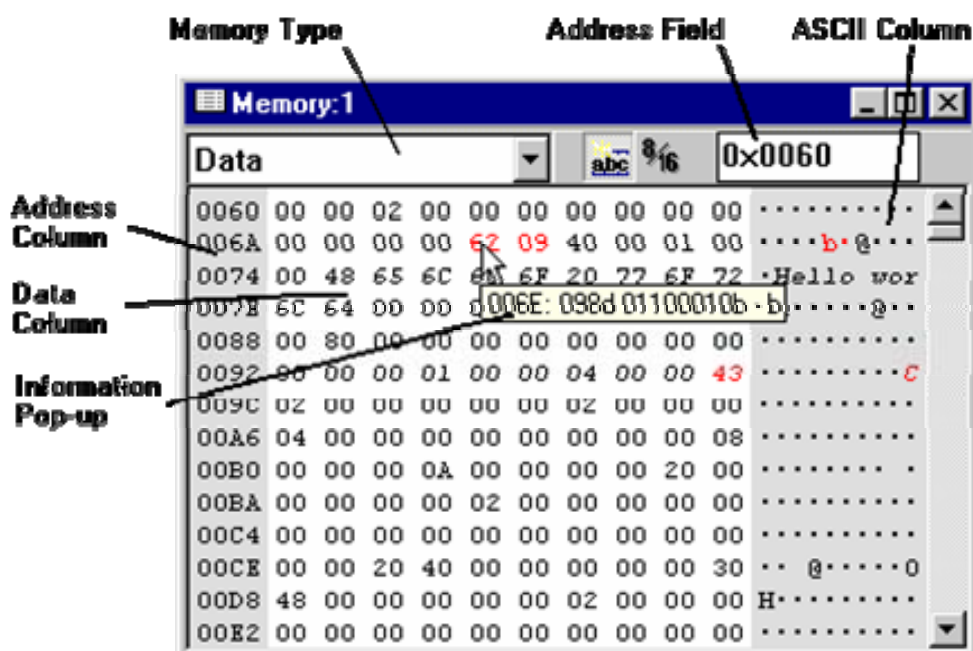


Рис. 9. Дамп памяти

При позиционировании курсора мыши на ячейку происходит вывод всплывающего окна дополнительной информации по данной позиции. Кроме того, вид дополнительной информации может задаваться пользователем.

Содержимое ячеек памяти может быть изменено путем непосредственной записи данных в ячейку. В стандартном режиме при каждом нажатии клавиши в память заносится введенная величина. Для предварительного набора данных с последующей записью в память дважды нажмите левую кнопку мыши для запуска окна редак-

тора. Данные, заносимые в окне редактора, не заносятся в память до тех пор, пока окно не закрыто. Отмена редактирования – клавиша Esc.

Использование окна редактора необходимо в тех случаях, когда редактируется содержимое портов ввода-вывода, поскольку каждая тетрада будет заноситься в порт не непосредственно, а в составе готового к записи байта (например, при инициализации UART).

В окне дампа памяти применяется следующая цветовая маркировка. Позиции, прошедшие отладку (Останов/Пошаговое выполнение), изображаются красным цветом. Имеются в виду только те позиции, которые отображены на дисплее в настоящий момент. Адреса ячеек и колонка представления данных в ASCII-формате всегда отображаются серым цветом. Когда программа обращается к двухбайтовому слову в памяти, его адрес изменяет цвет на голубой.

Можно просматривать дамп любого типа памяти (памяти данных, памяти программ, портов ввода-вывода и EEPROM-память). Тип памяти выбирается в верхней левой части окна дампа памяти. Ограничения на тип просматриваемой памяти вводятся при установке и идентификации используемого контроллера.

Окно Watch. В окне Watch отображаются символьные переменные, определенные пользователем (рис. 10). Одновременно может быть открыто не более одного окна Watch. Распределение символьных переменных по адресам происходит между сеансами. Окно Watch содержит четыре закладки, содержащие все переменные. Переключение закладок в нижней части окна Watch.

Watch	Value	Type	Address
<input type="checkbox"/> pCircBuf	0xCE	tiny *	(REG) R16
└─ <input type="checkbox"/> ->	{0xD3, 0x0A '□', 0x...	struct	(SRAM) 0x00CE
└─ <input type="checkbox"/> pBuffer	0xD3	tiny unsigned char*	(SRAM) 0x00CE
└─ ->	0x0A '□'	unsigned char	(SRAM) 0x00D3
└─ size	0x0A '□'	unsigned char	(SRAM) 0x00CF
└─ inPos	0x01 '□'	unsigned char	(SRAM) 0x00D0
└─ outPos	0x00 '\0'	unsigned char	(SRAM) 0x00D1
└─ free	0x09 '□'	unsigned char	(SRAM) 0x00D2
└─ c	0x0A '□'	unsigned char	(REG) R20

Watch1 Watch2 Watch3 Watch4

Рис. 10. Окно Watch

Окно Watch отображает как простые, так и сложные типы данных. Сложные типы данных, такие как массивы и структуры, изображаются в виде дерева. При таком подходе вся структура данных изображена на дисплее. Для того чтобы развернуть ветвь дерева, щелкните по значку «плюс». Окно Watch состоит из четырех колонок. В первой колонке отображаются символьные имена переменных, в следующей – размерность переменной, в третьей – тип и в четвертой – адрес. По умолчанию окно Watch пустое, т.е. те переменные, за которыми необходимо следить, нужно добавить в окно Watch. Если поместить указатель мыши на закладку окна, то всплывет информация о количестве установленных переменных. Окно Watch создается через меню *View > Watch* или нажатием кнопки *Watch Window* на панели инструментов *Views*. Для закрытия окна нажмите кнопку *Close Window* в правой верхней части окна. Каждое очередное нажатие *Watch* в меню или на панели инструментов включает и выключает окно. Простые переменные будут сохранены при закрытии или переключении окон, однако сложные структуры будут нарушены при выходе.

Существует несколько способов вставки символов в окно Watch: нажмите кнопку *Add Watch* на панели инструментов отладки или выберите *Add Watch* в меню Watch; нажмите *Enter* или *Insert* на клавиатуре или выберите *Add Watch* левой кнопкой в программном меню окна Watch; выберите символ в текущем окне и перетащите его с помощью мыши в нужное окно Watch.

Можно удалить текущую выбранную переменную из окна, нажав кнопку удаления на панели инструментов или воспользовавшись меню *Watch*, выбрав *Delete*. Если окно Watch является активным, можно просто нажать клавишу *Del*. Команда «очистить все» (*Delete All*) доступна через меню Watch. При выполнении команды данной команды очистки все переменные будут удалены из текущего окна Watch.

Изменение формата представления переменных происходит после двойного щелчка в области формата переменной. Поддерживаются двоичный *Ob*, десятичный *Od* и шестнадцатеричный *Oh*.

Для редактирования сложных типов данных разверните дерево и редактируйте каждый элемент.

Окно Project. В окне Project создаются новые проекты и открываются старые (рис. 11). В окне отображается информация о файлах, группах и микроконтроллерах, участвующих в проекте. Наименование проекта отображается в заголовке проекта. Функции, дос-

тупные проекту, зависят от типа проекта, созданного пользователем. Ниже даны общие характеристики проекта. Все функции программного меню окна можно вызвать через правую кнопку мыши.

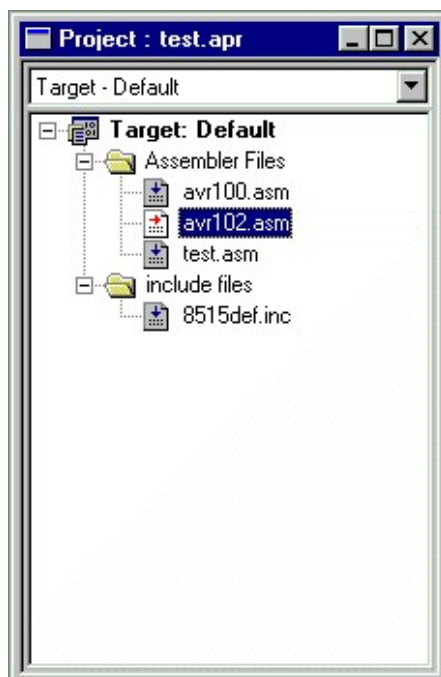


Рис. 11. Окно Project

Выберите группу, в которую необходимо добавить файл. После открытия окна файлового диалога дважды щелкните на требуемом файле. Для того чтобы добавить новый файл, введите имя несуществующего файла и нажмите «открыть» (*open*). Можно добавлять неограниченное число контроллеров. Каждый контроллер в проекте может иметь свою собственную выходную конфигурацию. В этом случае в основном используется компилятор проектов. В проекте может присутствовать неограниченное число групп. Это дает возможность удобного расположения файлов внутри больших проектов. Можно менять файлы в группах и удалять ненужные. Двойным щелчком на файле вызывается редактор. Первый файл, добавленный в проект, будет объявлен как загружаемый. Этот файл отображается как белая «иконка» с красной стрелкой. Изменить его положение можно посредством программного меню. Используйте INSERT для добавления новых файлов или DELETE – для удаления выбранного файла. Используйте клавиши перемещения для перемещения по таблице файлов. Нажмите ENTER для открытия выбранного файла.

При выполнении операции *Close* в меню File все открытые окна в текущей сессии будут закрыты. Также будет сохранен файл проекта в той директории, где хранятся объектные файлы проекта.

Объектный файл перезаписывается постоянно и, если произошли какие-либо изменения, появляется сообщение пользователю. В этом случае объектный файл может быть перезагружен (*Reload*).

Поддерживается копирование через буфер Windows. Если функция копирования *Copy* доступна в текущем окне, команда *Copy* присутствует на панели инструментов (правая кнопка мыши).

Меню отладчика Debug. *Сброс Reset (SHIFT+F5)*. Команда сброса эмулирует системный сигнал сброса Reset. Если команда дана в процессе выполнения программы, программа будет остановлена. После выполнения команды сброса информация во всех окнах будет обновлена.

Выполнить Go (F5). Команда Go в меню отладчика Debug начинает выполнение программы. Программа будет выполняться до тех пор, пока не поступит команда «стоп» от пользователя или не встретится точка останова.

Команда Go доступна только в режиме останова программы.

Прервать Break (CTRL-F5). Команда Break в меню отладчика Debug останавливает выполняющуюся программу. При остановке программы информация во всех окнах будет обновлена. Команда Break доступна только в режиме выполнения программы.

Трассировка Trace Into (F11). Команда Trace в меню отладчика Debug выполняет одну инструкцию программы. Когда AVR Studio находится в режиме source, будет выполнена одна инструкция этого режима и в режиме дизассемблера выполняется одна ассемблерная команда. Корректировка информации во всех окнах происходит после выполнения инструкции.

Полный шаг Step Over (F10). Команда Step Over в меню отладчика Debug также выполняет одну инструкцию. Однако если инструкция является командой вызова подпрограммы, то подпрограмма также будет выполнена. Если в процессе выполнения встретится контрольная точка, то выполнение будет остановлено. Корректировка информации во всех окнах происходит после выполнения инструкции.

Шаг до завершения Step Out (SHIFT+F11). Команда Step Out в меню отладчика Debug полностью выполняет текущую функцию. Если в процессе выполнения встретится контрольная точка, то выполнение будет остановлено. Если выполняемая функция является функцией высокого уровня в иерархии подпрограмм, то будут выполнены все нижележащие функции, причем выполнение будет происходить до ближайшей контрольной точки или до тех пор, пока не

будет дана команда останова. Корректировка информации во всех окнах происходит после выполнения инструкции.

Выполнить с текущей позиции курсора до указателя Run To Cursor (F7). Команда Run to Cursor в отладочном меню запускает программу на выполнение с текущей позиции курсора до команды в текущей позиции указателя команд. При использовании команды Run to Cursor точки останова, определенные пользователем, игнорируются. В случае заикливания программы необходимо остановить ее вручную. После выполнения программы информация во всех окнах будет приведена в соответствие. Команда доступна только в активном окне.

Трассировка заданного количества команд Multi Trace Into. Команда Multi Trace Into в меню отладчика Debug выполняет необходимое количество команд. В обычном режиме AVR Studio выполняются команды этого режима, а в режиме дизассемблера – дизассемблированные команды. Количество необходимых к выполнению команд задается в опциях (меню Options) отладчика. После выполнения заданного количества команд содержимое информационных окон будет обновлено (по умолчанию) или устанавливается после каждой команды. Команда Multi Trace Into будет выполняться до тех пор, пока не достигнет количества заданных команд, не будет прервана пользователем или не встретит точку останова.

Автоматическая трассировка Auto Trace Into. Команда Auto Trace в меню отладчика Debug выполняет команды в соответствии с определенным режимом, заданным в меню Options отладчика. В обычном режиме AVR Studio выполняются команды этого режима, а в режиме дизассемблера – дизассемблированные команды. После выполнения трассировки информация во всех окнах будет скорректирована. Задержка между трассируемыми командами задается в меню отладчика Options. Команда Auto Trace выполняется до тех пор, пока не будет остановлена пользователем или не встретит точку останова.

Установить контрольную точку (F9). Эта команда устанавливает контрольную точку останова для команды в текущей позиции курсора. Помните, что контрольная точка доступна только в активном окне.

Сбросить все контрольные точки (Clear all breakpoints). Эта команда сбрасывает все установленные контрольные точки, в том числе и точки, пройденные трассировщиком ранее.

Показать список контрольных точек Show list (Ctrl-B). При выборе инструкции Show list вызывается диалоговое окно показа

списка контрольных точек. В окне диалога пользователь может осуществлять контроль над точками, добавлять новые точки, удалять и активировать или деактивировать контрольные точки. Установленные контрольные точки отображаются коричневыми маркерами в левой части текущего окна.

Установить Toggle Trace and Trigger (F8). Эта команда добавляет точки трассировки в текущей позиции курсора в активном окне. Если точка ранее была установлена, то по F8 она будет удалена.

Очистить все (Clear all Trace & Trigger). Эта команда удаляет все точки Trace & Trigger.

Очистка памяти трассировки (Clear Trace Memory) – инициализация буфера трассировки.

Поиск в буфере трассировки (Search in Trace Memory). Эта команда открывает диалоговое окно поиска в буфере трассировки, поиск осуществляется до момента совпадения в предыстории отладки.

Add Watch (Ins). Для того чтобы добавить переменную, пользователь должен выбрать Add Watch из меню Watch или нажать кнопку Add Watch на панели Debug. Если окно Watch не присутствует, когда команда Add Watch дана, то окно Watch создается. Если окно Watch является активным, новые переменные также могут быть добавлены по нажатию клавиши Ins.

Delete Watch (Del). Пользователь может удалить переменную, сначала выделив удаляемый символ в окне Watch, затем дав команду Delete Watch из меню Watch или нажав кнопку на панели Debug. Выбор переменной осуществляется установкой курсора мыши к имени переменной и нажатием левой клавиши мыши. Если окно Watch является активным, то выделенный символ можно удалить нажатием клавиши Delete.

Delete All. Команда Delete all watches доступна из меню Watch. По этой команде все определенные переменные удаляются из окна Watch.

Все переменные в окнах Watch отображаются в шестнадцатеричном формате.

Клавиши быстрого вызова функций. В табл. 2 приведены клавиши быстрого вызова функций.

Клавиши быстрого вызова функций

Команда	Сочетание
Переключиться на окно Регистр	Alt+0
Переключиться на окно Слежения (Watch)	Alt+1
Переключиться на окно Сообщения	Alt+2
Переключиться на окно Процессор	Alt+3
Добавить окно памяти (Memory)	Alt+4
Добавить окно ввода/вывода (I/O)	Alt+5
Показать список контрольных точек (Breakpoints)	Ctrl+B
Копировать в буфер	Ctrl+C
Открыть файл	Ctrl+O
Искать следующий элемент	Ctrl+N
Перезагрузить	Ctrl+R
Искать в основном окне	Ctrl+S
Выполнить (Run)	F5
Останов (Break)	Ctrl+F5
Сброс (Reset)	Shift+F5
Ассемблировать	F7
Включить trace & trigger	F8
Включить контрольную точку	F9
По шагам поверх подпрограмм	F10
По шагам заходя в подпрограммы	F11

Практическая часть

Лабораторная работа № 1 Знакомство с ПО AVR Studio

Цель работы: изучение назначения и особенностей архитектуры однокристальных микроконтроллеров; ознакомление с архитектурой и программной моделью AVR-микроконтроллеров; изучение этапов разработки ПО для встраиваемых микропроцессоров; приобретение навыков работы в среде AVR Studio.

Теоретическая часть

AVR Studio 4 – профессиональная интегрированная среда разработки (Integrated Development Environment – IDE), предназначенная для написания и отладки прикладных программ для AVR микропроцессоров в среде Windows. AVR Studio 4 содержит ассемблер и симулятор.

В режиме симулятора AVR Studio позволяет программисту наблюдать за логикой выполнения программы, т.е. видеть содержимое регистров, памяти, портов, наблюдать за выполнением команд и т.д.

Особенность отладки ПО устройств на базе встраиваемых МП (в том числе однокристальных микроконтроллеров) состоит в отсутствии в их составе развитых средств для реализации пользовательского интерфейса и ограниченных возможностях системного ПО. В то же время именно для встраиваемых микропроцессорных систем этап отладки является чрезвычайно ответственным, так как для них характерна тесная взаимосвязь работы ПО и аппаратных средств.

Взаимодействие микропроцессора (микроконтроллера) с датчиками и исполнительными устройствами происходит путем передачи данных через регистры периферийных устройств (регистры ввода-вывода). Отдельные разряды таких регистров задают режимы работы периферийных устройств, имеют смысл готовности к обмену, завершения передачи данных и т.п. Состояние этих разрядов может устанавливаться как программно, так и аппаратно. При отладке ПО часто приходится переходить на уровень межрегистровых передач и проверять правильность установки отдельных разрядов. Кроме того, на этапе отладки может производиться оптимизация алгоритма, нахождение критических участков кода и проверка надежности разработанного ПО.

Для решения указанных задач применяются аппаратные и программные средства отладки ПО (рис. 12).

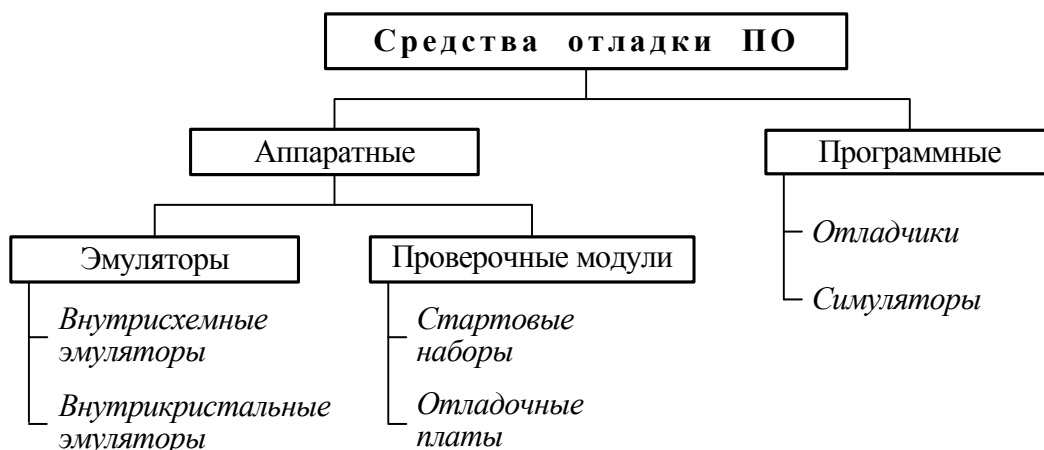


Рис. 12. Классификация средств отладки программного обеспечения

К **аппаратным средствам отладки** относятся аппаратные эмуляторы и проверочные модули.

Аппаратные эмуляторы предназначены для отладки программного и аппаратного обеспечения микропроцессорных систем в режиме реального времени. Они работают под управлением «ведущего» компьютера, оснащенного специальным ПО – программами-отладчиками (см. ниже). Основными видами аппаратных эмуляторов являются:

- внутрисхемные эмуляторы или эмуляторы-приставки, замещающие микропроцессор в отлаживаемой системе;
- внутрикристальные эмуляторы, представляющие собой одно из внутренних устройств микропроцессора.

Внутрисхемный эмулятор (In-Circuit Emulator, ICE) – это устройство, содержащее аппаратный имитатор процессора и схему управления имитатором. При отладке с помощью эмулятора микропроцессор извлекается из отлаживаемой системы, на его место подключается контактная колодка, количество и назначение контактов которой идентично выводам замещаемого микропроцессора (рис. 13). С помощью гибкого кабеля контактная колодка соединяется с эмулятором. Управление процессом отладки осуществляется с персонального компьютера. Эмуляторам-приставкам присущи следующие недостатки: высокая стоимость, недостаточная надежность, высокое энергопотребление, влияние на электрические характеристики цепей, к которым подключается эмулятор.

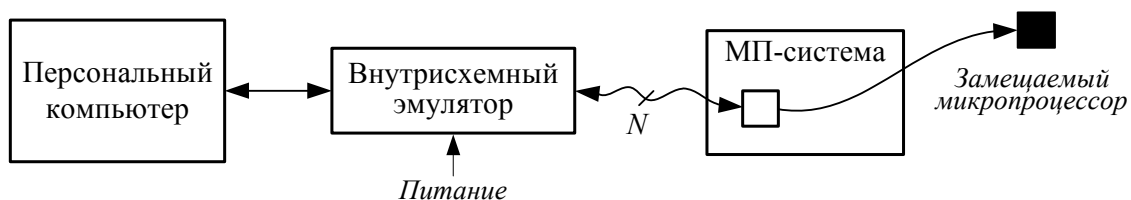


Рис. 13. Отладка с помощью внутрисхемного эмулятора:

N – количество выводов процессора

Внутрикристальные эмуляторы (On-Chip Emulator) позволяют проводить отладку программ без извлечения микропроцессора из системы. При этом осуществляется непосредственный контроль за выполнением программы, так как средства внутрикристальной отладки обеспечивают прямой доступ к регистрам, памяти и периферии микропроцессора. Наиболее распространенным средством внутрикристальной отладки является последовательный интерфейс **IEEE 1149.1**, известный как **JTAG** (Joint Test Action Group – Объединенная рабочая группа по автоматизации тестирования). Последовательный отладочный порт **JTAG** микропроцессора с помощью специального устройства сопряжения подключается к компьютеру, чем обеспечивается доступ к отладочным средствам процессора (рис. 14). Такой способ отладки также называют *сканирующей эмуляцией*. Достоинствами этого способа являются возможность выполнения различных действий на процессоре без его изъятия из системы, использование малого числа выводов процессора и поддержка его максимальной производительности без изменения электрических характеристик системы [3].

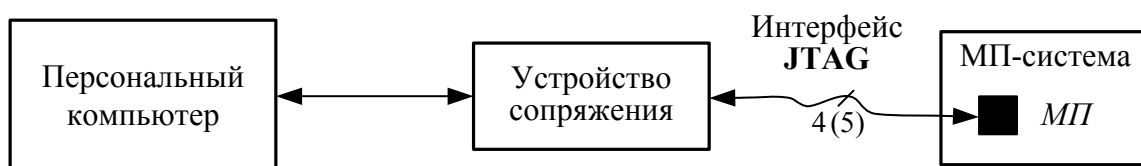


Рис. 14. Отладка с помощью внутрикристального эмулятора

Проверочные модули предназначены для быстрой отладки программного обеспечения в реальном масштабе времени. Проверочные модули бывают двух видов: стартовые наборы и отладочные платы.

Стартовые наборы (Starter Kit) предназначены для обучения работе с конкретным микропроцессором. Стартовый набор позволяет изучить характеристики микропроцессора, отладить не слишком

сложные программы, выполнить несложное макетирование, проверить возможность применения микропроцессора для решения конкретной задачи. В состав стартового набора входят плата, ПО и комплект документации. На плате устанавливаются микропроцессор, устройство загрузки программ, последовательные или параллельные порты, разъемы для связи с внешними устройствами и другие элементы. Плата подключается к компьютеру через параллельный или последовательный порт. Стартовые наборы удобны на начальном этапе работы с микропроцессором.

Отладочные платы (Evaluation Board) предназначены для проверки разработанного алгоритма в реальных условиях. Они позволяют проводить отладку и оптимизацию алгоритма с использованием установленной на плате периферии, а также изготовить на базе платы законченное устройство. Обычно на плате размещаются микропроцессор, схемы синхронизации, интерфейсы расширения памяти и периферии, схема электропитания и др. Плата подключается к компьютеру через параллельный или последовательный порт или непосредственно устанавливается в слот **PCI**.

Основными **программными средствами отладки** являются симуляторы и отладчики.

Симуляторы (simulator) или *симуляторы системы команд* представляют собой программы, имитирующие работу того или иного процессора на уровне его команд. Симуляторы обычно используются для проверки программы или ее отдельных частей перед испытанием на аппаратных средствах.

Отладчики (debugger) представляют собой программы, предназначенные для анализа работы созданного программного обеспечения. Можно указать следующие возможности отладчиков.

Пошаговое выполнение. Программа выполняется последовательно, команда за командой, с возвратом управления отладчику после каждого шага.

Прогон. Выполнение программы начинается с указанной команды и осуществляется без остановки до конца программы.

Прогон с контрольными точками. При выполнении программы происходит останов и передача управления отладчику после выполнения команд с адресами, указанными в списке контрольных точек.

Просмотр и изменение содержимого регистров и ячеек памяти. Пользователь имеет возможность выводить на экран и изменять (модифицировать) содержимое регистров и ячеек памяти.

Отладчики ПО встраиваемых микропроцессоров обычно используются совместно с внутрисхемными или внутрикристалльными эмуляторами, а также могут работать в режиме симулятора. Некоторые отладчики позволяют также выполнять *профилирование*, т.е. определять действительное время выполнения некоторого участка программы. Иногда функцию профилирования выполняет специальная программа – *профилировщик* (profiler).

Средства отладки ПО AVR-микроконтроллеров. Аппаратные средства отладки программного обеспечения AVR-микроконтроллеров представлены внутрисхемным эмулятором **ICE50**, внутрикристалльным эмулятором **JTAG ICE**, а также стартовым набором **STK500**.

К программным средствам отладки ПО AVR-микроконтроллеров относятся отладчик и симулятор, входящие в состав среды **AVR Studio**. Отладчик среды **AVR Studio** позволяет проводить отладку программ как в исходных кодах (например, ассемблера), так и в кодах дизассемблера (оттранслированной или скомпилированной программы, записанной с помощью мнемоник ассемблера). Вызов окна с кодом дизассемблера производится командой Disassembler меню View или командой Goto Disassembly контекстного меню редактора исходного текста. Обратное переключение в окно исходного текста осуществляется командой Goto Source контекстного меню окна **Disassembler**.

Отладчик среды **AVR Studio** может использоваться с внутрисхемным эмулятором **ICE50**, внутрикристалльным эмулятором **JTAG ICE**, отладочной платой **STK500** или симулятором. Указание способа отладки производится при создании проекта. Симулятор среды **AVR Studio** предназначен для предварительной отладки программ без применения аппаратных средств. В дальнейшем в настоящем лабораторном практикуме для отладки создаваемых программ предполагается применение отладчика среды **AVR Studio** в режиме симулятора.

Отладка ПО в среде AVR Studio. Команды отладчика в программе **AVR Studio** находятся в меню Debug.

Переход в режим отладчика в среде **AVR Studio** осуществляется автоматически при использовании для трансляции программы команды Build and Run или командой Start Debugging меню Debug при использовании для трансляции команды Build. Выход из режима отладчика производится командой Stop Debugging меню Debug.

Пошаговое выполнение программы задается командами Step Into, Step Over меню Debug. Команда Step Into позволяет выполнить одну

команду программы (в том числе команду вызова подпрограммы). Для завершения выполнения подпрограммы может использоваться команда Step Out. Команда Step Over также выполняет одну команду программы, но если это команда вызова подпрограммы, последняя полностью выполняется за один шаг. Следующая выполняемая команда (команда, адрес которой содержится в программном счетчике) обозначается символом ➡ в окне исходного текста программы. Сброс выполнения программы осуществляется с помощью команды Reset.

Прогон (запуск или продолжение выполнения) программы осуществляется командой Run. Для остановки выполнения программы служит команда Break.

Контрольные точки представляют собой специальные маркеры для программы-отладчика и могут быть трех типов: точки останова, точки трассировки и точки наблюдения.

Точки останова задаются командой Toggle Breakpoint меню Debug или контекстного меню редактора исходного текста программы. Точка останова обозначается в редакторе исходного текста символом слева от помечаемой строки. Просмотреть заданные точки останова можно на закладке **Breakpoints** окна **Output**; там же точки останова могут быть запрещены (путем сброса флажка напротив точки останова) и разрешены (путем установки флажка). При достижении точки останова во время прогона программы ее выполнение приостанавливается. Повторный вызов команды установки точки останова на той же строке программы приводит к удалению точки останова. Удалить все заданные точки останова позволяет команда Remove Breakpoints меню Debug или команда Remove all Breakpoints контекстного меню закладки **Breakpoints** окна **Output**. Параметры точки останова задаются в диалоговом окне **Breakpoint Condition**, вызов которого осуществляется командой Breakpoints Properties контекстного меню редактора исходного текста программы. Установка флажка **Iterations** позволяет задать количество итераций (повторных выполнений) команды до останова прогона программы. При установке флажка **Watchpoint** по достижению точки останова производится только обновление значений регистров и ячеек памяти в окнах просмотра. Флажки **Iterations** и **Watchpoint** не должны устанавливаться одновременно. Установка флажка **Show message** обеспечивает отображение сообщений о достижении точки останова на закладке **Breakpoints** окна **Output**. Вызов диалогового окна задания свойств и удаление точки останова могут быть произведены из контекстного меню закладки **Breakpoints** окна **Output**.

Точки трассировки предназначены для контроля выполнения программы в режиме реального времени. Трассировка позволяет отслеживать так называемую трассу программы – изменение содержимого регистров и ячеек памяти при выполнении определенных команд (команд, по адресам которых заданы точки трассировки). В среде **AVR Studio** функция трассировки может использоваться только при отладке программы с применением внутрисхемного эмулятора; при работе в режиме симулятора функция трассировки недоступна.

Точки наблюдения задаются командой Add to Watch контекстного меню редактора исходного текста программы. Точки наблюдения представляют собой символические имена регистров или ячеек памяти, содержимое которых необходимо отслеживать. При выполнении команды Add Watch на экране появляется окно **Watches**, разделенное на четыре столбца: **Name** (символическое имя точки наблюдения), **Value** (значение), **Type** (тип), **Location** (местонахождение). Новая точка наблюдения может быть также задана в выделенной ячейке столбца **Name** окна **Watches** или командой Quickwatch в окне редактора исходного текста программы (при этом курсор должен находиться на имени регистра или ячейки памяти). Значения, отображаемые в столбце **Value**, обновляются при изменении содержимого соответствующего регистра или ячейки памяти. Удалить заданные точки наблюдения можно из окна **Watches**.

Отладчик среды **AVR Studio** также обеспечивает следующие функции: выполнение до курсора (команда Run to Cursor меню Debug) и последовательное выполнение команд с паузами между ними (команда Auto Step меню Debug).

Для удобства использования в процессе отладки ряд команд отладчика доступен с клавиатуры (табл. 3).

Таблица 3

Команда отладчика	Клавиша	Команда отладчика	Клавиша
Run	F5	Step Into	F11
Break	Ctrl+F5	Step Out	Shift+F11
Reset	Shift+F5	Step Over	F10
Run to Cursor	Ctrl+F10	Toggle Breakpoint	F9

Для просмотра и изменения содержимого регистров и ячеек памяти служат команды Registers, Memory, Memory 1, Memory 2, Memory 3 меню View.

По команде Registers на экране отображается окно **Registers**, в котором приводятся шестнадцатеричные представления содержи-

мого РОН. Изменение (модификация) содержимого регистров производится путем двойного щелчка мышью. Наблюдение за содержимым РОН может быть также произведено с помощью дерева устройств микроконтроллера, находящегося на закладке **I/O** окна **Workspace**. Для этого необходимо раскрыть объекты **Register 0-15** и **Register 16-31** щелчком мыши по знаку «+».

Команды **Memory**, **Memory 1**, **Memory 2**, **Memory 3** обеспечивают вызов окон **Memory**, служащих для отображения содержимого ячеек оперативной и энергонезависимой памяти данных, памяти программ, регистров ввода-вывода и РОН. Выбор типа памяти, отображаемой в окне **Memory**, производится с помощью списка, расположенного в панели управления окна (**Data** – оперативная память данных, **Eeprom** – энергонезависимая память данных, **I/O** – регистры ввода-вывода, **Program** – память программ, **Register** – РОН).

Для наблюдения за состоянием процессора необходимо раскрыть объект **Processor** закладки **I/O** окна **Workspace**. При этом будет отображена следующая информация: содержимое программного счетчика (**Program Counter**); содержимое указателя стека (**Stack Pointer**), количество тактов, прошедших с начала выполнения (**Cycle Counter**); содержимое 16-разрядных регистров-указателей **X**, **Y** и **Z**; тактовая частота (**Frequency**); затраченное на выполнение время (**Stop Watch**).

Для контроля содержимого регистров ввода-вывода необходимо раскрыть объект **I/O * закладки I/O** окна **Workspace**, где * – тип микроконтроллера. Регистры ввода-вывода, входящие в объект **I/O**, сгруппированы по типам периферийных устройств.

Модифицированные значения содержимого регистров и ячеек памяти действуют только во время текущего сеанса отладки, в исходный текст программы изменения не заносятся.

Практическая часть

Запустим AVR Studio, выберем режим создания нового проекта (**New Project**).

Создание первого проекта

Для более полного знакомства напомним простую программу. Для этого создаем новый проект (в верхнем меню **Project\new project** или в стартовом окне **New Project**) (рис. 15). В открывшемся окне в графе **Project type** выберем тип проекта **Atmel AVR Assembler**, в графе **Project name** указываем имя нового проекта, в строке **Location**

необходимо указать путь к будущему проекту (путь не должен быть слишком длинным, содержать пробелов и иметь русских символов) также необходимо отметить галочкой **Create folder** (тогда проект будет создан в отдельной папке) (рис. 16). Нажимаем «**Finish**». Если нажать «**Next**», то перед нами откроется окно выбора устройства, под которое мы будем писать программу.

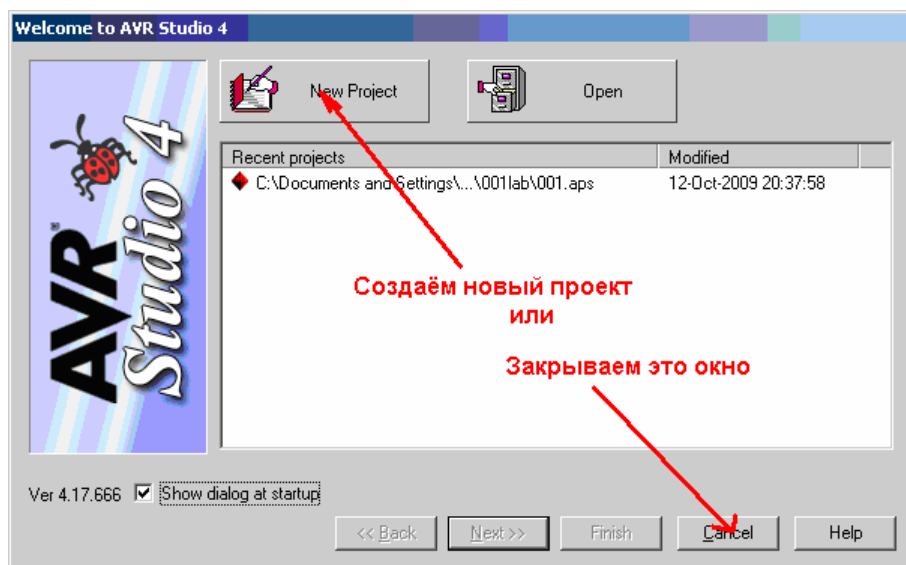


Рис. 15. Стартовое окно

Первую программу напишем для микроконтроллера Atmega16.

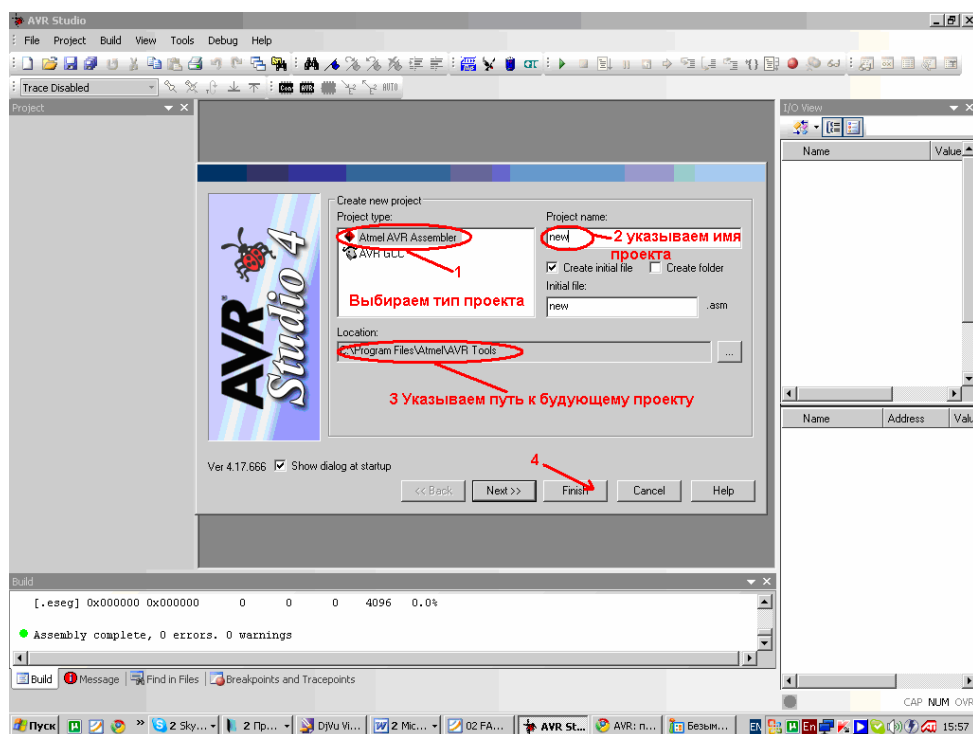


Рис. 16. Окно AVR Studio

AVR Studio в режиме отладки программы

Основными функциями AVR Studio являются функция разработки и функция отладки программ для микроконтроллеров фирмы Atmel. С функцией разработки мы уже познакомились, теперь запустим режим отладки. Для этого:

1. Запускаем AVR Studio (если еще не запустили).
2. Создаем новый проект.
3. Напишем простейшую программу для ATmega16 и откомпилируем ее, для этого в AVR Studio в верхнем меню необходимо открыть вкладку **Build** и нажать опять же на **build** или можно воспользоваться горячей клавишей **F7**.

После компиляции, если программа написана верно, в логе событий мы увидим сообщение: «Assembly complete, 0 errors. 0 warnings».

;пример для эмуляции программы в AVR Studio

.include "m16def.inc";подключение библиотеки

.list;включение листинга

.def temp0=r16;определение рабочих регистров

.def temp1=r17

.def temp2=r18

.def temp3=r19

;-----

metka:

ldi temp0,0x00;записываем ноль в регистр temp0

ldi temp0,0xFF;записываем 0xff в регистр temp0

ldi temp1,0x00;записываем ноль в регистр temp1

ldi temp1,0xAA;записываем 0xAA в регистр temp1

ldi temp2,0x00;записываем ноль в регистр temp2

ldi temp2,0xCC;записываем 0xCC в регистр temp2

ldi temp3,0x00;записываем ноль в регистр temp3

mov temp3,temp2;пересылка данных из temp2 в temp3

rjmp metka;переход к метке

4. Настроим эмулятор.

Для этого в верхнем меню откроем вкладку **Debug** и выберем **select platform and device...**, в появившемся окне выберем платформу **AVR simulator** и устройство **ATmega16** после чего нажмем **Finish** (рис. 17).

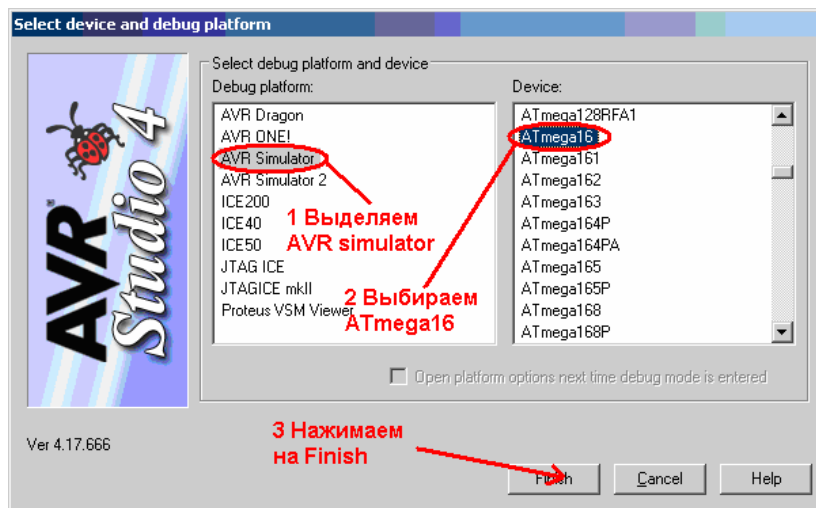


Рис. 17. Окно выбора целевого устройства

5. Запустим эмуляцию, нажав **Start debugging** в той же вкладке **Debug**.

*Эмуляция в AVR Studio – пошаговое выполнение команд с возможностью контроля их выполнения непосредственно в структуре процессора.

В окне **I/O view** вы можете контролировать состояние портов, памяти и т.д., а в окне **Processors** видеть как изменяются значения регистров и следить за процессом эмуляции (рис. 18).

Желтая стрелка, находящаяся слева от команды, показывает, что именно эта команда будет выполнена при следующем шаге эмуляции.

6. Начнем выполнение программы, для этого нажимаем на **step Into(F11)** во вкладке **Debug**.

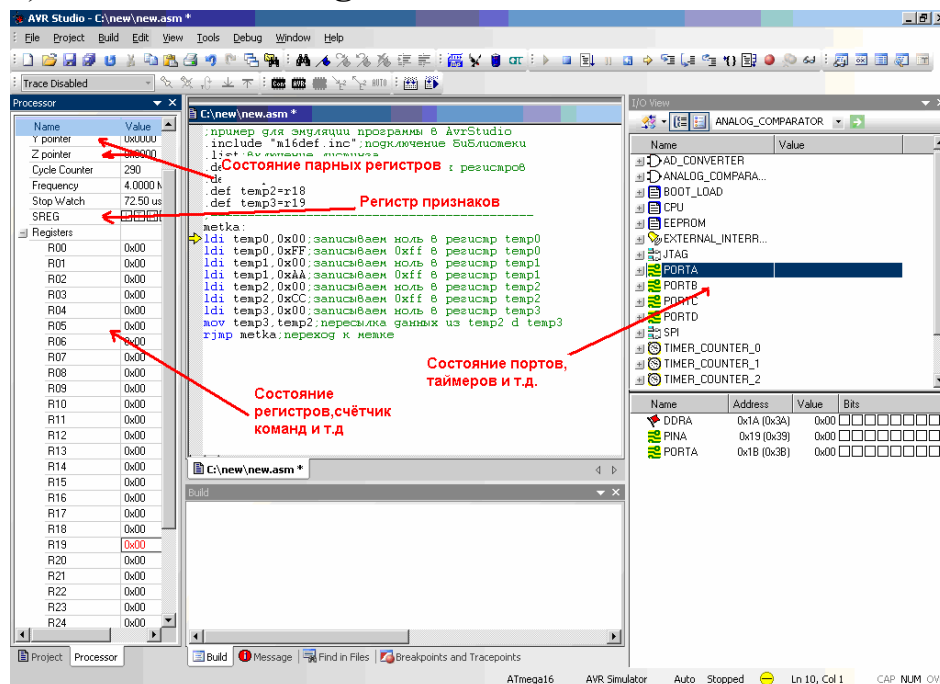


Рис. 18. Режим отладки

Так как в нашей программе мы изменяем только состояние регистров, то за ними мы и наблюдаем, таким образом мы можем контролировать выполнение программы, и если что-то пойдет не так, сможем быстро устранить ошибку.

6.1. Выполнить программу в пошаговом режиме, отслеживая изменение содержимого используемых в программе регистров. Обратить внимание на изменение содержимого программного счетчика. Сравнить содержимое программного счетчика при выполнении команд с их адресами в памяти программ, приведенными в листинге трансляции и окне памяти программ.

6.2. Выполнить прогон программы. Проверить правильность результата работы программы.

6.3. Задать точку останова на команде загрузки в РОН числа 0хСС. Включить режим отображения сообщений о достижении точки останова. Выполнить прогон программы с контрольными точками. Задать точку останова на команде умножения. Выполнить прогон программы с контрольными точками. Удалить заданные точки останова.

6.4. Задать точки наблюдения в используемых РОН. Выполнить программу в пошаговом режиме, отслеживая изменение их содержимого.

Содержание отчета

Отчет должен содержать титульный лист с указанием номера и названия лабораторной работы, номера группы и фамилий выполнивших работу; цель работы; схему программной модели AVR-микроконтроллера; перечень этапов разработки прикладного ПО для встраиваемых МП (МК); распечатку листинга трансляции созданной программы с расшифровкой одной из строк.

Контрольные вопросы

1. Каково назначение однокристальных микроконтроллеров?
2. Какие особенности архитектуры однокристальных микроконтроллеров вы знаете?
3. Опишите архитектуру и программную модель AVR-микроконтроллеров.
4. Какие этапы разработки ПО для встраиваемых микропроцессоров?
5. Каков формат строки программы на ассемблере для AVR-микроконтроллеров?
6. Какой состав листинга трансляции?

Лабораторная работа № 2

Способы адресации операндов

Цель работы: изучение способов адресации операндов в AVR-микроконтроллерах; сравнение различных способов адресации по быстродействию и размеру программного кода.

Теоретическая часть

В зависимости от количества используемых операндов возможны три типа команд AVR-микроконтроллера: безадресные, одноадресные и двухадресные. В первом типе команд присутствует только код операции (КОП), определяющий выполняемую командой функцию. В командах второго и третьего типов помимо кода операции содержится адресная часть, устанавливающая способ доступа соответственно к одному или двум участвующим в команде операндам (аргументам команды). Способ формирования адреса операнда, указание на который содержится в команде, называется *адресацией* (addressing). С помощью того или иного способа адресации вычисляется физический адрес, подающийся на шину адреса процессора для выбора ячейки памяти или регистра, используемых в команде.

В соответствии с типом адресуемой памяти способы адресации в AVR-микроконтроллерах можно разделить на способы адресации РОН и регистров ввода-вывода, способы адресации оперативной памяти данных (ОЗУ) и способы адресации памяти программ. Возможность использования различных способов адресации позволяет сократить размер и время выполнения программ.

Для адресации РОН и регистров ввода-вывода предусмотрен всего один режим – прямая регистровая адресация.

При *прямой регистровой адресации РОН* операндом является содержимое регистра общего назначения, указанного в команде. Команды с прямой регистровой адресацией могут адресовать один (**Rd**) или два (**Rr** и **Rd**) РОН (рис. 19; взаимное расположение КОП и адресной части в разрядах команды здесь и далее показано условно). Во втором случае результат выполнения команды сохраняется в регистре **Rd**. Прямая регистровая адресация РОН применяется во всех арифметических и логических командах, а также в некоторых командах работы с битами, так как эти команды выполняются в АЛУ только над содержимым РОН. Команды, вторым операндом которых является константа, могут использовать в качестве первого операнда только регистры из старшей половины РОН (**R16...R31**).

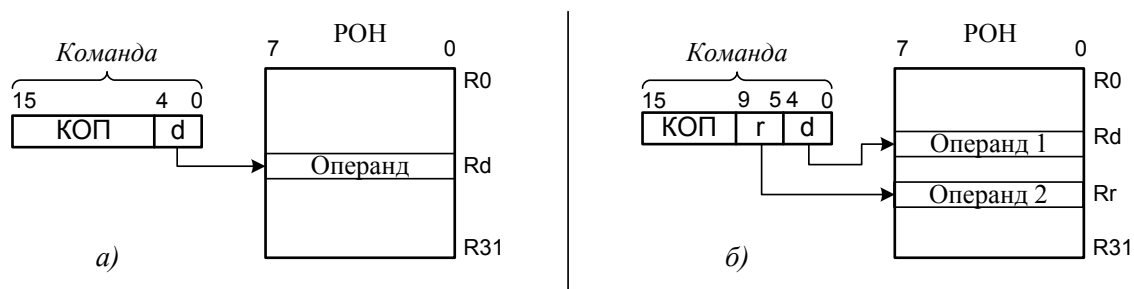


Рис. 19. Прямая регистровая адресация одного (а) и двух (б) РОН

При *прямой регистровой адресации регистра ввода-вывода* операнд содержится в регистре ввода-вывода, указанном в команде. Адрес регистра ввода-вывода хранится в шести разрядах слова команды (рис. 20, на котором **n** определяет адрес регистра-источника или регистра-приемника в РОН). Прямая регистровая адресация регистров ввода-вывода используется в командах чтения **IN** и записи **OUT** регистра ввода-вывода, а также в ряде других команд работы с регистрами ввода-вывода.

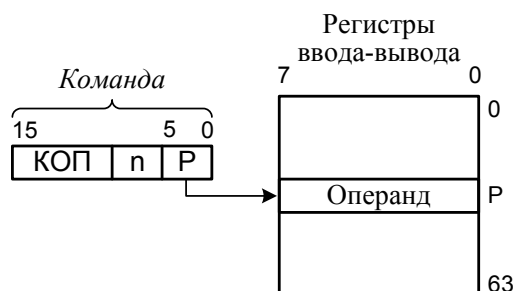


Рис. 20. Прямая регистровая адресация регистра ввода-вывода

Примеры использования прямой регистровой адресации:

; прямая регистровая адресация одного РОН

CLR R1 ; очистка всех разрядов регистра R1

; прямая регистровая адресация двух РОН

ADD R11, R12 ; сложение содержимого регистров R11 и R12

Для **адресации оперативной памяти данных** используются пять способов адресации: непосредственная, косвенная, косвенная со смещением, косвенная с предкрементом и косвенная с постинкрементом.

1. При *непосредственной адресации оперативной памяти данных* операндом является содержимое ячейки ОЗУ, адрес которой ука-

зан в команде. Адрес операнда содержится в 16 младших разрядах 32-разрядной команды (рис. 21, на котором **Rr/Rd** определяет адрес регистра-источника или регистра-приемника в POH). Непосредственная адресация используется в команде **LDS** (Load Direct from Data Space) загрузки из ОЗУ и в команде **STS** (Store Direct to Data Space) загрузки в ОЗУ. Зарезервировать байты в ОЗУ позволяет директива **.byte**. Для того чтобы на выделенную область памяти можно было ссылаться, директиве **.byte** должна предшествовать метка. Директива **.byte** имеет один параметр – количество выделяемых байт – и может использоваться только в сегменте данных, определяемом с помощью директивы **.dseg** (data segment). Задать требуемое размещение выделяемой области памяти позволяет директива **.org** (origin – смещение). Начало программного сегмента указывается с помощью директивы **.cseg** (code segment). Директивы **.dseg** и **.cseg** не имеют параметров. Выделенные в оперативной памяти данные байты не инициализируются.

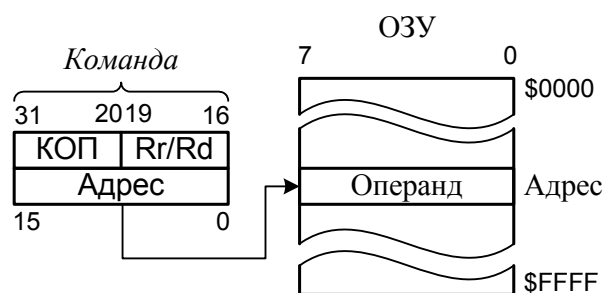


Рис. 21. Непосредственная адресация ячейки оперативной памяти данных

Пример использования непосредственной адресации:

```
; непосредственная адресация
.dseg                                ; сегмент данных (оперативная память
данных)
.org $0065                          ; по адресу $0065
cnt1:    .byte 1                    ; резервирование 1 байта для cnt1

.cseg                                ; программный сегмент (память про-
грамм)
;...
LDS R10, cnt1    ; загрузка cnt1 в R10
```

2. При *косвенной адресации оперативной памяти данных* операндом является содержимое ячейки ОЗУ, адрес которой находится в регистре **X**, **Y** или **Z** (рис. 22). Косвенная адресация используется в команде **LD** (Load Indirect) косвенной загрузки из ОЗУ и команде **ST** (Store Indirect) косвенной загрузки в ОЗУ.

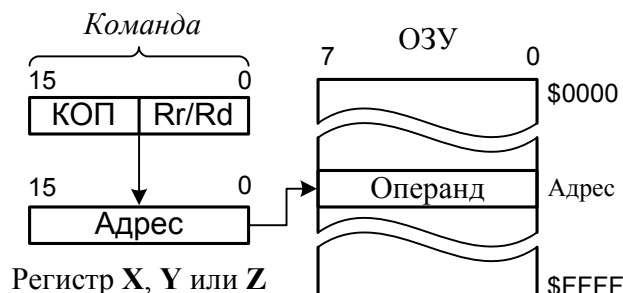


Рис. 22. Косвенная адресация оперативной памяти данных

Пример использования косвенной адресации:

; косвенная адресация

.dseg ; сегмент данных (оперативная память данных)

cnt1: .byte 1 ; резервирование 1 байта для cnt1

.cseg ; программный сегмент (память программ)

;...

LDI R30, low(cnt1) ; загрузка в R30 младшего байта адреса cnt1

LDI R31, high(cnt1) ; загрузка в R31 старшего байта адреса cnt1

LD R1, Z ; загрузка cnt1 в R1, т. е. R1 <- (R31:R30)

В приведенном примере использованы функции **low** и **high**, возвращающие соответственно младший и старший байты выражения (в данном случае – адреса ячейки памяти с символическим именем **cnt1**).

3. При *косвенной адресации оперативной памяти данных со смещением* адрес операнда в оперативной памяти данных вычисляется путем прибавления к содержимому регистра **Y** или **Z** смещения, указанного в команде. Смещение содержится в шести разрядах слова команды (рис. 23). Косвенная адресация со смещением используется в команде **LDD** (Load Indirect with Displacement) косвенной загрузки из ОЗУ со смещением и в команде **STD** (Store Indirect with Displacement) косвенной загрузки в ОЗУ со смещением.

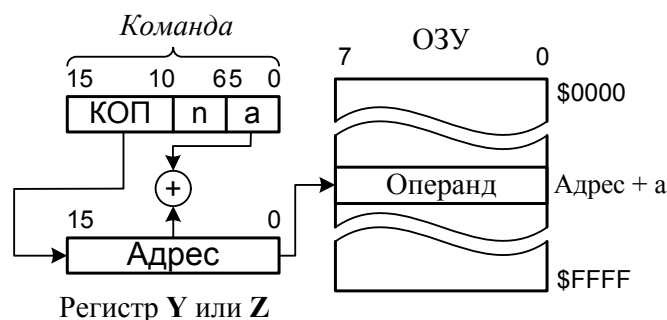


Рис. 23. Косвенная адресация оперативной памяти данных со смещением

Пример использования косвенной адресации со смещением:

; косвенная адресация со смещением

.dseg ; сегмент данных (оперативная память данных)

arr: .byte 5 ; резервирование 5 байт для массива arr

.cseg ; программный сегмент (память программ)

LDI R30, low(arr) ; загрузка в R30 младшего байта адреса arr

LDI R31, high(arr) ; загрузка в R31 старшего байта адреса arr

;...

LDD R10, Z + 0 ; загрузка первого элемента массива arr в R10

LDD R11, Z + 1 ; загрузка второго элемента массива arr в R11

4. При *косвенной адресации оперативной памяти данных с предекрементом* (лат. *decrementum* – уменьшение, убыль) перед выполнением команды содержимое указанного в команде регистра **X**, **Y** или **Z** декрементируется (уменьшается на единицу); декрементированное содержимое регистра **X**, **Y** или **Z** является адресом операнда в оперативной памяти данных (рис. 24). Косвенная адресация с предекрементом используется в команде **LD** косвенной загрузки из ОЗУ и команде **ST** косвенной загрузки в ОЗУ.

; косвенная адресация с предекрементом

LD R10, -Z ; $Z \leftarrow Z - 1$, $R10 \leftarrow (Z)$

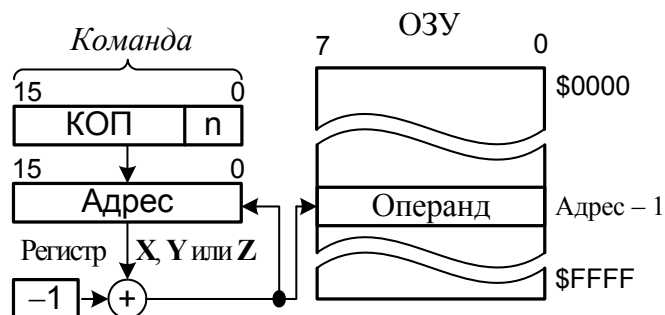


Рис. 24. Косвенная адресация оперативной памяти данных с предекрементом

5. При *косвенной адресации оперативной памяти данных с постинкрементом* (лат. *incrementum* – увеличение, рост) адресом операнда в оперативной памяти данных является содержимое регистра **X**, **Y** или **Z**, указанного в команде; после выполнения команды содержимое регистра **X**, **Y** или **Z** инкрементируется, т. е. увеличивается на единицу (рис. 25). Косвенная адресация с постинкрементом используется в команде **LD** косвенной загрузки из ОЗУ и команде **ST** косвенной загрузки в ОЗУ. Например:

; косвенная адресация с постинкрементом
LD R10, Z+ ; R10 <- (Z), Z <- Z + 1

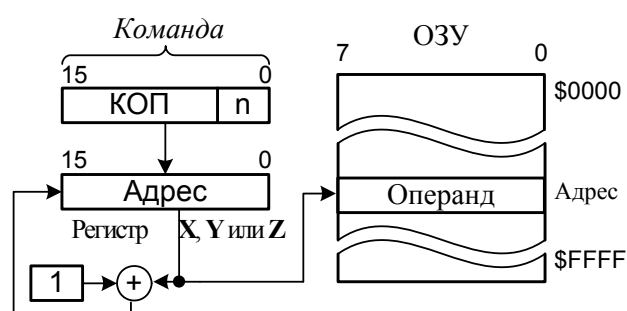


Рис. 25. Косвенная адресация оперативной памяти данных с постинкрементом

Для **адресации памяти программ** используется непосредственная адресация, косвенная адресация, относительная адресация, адресация константы и адресация константы с постинкрементом.

При *непосредственной адресации памяти программ* выполнение программы продолжается с адреса, указанного в команде (рис. 26; **FLASHEND** – символическое имя адреса последней ячейки памяти программ). Непосредственная адресация памяти программ используется в командах **JMP** и **CALL**.

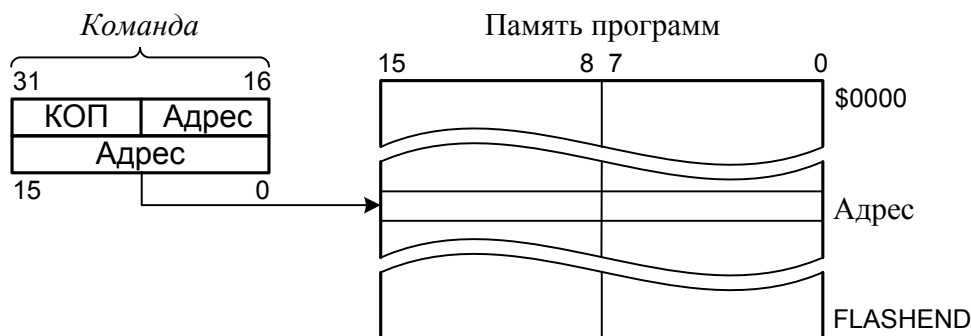


Рис. 26. Непосредственная адресация памяти программ

При *косвенной адресации памяти программ* выполнение программы продолжается с адреса, содержащегося в регистре **Z**, т.е. в программный счетчик загружается содержимое регистра **Z** (рис. 27). Косвенная адресация памяти программ используется в командах **IJMP** и **ICALL**.

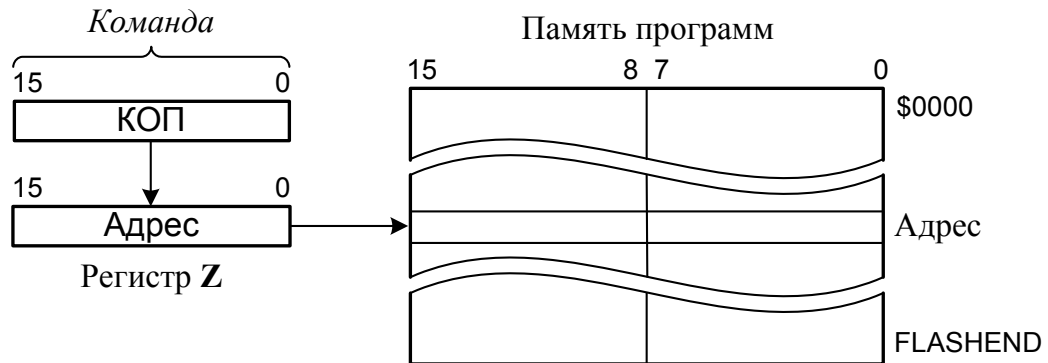


Рис. 27. Косвенная адресация памяти программ

При *относительной адресации памяти программ* выполнение программы продолжается с адреса $(PC + k + 1)$, где **PC** – содержимое программного счетчика; **k** – указанный в команде относительный адрес, который может принимать значения от -2048 до 2047 (рис. 28). Относительная адресация памяти программ используется в командах **RJMP** и **RCALL**.

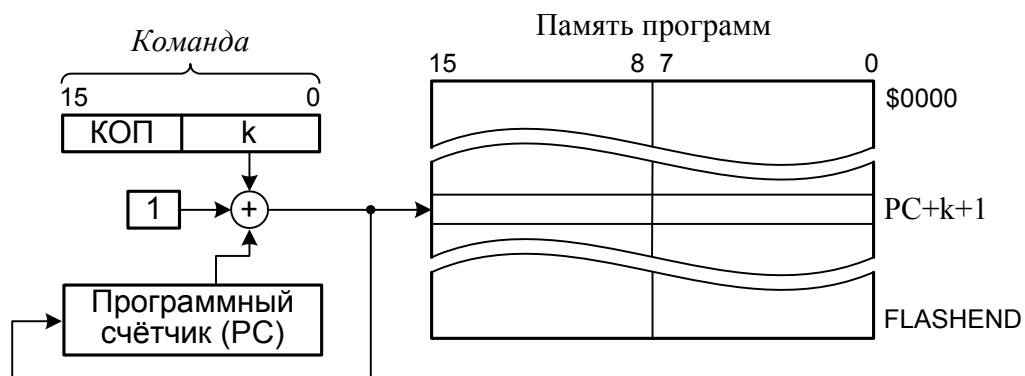


Рис. 28. Относительная адресация памяти программ:
k – относительный адрес

При *адресации константы* адрес байта константы содержится в регистре **Z** (рис. 29): старшие 15 разрядов занимает адрес ячейки памяти программ (от 0 до 32 К); состояние младшего разряда (LSB) определяет выбор младшего (LSB = 0) или старшего (LSB = 1) байта ад-

ресуемой ячейки памяти. Адресация константы в памяти программ используется в командах **LPM** (Load Program Memory), которая загружает адресованный регистром **Z** байт в указанный в команде регистр. Если регистр-приемник не указан (команда используется без операндов), байт загружается в регистр **R0**. Команда **ELPM** (Extended Load Program Memory) служит для загрузки константы из памяти программ объемом более 64 К слов. При этом для расширения регистра-указателя **Z** используется регистр **RAMPZ**, связанный с регистром **Z**.

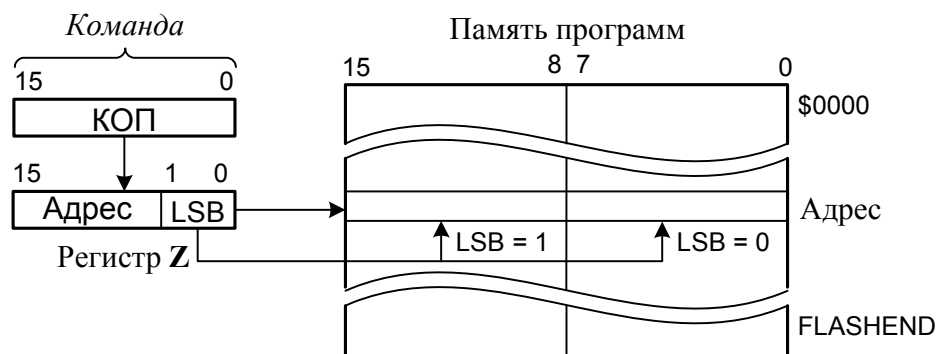


Рис. 29. Адресация константы в памяти программ

Задать данные в память программ позволяет директива **.db** (define bytes). Для того чтобы на заданные ячейки памяти можно было ссылаться, директиве должна предшествовать метка. Параметры директивы – последовательность выражений, разделенных запятыми; каждое выражение должно быть числом в диапазоне $-128 \dots 255$ или в результате вычисления давать результат в этом же диапазоне, в противном случае число усекается до байта. Директива **.db** размещается в программном сегменте и может использоваться совместно с директивой **.org**. Задавать положение данных в памяти программ следует таким образом, чтобы была исключена возможность непреднамеренного перехода к выполнению их как команд микроконтроллера.

Пример использования адресации константы в памяти программ:

```
LDI R31, high(var<<1) ; старший байт регистра Z
LDI R30, low(var<<1)  ; младший байт регистра Z
LPM R16, Z             ; загрузка числа $50 в R16
;...
.org $0025              ; по адресу $0025
var: .db $50, 137       ; константы $50 и 137
```

В приведенном примере использован оператор \ll , выполняющий сдвиг операнда влево на указанное число разрядов.

При *адресации константы в памяти программ с постинкрементом* адрес байта константы содержится в регистре **Z** и формируется так же, как и при адресации константы (рис. 29). Адресованный регистром **Z** байт загружается в указанный регистр; после выполнения команды содержимое регистра **Z** инкрементируется. Адресация константы в памяти программ с постинкрементом используется в командах **LPM** и **ELPM**. Например:

```
LDI R31, high(var<<1) ; старший байт регистра Z
LDI R30, low(var<<1)  ; младший байт регистра Z
LPM R16, Z+           ; загрузка числа $50 в R16, инкремент Z
LPM R17, Z            ; загрузка числа 137 в R17
;...
var: .db $50, 137      ; константы $50 и 137
```

Практическая часть

Составить программу сложения двух целых 8-разрядных чисел:

1) с использованием *прямой регистровой адресации РОН*. Результат сложения в этом и последующих пунктах задания сохранить в РОН. Значения операндов взять из задания лабораторной работы № 1 (числа **A** и **B**);

2) с использованием *непосредственной адресации оперативной памяти данных*. Для этого зарезервировать в ОЗУ байты под слагаемые с помощью директив **.byte**. Занести слагаемые в зарезервированные ячейки ОЗУ командой **STS** с непосредственной адресацией. Сложить операнды, предварительно загрузив их в РОН командой **LDS** с непосредственной адресацией;

3) с использованием *косвенной адресации оперативной памяти данных*. Адреса слагаемых в ОЗУ загрузить в регистры **X** и **Y** с помощью команды **LDI** и функций **low** и **high**. Занести слагаемые в зарезервированные ячейки ОЗУ командой **ST** с косвенной адресацией. Сложить операнды, предварительно загрузив их в РОН командой **LD** с косвенной адресацией;

4) с использованием *косвенной адресации оперативной памяти данных со смещением*. Для этого зарезервировать в ОЗУ байты под слагаемые в одной директиве **.byte**. Адрес начала блока данных за-

грузить в регистр **Z** с помощью команды **LDI** и функций **low** и **high**. Занести слагаемые в зарезервированные ячейки ОЗУ командой **STD** с косвенной адресацией со смещением. Сложить операнды, предварительно загрузив их в РОН командой **LDD** с косвенной адресацией со смещением;

5) с использованием *косвенной адресации оперативной памяти данных с предекрементом*. Для этого зарезервировать в ОЗУ байты под слагаемые в одной директиве **.byte**. В регистр **Z** загрузить адрес ячейки ОЗУ, следующей за блоком зарезервированных байтов. Занести слагаемые в зарезервированные ячейки ОЗУ командой **ST** с косвенной адресацией с предекрементом. Сложить операнды, предварительно загрузив их в РОН командой **ld** с косвенной адресацией ячеек ОЗУ с предекрементом;

6) с использованием *косвенной адресации оперативной памяти данных с постинкрементом*. Для этого зарезервировать в ОЗУ байты под слагаемые в одной директиве **.byte**. В регистр **Z** загрузить адрес начала блока зарезервированных байтов. Занести слагаемые в зарезервированные ячейки ОЗУ командой **ST** с косвенной адресацией с постинкрементом. Сложить операнды, предварительно загрузив их в РОН командой **LD** с косвенной адресацией с постинкрементом;

7) с использованием *адресации константы в памяти программ*. Для этого задать слагаемые в памяти программ в одной директиве **.db**. Сложить операнды, предварительно загрузив их в РОН с помощью команды **LPM**. Для пересылки слагаемых между РОН использовать команду **MOV**;



8) с использованием *адресации константы в памяти программ с постинкрементом*. Для загрузки слагаемых в РОН использовать команду **LPM** с постинкрементом.

В диалоговом окне **AVR Simulator Options** в разделе **Device Selection** установить тактовую частоту моделирования работы микроконтроллера, равную 8,0 МГц (поле **Frequency**).

Выполнить трансляцию и отладку созданных программ. По данным, выводимым после трансляции на закладке **Build** окна **Output**, проанализировать использование памяти программ (**Program memory usage**) под код программы (**Code**) и константы (**Constants**), оценить объем неиспользованной (**Unused**) и общей занятой (**Total**) памяти. Занести эти сведения в отчет (табл. 4).

Таблица 4

Вариант программы	1	2	3	4	5	6	7	8
Объем памяти программ, занятой под код программы, слов								
Объем памяти программ, занятой под константы, слов								
Объем неиспользованной памяти программ, слов								
Общий объем занятой памяти программ, слов								
Число тактов выполнения программы								
Время выполнения программы, мкс								

П р и м е ч а н и е. Все программы рекомендуется включить в один проект. При этом программа, подлежащая трансляции, задается командой Set as entry file контекстного меню дерева иерархии проекта. Подлежащая трансляции программа помечается в дереве иерархии проекта символом  (все прочие исходные файлы проекта имеют символ ).

При отладке программ использовать средства наблюдения за содержимым регистров и ячеек памяти. По полю **Cycle Counter** объекта **Processor** закладки **I/O** окна **Workspace** определить число тактов выполнения программы, по полю **Stop Watch** – время выполнения программы (до выполнения команды, организующей бесконечный цикл). Зафиксировать эти сведения в отчете (табл. 4). По результатам выполнения программ сделать выводы.

Содержание отчета

Отчет должен содержать титульный лист с указанием номера и названия лабораторной работы, номера группы и фамилий выполнивших работу; цель работы; листинги трансляции программ и сведения, указанные в задании; схемы образования адреса для использованных способов адресации.

Контрольные вопросы

1. Опишите виды адресации РОН и регистров ввода-вывода AVR-микроконтроллеров.
2. Какие существуют способы адресации памяти данных AVR-микроконтроллеров?
3. Какие способы адресации памяти программ AVR-микроконтроллеров вы знаете?
4. Каковы особенности выполнения арифметических и логических операций в AVR-микроконтроллерах?
5. Каково назначение и использование регистров X, Y и Z?

Лабораторная работа № 3

Арифметические и логические команды

Цель работы: изучение команд сложения, вычитания, операций «и, или, не» с регистрами и константами, а также установки, сброса и сдвига разрядов, команд установки и сброса флагов, и команд сравнения РОН.

Теоретическая часть

Арифметические команды:

add Rd,Rr сложение двух РОН без учета переноса
adc Rd,Rr сложение двух РОН с учетом переноса
adiw Rd,k сложение регистровой пары с константой
sub Rd,Rr вычитание двух РОН без учета переноса
sbc Rd,Rr вычитание двух РОН с учетом переноса
sbiw Rd,k вычитание константы из регистровой пары
subi Rd,k вычитание константы из регистра
sbci Rd,k вычитание константы из регистра с учетом переноса
inc Rd увеличение содержимого регистра на единицу
dec Rd уменьшение содержимого регистра на единицу
clr Rd очистка регистра (операция «исключающее или» регистра с самим собой)
ser Rd установка регистра
and Rd,Rr логическое «и»
andi Rd,k логическое «и» с константой
or Rd,Rr логическое «или»
ori Rd,k логическое «или» с константой
eor Rd,Rr логическое исключающее «или»
com Rd побитная инверсия
neg Rd дополнительный код (инверсия знака)

Команды операций с битами:

CBR Rd, K Сброс разряда(ов) РОН
SBR Rd, K Установка разряда(ов) РОН
CBI A, b Сброс разряда PBB
SBI A, b Установка разряда PBB

LSL Rd Логический сдвиг влево (с установкой переноса)
LSR Rd Логический сдвиг вправо (с установкой переноса)
ROL Логический сдвиг влево через перенос
ROR Логический сдвиг вправо через перенос
ASR Арифметический сдвиг вправо
BCLR S Сброс флага
BSET S Установка флага
BLD Rd, b Загрузка разряда POH из флага T
BST Rr, b Загрузка разряда POH из флага T
CLC Сброс флага переноса
SEC Установка флага переноса
CLN Сброс флага отрицательного числа
SEN Установка флага отрицательного числа
CLZ Сброс флага нуля
SEZ Установка флага нуля
CLI Общий запрет прерываний
SEI Общее разрешение прерываний
CLS Сброс флага знака
SES Установка флага знака
CLV Сброс флага переполнения дополнительного кода
SEV Установка флага переполнения
дополнительного кода
CLT Сброс пользовательского флага T
SET Установка пользовательского флага T

Rd – POH, A – порт, K – константа (от 1 до 255), b – номер разряда порта (от 0 до 7),

S-произвольный флаг в регистре состояний SREG

Практическая часть

Запустить **AVR Studio** в режиме эмуляции и выполнить в пошаговом режиме программу 1, при этом наблюдая за состоянием регистров.

Программа 1:

```
.include "m16def.inc";подключение библиотеки
.list;включение листинга
.def temp0=r16;определение рабочих регистров
.def temp1=r17
.def temp2=r18
.def temp3=r19
.def temp4=r20
.def temp5=r21
.def temp6=r22
;-----
metka:
ldi temp0,0x00;записываем ноль в регистр temp0
ldi temp0,0xFF;записываем 0xff в регистр temp0
ldi temp1,0x00;записываем ноль в регистр temp1
ldi temp1,0xAA;записываем 0xAA в регистр temp1
ldi temp2,0x00;записываем ноль в регистр temp2
ldi temp2,0xCC;записываем 0xCC в регистр temp2
ldi temp3,0x00;записываем ноль в регистр temp3
mov temp3,temp2;пересылка данных из temp2 в temp3
add temp1,temp2;складываем temp1 и temp2 без учета переноса
sbc temp0,temp1; вычитаем temp1 из temp0 с учетом переноса
subi temp4,0x11 ;вычитание константы из регистра temp4
inc temp5; увеличение содержимого регистра на единицу
inc temp5; увеличение содержимого регистра на единицу
inc temp5; увеличение содержимого регистра на единицу
dec temp5; уменьшение содержимого регистра на единицу
dec temp5; уменьшение содержимого регистра на единицу
dec temp5; уменьшение содержимого регистра на единицу
clr temp5 ;очистка регистра (операция "исключающее или" регистра с самим собой)
or temp0,temp1; логическое "или"
com temp6; побитная инверсия
neg temp6; дополнительный код(инверсия знака)
rjmp metka;переход к метке
```

Внести следующие изменения в программу:

1. Написать программы сложения и вычитания двух 8-разрядных чисел с записью результата в ячейку памяти.
2. Написать программы сложения двух 16-разрядных чисел.
3. В соответствии с вариантом сложить содержимое Rd1 и Rd2, вычесть из результата константу К и проверить состояние бита отрицательности, после чего записать результат в ячейку памяти R.

Вариант	R	K	Rd1	Rd2
1	0060	0xFF	0x01	0x10
2	0061	0xEE	0x02	0x20
3	0062	0xDD	0x03	0x30
4	0063	0xCC	0x04	0x40
5	0064	0xBB	0x05	0x50
6	0065	0xAA	0x06	0x60
7	0066	0x99	0x07	0x70
8	0067	0x88	0x08	0x80
9	0068	0x77	0x09	0x90
10	0069	0x66	0x0A	0xA0
11	006A	0x55	0x0B	0xB0
12	006B	0x44	0x0C	0xC0
13	006C	0x33	0x0D	0xD0
14	006D	0x22	0x0E	0xE0
15	006E	0x11	0x0F	0xF0

Запустить **AVR Studio** в режиме эмуляции и выполнить программу в пошаговом режиме, при этом наблюдая за состоянием регистров и флагов.

Программа 2:

```
.include "m16def.inc";подключение библиотеки
.list;включение листинга
.def Rd=r16;определение рабочих регистров
.def Rr=r17
.def temp2=r18
.def temp3=r19
.def temp4=r20
.def temp5=r21
```



```
.def temp6=r22
;-----
sbr Rd,0x1 ;установка Rd в единицу
cbr Rd,0x1;сброс нулевого разряда регистра Rd
sbi porta,0 ;установка нулевого разряда PVB порта A в единицу
cbi porta,0 ;сброс нулевого разряда PVB порта A
sbr Rr,0xf0 ;установка четырех младших разрядов Rr
```

Внести следующие изменения в программу:

1. Записать в регистры r17 и r18 соответственно числа 0x00 и 0xFF.
2. Установить в единицу четыре старших разряда регистра r17. Сбросить четыре младших разряда регистра r18.
3. Установить в единицу нулевой, второй, четвертый и шестой разряды PVB порта B.
4. Пронаблюдать в пошаговом режиме за состоянием всех вышеперечисленных регистров.

Содержание отчета

Отчет должен содержать титульный лист с указанием номера и названия лабораторной работы, номера группы и фамилий выполнивших работу; цель работы; листинги трансляции программ и сведения, указанные в задании; этапы преобразования числа в РОН при выполнении различных видов арифметических, логических и битовых команд.

Контрольные вопросы

1. Какие арифметические команды микроконтроллера вы знаете?
2. Опишите логические команды. Какие допустимые операнды для этих команд?
3. Какие существуют варианты изменения определенного бита в заданном РОН с помощью логических команд?
4. Какие происходят изменения определенного бита в заданном РОН с помощью команд установки битов?
5. Как переслать заданный бит из одного РОН в другой?

Лабораторная работа № 4

Реализация типовых структур алгоритмов

Цель работы: изучение принципов реализации типовых алгоритмических структур на примере ветвлений и циклических программ.

Теоретическая часть

Любая процедура управления или обработки данных представляет собой совокупность некоторых алгоритмических структур, с помощью которых выполняются требуемые операции. Наиболее распространенными алгоритмическими структурами являются ветвления (branching) и циклы (loop) [5].

Ветвления используются для выполнения различных частей программы (разделения ветвей алгоритма) в зависимости от некоторых условий.

В *циклах* одна и та же операция выполняется над содержимым нескольких последовательно расположенных в памяти ячеек или элементов данных. Использование циклических программ целесообразно при обработке массивов, таблиц и подобных по структуре данных. Числом повторений цикла управляют счетчики, а обрабатываемый при данном проходе цикла элемент определяется с помощью индекса или указателя.

В циклической программе можно выделить четыре основных блока.

1. *Блок инициализации* (от лат. *initium* – начало), в котором производится присвоение начальных значений переменным, счетчикам, индексам и указателям. Указатели представляют собой адреса данных в памяти.

2. *Блок обработки*, в котором выполняются требуемые вычисления, т.е. одинаковые повторяющиеся действия над различными последовательно расположенными в памяти данными.

3. *Блок управления циклом*, в котором изменяются значения счетчиков и индексов (указателей) перед выполнением следующей повторяющейся операции, а также производится проверка условия выхода из цикла.

4. *Заключительный блок*, в котором производится сохранение полученных результатов.

Блоки 2 и 3 составляют *тело цикла* (loop body). Для повышения быстродействия и сокращения размера циклических программ сле-

дует разгружать тело цикла от операций, которые могут быть выполнены за его пределами.

Для организации циклических программ, а также ветвлений в программах используются команды безусловных и условных переходов. Кроме того, для построения циклов могут применяться специальные команды циклов, выполняющие несколько действий одновременно (в системе команд AVR-микроконтроллеров отсутствуют).

Команды безусловных переходов JMP, RJMP, IJMP и EIJMP передают управление по указанному в команде адресу памяти программ. Команда **JMP** (Jump – переход) позволяет передавать управление внутри всего объема памяти программ. Команда **RJMP** (Relative Jump – относительный переход) обеспечивает переход в пределах ± 2 К слов (± 4 Кбайт) относительно текущего содержимого программного счетчика. По команде **IJMP** (Indirect Jump – косвенный переход) выполняется косвенный переход по адресу, указанному регистром **Z**; максимальное смещение составляет 64 К слов (128 Кбайт). Команда **EIJMP** (Extended Indirect Jump – расширенный косвенный переход) обеспечивает косвенный переход по всему объему памяти программ; для расширения программного счетчика используется регистр **EIND**. При выполнении команд безусловных переходов в программный счетчик загружается адрес ячейки памяти программ, на которую передается управление.

Команды условных переходов передают управление по указанному адресу памяти программ в случае выполнения некоторых условий.

Команды **BRxx** (Branch if ... – перейти, если ...) выполняют переход на расстояние $-64 \dots +63$ слова относительно текущего содержимого программного счетчика по результатам проверки разрядов регистра состояния **SREG** (кодов или флагов условий). Регистр состояния **SREG** находится в адресном пространстве регистров ввода-вывода. Коды условий (**C**, **Z**, **N**, **V**, **S**, **H**) формируются в регистре состояния при выполнении арифметических, логических команд и команд работы с битами и представляют собой признаки результата операции. Разряд **C** (carry – перенос) устанавливается, если при выполнении команды был перенос из старшего разряда результата. Разряд **Z** (zero – нуль) устанавливается, если результат выполнения команды равен нулю. Разряд **N** (negative – отрицательный результат) устанавливается, если старший значащий разряд результата равен 1 (правильно показывает знак результата, если не было переполнения разрядной сетки числа со знаком). Разряд **V** (overflow – переполне-

ние) устанавливается, если при выполнении команды произошло переполнение разрядной сетки числа со знаком. Разряд $S = N \oplus V$ (sign – знак) правильно показывает знак результата при переполнении разрядной сетки числа со знаком. Разряд **H** (half carry – полуперенос) устанавливается, если при выполнении команды был перенос из третьего разряда результата.

Для организации ветвлений при сравнении операндов команды **BRxx** используются совместно с командами **CP** (Compare) сравнения содержимого двух РОН, **CPC** (Compare with Carry) сравнения с учетом признака переноса и **CPI** (Compare with Immediate) сравнения с константой. Команды ветвления **BRxx** отличаются для операндов без знака и со знаком. Числа без знака представляются прямым кодом, числа со знаком – дополнительным кодом.

Команды условных переходов, используемые для ветвлений при сравнении операндов, сведены в табл. 5.

Таблица 5

Условие	Логическое выражение	Команда		Операнды
		сравнения	перехода	
$R_d > R_r$	$Z \cdot (N \oplus V) = 0$	CP Rr, Rd	BRLT	со знаком
	$C + Z = 0$	CP Rr, Rd	BRLO	без знака
$R_d \geq R_r$	$(N \oplus V) = 0$	CP Rd, Rr	BRGE	со знаком
	$C = 0$	CP Rd, Rr	BRSH/BRCC	без знака
$R_d = R_r$	$Z = 1$	CP Rd, Rr	BREQ	со знаком, без знака
$R_d \neq R_r$	$Z = 0$	CP Rd, Rr	BRNE	со знаком, без знака
$R_d \leq R_r$	$Z + (N \oplus V) = 1$	CP Rr, Rd	BRGE	со знаком
	$C + Z = 1$	CP Rr, Rd	BRSH	без знака
$R_d < R_r$	$(N \oplus V) = 1$	CP Rd, Rr	BRLT	со знаком
	$C = 1$	CP Rd, Rr	BRLO/BRCS	без знака

К командам условных переходов также относится команда **CPSE** (Compare and Skip if Equal – сравнить и пропустить, если равно), которая сравнивает содержимое двух РОН и пропускает следующую за ней команду, если содержимое одинаково.

Команды **SBRs**, **SBRC**, **SBIS**, **SBIC** (Skip if Bit in Register [I/O Register] is Set [Cleared] – пропустить, если разряд в регистре общего назначения [ввода-вывода] установлен [сброшен]) пропускают следующую команду в случае выполнения соответствующего условия.

При обработке массивов в циклических программах эффективно использование косвенной адресации памяти данных с предекрементом и постинкрементом, а также косвенной адресации памяти данных со смещением.

На рис. 30 приведен фрагмент программы, в которой число 100 заносится в ячейки массива из пяти байт. Для проверки условия выхода из цикла и передачи управления используется команда **BRNE**. Предел повторений цикла равен 5, шаг равен -1 , параметр цикла (счетчик) содержится в регистре **R16**.

Практическая часть

1. Дополнить фрагмент программы, приведенный на рис. 30, необходимыми директивами. Изменить число, заносимое в ячейки массива, в соответствии с заданным вариантом (табл. 6). Выполнить программу в пошаговом режиме с помощью симулятора-отладчика.

2. Произвести изменения в программе: заменить команды **ADD** (сложение) и **SUB** (вычитание) на **INC** (инкремент) и **DEC** (декремент) соответственно.

```
; ...
array:      .byte 5           ; 5 байт для массива array
; ...

    LDI R16, 5                ; предел повторений цикла
    LDI R17, 100              ; число, заносимое в массив array
    LDI R18, 1

    LDI R30, low(array)       ; младший байт адреса массива array
    LDI R31, high(array)      ; старший байт адреса массива array

loop:                          ; тело цикла
    ST Z, R17                 ; занесение числа 100 в массив array
    ADD R30, R18              ; адрес следующего байта массива array
    SUB R16, R18              ; счетчик числа проходов, шаг равен -1
    BRNE loop                 ; повторить, если счетчик не равен ну-
лю
; ...
```

Рис. 30. Фрагмент программы циклической обработки массива

Таблица 6

Но- мер вари- анта	Чис- ло	Массив I	Массив II	Но- мер вари- анта	Чис- ло	Массив I	Массив II
1	99	13; 78; 1; 24; 18	81; 10; 201; 33; 8	16	84	65; 2; 43; 10; 125	84; 95; 5; 116; 48
2	98	5; 61; 75; 17; 27	42; 137; 72; 9; 53	17	83	14; 23; 83; 30; 66	47; 50; 36; 21; 74
3	97	33; 44; 29; 81; 20	7; 100; 38; 49; 99	18	82	34; 18; 136; 27; 5	94; 52; 47; 85; 21
4	96	24; 31; 6; 55; 71	30; 127; 23; 8; 17	19	81	23; 75; 30; 15; 41	110; 4; 39; 40; 33
5	95	68; 41; 25; 13; 57	48; 4; 15; 36; 121	20	80	71; 52; 19; 24; 88	37; 44; 26; 60; 18
6	94	45; 55; 2; 109; 33	9; 57; 15; 22; 207	21	79	49; 117; 29; 6; 21	51; 14; 57; 23; 48
7	93	23; 13; 67; 39; 48	47; 180; 3; 10; 55	22	78	83; 16; 54; 27; 30	94; 35; 76; 55; 81
8	92	34; 92; 8; 20; 71	36; 76; 23; 99; 40	23	77	37; 65; 29; 86; 24	81; 23; 70; 64; 32
9	91	28; 0; 139; 36; 17	128; 35; 5; 68; 72	24	76	51; 36; 48; 25; 80	78; 94; 8; 24; 128
10	90	61; 40; 22; 27; 66	59; 31; 129; 18; 63	25	75	13; 41; 27; 82; 77	53; 67; 15; 56; 30
11	89	7; 56; 29; 16; 104	87; 23; 90; 44; 62	26	74	94; 2; 17; 38; 45	17; 0; 49; 69; 32
12	88	49; 24; 49; 84; 15	75; 3; 12; 64; 227	27	73	6; 60; 73; 18; 44	100; 22; 37; 9; 56
13	87	51; 33; 19; 48; 80	145; 26; 1; 13; 88	28	72	48; 14; 23; 50; 65	62; 58; 46; 59; 33
14	86	67; 30; 25; 52; 38	35; 62; 8; 59; 46	29	71	31; 52; 17; 24; 78	3; 88; 53; 162; 72
15	85	120; 36; 7; 10; 45	53; 47; 35; 62; 81	30	70	66; 70; 42; 13; 29	42; 15; 76; 38; 86

3. В последнем варианте программы исключить команду **INC** за счет использования косвенной адресации памяти данных с пост-инкрементом.

4. Изменить порядок подсчета числа проходов цикла, задав изменение параметра цикла (счетчика) не с предела, а с нуля.

5. Составить программу пересылки массива из памяти программ в память данных. Массив в памяти программ задать директивой `.db`; значения элементов массива взять из табл. 6 (массив I) в соответствии с заданным вариантом.

6. Составить программу суммирования элементов массива. Значения элементов массива задать аналогично п. 5.

7. Составить программу поэлементного сложения двух массивов. Значения элементов массивов взять из табл. 6 (массив I и массив II) в соответствии с заданным вариантом.

8. Составить программу слияния двух массивов, где в результате попарного сравнения двух элементов из разных массивов образуется новый массив из наибольших элементов. Значения элементов массивов взять аналогично п. 7.

Содержание отчета

Отчет должен содержать титульный лист с указанием номера и названия лабораторной работы, номера группы и фамилий выполнивших работу; цель работы; листинги трансляции программ в соответствии с заданием.

Контрольные вопросы

1. Назовите наиболее распространенные алгоритмические структуры.

2. Каковы назначение и общая структура циклических программ?

3. Какие существуют команды, используемые для организации ветвлений и циклов?

4. Опишите назначение и выполнение команд безусловных переходов.

5. Назначение и выполнение команд условных переходов.

6. Опишите механизм использования кодов (флагов) условий в командах условных переходов.

Лабораторная работа № 5

Организация подпрограмм

Цель работы: изучение принципов организации подпрограмм и передачи параметров; освоение команд, обеспечивающих взаимодействие вызывающей программы с подпрограммой.

Теоретическая часть

В большой и сложной программе можно выделить последовательности команд, выполняющие некоторые законченные функции (процедуры). Если такие последовательности команд оформить в виде отдельных модулей – *подпрограмм* (routine, subroutine), то в программе эти команды могут быть заменены вызовами соответствующих подпрограмм. Такое модульное (структурное) программирование дает следующие преимущества:

- ускоряется и упрощается отладка всей программы;
- подпрограммы, реализующие универсальные функции, могут использоваться при разработке других программ;
- в одной программе могут быть объединены модули, полученные после трансляции программ, написанных на разных языках программирования.

Для взаимодействия вызывающей программы с подпрограммой необходимо выполнение следующих условий:

- вызывающей программе должно быть известно положение подпрограммы в общей структуре программы;
- должны быть определены способы вызова подпрограммы и возврата из нее;
- должен быть выбран способ обмена данными между вызывающей программой и подпрограммой.

Положение подпрограммы в общей структуре программы определяется по ее имени. Имя подпрограммы на ассемблере, задаваемое символической меткой, является стартовым адресом исполняемой части подпрограммы.

Вызов подпрограммы подразумевает передачу в нее управления ходом выполнения команд. Передача управления осуществляется путем загрузки в программный счетчик (РС) стартового адреса подпрограммы. При этом необходимо обеспечить сохранение адреса возврата из подпрограммы, т.е. адреса команды, следующей за командой

вызова подпрограммы. Для хранения адресов возврата из подпрограмм используется *стек* (stack).

Стек – специальным образом организованная последовательность ячеек памяти с дисциплиной обслуживания «последним пришел – первым вышел» (**LIFO**: Last-In – First-Out). При занесении в стек новых данных предыдущие данные сохраняются, но становятся временно недоступными («опускаются»). Выгружаются из стека («поднимаются», «выталкиваются») данные в обратном порядке. Стек может быть реализован как аппаратно, так и программно. Аппаратный стек (Hardware Stack) выполняется непосредственно в составе процессора в виде набора специальных регистров и, как правило, имеет небольшую глубину. Программный стек (Software Stack) организуется в оперативной памяти; глубина определяется размером и степенью использования памяти.

Адрес очередной свободной ячейки стека содержится в специальном регистре – указателе стека **SP** (Stack Pointer). При записи в стек число помещается в ячейку с адресом, содержащимся в указателе стека, после чего содержимое указателя стека уменьшается на единицу (рис. 31,а). При чтении из стека производится выборка содержимого ячейки по адресу, на единицу большему содержимого указателя стека (рис. 31,б). Таким образом, при записи стека и чтении из него содержимое указателя стека изменяется.

Механизм использования стека для хранения адресов возврата из подпрограмм состоит в следующем. Когда в программе встречается команда вызова подпрограммы, в ячейку памяти, определяемую указателем стека, записывается адрес возврата; значение указателя стека декрементируется; в программный счетчик загружается стартовый адрес подпрограммы.

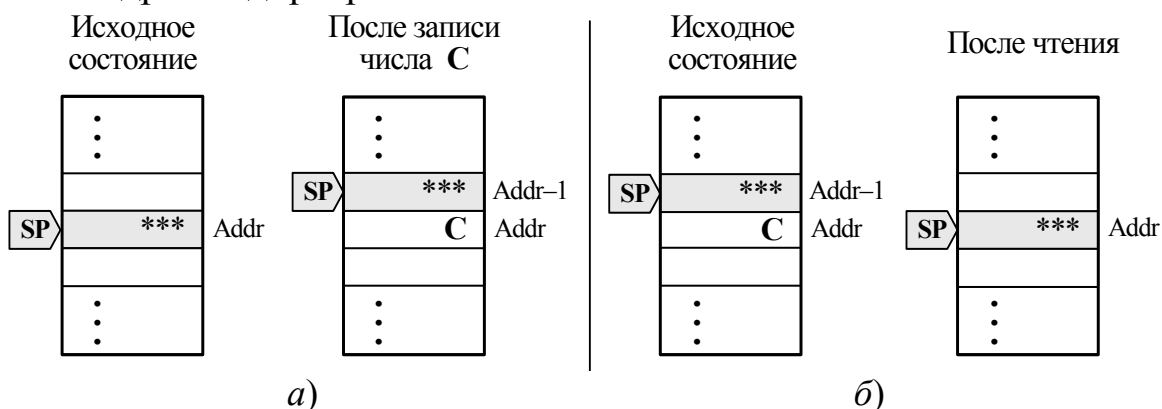


Рис. 31. Операции записи в стек (а) и чтения из стека (б):

*** – очередная свободная ячейка стека; Addr – адрес

Далее выполняются команды подпрограммы. Когда в подпрограмме встречается команда возврата в вызывающую программу, значение указателя стека инкрементируется; сохраненный в стеке адрес возврата загружается в программный счетчик (рис. 32). Использование стека для работы с подпрограммами позволяет одной подпрограмме вызывать другую, т.е. обеспечивает возможность вложения подпрограмм. Глубина вложения подпрограмм ограничивается размером стека.

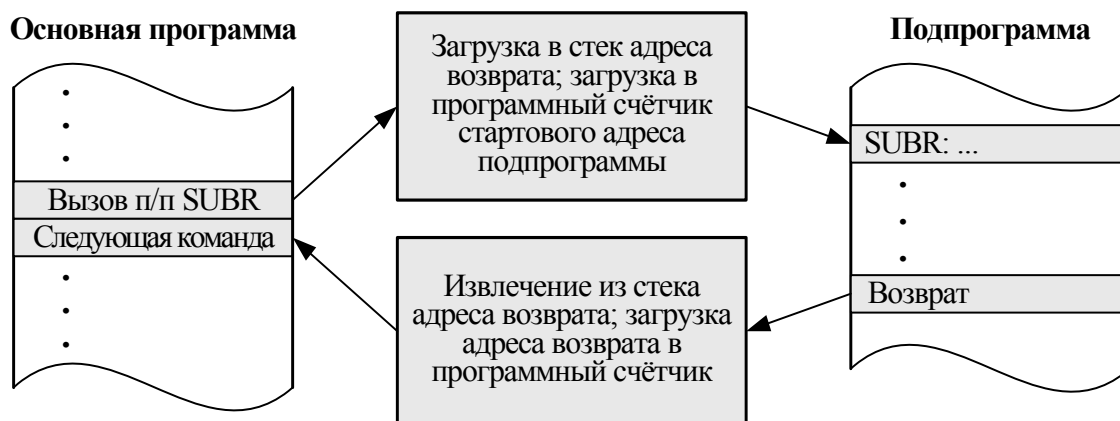


Рис. 32. Механизм вызова подпрограммы и возврата в вызывающую программу

В большинстве AVR-микроконтроллеров стек размещается в оперативной памяти. Указатель стека представляет собой пару 8-разрядных регистров **SPH** (старший байт указателя стека) и **SPL** (младший байт указателя стека), находящихся в адресном пространстве регистров ввода-вывода. В микроконтроллерах, размер оперативной памяти которых не превышает 256 байт, для хранения указателя стека используется только один регистр **SPL (SP)**. Микроконтроллеры, не имеющие оперативной памяти, содержат трехуровневый аппаратный стек.

Организуемый в оперативной памяти стек «растет» от старших адресов к младшим. Учитывая, что начальное значение указателя стека после сброса микроконтроллера равно нулю, в инициализирующей (начальной) части программы необходимо произвести его установку, если предполагается использование хотя бы одной подпрограммы. При организации стека во внутренней оперативной памяти это может быть сделано, например, следующим образом:

LDI R16, low(RAMEND) ; младшая часть адреса
RAMEND

OUT SPL, R16 ; инициализация SPL

LDI R16, high(RAMEND) ; старшая часть адреса
RAMEND

OUT SPH, R16 ; инициализация SPH

Команда **OUT** осуществляет запись содержимого регистра общего назначения в регистр ввода-вывода; **RAMEND** – символическое имя адреса последней ячейки внутренней оперативной памяти. Для того чтобы использовать в программе символические имена адресов периферийных устройств AVR-микроконтроллера, необходимо при помощи директивы **.include** подключить файл определения адресов периферийных устройств (**inc**-файл). Например, для микроконтроллера **ATmega8535** следует подключить файл **m8535def.inc**:

```
.include "m8535def.inc"
```

К подключаемому **inc**-файлу должен быть указан путь в поле **Include Path** окна **AVR Assembler**, вызов которого осуществляется командой AVR Assembler Setup меню Project. При использовании директивы **.include** нет необходимости включать в программу директиву **.device**, так как она содержится в соответствующем **inc**-файле.

Избежать появления в листинге трансляции текста **inc**-файла позволяет директива **.nolist** отключения листинга. Директива **.nolist** используется совместно с директивой **.list** включения листинга (по умолчанию режим генерации листинга включен). Исключение **inc**-файла из листинга трансляции может быть произведено следующим образом:

```
.nolist ; отключить генерацию листинга  
.include "m8535def.inc" ; подключить inc-файл  
.list ; включить генерацию листинга
```

Вызов подпрограммы на языке ассемблера AVR-микроконтроллеров осуществляется командами **RCALL**, **ICALL**, **CALL** и **EI-CALL**. Команда **RCALL** (Relative Call – относительный вызов) обеспечивает вызов подпрограммы, смещение начального адреса которой относительно текущего значения программного счетчика лежит в

пределах ± 2 К слов (± 4 Кбайт), с помощью относительной адресации памяти программ. Команда **ICALL** (Indirect Call – косвенный вызов) производит косвенный вызов подпрограммы по адресу памяти программ, содержащемуся в регистре-указателе **Z**; максимальное смещение начального адреса подпрограммы составляет 64 К слов (128 Кбайт). Команда **CALL** (Call – вызов) обеспечивает вызов подпрограммы из памяти программ размером до 4 М слов (8 Мбайт) с использованием непосредственной адресации. Команда **ETCALL** (Extended Indirect Call – расширенный косвенный вызов) позволяет выполнять косвенный вызов подпрограмм из памяти программ размером до 4 М слов (8 Мбайт). При выполнении команд **RCALL**, **ICALL**, **CALL** и **ETCALL** текущее значение программного счетчика заносится в стек (2 байта в микроконтроллерах с 16-разрядным программным счетчиком или 3 байта в микроконтроллерах с 22-разрядным программным счетчиком). Содержимое указателя стека (**SPH:SPIL**) уменьшается соответственно на 2 или 3.

Примеры использования команд вызова подпрограмм:

```
RCALL    subr1      ; относительный вызов подпрограммы
subr1

; косвенный вызов подпрограммы subr2
LDI  R30, low(subr2)
LDI  R31, high(subr2)
ICALL                                ; команда icall не имеет операндов
```

Для возврата из подпрограмм используется команда **RET**. При выполнении команды **RET** адрес возврата загружается из стека в программный счетчик. При этом содержимое указателя стека увеличивается на 2 или 3 в зависимости от разрядности программного счетчика (см. выше).

Стек может также использоваться для сохранения содержимого РОН на время выполнения подпрограмм. Для сохранения в стеке и извлечения из стека содержимого РОН служат команды **PUSH** и **POP**. Команда **PUSH** заносит содержимое регистра в стек по адресу, хранящемуся в указателе стека, при этом значение указателя стека уменьшается на единицу (**SPH:SPIL** = **SPH:SPIL** – 1). Команда **POP**

выполняет обратные действия: значение указателя стека увеличивается на единицу ($\text{SPH:SPL} = \text{SPH:SPL} + 1$); содержимое ячейки памяти по адресу, хранящемуся в указателе стека, загружается в регистр.

Программы с использованием подпрограмм обычно начинаются с команды относительного перехода к основной программе, в которой прежде всего производится инициализация стека (рис. 33).

```
.nolist                ; отключить генерацию листинга
.include "m8535def.inc" ; подключить inc-файл
.list                  ; включить генерацию листинга

    RJMP    RESET      ; переход к основной программе

PODPR:                ; подпрограмма PODPR
    ; ...
    RET              ; возврат в основную программу

RESET:                ; основная программа
    LDI     R16, low(RAMEND)
    OUT     SPL, R16    ; инициализация SPL
    LDI     R16, high(RAMEND)
    OUT     SPH, R16    ; инициализация SPH

    ; ...
    RCALL    PODPR      ; вызов подпрограммы PODPR
    ; ...
```

Рис. 33. Пример программы с использованием подпрограммы

При работе с подпрограммами важно обеспечить передачу параметров из вызывающей программы в подпрограмму и возврат результатов выполнения подпрограммы обратно в вызывающую программу. В ассемблере AVR-микроконтроллеров способы обмена данными между вызывающей программой и подпрограммой не форма-

лизованы. Для передачи параметров могут использоваться регистры общего назначения, ячейки оперативной памяти и стек.

Передача параметров через регистры общего назначения пригодна только для небольшого числа параметров, так как число РОН ограничено и занятые под параметры регистры уже нельзя использовать в подпрограмме для участия в других вычислениях и хранения других данных. Однако это самый простой и прозрачный способ передачи параметров, обеспечивающий самый быстрый доступ к передаваемым параметрам.

Использование оперативной памяти для передачи параметров требует жесткой регламентации правил обращения к ним. Например, можно организовать в памяти массив (таблицу) для непосредственного хранения значений передаваемых параметров или их адресов; адрес начала массива занести в РОН. Имея начальный адрес массива, вызывающая программа и подпрограмма смогут получить доступ к требуемым параметрам.

При *передаче параметров через стек* перед вызовом подпрограммы передаваемые параметры заносятся в стек. Следует помнить, что после выполнения команды вызова подпрограммы в стек будет добавлен адрес возврата в вызывающую программу. Задавшись значением указателя стека в качестве базового адреса и используя косвенную адресацию памяти данных со смещением, в подпрограмме можно получить доступ к содержащимся в стеке параметрам. Например, если в основной программе перед вызовом подпрограммы занести в стек значение некоторого параметра:

```
LDI R16, $33 ; R16 <- $33
```

```
PUSH R16 ; сохранение содержимого регистра R16  
в стеке
```

то в подпрограмме можно получить к нему доступ:

```
IN R30, SPL ; младший байт указателя стека
```

```
IN R31, SPH ; старший байт указателя стека
```

```
LDD R20, Z+3; загрузка числа $33 из стека в регистр R20
```

(команда **IN** служит для считывания содержимого регистра ввода-вывода в РОН). Аналогично можно использовать стек для хранения

адреса массива передаваемых параметров, расположенного в оперативной памяти данных.

Практическая часть

1. Составить программу, использующую для вызова подпрограммы команду **RCALL**. Выполняемую подпрограммой функцию взять из задания лабораторной работы № 4 (по указанию преподавателя). Для передачи параметров в подпрограмму использовать регистры общего назначения. Выполнить программу в пошаговом режиме, отслеживая изменение содержимого программного счетчика, указателя стека, а также занесение в стек адреса возврата из подпрограммы.

2. Выполнить задание п. 1, используя стек для передачи параметров в подпрограмму. Проследить в симуляторе занесение передаваемых параметров в стек.

3. Выполнить задание п. 2, применив для вызова подпрограммы команду **ICALL**.

Содержание отчета

Отчет должен содержать титульный лист с указанием номера и названия лабораторной работы, номера группы и фамилий выполнивших работу; цель работы; листинги трансляции программ в соответствии с заданием.

Контрольные вопросы

1. Каковы условия взаимодействия вызывающей программы и подпрограммы?

2. Опишите принцип организации и назначение стека.

3. Каков механизм вызова подпрограмм?

4. Какие команды работы с подпрограммами на ассемблере микроконтроллеров семейства AVR вы знаете?

5. Какие существуют способы обмена данными между вызывающей программой и подпрограммой?

Лабораторная работа № 6

Система прерываний

Цель работы: изучение системы прерываний на примере прерывания по переполнению встроенного таймера-счетчика AVR-микроконтроллера.

Теоретическая часть

При работе реальной микропроцессорной системы в ней или вне ее могут произойти события, требующие немедленной реакции. Такая реакция обеспечивается процедурой прерывания (interrupt), которая состоит в том, что выполнение текущей программы приостанавливается, запоминается состояние на момент прерывания, выполняется другая программа, после чего восстанавливается сохраненное до прерывания состояние процессора и продолжается выполнение прерванной программы. Сигнал, вызвавший прерывание текущей программы, называется *запросом на прерывание* (interrupt request – IRQ); источник этого сигнала – *источником прерывания*; последовательность действий, выполняемая по запросу на прерывание, – *обслуживанием прерывания*, а выполняемая по прерыванию программа – *подпрограммой обработки прерывания* (interrupt handler, interrupt routine).

Различают два типа источников прерывания – аппаратные и программные. Источниками аппаратных прерываний служат внешние и внутренние периферийные устройства. Запросом на прерывание от внешнего источника является активный сигнал на соответствующем выводе процессора; источник прерывания определяется по выводу, на котором появляется такой сигнал. К источникам программного прерывания относятся специальные команды прерываний (trap) – управляемые программные прерывания и особые условия (exception – исключение) – неуправляемые программные прерывания, являющиеся реакцией процессора на исключительную ситуацию, возникшую при выполнении некоторой команды (переполнение, деление на нуль и т.п.). Запросом на прерывание от программного источника является непосредственно команда прерывания или установка бита (битов), фиксирующих возникновение особого условия. Общее количество источников аппаратных и программных прерываний может быть различным – от единиц до нескольких десятков.

Процедура обслуживания прерываний по запросам от нескольких источников в различных процессорах выполняется по-разному.

Тем не менее основные принципы реализации механизма прерываний являются общими. Управление процедурой прерываний осуществляется специальными устройствами в составе аппаратного обеспечения процессора (контроллерами, схемами управления и т.п.). Основными средствами управления прерываниями являются:

- векторы прерываний;
- приоритеты прерываний;
- операция маскирования прерываний;
- флаги прерываний.

В микроконтроллерах указанные средства управления прерываниями реализуются следующим образом.

Для управления прерываниями от N источников в адресном пространстве памяти программ выделяется специальная область из N ячеек памяти (или N блоков, состоящих из нескольких ячеек). В каждой из этих ячеек размещаются команды перехода к соответствующей подпрограмме обработки прерывания или (в случае блока из нескольких ячеек) непосредственно команды, которые необходимо выполнить по запросу на прерывание. Эти ячейки памяти (блоки) называются *векторами прерываний* (или просто векторами), адрес ячейки (первой ячейки каждого блока) – адресом вектора прерывания. Таким образом, каждому источнику прерывания ставится в соответствие свой адрес вектора прерывания. Совокупность N векторов образует таблицу векторов прерываний, которая обычно располагается, начиная с нулевого адреса памяти программ.

Приоритеты прерываний (interrupt priority) определяют очередность обслуживания запросов на прерывания. Введение приоритетов необходимо, если возможно одновременное (в течение одного периода тактовой частоты) поступление запросов на прерывание от различных источников или поступление нового запроса на прерывание во время обслуживания прерывания по ранее поступившему запросу. Виды и структура приоритетов прерываний определяются архитектурой процессора. Наиболее простым способом задания приоритетов является последовательное присвоение значений приоритетов в таблице векторов прерываний от высшего к низшему. Высший приоритет всегда имеет аппаратный сброс; далее располагаются векторы прерываний от других источников.

Для того чтобы запретить обслуживание неиспользуемых прерываний, служит операция *маскирования*. В зависимости от возможности маскирования источники прерывания делятся на маскируемые (maskable), прерывания от которых могут разрешаться или запрещаться, и немаскируемые (nonmaskable), прерывания от которых не

могут запрещаться. Маскирование может быть общим и индивидуальным. При общем (глобальном) маскировании все прерывания, кроме немаскируемых, запрещены независимо от их индивидуально-го маскирования. Индивидуальное маскирование позволяет запрещать (разрешать) прерывание от каждого источника отдельно.

Флаги прерываний представляют собой разряды специальных регистров, устанавливающиеся при поступлении запроса на прерывание от некоторого источника.

Процедура обслуживания прерывания может быть упрощенно представлена состоящей из следующих этапов:

- приема запросов на прерывание;
- арбитража прерываний;
- выполнения подпрограммы обслуживания прерывания.

При приеме запроса на прерывание от немаскируемого источника сразу осуществляется переход к следующему этапу его обслуживания – арбитражу. Запрос на прерывание от маскируемого источника обрабатывается по более сложному алгоритму. При поступлении запроса устанавливается соответствующий флаг прерывания. Далее проверяется наличие общего маскирования прерываний. Если режим общего маскирования установлен, то запросы на прерывания от всех маскируемых источников игнорируются и продолжается выполнение текущей программы. Если режим общего маскирования не задан, то запрещение или разрешение данного прерывания определяется наличием (отсутствием) индивидуального маскирования. Если данное прерывание замаскировано, то запросы на прерывание от данного источника запрещены и продолжается выполнение текущей программы. В противном случае прерывания от данного источника разрешены и для него начинается следующий этап обслуживания – арбитраж.

Арбитраж прерываний служит для определения прерывания с наивысшим приоритетом из очереди поступивших запросов на прерывание. После арбитража начинается выполнение выбранного запроса на прерывание.

Выполнение прерывания состоит в переходе к подпрограмме обслуживания прерывания, ее выполнении и возврате к выполнению текущей программы. Перед выполнением прерывания производится общее маскирование, т.е. запрещение всех прерываний, кроме немаскируемых, а также очищается флаг обслуживаемого прерывания. Собственно выполнение прерывания начинается с обращения к вектору прерывания обслуживаемого источника.

Обслуживаемое прерывание может быть прервано по запросам от источников, имеющих более высокий приоритет. Прерывания, для обслуживания которых прерывается выполнение подпрограммы обработки другого прерывания, называются *вложенными*. Процедура их обслуживания аналогична обслуживанию обычных прерываний; отличие состоит лишь в том, что прерывается выполнение не основной программы, а подпрограммы обработки прерывания от источника с более низким приоритетом.

В микропроцессорных системах механизм прерываний используется для обмена информацией с различными устройствами ввода-вывода. Такой способ обмена данными называется *обменом по прерываниям*. Типичными примерами запросов на прерывание являются запросы по готовности результата аналого-цифрового преобразования, готовности устройства к приему (передаче) информации, переполнению некоторого регистра и т.п. Использование механизма прерываний позволяет значительно повысить производительность системы при работе с медленно действующими устройствами, обслуживание которых в таком случае занимает процессорное время только при их готовности к обмену.

В AVR-микроконтроллерах механизм прерываний реализуется следующим образом. Управление прерываниями осуществляется с помощью схемы прерываний (см. рис. 1). Область векторов прерываний размещается в начале памяти программ; каждый вектор состоит из одной ячейки. При необходимости область векторов прерываний может быть перемещена в другое место памяти программ. Прерывания с младшими адресами имеют больший уровень приоритета. Источниками всех прерываний являются аппаратные средства (внешние или внутренние); источники программных прерываний отсутствуют. Все источники прерываний являются маскируемыми. Общее маскирование осуществляется очисткой бита **I** глобального разрешения прерываний в регистре состояния **SREG**. Количество векторов прерываний в AVR-микроконтроллерах составляет от 3 до 35 в зависимости от типа. Например, в микроконтроллере **ATmega8535** имеется 21 вектор прерывания: 3 от внешних источников и 18 от внутренней периферии.

Работа с внешними прерываниями осуществляется с помощью регистра управления **GICR** (General Interrupt Control Register) и регистра флагов **GIFR** (General Interrupt Flag Register), расположенных в адресном пространстве регистров ввода-вывода. Установка разряда 7 (**INT1**) регистра управления **GICR** разрешает внешнее прерывание **INT1**, установка разряда 6 (**INT0**) – внешнее прерывание **INT0**, уста-

новка разряда 5 (**INT2**) – внешнее прерывание **INT2**. Разряд 7 (**INTF1**) регистра флагов **GIFR** устанавливается при поступлении запроса на прерывание **INT1**, разряд 6 (**INTF0**) – запроса на прерывание **INT0**; разряд 5 (**INTF2**) – запроса на прерывание **INT2**. Очистка установленных флагов прерываний производится записью единиц в соответствующие разряды регистра **GIFR**.

Режим запуска внешних прерываний **INT0** и **INT1** задают разряды 0...3 (**ISC00**, **ISC01**, **ISC10**, **ISC11**) регистра управления **MCUCR**. Запись в разряды **ISC00**, **ISC01** соответственно значений 0, 0 задает режим запуска внешнего прерывания **INT0** по низкому уровню; 0, 1 – по отрицательному фронту; 1, 1 – по положительному фронту; значения 1, 0 не используются. Аналогично с помощью разрядов **ISC10**, **ISC11** задается режим запуска внешнего прерывания **INT1**. Режим запуска внешнего прерывания **INT2** задается разрядом 6 (**ISC2**) регистра управления и состояния **MCUCSR**: 0 – по отрицательному фронту; – по положительному фронту.

Для управления прерываниями от внутренних периферийных устройств в адресном пространстве регистров ввода-вывода также предусмотрены специальные регистры. Например, управление прерываниями по запросам от встроенных таймеров-счетчиков осуществляется с помощью регистра масок **TIMSK** (Timer/Counter Interrupt Mask Register) и регистра флагов **TIFR** (Timer/Counter Interrupt Flag Register). Кроме того, с каждым аппаратным устройством **AVR**-микроконтроллера ассоциированы управляющие регистры, расположенные в адресном пространстве регистров ввода-вывода. Например, управление встроенным 8-разрядным таймером-счетчиком **T/C0** (Timer/Counter0) осуществляется с помощью регистра **TCCR0** (Timer/Counter0 Control Register) и регистра и **TCNT0** (Timer/Counter0). Разряды 0...2 (**CS00**, **CS01**, **CS02**) регистра **TCCR0** задают режим работы таймера-счетчика **T/C0**: при записи в разряды **CS00**, **CS01**, **CS02** соответственно значений 0, 0, 0 таймер-счетчик остановлен; 1, 0, 0 – содержимое регистра **TCNT0** инкрементируется на каждом такте тактового генератора; 0, 1, 0 – на каждом 8-м такте; 1, 1, 0 – на каждом 64-м такте; 0, 0, 1 – на каждом 256-м такте; 1, 0, 1 – на каждом 1024-м такте; значения 0, 1, 1 и 1, 1, 1 устанавливают режим подсчета числа импульсов внешнего источника по отрицательному и положительному фронту соответственно. Таймер-счетчик **T/C0** генерирует запрос на прерывание при переполнении регистра **TCNT0**. В регистре масок **TIMSK** прерыванию при переполнении таймера-счетчика **T/C0** соответствует разряд 1 (**TOIE0**); в регистре флагов **TIFR** – разряд 1 (**TOV0**). Установка разряда **TOIE0** раз-

решает прерывание по переполнению регистра **TCNT0**; флаг **TOIF0** устанавливается при поступлении запроса на прерывание по переполнению регистра **TCNT0**.

Пример программы с использованием прерываний приведен на рис. 34.

```

; область векторов прерываний
.org $0000
RJMP  RESET                ; переход к основной программе
.org INT0addr
RJMP  EXT_INT0             ; внешнее прерывание INT0
.org OVF0addr
RJMP  TMR0_INT             ; прерывание по таймеру T/C0

; подпрограмма обработки внешнего прерывания INT0
EXT_INT0:
; ...
RETI                        ; возврат

; подпрограмма обработки прерывания по таймеру T/C0
TMR0_INT:
; ...
RETI                        ; возврат

RESET:                      ; основная программа

; инициализация стека
; ...

; инициализация внешнего прерывания INT0
LDI    R16, (1<<ISC01)|(1<<ISC00)
OUT    MCUCR, R16          ; по положительному фронту

LDI    R16, (1<<INTF1)|(1<<INTF0)
OUT    GIFR, R16           ; очистка флагов внешних прерываний

LDI    R16, 1<<INT0
OUT    GICR, R16           ; разрешение внешнего прерывания INT0

; инициализация прерывания по таймеру T/C0
LDI    R16, 1<<CS00
OUT    TCCR0, R16          ; деления частоты нет

LDI    R16, 1<<TOIE0
OUT    TIMSK, R16          ; разрешение прерывания по таймеру T/C0

SEI                                ; общее разрешение прерываний

forever:
NOP                                ; пустая команда (no operation)
RJMP  forever                    ; бесконечный цикл

; ...

```

Рис. 34. Пример программы с использованием прерываний

Программы с использованием прерываний начинаются с определения области векторов прерываний. Адреса векторов прерываний указываются символическими именами и помощью директив **.org**. По адресам векторов прерываний размещают команды относительного перехода к подпрограммам обработки прерываний, которые обычно располагают непосредственно после области векторов прерываний. Подпрограммы обработки прерываний завершаются командами **RETI** возврата в основную программу. Команда **RETI** выполняет те же действия, что и команда **RET**, а также восстанавливает бит **I** общего (глобального) разрешения прерываний в регистре состояния **SREG**.

В основной программе производится инициализация стека и прерываний. Инициализация прерываний осуществляется путем установки определенных разрядов в соответствующих регистрах ввода-вывода; при этом в командах используются символические обозначения как самих регистров, так и отдельных их разрядов. После инициализации прерываний производится общее разрешение прерываний путем установки бита **I** в регистре состояния **SREG**. Для этого предусмотрена специальная команда **SEI** (Set Global Interrupt Flag).

Процедура обслуживания прерываний в AVR-микроконтроллерах выполняется согласно приведенному выше алгоритму. Для организации вложенных прерываний необходимо в подпрограмме обработки прерывания восстанавливать бит **I** общего разрешения прерываний в регистре состояния **SREG**.

Практическая часть

1. Дополнить программу, приведенную на рис. 34, необходимыми директивами и командами. В подпрограмму обработки прерывания по таймеру-счетчику **T/C0** поместить команду загрузки числа в **РОН**. Выполнить программу в пошаговом режиме. Проследить изменение содержимого стека при обработке прерывания, а также установку и сброс бита **I** общего разрешения прерываний и флага **TOV0** прерывания по таймеру-счетчику **T/C0**. Для контроля содержимого регистров таймера-счетчика **T/C0** раскрыть пункт **TIMER_COUNTER_0** объекта **I/O ATMEGA8535** закладки **I/O** окна **Workspace**.

2. Исследовать процедуру обработки вложенных прерываний, внося соответствующие изменения в программу. В подпрограмму обработки прерывания по внешнему прерыванию **INT0** поместить команду очистки **РОН**, используемого в подпрограмме обработки

прерывания по таймеру-счетчику **T/C0**. В симуляторе после перехода в подпрограмму обработки прерывания по таймеру-счетчику **T/C0** смоделировать поступление сигнала внешнего прерывания **INT0**. Для этого в симуляторе установить флаг **INTF0** в регистре **GIFR** группы **EXTERNAL_INTERRUPT** объекта **I/O ATMEGA8535** закладки **I/O** окна **Workspace**. Проследить изменение содержимого стека при обработке вложенных прерываний.

Содержание отчета

Отчет должен содержать титульный лист с указанием номера и названия лабораторной работы, номера группы и фамилий выполнивших работу; цель работы; листинги трансляции программ в соответствии с заданием.

Контрольные вопросы

1. Каково назначение прерываний?
2. Опишите типы прерываний.
3. Какие существуют средства управления прерываниями?
4. Опишите порядок и цель операции маскирования прерываний.
5. Расскажите об этапах процедуры прерывания.
6. Какова реализация прерываний в **AVR**-микроконтроллерах?

Список литературы

1. URL: <http://www.atmel.com/products/microcontrollers/avr/default.aspx>
2. Фрунзе, А. В. Микроконтроллеры? Это же просто! / А. В. Фрунзе. – М. : ИД «Додэка-XXI», 2007. – Т. 1. – 312 с.
3. Гребнев, В. В. Микроконтроллеры семейства AVR фирмы Atmel / В. В. Гребнев. – М. : РадиоСофт, 2002. – 176 с.
4. Трампет, В. AVR-RISC микроконтроллеры / В. Трампет ; пер. с нем. – Киев : МК-Пресс, 2006. – 464 с.
5. Баранов, В. Н. Применение микроконтроллеров AVR: схемы, алгоритмы, программы / В. Н. Баранов. – М. : ИД «Додэка-XXI», 2004. – 288 с.
6. Евстифеев, А. В. Микроконтроллеры AVR семейств Tiny и Mega фирмы Atmel / А. В. Евстифеев. – М. : ИД «Додэка-XXI», 2004. – 560 с.
7. Белов, А. В. Самоучитель разработчика устройств на микроконтроллерах AVR / А. В. Белов. – СПб. : Наука и техника, 2008. – 544 с.

СОДЕРЖАНИЕ

Введение	3
Глава 1. Гарвардская архитектура	7
Классическая гарвардская архитектура.....	7
Модифицированная гарвардская архитектура	8
Гибридные модификации с архитектурой фон Неймана.....	8
Глава 2. Архитектура RISC	10
Глава 3. Общее описание микроконтроллеров AVR.....	14
Система команд микроконтроллеров AVR	14
Семейства и версии микроконтроллеров	15
Краткие характеристики встроенной периферии МК.....	16
Глава 4. Описание микроконтроллера ATmega.....	18
Программная модель AVR-микроконтроллеров.....	20
Периферия	22
Питание.....	26
Программирование микроконтроллеров.....	26
Глава 5. Описание ассемблера AVR.....	30
Требования к исходному коду.....	30
Инструкции процессоров AVR	30
Арифметические и логические инструкции	31
Инструкции ветвления	32
Инструкции передачи данных	35
Инструкции работы с битами	37
Директивы ассемблера	39
Выражения.....	48
Глава 6. Работа с пакетом AVR Studio 4.....	54
Окна и режимы AVR Studio	57
Практическая часть.....	69
Лабораторная работа № 1. Знакомство с ПО AVR Studio	69
Лабораторная работа № 2. Способы адресации операндов	81
Лабораторная работа № 3. Арифметические и логические команды.....	92
Лабораторная работа № 4. Реализация типовых структур алгоритмов.....	97
Лабораторная работа № 5. Организация подпрограмм.....	103
Лабораторная работа № 6. Система прерываний	111
Список литературы	119

Учебное издание

Кочегаров Игорь Иванович,
Трусов Василий Анатольевич

Микроконтроллеры семейства AVR.
Лабораторный практикум

Редактор *Ж. А. Лубенцова*
Компьютерная верстка *М. Б. Жучковой*

Подписано в печать 21.11.12.
Формат 60×84¹/₁₆. Усл. печ. л. 7,09.
Тираж 50. Заказ № 940.

Издательство ПГУ.
440026, Пенза, Красная, 40.
Тел./факс: (8412) 56-47-33; e-mail: iic@mail.pnzgu.ru