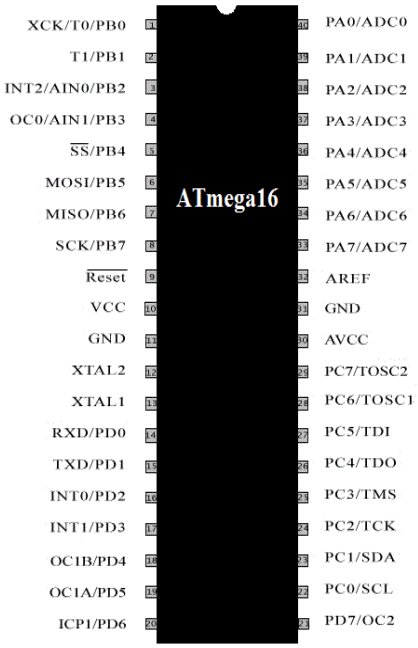
Томский межвузовский центр
дистанционного образования
ТМЦДО

К.В. Бородин

МИКРОПРОЦЕССОРНЫЕ УСТРОЙСТВА И СИСТЕМЫ

Учебное пособие



ТОМСК — 2015

Федеральное агентство по образованию

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

К.В. Бородин

МИКРОПРОЦЕССОРНЫЕ УСТРОЙСТВА И СИСТЕМЫ

Учебное пособие

2015

Корректор: Осипова Е.А.

Бородин К.В.

Микропроцессорные устройства и системы: Учебное пособие. — Томск: Кафедра промышленной электроники, 2016. — 109с.

Рассмотрены программная модель, система команд и характеристики периферийных устройств микроконтроллеров AVR фирмы Atmel семейства Mega. Показано использование компилятора AVR Studio 6.2 для отладки программ на ассемблере и языке Си.

Для студентов вузов радиоэлектронного профиля и инженеров-проектировщиков средств и систем автоматики и промышленной электроники.

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Целью курса является изучение принципов построения и организации микропроцессорных систем (МПС), особенностей проектирования электронных систем управления на их основе и знакомство с отладочными средствами микропроцессорных устройств. Знакомство с процессом написания кода, программирования и отладки осуществляется на микроконтроллерах фирмы Atmel Corporation (AVR Mega) на специальной отладочной плате, разработанной на кафедре промышленной электроники ТУСУР, и интегрированной среды разработки Atmel Studio. Языком программирования является ассемблер и C++. В данном курсе акцент делается на написание основного кода программ на языке C++, как наиболее востребованного и часто используемого. Ассемблер изучается с целью дальнейшей отладки написанных программ, выявления ошибок и вставки простейших команд в тело основного кода. Отладочный макет используется для контроля и управления различными внешними периферийными устройствами.

В результате изучения курса студенты должны иметь представление о классификации, возможностях и применениях микропроцессорных устройств и систем, о средствах и способах автономной отладки аппаратурных средств (АС) и программных средств (ПС) МПС, знать архитектуру и основные конфигурации микропроцессорных систем, особенности процесса интеграции АС и ПС МПС, уметь проектировать микропроцессорные устройства и системы управления периферийными устройствами и получить навыки проведения комплексной отладки и тестирования МПС. Знания программирования, полученные в курсе, позволят в дальнейшем использовать другие пакеты программ такие как: IAR, Keil uVision, Texas Code Composer Studio и другие.

В результате изучения дисциплины студент должен:

Знать: архитектуру и основные конфигурации микропроцессорных систем, особенности процесса интеграции АС и ПС МПС;

Уметь: проектировать микропроцессорные устройства и системы управления периферийными устройствами;

Владеть: навыками проведения комплексной отладки и тестирования МПС.

1. СТРУКТУРА МИКРОПРОЦЕССОРОВ

1.1. Основные понятия

Основной особенностью *микроконтроллера* является то, что он, кроме *микропроцессора*, может содержать на одном кристалле ОЗУ и ПЗУ нескольких типов, блоки ввода-вывода, управления и синхронизации, и, главным образом, набор различных периферийных блоков. Хотя современные микропроцессоры тоже могут содержать на кристалле дополнительные блоки, принято считать, что микропроцессоры применяются для создания персональных компьютеров, а микроконтроллеры – для создания различных встраиваемых систем, систем управления телекоммуникациями, портативным и технологическим оборудованием. Его назначение следует из самого названия «микроконтроллер» (control – управление) [2].

Исторически сложились следующие варианты деления архитектур микропроцессоров по количеству встроенных команд в ядро.

CISC (Complex Instruction Set Computer) – архитектура реализована во многих типах микропроцессоров, выполняющих большой набор разноформатных команд с использованием многочисленных способов адресации. Эта классическая архитектура процессоров, которая начала свое развитие в 1940-х годах с появлением первых компьютеров. Типичным примером CISC процессоров являются микропроцессоры семейства Pentium.

Они выполняют более 200 команд разной степени сложности, которые имеют размер от 1 до 15 байт и обеспечивают более 10 различных способов адресации. Такое большое многообразие выполняемых команд и способов адресации позволяет программисту реализовать наиболее эффективные алгоритмы решения различных задач. Однако при этом существенно усложняется структура микропроцессора, особенно его устройства управления, что приводит к увеличению размеров и стоимости кристалла, снижению производительности. В то же время многие команды и способы адресации используются достаточно редко.

RISC (Reduced Instruction Set Computer) – архитектура отличается использованием ограниченного набора команд фиксированного формата. Современные RISC-процессоры обычно реализуют не более 100 команд, имеющих фиксированный формат длиной 4 байта. Также значительно сокращается число используемых способов адресации. Обычно в RISC-процессорах все команды обработки данных выполняются только с регистровой или непосредственной адресацией. При этом для сокращения количества обращений к памяти RISC-процессоры имеют увеличенный объем внутреннего РЗУ — от 32 до нескольких сотен регистров, тогда как в CISC-процессорах число регистров общего назначения обычно составляет 8 – 16. Обращение к памяти в RISC-процессорах используется только в операциях загрузки данных в РЗУ или пересылки результатов из РЗУ в память. При этом используется небольшое число наиболее простых способов адресации: косвеннорегистровая, индексная и некоторые другие. В результате существенно упрощается структура микропроцессора, сокращаются его размеры и стоимость.

VLIW (Very Large Instruction Word) – архитектура с очень длинными командами (128 бит и более), отдельные поля которых содержат коды, обеспечивающие выполнение различных операций. Таким образом, одна команда вызывает выполнение сразу нескольких операций параллельно в

различных операционных устройствах, входящих в структуру микропроцессора. При трансляции программ, написанных на языке высокого уровня, соответствующий компилятор производит формирование «длинных» VLIW-команд, каждая из которых обеспечивает реализацию процессором целой процедуры или группы операций. Данная архитектура реализована в некоторых типах микропроцессоров (PA8500 компании «Hewlett-Packard», Itanium — совместная разработка «Intel» и « Hewlett-Packard», некоторые типы DSP — цифровых процессоров сигналов) и является весьма перспективной для создания нового поколения сверхвысокопроизводительных процессоров.

Кроме набора выполняемых команд и способов адресации важной архитектурной особенностью микропроцессоров является

используемый вариант реализации памяти и организация выборки команд и данных. По этим признакам различаются процессоры с Принстонской и Гарвардской архитектурой. Эти архитектурные варианты были предложены в конце 1940-х годов специалистами соответственно Принстонского и Гарвардского университетов США для разрабатываемых ими моделей компьютеров.

Принстонская архитектура (архитектура Фон-Неймана), характеризуется использованием общей оперативной памяти для хранения команд (программ) и памяти данных, а также для организации стека. Для обращения к этой памяти используется общая системная шина, по которой в процессор поступают и команды, и данные.

Достоинством архитектуры является наличие общей памяти, что позволяет оперативно и эффективно перераспределять ее объем для хранения отдельных массивов команд, данных и реализации стека в зависимости от решаемых задач, а использование общей шины для передачи команд и данных значительно упрощает отладку, тестирование и текущий контроль функционирования системы, повышает ее надежность. Поэтому Принстонская архитектура в течение долгого времени доминировала в вычислительной технике.

Недостатком является необходимость последовательной выборки команд и обрабатываемых данных по общей системной шине, что ограничивает наращивание производительности цифровой системы.

Гарвардская архитектура характеризуется физическим разделением памяти команд (программ) и памяти данных. В ее оригинальном варианте использовался также отдельный стек для хранения содержимого программного счетчика, который обеспечивал возможности выполнения вложенных подпрограмм.

Каждый внутренний блок памяти соединяется с процессорным ядром отдельной шиной, что позволяет одновременно с чтением-записью данных при выполнении текущей команды производить выборку и декодирование следующей команды. Благодаря такому разделению потоков команд и данных и совмещению операций их выборки реализуется более высокая производительность, чем при использовании Принстонской архитектуры.

Недостатком Гарвардской архитектуры является сложность изготовления кристалла с большим количеством шин, а также с фиксированным объемом памяти, выделенной для команд и данных, значение которой не может оперативно перераспределяться в соответствии с требованиями решаемой задачи. Поэтому приходится использовать память большего объема, коэффициент использования которой при решении разнообразных задач оказывается более низким, чем в системах с Принстонской архитектурой.

Однако развитие микроэлектронных технологий позволило в частично преодолеть указанные недостатки. Гарвардская архитектура широко применяется во внутренней структуре современных высокопроизводительных микропроцессоров, где используется отдельная кэш-память для хранения команд и данных. В то же время во внешней структуре большинства микропроцессоров реализуются принципы Принстонской архитектуры.

1.2. Архитектура микроконтроллеров ATMega16, программная модель

В настоящее время в серийном производстве находятся три основных семейства AVR — Tiny, Mega и xMega . Микроконтроллеры семейства Tiny (до 20МГц) имеют небольшой объем памяти программ и весьма ограниченную периферию и специально предназначены для систем, чувствительных к размерам и стоимости. Самые миниатюрные микросхемы tinyAVR имеют габариты 1,5 x 1,4 мм. Их можно использовать в качестве одночиповых решений для компактных систем. Способны работать при напряжении питания выше 0.7В, что соответствует одному аккумулятору питания. Они выпускаются в 8-выводных корпусах и являются самыми дешевыми.

Более развитую периферию, б***о***льшие объемы памяти данных и программ имеют микроконтроллеры семейства Mega (до 20МГц), которые рассматриваются в данном курсе. Самая быстрая и дорогая линейка xMega (до 32 МГц) имеет в своем ядре аппаратные блоки шифрования AES / DES и расширенную периферию.

Все микроконтроллеры построены на модифицированном фирмой AVR ядре RISC. Сама идея создания нового RISC-ядра родилась в 1994 году в Норвегии. В 1995 году два его изобрета-

теля Альф Боген (Alf-Egil Bogen) и Вегард Воллен (Vegard Wollen) предложили корпорации Atmel выпускать новый 8-разрядный RISC-микроконтрол-лер как стандартное изделие и снабдить его Flash-памятью программ на кристалле. Руководство Atmel Corp. приняло решение инвестировать данный проект. В 1996 году был основан исследовательский центр в городе Тронхейм (Норвегия). Оба изобретателя стали директорами нового центра, а микроконтроллерное ядро было запатентовано и получило название AVR (**A**lf-Egil Bogen + **V**egard Wollen + **R**ISC). Первый опытный кристалл 90S1200 был выпущен на стыке 1996—1997 годов, а с 3 квартала 1997 года корпорация Atmel приступила к серийному производству нового семейства микроконтроллеров и к их рекламной и технической поддержке.

Рассмотрим линейку ATMega подробно. Основной нишей семейства ATMega являются дешевые конечные изделия общего пользования, не требующие серьезных вычислительных затрат и высокой скорости реакции: цифровые плата индикации для отображения информации, обработчики клавиатуры, простейшие устройства управления шаговыми двигателями, преобразователь интерфейсов либо ретранслятор сигналов и т.п. Часто применяются в составе сложных изделий для уменьшения нагрузки на центральный высокопроизводительный микроконтроллер.

Микроконтроллер выполняет одну полноценную инструкцию за один такт, что соответствует производительности 1 MIPS/МГц. Максимальная частота работы составляет 16МГц или 62.5нс (1/16МГц) на 1 простейшую операцию.

Микроконтроллер AVR ATmega16 содержит типовой набор периферийных блоков (Рис. 1):

– порты ввода/вывода, позволяют вывести 1 (5 вольт) либо 0 (земля) на ножку микросхемы;

– три внутренних таймера, позволяющих отсчитывать интервалы времени, по истечении которых выполнять определенные действия;

– широтно-импульсный модулятор (ШИМ), позволяет изменять скважность импульсов (их заполнение);

– один 10 битный АЦП позволяет сделать 1024 (2^{10}) сравнения при преобразовании аналогового сигнала

– аналоговый компаратор для одной задачи;

– цифровые шины данных (UART, SPI, I2C) как для связи с другим контроллером, так и с различными датчиками и устройствами

– вспомогательные внутренние периферийные блоки (монитор питания, сторожевой таймер и т.д.).

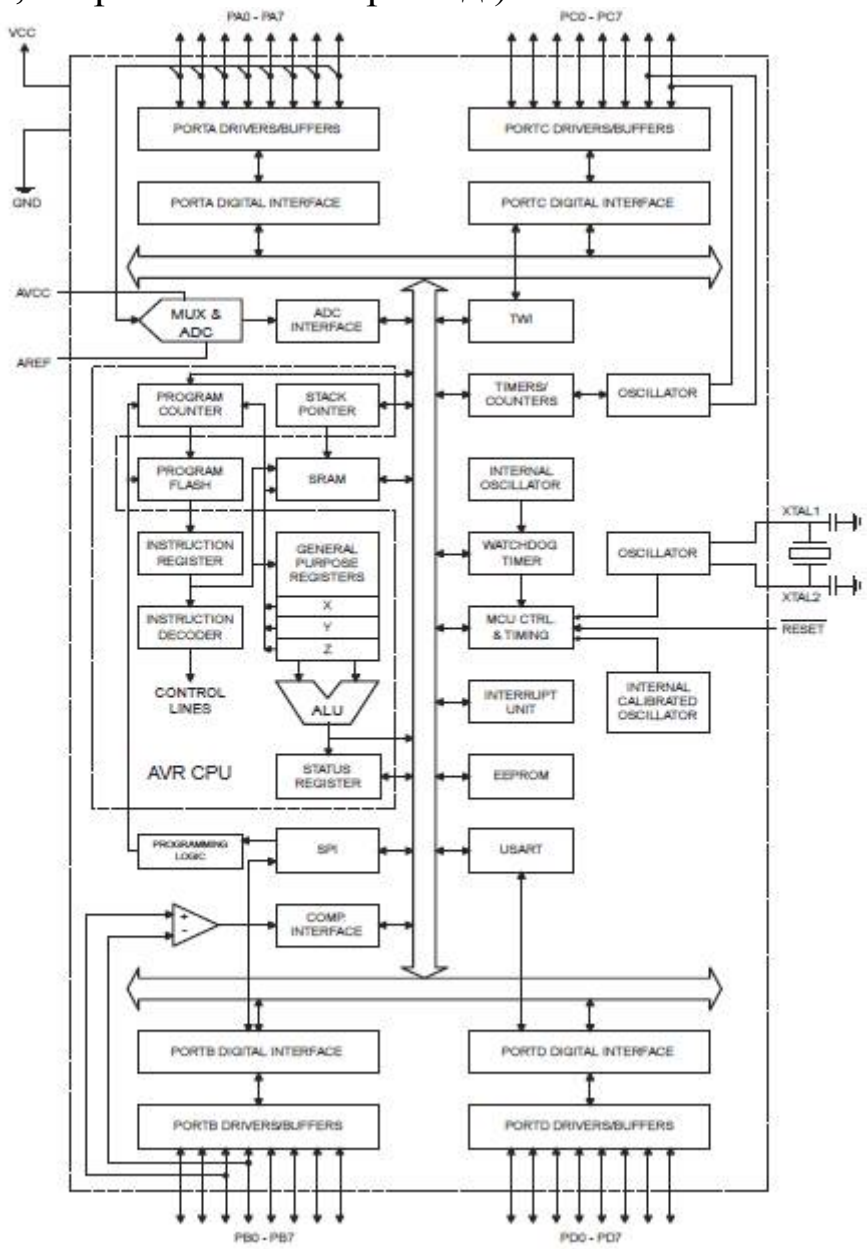


Рис. 1 – Архитектура ATMega 16

Следует отметить, что данные блоки могут работать независимо друг от друга, т.е. если в основной программе запущен аналого-цифровой преобразователь, ШИМ и принимаются данные по интерфейсам, то производительность центрального процессо-

ра не уменьшится. Если каждый принятый, оцифрованный либо выданный байт данных будет в основном коде программы обрабатываться, то это время обработки следует учесть.

Специальные разновидности микросхем могут содержать внутренние блоки USB, Ethernet, радиомодем, более точный АЦП, либо несколько одинаковых цифровых интерфейсов. Если стоит задача передавать данные по UART (COM Port) между несколькими устройствами, то следует выбрать нужную разновидность контроллера, в котором имеется четыре аппаратных блока UART для реализации задачи. Однако, цена у последнего будет выше, и возможно, будет отсутствовать часть базового функционала.

1.3. Тематические задания

- Основные свойства, области использования и особенности микропроцессоров общего назначения.
- Сравнительная характеристика RISC архитектуры.
- Сравнительная характеристика Принстонской архитектуры(Фон-Неймана).
- Сравнительная характеристика Гарвардской архитектуры.
- Характеристика аспектов построения процессорного ядра.

2. ОСНОВНЫЕ ПЕРИФЕРИЙНЫЕ МОДУЛИ МИКРОКОНТРОЛЛЕРОВ

2.1. Порты ввода/вывода A, B, C, D (I/O)

Порты ввода/вывода (Порт A, Порт B, Порт C, Порт D) AVR имеют число независимых линий "вход/выход" от 3 до 53 и обозначены стрелкой. Направление стрелки позывает возможное направление порта. Каждая линия порта может быть запрограммирована на вход (например, считывание значений кнопки) или на выход (светодиоды, микросхемы и т.п.). Каждая линия порта соответствует конкретной отдельной ножке микросхемы.

Нагрузочная способность:

- Выходные драйверы микроконтроллера обеспечивают нагрузочную способность 20 мА на линию порта (втекающий ток),

что позволяет, например, непосредственно подключать к микроконтроллеру светодиоды и биполярные транзисторы.

- Общая токовая нагрузка на все линии одного порта (A, B,C,D) не должна превышать 80-200 мА (в зависимости от корпуса и порта)
- Общая нагрузка на все линии всех портов 400 мА (все значения приведены для напряжения питания 5 В).

Возможно непосредственное подключение семисегментных индикаторов к микроконтроллеру если параметры нагрузочной способности не будут превышены при выводе цифры «88»

Через порты процессорное ядро взаимодействует с различными внешними устройствами: считывает значения входных сигналов и устанавливает значения выходных сигналов. Во встраиваемых системах в качестве внешних устройств чаще всего рассматриваются датчики, исполнительные устройства, устройства ввода-вывода данных оператором, устройства внешней памяти.

По типу сигнала различают порты:

1. Дискретные (цифровые) — используются для ввода/вывода дискретных значений логического «0» или «1». Иногда обозначаются как GPIO (Port input/output)

2. Аналоговые — через них вводятся сигналы на вход АЦП или других аналоговых схем и выводятся выходные сигналы ЦАП или других аналоговых схем. Обычно выведены отдельной линией на микропроцессоре.

3. Перестраиваемые — настраиваются на аналоговый или цифровой режим работы.

По направлению передачи сигнала различают:

1. Однонаправленные порты, предназначенные только для ввода (входные порты, порты ввода) или только для вывода (выходные порты, порты вывода).

2. Двунаправленные порты, направление передачи которых определяется в процессе программно-управляемой настройки схемы.

3. Порты с альтернативной функцией. Отдельные линии этих портов связаны со встроенными периферийными устройствами, такими, как таймер, контроллеры последовательных приемопере-

датчиков. Если соответствующий периферийный модуль не задействован, то линии можно использовать как обычные порты. Если модуль активизирован, то связанные с ним линии автоматически или «вручную» (программно) конфигурируются в соответствии с функциональным назначением и не могут быть использованы в качестве универсальных портов ввода-вывода. В некоторых случаях порты могут использоваться только для связи с периферийным модулем (например, входы АЦП в некоторых процессорах).

По алгоритму обмена различают порты:

1. С программно-управляемым (программным) вводом/выводом – установка и считывание данных определяется только ходом вычислительного процесса. Нет защиты от повторного считывания-записи одного и того же (не изменившегося) значения на выводе и считывания-записи во время переходного процесса на выводе.

2. Со стробированием – каждая операция ввода вывода подтверждается импульсом синхронизации (стробом) со стороны источника сигнала (при выводе – процессор, при вводе – внешнее устройство). Считывание информации приемником происходит только по стробу, что позволяет защититься от приема данных во время переходного процесса входного сигнала.

3. С полным квитированием. Данный режим чаще всего используется для обмена данными с другой вычислительной системой по параллельной шине. Кроме сигналов синхронизации со стороны передатчика используются сигналы подтверждения (готовности к следующему обмену) со стороны приемника. Это позволяет управлять интенсивностью обмена обоим взаимодействующим сторонам и предотвращает потерю данных, когда одна из них перегружена. Пример порта с квитированием служит порт LPT персонального компьютера. Во встроенных модулях процессоров данный режим чаще всего реализуется программно/аппаратно.

Каждый порт обслуживают три регистра:

- регистр данных – считывает значение (например, кнопки);
- регистр направления – настраивает на ввод/вывод;
- регистр выводов – выводит значение (ноль/единицу).

Описание регистров см в главе 6.

2.2. Аналоговый компаратор (AC)

Иногда требуется сравнить два напряжения между собой и если одно больше/меньше второго, то выполнить функцию. Примером может служить поддержание заданной температуры: если аналоговый датчик температуры выдал значение (в вольтах) меньше установленного, то можно включить нагрев, как только два напряжения сравняются, то результатом сравнения будет логическое значение, которое может быть прочитано из программы. Результатом сравнения является значение флага - либо 0 либо 1

В компараторе имеется дополнительная возможность отслеживать как произошло изменение: сигнал изменился с нуля до единицы либо с единицы до нуля.

Два входа компаратора выведены на конкретные отдельные ножки микросхемы, которые переназначить на другие порты («пины») нельзя.

2.3. Аналого-цифровой преобразователь (A/D CONVERTER)

Аналого-цифровой преобразователь (АЦП) служит для получения числового (количественного) значения напряжения, поданного на его вход. Результатом является целое беззнаковое число, сохраненное в отдельном регистре данных АЦП. Преобразование аналогового сигнала в цифровой длительная операция. Окончание преобразования обозначается либо отдельным флагом АЦП, который может циклически опрашиваться в программе, либо прерыванием выполнения основной программы и переходом на отдельную процедуру прерывания.

Какой из выводов («пинов») микроконтроллера будет являться входом АЦП, определяется числом, занесенным в соответствующий регистр. Имеется возможность переназначить вход АЦП на один из 8 выводов микросхемы, что позволяет последовательно оцифровать до 8 различных аналоговых сигналов.

2.4. Последовательный периферийный интерфейс SPI

Последовательный высокоскоростной периферийный трехпроводный интерфейс SPI (Serial Peripheral Interface) предназна-

чен для организации обмена данными между двумя устройствами. Самый быстрый последовательный встроенный интерфейс из имеющихся в микроконтроллере. Обычно применяется для связи внутри изделия: карты памяти SD, микросхемы памяти FLASH-ПЗУ, внешние АЦП/ЦАП, встроенные видеокамеры, другие микроконтроллеры, расположенные рядом.

Высокая скорость работы компенсируется небольшим расстоянием работы и меньшей помехозащищенностью в сравнении с UART, I2C и TWI.

Выводы SPI выведены на конкретные отдельные ножки микроконтроллера, которые переназначить на другие порты («пины») в программе нельзя.

Через интерфейс SPI может осуществляться программирование микроконтроллера.

2.5. Двухпроводной последовательный интерфейс TWI (I2C)

Двухпроводной последовательный интерфейс TWI (Two-wire Serial Interface) является полным аналогом базовой версии интерфейса I2C (двух проводная двунаправленная шина) фирмы Philips. Этот внешний интерфейс позволяет объединить вместе до 128 различных устройств с помощью двунаправленной шины, состоящей из линии тактового сигнала (SCL) и линии данных (SDA). Все 128 устройств соединены параллельно всего тремя проводами (SDA, SCL, GND), что облегчает монтаж. Часто применяется в различных датчиках: температуры, влажности, пожара, давления и т.п. Скорость передачи данных выше чем в UART, но ниже чем SPI

Выводы TWI/I2C выведены на конкретные отдельные ножки микроконтроллера, которые переназначить на другие порты («пины») в программе нельзя.

2.6. Интерфейс JTAG для отладки

Интерфейс JTAG был разработан группой ведущих специалистов попроблемам 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.

Позволяет в режиме реального времени наблюдать за состоянием регистров, портов, переменных в программе. Пошагово выполнять математические действия в написанной программе, отслеживать переходы и запросы ожидания.

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.

2.7. Таймеры/счетчики (TIMER/COUNTERS)

Микроконтроллеры AVR имеют в своем составе от 1 до 4 независимых таймеров с разрядностью 8 ($2^8 = 256$) или 16 ($2^{16}= 65\,535$) бит, которые могут работать и как таймеры от внутреннего/внешнего источника тактовой частоты, и как счетчики внешних событий (нажатие кнопки, входные импульсы и др.).

Таймеры используются для точного формирования временных интервалов, подсчета импульсов на выводах микроконтроллера, формирования последовательности импульсов. В режиме ШИМ (PWM) таймер/счетчик может представлять собой широтно-импульсный модулятор и используется для генерирования сигнала с программируемыми частотой и скважностью.

Результатом завершения счета является значение флагов - либо 0 либо 1.

Таймеры/счетчики способны вырабатывать различные запросы прерываний. Выводы таймеров/счетчиков выведены на конкретные ножки микроконтроллера, которые переназначить на другие порты («пины») в программе нельзя.

2.8. Сторожевой таймер (WDT)

Сторожевой таймер (WatchDog Timer) можно дословно перевести как Watch – смотреть, следить и Dog – собака. Имеет отдельные от других таймеров настройки, флаги и регистры. Основная задача следить за тем, чтобы при зацикливании («зависании») программы перезагрузить микроконтроллер и вернуть его в

рабочее состояние. Он имеет свой собственный встроенный в микроконтроллер RC-генератор, работающий на частоте 1 МГц.

Идея использования сторожевого таймера предельно проста и состоит в регулярном его сбрасывании под управлением программы или внешнего воздействия до того, как закончится его переполнение 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), либо зациклился, то произойдет полный сброс процессора, инициализирующий все регистры и приводящий систему в рабочее состояние.

Результатом является формирование импульса сброса микроконтроллера и его перезагрузка предотвращающая некорректную работу.

Внешние выводы («пины»), отвечающие за сторожевой таймер, отсутствуют.

2.9. Сброс при снижении напряжения питания (BOD)

BOD (Brown-Out Detection) отслеживает напряжение источника питания. При снижении напряжения питания ниже некоторого установленного программистом значения схема сброса переводит микроконтроллер в сброшенное удерживаемое состояние. Когда напряжение питания вновь увеличится выше порогового значения, запускается таймер задержки для игнорирования переходных процессов. После формирования задержки внутренний сигнал сброса снимается и происходит запуск микроконтроллера с начала программы.

2.10. Прерывания (INTERRUPTS)

Встроенная система прерываний позволяет гибко обрабатывать результаты работы параллельно запущенных задач. Прерывание прекращает нормальный ход основной программы для выполнения приоритетной задачи, определяемой внутренним или внешним событием. В каждом периферийном блоке существует свой уникальный набор событий. Для таймера - достижение максимального или минимального значения и др.; для компаратора - равенство напряжений, для АЦП - окончание преобразования,

для приемо/передатчиков – принятый микроконтроллером байт данных либо пустой буфер отправки данных др.

Для каждого такого события разрабатывается отдельная программа с уникальным именем, которую называют подпрограммой обработки запроса на прерывание (для краткости – подпрограммой прерывания). В каждом компиляторе (IAR, AVR Studio, Texas) обозначение уникально, чем затрудняется компиляция кода в разных программных продуктах. При возникновении события, вызывающего прерывание, микроконтроллер сохраняет в стеке содержимое счетчика команд, прерывает выполнение центральным процессором основной программы и переходит к выполнению подпрограммы обработки прерывания. После выполнения подпрограммы прерывания осуществляется восстановление предварительно сохраненного в стеке значения счетчика команд, и процессор возвращается к выполнению прерванной программы.

Чем больше размер подпрограммы прерывания, тем дольше в микроконтроллере будет остановлена основная программа. Поэтому следует стараться уменьшать размер кода в подпрограммах прерываний.

-Что если в момент выполнения прерывания произойдет другое прерывание, например пришли данные по другому интерфейсу (SPI, UART…)?

Для каждого события установлен приоритет согласно адресу его вектора прерывания – чем меньше адрес, тем больше приоритет. Вектор прерывания – адрес к которому переходит микроконтроллер при возникновении соответствующего прерывания. Соответственно наивысший приоритет имеет событие – сброс, вектор прерывания которого имеет нулевой адрес. По данному адресу расположен вход в основную *main()* процедуру, поэтому при включении микроконтроллер начинает выполнять основной код программы.

Понятие приоритет означает, что при одновременном возникновении двух событий первым обработается событие с большим приоритетом.

Таким образом, если в момент обработки прерывания приоритет нового прерывания будет меньше, то остановки выполнения не произойдет. Весь код в прерывании будет выполнен, ука-

затель вернется в основную программу main(), после чего будет обработан запрос на новое прерывание.

2.11. Тактовый генератор

Тактовый генератор вырабатывает синхронизирующие импульсы для всех устройств микроконтроллера. Каждый отдельный периферийный блок (АЦП, Таймер, шины передачи и т.д.) работает на своей частоте, но кратной основной частоте ядра в четное число раз (2,4,8…). Имеется встроенный тактовый генератор, часто используемый, если от микроконтроллера не требуется высокого быстродействия.

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.

Имеется 2 внешних вывода, отвечающие за подключение внешнего генератора. Могут не использоваться, если микроконтроллер работает от внутреннего генератора.

2.12. Система реального времени (RTC)

Предположим, требуется отсчитывать секундные, минутные, часовые интервалы времени. Для решения можно использовать один из четырех встроенных таймеров, однако при рабочей частоте 20МГц единичный отсчет составит 1/20МГц = 50нс и понадобится 20 миллионов отсчетов для достижения 1 секунды. Существует более удобный способ – использовать блок реального времени. Отдельный таймер/счетчик RTC имеет свой собственный предделитель, который может быть программным способом подключен или к основному внутреннему источнику тактовой частоты микроконтроллера, или к дополнительному асинхронному источнику опорной частоты (кварцевый резонатор или внешний синхросигнал). Для этой цели зарезервированы два внешних вывода микроконтроллера. Внутренний осциллятор, нагруженный на счетный вход таймера/счетчика RTC, оптимизирован для

работы с внешним «часовым» кварцевым резонатором 32,768 кГц, что в итоге дает точные секундные интервалы.

2.13. Память

В микроконтроллере, как и в персональном компьютере либо ноутбуке, тоже присутствуют различные типы памяти: жесткий диск - память программ, внутренняя память центрального процессора (кэш) – регистры общего назначения, оперативная память (RAM) – ОЗУ. В микроконтроллерах AVR реализована Гарвардская архитектура, в соответствии с которой разделены не только адресные пространства памяти программ и памяти данных, но и шины доступа к ним. Каждая из областей памяти данных (оперативная память и EEPROM) также расположена в своем адресном пространстве.

Память программ (Flash ROM или Flash ПЗУ)

Память программ предназначена для хранения последовательности команд, управляющих функционированием микроконтроллера, и имеет 16- битную организацию. Все AVR имеют Flash-память программ, которая может быть различного размера – от 1 до 256 Кб. Допускает многократное стирание и запись информации (гарантированное число циклов перезаписи не менее 10 тыс. циклов). Данные сохраняются после выключения питания.

Память данных

Память данных разделена на три части: регистровая память, оперативная память (ОЗУ – оперативное запоминающее устройство или SRAM) и энергонезависимая память (EEPROM).

Регистровая память (РОН и РВВ)

Регистровая память включает 32 регистра общего назначения (РОН или GPR), объединенных в файл, и служебные регистры ввода/вывода (РВВ). И те, и другие расположены в адресном пространстве ОЗУ, но не являются его частью (Рис. 2). Каждый из 32-х регистров общего назначения длиной 1 байт непосредственно связан с арифметико-логическим устройством (ALU) процессора. Другими словами, в AVR существует 32 регистра-

аккумулятора. Так, два операнда извлекаются из регистрового файла, выполняется команда и результат записывается обратно в регистровый файл в течение только одного машинного цикла, причем этот цикл равен периоду тактового генератора. Старшие регистры объединены парами и образуют три 16-разрядных регистра X, Y и Z, предназначенных для косвенной адресации ячеек памяти (AVR без RAM имеют только один 16-бит-ный регистр Z).

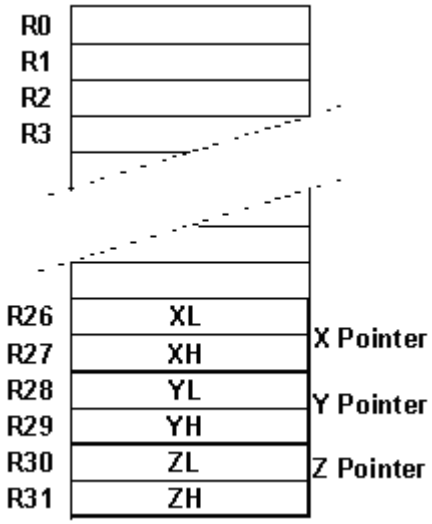


Рис. 2 – Регистровый файл ATMega

Все математические операции происходят через данные регистры. При написании, например, операции сложения C=A+B на языке C++ компилятор преобразует код так, чтобы числа A и B были расположены в каких-либо регистрах R0-R31, а результат был считан из R** и условно передан переменной C.

Следует помнить, что любой код C++ преобразуется сначала в ассемблерный код конкретного микроконтроллера (с ограниченным числом команд), который в последствии транслируется в машинные циклы.

По сути, работа микроконтроллера заключается в управлении этими регистрами.

Данные обнуляются после выключения питания.

Память EEPROM

Блок энергонезависимой электрически стираемой памяти данных EEPROM, доступен программе микроконтроллера непосредственно в ходе ее выполнения. Часто используется для хра-

нения промежуточных данных, различных констант, настроек устройства, таблиц перекодировок, калибровочных коэффициентов и т.п. Например, настройки уровня громкости в проигрывателе, последний просмотренный канал в телевизоре, настройки радио – станций, эквалайзера и др. будут сохранены при выключении/включении устройства. EEPROM также может быть загружена извне (отдельно от прошивки основной программы) как через SPI интерфейс, так и с помощью обычного программатора. Число циклов перезаписи — не менее 100000. Память программ имеет в 10-100 раз меньший ресурс.

Два программируемых бита секретности позволяют защитить память программ и энергонезависимую память данных EEPROM от несанкционированного считывания. Для записи в память всего лишь требуется при объявлении переменных указать, что она (например, *temp*) расположена в eeprom.

Данные сохраняются после выключения питания.

Оперативная память (ОЗУ или RAM)

Внутренняя оперативная статическая память Static RAM (SRAM) имеет байтовый формат и используется для оперативного хранения данных. При объявлении переменной, например, *char temp*, будет выделен/зарезервирован 1 байт, а при объявлении строки для вывода на индикатор *char stroka[] = "Hello World"* будет выделено 12байт

Таким образом, все объявленные переменные, массивы и матрицы располагаются в данной памяти.

Размер оперативной памяти может варьироваться у различных чипов от 64 байт до 32 Кб (или больше). Число циклов чтения и записи в RAM не ограничено. В старших ячейках SRAM обычно организуется стек, который заполняется в сторону уменьшения адреса ячейки. Данные обнуляются после выключения питания.

2.14. Тематические задания

- Основные параметры и особенности применения двунаправленных портов, настраиваемых на ввод или вывод программированием бита в регистре направления передачи.

- Принцип работы и основные параметры модуля АЦП на основе АЦП последовательного приближения.
- Принцип работы и основные свойства «классического» таймера.
- Основные возможности сторожевого таймера.
- Область применения системы реального времени.
- Назовите последовательные интерфейсы данных и их области применения.
- Разновидности встроенной памяти и области ее применения в микроконтроллерах.
- Принципы работы прерываний их приоритеты.

3. СИСТЕМА КОМАНД МИКРОКОНТРОЛЛЕРОВ AVR

3.1. Регистры состояния

Регистровый файл, блок регистров ввода/вывода и память данных образуют единое адресное пространство, что дает возможность при программировании обращаться к 32 оперативным регистрам и к регистрам ввода/вывода как к ячейкам памяти, используя команды доступа к RAM (в том числе и с косвенной адресацией). Младшие 32 адреса ($0 — $1F) соответствуют оперативным регистрам. Следующие 64 адреса ($20 — $5F) зарезервированы для регистров ввода/вывода. Внутренняя RAM начинается с адреса $60 (знак $ указывает на шестнадцатеричную систему счисления) [1].

Все математические операции в контроллерах AVR, как и в МК51, выполняются через аккумулятор. Выполнять арифметико-логические операции и операции сдвига непосредственно над содержимым ячеек памяти нельзя, как и нельзя записать константу или очистить содержимое ячейки памяти. Система команд AVR позволяет лишь выполнять операции обмена и пересылки данных между ячейками RAM и оперативными регистрами. Операции сложения, вычитания, умножения числа А на В происходят с использованием заранее определенных регистров, в которых число А расположено, например в регистре R1, число В в регистре R2, а расположение результата будет зависеть от выбранной математической операции. Следовательно, для удобства работы и увеличения производительности требуется расширенная система ко-

манд пересылки данных и адресации ячеек памяти. Достоинством системы команд AVR можно считать разнообразные режимы адресации ячеек памяти. Кроме прямой адресации, имеются следующие режимы: косвенная, косвенная с постинкрементом, косвенная с предекрементом и косвенная со смещением.

Регистры ввода/вывода располагаются в так называемом адресном пространстве ввода/вывода размером 64 байт. Их можно разделить на две группы: служебные регистры микроконтроллера и регистры, относящиеся к периферийным устройствам (в том числе порты ввода/вывода). Подробное описание данных регистров происходит одновременно с изучением конкретного периферийного узла.

Среди регистров ввода/вывода есть регистр, используемый наиболее часто в процессе выполнения программы. Это регистр статуса SREG. Он располагается по адресу $3F и содержит набор флагов (Таблица 1), показывающих текущее состояние микроконтроллера. Большинство флагов автоматически устанавливаются в соответствии с результатом выполнения команд. Все разряды SREG доступны как для записи, так и для чтения. После сброса микроконтроллера регистр обнулен.

Таблица 1 — Разряды регистра состояния SREG

Разряд	Название	Описание
7	**I**	**Общее разрешение прерываний.** Для разрешения прерываний этот флаг должен быть установлен в 1. Флаг сбрасывается аппаратно после входа в подпрограмму обслуживания прерываний и восстанавливается командой RETI для разрешения обработки следующих прерываний
6	**T**	**Хранение копируемого бита.** Заданный разряд любого РОН может быть скопирован в этот разряд командой BST или установлен в соответствии с содержимым данного разряда командой BLD
5	**H**	**Флаг потетрадного переноса.** Этот флаг устанавлива-

Разряд	Название	Описание
		ется в 1, если произошел перенос из младшей тетрады байта (из 3-го разряда в 4-й) или заем из старшей тетрады при выполнении некоторых арифметических операций
4	**S**	**Флаг знака.** Этот флаг равен результату операции «Исключающее ИЛИ» между флагами N и V. Он устанавливается в 1, если результат выполнения арифметической операции меньше нуля
3	**V**	**Флаг переполнения дополнительного кода.** Этот флаг устанавливается в 1 при переполнении разрядной сетки знакового результата
2	**N**	**Флаг отрицательного значения.** Этот флаг устанавливается в 1, если старший разряд результата операции (7 разряд) равен 1
1	**Z**	**Флаг нуля**. Этот флаг устанавливается в 1 при нулевом результате выполнения операции
0	**C**	**Флаг переноса.** Этот флаг устанавливается в 1, если в результате выполнения операции произошел выход за границы байта

Все регистры ввода/вывода могут считываться и записываться через оперативные регистры при помощи команд IN, OUT (см. группу команд передачи данных). Регистры ввода/вывода, имеющие адреса в диапазоне $00 — $1F, обладают возможностью побитовой адресации. Непосредственная установка и сброс отдельных битов этих регистров выполняется командами SBI и CBI (см. группу команд работы с битами). Для признаков результата операции, которые являются битами регистра ввода/вывода SREG, имеется целый набор команд установки и сброса. Команды условных переходов в качестве своих операндов могут иметь как биты-признаки результата операции, так и отдельные разряды побитно адресуемых регистров ввода/вывода.

Следует также иметь в виду, что у разных типов AVR одни и те же регистры ввода/вывода могут иметь различные адреса. Для того, чтобы обеспечить переносимость программного обеспечения с одного типа кристалла на другой, следует использовать в программе стандартные, принятые в оригинальной фирменной документации, символические имена регистров ввода/вывода, а соответствие этих имен реальным адресам задавать, подключая в начале своей программы (при помощи директивы ассемблера **.INCLUDE**) файл определения адресов регистров ввода/вывода. Файлы определения адресов регистров ввода/вывода имеют расширение **.inc**. Они уже созданы разработчиками фирмы ATMEL и свободно распространяются вместе с документацией на AVR-микроконтроллеры. В этих файлах задается соответствие символических имен основным адресам регистров ввода/вывода.

Младшие адреса памяти программ имеют специальное назначение. Адрес $000 является адресом, с которого начинает выполняться программа после сброса процессора. Начиная со следующего адреса ячейки памяти программ образуют область векторов прерывания. В этой области для каждого возможного источника прерывания отведен свой адрес, по которому (в случае использования данного прерывания) размещают команду относительного перехода RJMP на подпрограмму обработки прерывания. Следует помнить, что адреса векторов прерывания одних и тех же аппаратных узлов для разных типов AVR могут иметь разное значение. Поэтому для обеспечения переносимости программного обеспечения удобно, так же как и в случае с регистрами ввода/вывода, использовать символические имена адресов векторов прерывания, которые определены в соответствующем **inc**-файле.

В ячейках оперативной памяти организуется системный стек, который используется автоматически для хранения адресов возврата при выполнении подпрограмм, а также может использоваться программистом для временного хранения содержимого оперативных регистров (команды PUSH и POP). Стек растет от старших адресов к младшим, поэтому, учитывая, что начальное значение указателя стека после сброса равно нулю, программист AVR обязательно должен в инициализирующей части программы позаботиться об установке указателя стека, если он предполагает использовать хотя бы одну подпрограмму. Микроконтроллеры,

не имеющие RAM (семейства Tiny), содержат трехуровневый аппаратный стек.

3.2. Принцип реализации выполнения программы

В режиме выполнения основной программы процессор выбирает из памяти очередную команду программы и выполняет соответствующую операцию. Команда представляет собой многоразрядное двоичное число, которое состоит из двух частей (полей) — кода операции (КОП) и кода адресации операндов (КАД). Код операции КОП задает вид операции, выполняемой данной командой, а код адресации КАД определяет выбор операндов (способ адресации), над которыми производится заданная операция.

Для хранения адреса очередной команды служит специальный регистр процессора – программный счетчик PC (Program Counter), содержимое которого автоматически увеличивается на 1 после выборки следующего байта команды. Таким образом, обеспечивается последовательная выборка команд в процессе выполнения программы. При выборке очередной команды содержимое PC поступает на шину адреса, обеспечивая считывание из ОЗУ следующей команды выполняемой программы. При реализации безусловных или условных переходов (ветвлений) или других изменений последовательности выполнения команд выполняется загрузка в PC нового содержимого, в результате чего производится переход к другой ветви программы или подпрограмме.

Принятая из ОЗУ команда поступает в регистр команд, входящий в состав УУ процессора. Затем производится дешифрация команды, в процессе которой определяется вид выполняемой операции (расшифровка КОП) и формируется адрес необходимых операндов (расшифровка КАД). В соответствии с кодом поступившей команды УУ процессора генерирует последовательность микрокоманд, обеспечивающих выполнение заданной операции. Каждая микрокоманда выполняется в течение одного машинного такта — периода тактовых импульсов, задающих рабочую частоту всех внутренних узлов и блоков микропроцессора. Таким образом, тактовая частота микропроцессора определяет время выполнения отдельных микрокоманд, последовательность

которых обеспечивает получение необходимого результата операции (поступившей команды).

Для выполнения каждой поступившей команды требуется определенное количество командных циклов и тактов. Командным циклом называется промежуток времени, требуемый для выполнения обращения к ОЗУ или внешнему устройству с помощью системной шины. Обычно реализация такого цикла занимает от 2 до 4 системных тактов (периодов синхросигналов шины), которые требуются для установки требуемого адреса, выдачи сигналов, определяющих вид цикла – чтение или запись, получения сигнала готовности к обмену (от памяти или внешних устройств) и собственно передачи данных или команд.

При выполнении каждой команды в первых циклах производится ее выборка из памяти по адресу, который задается содержимым программного счетчика PC. Последующая дешифрация выбранной команды определяет необходимое число циклов для ее последующего выполнения. Если для выполнения команды не требуется считывание операндов из памяти (внешних устройств) или запись в память (вывод на внешние устройства) результатов операции, то такая команда выполняется за один цикл.

При считывании операндов из памяти (внешних устройств) или записи результата в память (вывод на внешние устройства) требуется выполнение дополнительных циклов чтения (ввода) или записи (вывода). В зависимости от разрядности обрабатываемых операндов и разрядности используемой системной шины число циклов, необходимых для выполнения команд, может быть различным: от 1 (выборка команды) до 4 – 5 (зависит от команды, разрядности шин и операндов).

Машинным (процессорным) тактом в микропроцессорных системах является длительность периода тактовых сигналов Tt, которая задается тактовой частотой микропроцессора Ft = 1/Tt. При выполнении операций, не требующих обращений к системной шине, эта частота определяет производительность микропроцессора.

3.3. Вызов подпрограммы на языке низкого уровня.

Обращение к подпрограмме реализуется при поступлении в микропроцессор специальной команды CALL, которая указывает

адрес первой команды вызываемой подпрограммы. Этот адрес загружается в PC, обеспечивая в следующем командном цикле выборку первой команды подпрограммы. Предварительно выполняется процедура сохранения в специальном регистре или ячейке памяти (в стеке) текущего содержимого PC, где хранится адрес следующей команды основной программы, чтобы обеспечить возвращение к ней после выполнения подпрограммы. Возврат к основной программе реализуется при поступлении команды RETURN (мнемоническое обозначение RET), завершающей подпрограмму. По этой команде сохранявшееся содержимое PC снова загружается в программный счетчик, обеспечивая выполнение команды, которая в исходной программе следовала за командой CALL.

Особенность этой процедуры состоит в том, что большинство МП обеспечивают возможности вложения подпрограмм, т. е. реализуют при выполнении подпрограммы вызов новой подпрограммы с последующим возвращением к предыдущей подпрограмме. При вложении нескольких подпрограмм требуется сохранение нескольких промежуточных значений содержимого PC и последовательная загрузка этих значений в PC при возврате к предыдущим подпрограммам и к основной программе.

Для реализации этой процедуры используется стек – специальная память магазинного типа, работающая по принципу «последний пришел — первый ушел» (стек типа LIFO—«Last In-First Out»). Существуют различные варианты реализации стека.

Регистровый стек реализуется с помощью реверсивных сдвиговых регистров (их число строго ограничено). Каждая команда CALL вызывает ввод в стек очередного содержимого PC. По команде RETURN направление сдвига изменяется и производится извлечение из стека последнего поступившего содержимого PC. Таким образом обеспечивается выполнение вложенных подпрограмм. Возможное число вложенных подпрограмм определяется глубиной стека, т. е. разрядностью используемых регистров сдвига. Если число вложений превышает глубину стека, первые из введенных в стек значений PC теряются, т. е. возврат к основной программе не будет обеспечен. Поэтому при использовании регистрового стека необходим строгий контроль за числом вложений. Такая реализация стека применяется в системах, решающих зада-

чи с ограниченным числом вложенных подпрограмм (обычно не более 10 – 20).

Значительно более широкие возможности вложения подпрограмм обеспечивает реализация стека в ОЗУ. В этом случае часть ОЗУ выделяется для работы в качестве стека. Адресация к ячейкам стека производится с помощью специального регистра указателя стека SP (Stack Pointer), который вводится в состав УУ процессора. Регистр SP содержит адрес верхней заполненной ячейки стека, в которой хранится значение PC, записанное при выполнении команды CALL. При поступлении новой команды CALL содержимое SP автоматически уменьшается на 1, адресуя следующую, еще незаполненную ячейку стека. Полученный адрес SP–1 выдается на шину адреса, а на шину данных поступает содержимое PC, которое должно сохраняться в стеке. Таким образом, производится последовательное заполнение ячеек стека «снизу-вверх», при этом SP всегда адресует вершину стека. По команде RETURN текущее содержимое SP выдается на шину адреса, и по шине данных производится считывание с вершины стека последнего записанного значения PC. После этого содержимое SP увеличивается на 1, адресуя предыдущее значение PC, хранящееся в стеке. Так как ОЗУ обычно имеет значительный объем, то для размещения стека можно выделить достаточно большое количество ячеек памяти, обеспечивая необходимый уровень вложения подпрограмм.

3.4. Язык Ассемблера для микроконтроллеров AVR

Операнды могут быть таких 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).

Ассемблер не различает регистр символов.

В таблице ниже приведены наборы команд процессора. Например, записав математическое выражение на языке высокого уровня (Си) содержащее квадратные корни, экспоненты, синусы и т.п. они будут преобразованы компилятором в простейшие команды (Таблица 2 - Таблица 5). Список команд уникален. В новые модели производители контроллеров обычно стараются добавить поддержку дополнительных команд (например, аппаратное деление чисел, умножение, экспоненты…)

Таблица 2 — Арифметические и логические команды процессоров AVR Atmega

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 · Rr	Z,N,V,S	1
ANDI	Rd,K8	Логическое И с константой	Rd = Rd · K8	Z,N,V,S	1
OR	Rd,Rr	Логическое ИЛИ	Rd = Rd V Rr	Z,N,V,S	1
ORI	Rd,K8	Логическое ИЛИ с константой	Rd = Rd V K8	Z,N,V,S	1
EOR	Rd,Rr	Логическое исключающее ИЛИ	Rd = Rd 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

Мнемоника	Операнды	Описание	Операция	Флаги	Циклы
SBR	Rd,K8	Установить бит (биты) в регистре	Rd = Rd V K8	Z,C,N,V,S	1
CBR	Rd,K8	Сбросить бит (биты) в регистре	Rd = Rd · ($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 · Rd	Z,C,N,V,S	1
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 - K 6	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) << 1	Z,C	2
FMULS	Rd,Rr	Умножение дробных чисел со знаком	R1:R0 = (Rd *Rr) << 1	Z,C	2
FMULSU	Rd,Rr	Умножение дробного числа со знаком с числом без знака	R1:R0 = (Rd * Rr) << 1	Z,C	2

Таблица 3 — Команды ветвления AVR Atmega

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
RCALL	k	Относительный вызов подпро-	STACK = PC+1, PC = PC + k +	None	3/4*

Мнемоника	Операнды	Описание	Операция	Флаги	Циклы
		граммы	1		
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
SBRS	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
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

Мнемоника	Операнды	Описание	Операция	Флаги	Циклы
BRVC	k	Перейти если флаг переполнения очищен	if(V==0) PC = PC + k + 1	None	1/2
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 (128KB памяти программ). Для устройств с памятью программ свыше 128KB добавьте пять циклов плюс по три цикла на каждое ожидание.

Таблица 4 — Команды передачи данных AVR Atmega

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	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 пост-инкрементом	Rd = (X), X=X+1	None	2*
LD	Rd,-X	Косвенная загрузка с предекрементом	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 предекрементом	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 предекрементом	Z=Z-1, Rd = (Z)	None	2*

Мнмоника	Операнды	Описание	Операция	Флаги	Циклы
LDD	Rd,Z+q	Косвенная загрузка с замещением	Rd = (Z+q)	None	2*
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 предекрементом	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 предекрементом	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 предекрементом	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	-
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.

Таблица 5 — Команды работы с битами AVR Atmega

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
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
CLS	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 энергопотреб-	Смотрите описание инструкции	None	1

Мнемоника	Операнды	Описание	Операция	Флаги	Циклы
		ление)			
WDR	Нет	Сброс сторожевого таймера	Смотрите описание инструкции	None	1

3.5. Директивы ассемблера

Компилятор поддерживает ряд директив. Директивы представляют собой команды управления компилятором. Их не следует путать с командами микропроцессора. Директивы не прошиваются в контроллер, а используются для удобства написания и отладки кода. Вместо этого они используются для указания положения в программной памяти, определения макросов, инициализации памяти и т.д. Конечный размер выходного файла hex (прошивка) не увеличится при использовании директив. Все директивы начинаются с точки. Общее число более 20 и зависит от компилятора (т.к. это команды компилятору).

Часто используемые:

– INCLUDE - Вложить другой файл
– EXIT - Выйти из файла
– LIST/ NOLIST - Включить/выключить генерацию листинга
– EQU - Установить постоянное выражение
– SET - Установить переменный символический эквивалент выражения
– DEF - Назначить регистру символическое имя
– DB - Определить байты во флэш или EEPROM
– DW - Определить слова во флэш или EEPROM
– BYTE - Зарезервировать байты в ОЗУ
– CSEG - Определить программный сегмент
– DSEG - Определить сегмент данных
– ESEG - Определить сегмент EEPROM
– ORG - Установить положение в сегменте
– DEVICE - Определить устройство для которого компилируется программа

Директива ***.include*** подставляет текстовый файл в то место программы, где происходит ее употребление. Внутри подставляемого файла возможно использовать свои другие .include файлы. Путь к файлу указывается либо полный, либо просто имя файла, если файл расположен в директории проекта или в одной из служебных папок.

Директива	*.include*
Синтаксис	*.include "{путь к файлу}"*
Пример:	*.include "m16def.inc"; вставка стандартного заголовочного файла*

Директива ***.exit*** указывает ассемблеру место окончания файла исходного текста. Все команды, находящиеся после директивы, становятся недоступными для компилятора. Если .exit встречается в подключаемом файле, то сборка проекта заканчивается строкой, где расположена директива .include. При отсутствии директивы .exit, концом программы считается последняя строка исходного текста.

Директива	*.exit*
Синтаксис	*.exit*
Пример	*.exit ; конец файла*

Директивы ***.nolist*** и ***.list*** служат для управления файлом листинга, который обычно генерируется после сборки проекта. Первая из них запрещает, а другая, соответственно, разрешает вывод информации в файл. Директива .list отменяет действие .nolist и наоборот.

Директивы	*.nolist, .list*
Синтаксис	*.nolist, .list*
Пример	*.nolist ;запретить вывод текста файла “m16def.inc”* *.include "m16def.inc" ;в файл листинга программы* *.list ;продолжить вывод информации*

Директива ***.equ*** присваивает символьному имени некоторое числовое значение. Символьное имя должно быть уникальным,

не может начинаться с точки, цифры и спец. символов и не может быть изменено в процессе написания программы. Директива не может применяться для назначения символьных имен регистрам общего назначения. Математические формулы, используемые в выражении, вычисляются компилятором, а не микроконтроллером.

Директива	*.equ*
Синтаксис	*.equ {символьное имя} = {выражение}*
Пример	*.equ DDRA = 0x17; присвоение имени DDRA значения 0x17* *.equ PORTA = DDRA + 5; присвоение имени PORTA значения 0x1C*

Директива ***.set*** производит то же самое действие, что и .equ. Но в отличии от последней, символьное имя может быть переопределено в любом месте программы.

Директива	*.set*
Синтаксис	*.set {символьное имя} = {выражение}*
Пример	*.set OFFSET_X = 0x200; присвоение имени OFFSET значения 0x200* *------|ваш код|-------* *.set OFFSET_X = OFFSET_X + 1 ;переопределение значения OFFSET*

Директива ***.def*** присваивает ваше символьное имя одному из регистров общего назначения. В дальнейшем ходе программы данное имя может быть отменено директивой .undef.

Директивы	*.def, .undef*
Синтаксис	*.def {символьное имя} = { регистр общего назнач.}* *.undef {символьное имя}*
Пример	*.def temp = R15; присвоение регистру R15 имя temp* *------|ваш код|-------* *.undef temp ; отмена дальнейшего использования имени temp*

Директивы ***.db, .dw, .dd, .dq*** предназначены для резервирования памяти **во FLASH или EEPROM** микроконтроллера под

инициализированные данные. Все они могут применяться только в сегментах кода и EEPROM-памяти. Разница между этими директивами заключается в разрядности, представляемых данных. Директива .db резервирует байты, .dw – слова, .dd – двойные слова. В редких случаях может так же оказаться удобным использование директивы .dq, резервирующей 64-разрядные данные. Данные могут представлять собой буквы ASCII формате, строки теста, в последствии выводимых на дисплей, либо начальные значения уставок в приборе.

Директивы	*.db, .dw, .dd, .dq*
Синтаксис	*{метка}: .db {8-разрядные данные}* *{метка}: .dw {16-разрядные данные}* *{метка}: .dd {32-разрядные данные}* *{метка}: .dq {64-разрядные данные}*
Пример	*label:* *.db 0xFA, 250, -6, 0b11111010* *.dw 0xFADE, 64222, -1314, 0b1111101011011110* *.dd 0xFADEEFCA, 4208914378, -86052918* *.dq 0xFADEEFCAEFBACDEF,* *18077149609196178927, -521103510453211*

Директива ***.byte*** резервирует память под неинициализированные данные в сегментах SRAM и EEPROM.

Директива	*.byte*
Синтаксис	*{метка}: .byte {количество резервируемых данных}*
Пример	*.equ PAGESIZE = 0x20* *buffer: . byte 2*PAGESIZE; резервирование 64 байт в SRAM*

Директивы ***.dseg, .eseg, .cseg*** определяют начало сегментов кода, данных и EEPROM-памяти соответственно, т.е. размещает данные в разных типах памяти микроконтроллера (памяти программ, данных, EEPROM). Первая буква директивы обозначает тип памяти, например .eseg отвечает за e – EEPROM, d – данных, c – кода. В исходном файле каждый из сегментов может быть

представлен только в одном экземпляре. В случае если все эти директивы отсутствуют в программе, компилятор по умолчанию считает, что все операторы расположены в секции кода.

Директивы	*.dseg, .eseg, .cseg*
Синтаксис	*.dseg* *.eseg* *.cseg*
Пример	*.dseg ; начало сегмента данных* *buffer: .byte 32 ; резервирование 32 байт под буфер в SRAM* *.cseg ;начало сегмента кода* *rjmp initial* *string: .db "ATmega16",0 ; строка, хранящаяся во FLASH-памяти* *.eseg ;начало сегмента EEPROM-памяти* *_var: .byte 2 ;резервирование 2-ух байт под переменную _var* *_cnst: .db 0xAA ;резервирование байта под переменную _cnst = 0xAA*

Директива ***.org*** позволяет задать компилятору начальный адрес в пределах сегментов кода, данных и EEPROM-памяти. В случае применения в сегменте кода, директива определяет адрес размещения 16-разрядного слова программ.

Директива	*.org*
Синтаксис	*.org {начальный адрес}*
Пример	*.equ SRAM_START = 0x60* *.equ RAMEND = 0x045F* *.dseg ;начало сегмента данных* *.org SRAM_START ;резервирование 32 байт в SRAM под буфер,* *buffer: . byte 32 ;начиная с адреса 0x60* *.cseg ;начало сегмента кода*

	.org 0 ;вектор сброса по адресу 0 *rjmp initial* *.* *.org 0x50 ;начало основной программы с адреса 0x50* *initial:* *ldi temp,high(RAMEND) ;инициализация стека* *out SPH,temp* *ldi temp,low(RAMEND)* *out SPL,temp* *.*

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.

Директива	*.DEVICE*
Синтаксис	*.DEVICE {контроллер}*
Пример	*.DEVICE AT90S1200 ; Используется AT90S1200*

3.6. Форматы представления чисел

- Десятичный (принят по умолчанию): 10, 255.
- Шестнадцатеричный (два варианта записи): 0x0a, $0a, 0xff, $ff .
- Двоичный: 0b00001010, 0b11111111.
- Восьмеричный (начинаются с нуля): 010, 077.

Компилятор поддерживает ряд операций, и чем выше положение в таблице, тем выше приоритет операции (Таблица 6). Выражения могут заключаться в круглые скобки, такие выражения вычисляются перед выражениями за скобками.

Таблица 6 — Операторы присвоения языка ассемблер

Приоритет	знак	описание	Результат
14	!	Логическое отрицание	Возвращает 1 если выражение равно 0, и наоборот
14	~	Побитное отрицание	Возвращает выражение в котором все биты проинвертированы
13	*	Умножение	Возвращает результат умножения двух выражений
13	/	Деление	Возвращает целую часть результата деления левого выражения на правое
12	+	Суммирование	Возвращает сумму двух выражений
12	-	Вычитание	Возвращает результат вычитания правого выражения из левого
11	<<	Сдвиг влево	Возвращает левое выражение сдвинутое влево на число бит указанное справа
11	>>	Сдвиг вправо	Возвращает левое выражение сдвинутое вправо на число бит указанное справа
10	<	Меньше чем	Возвращает 1 если левое выражение меньше чем правое (учитывается знак), иначе 0
10	<=	Меньше или равно	Возвращает 1 если левое выражение меньше или равно чем правое (учитывается знак), иначе – 0.
10	>	Больше чем	Возвращает 1 если левое выражение больше чем правое (учитывается знак), иначе- 0.
10	>=	Больше или равно	Возвращает 1 если левое выражение больше или равно чем правое (учитывается знак), иначе 0.
9	==	Равно	Возвращает 1 если левое выражение равно правому (учитывается знак), иначе 0.
9	!=	Не равно	Возвращает 1 если левое выражение не равно правому (учитывается знак), иначе 0.
8	&	Побитное И	Возвращает результат побитового И выражений
7	^	Побитное исключающее ИЛИ	Возвращает результат побитового исключающего ИЛИ выражений
6	|	Побитное ИЛИ	Возвращает результат побитового ИЛИ выражений
5	&&	Логическое И	Возвращает 1 если оба выражения не равны нулю, иначе 0
4	||	Логическое ИЛИ	Возвращает 1 если хотя бы одно выражение не равно нулю, иначе 0

3.7. Тематические задания

- Пояснить разряды регистра состояния SREG
- Назначение программного счетчика
- Принцип выполнения программы

- Назначение и принцип работы указателя стека
- Перечислить часто используемые директивы ассемблера
- Пояснить математические и логические операторы присвоения

4. ЯЗЫК СИ ДЛЯ МИКРОКОНТРОЛЛЕРОВ

4.1. Математические и логические операции присвоения

Чтобы поместить число в переменную (в регистр), в Си есть **оператор присваивания**. Это знак = (называемый в математике «равно»). В Си этот знак не означает равенство. Знак = в Си означает — вычислить результат того, что справа от оператора присваивания, и поместить этот результат в переменную, находящуюся левее оператора присваивания.

Регистры микроконтроллера в программе на Си имеют названия, и описаны в оригинальной технической документации фирмы ATMEL AVR Data Sheets [6]. В программе они выступают в роли переменных, в которые можно что-нибудь писать и читать.

Ниже приведены примеры команд на Си, использующие оператор присваивания.

```
PORTB = PINB + 57; /*Взять (прочитать, считать) значение переменной (регистра) PINB, затем прибавить к нему число 57 и поместить результат в переменную PORTB */
PORTB&=0x5A; /*Прочитать значение переменной PORTB, затем выполнить «поразрядное (побитное) логическое И» между прочитанным значением и числом 0x5A и записать результат в переменную PORTB */
PORTB = 0x23; /*Не читая содержимое переменной PORTB, присвоить ей значение 0x23 */
```

Вместо & «И» могут быть и другие побитные логические операции: | «ИЛИ», ^ «Исключающее ИЛИ», ~ «инвертирование битов» и арифметические операции: + - * / %.

Запомните! Результатом поразрядных (побитных) логических операций (& | ^ ~) является число, которое может быть интерпретировано компилятором как «истина», если оно не ноль, и «ложь», если число ноль.

Целые числа в компиляторе могут быть записаны:

- в десятичной форме: 1234;
- в двоичной форме с префиксом 0b: 0b101001;
- в шестнадцатеричной форме с префиксом 0x: 0x5A;
- в восьмеричной форме с префиксом 0: 0775.

С оператором присваивания используются сокращения вида (Таблица 7).

Таблица 7 — Операторы присваивания языка Си

Длинная запись	Смысл	Сокращается до
x = x + 1;	добавить 1	x++; или ++x;
x = x - 1;	вычесть 1	x--; или --x;
x = x + y;	прибавить y	x += y;
x = x - y;	вычесть y	x -= y;
x = x * y;	умножить на y	x *= y;
x = x / y;	поделить на y	x /= y;
x = x % y;	остаток от деления	x %= y;
x--;	вычесть 1	x -= 1;
x++;	добавить 1	x += 1;

Есть в Си операции, которые изменяют значение переменной и без оператора присваивания, например:

```
PORTA++;  /* Взять значение переменной PORTA, добавить к ней 1 и записать результат обратно в PORTA — инкрементировать регистр PORTA */
PORTC--;  /* Эта строчка на Си означает обратное действие — декрементировать значение регистра PORTC */
```

Инкремент и декремент удобно использовать для изменения значения различных переменных-счетчиков. Важно помнить, что они имеют очень низкий приоритет. Поэтому, чтобы быть уверенным в порядке выполнения, желательно писать их отдельной строчкой программы.

Когда инкремент или декремент используется в выражении, то важно, где стоят два знака + или – (перед переменной или после переменной):

```
A=4;
B=7;
A=B++; /* Взять значение переменной B, присвоить его переменной A, затем добавить 1 к переменной B и сохранить результат в B. Теперь A будет содержать число 7, B будет содержать число 8 */
A=4;
```

```
B=7;
A=++B; /* Взять значение переменной B, затем добавить к нему 1 и сохранить результат в B и этот же результат присвоить переменной A. Теперь A будет содержать число 8 и B будет содержать число 8 */
```

Арифметические операции в Си:

```
x+y          //сложение
x-y          // вычитание
x * y   // умножение
x / y          /* деление. Если числа целые, результат — целое число с отброшенной дробной частью — не округленное! То есть если в результате деления на калькуляторе получается 6.23411 или 6.94, то результат будет просто целое число 6. Если числа с плавающей точкой, то есть float или double, записываются с точкой и числом после точки, то и результат будет число с плавающей точкой */
x % y   // вычислить остаток от деления нацело.
```

Примеры использования:

```
5 / 2          // даст  2
5 % 2          // даст 1
75 / 29        // даст 2
75 % 29        // даст 17
```

Операторы сравнения (или отношения) используются для сравнения переменных, чисел (констант) и выражений:

```
x < y     // x меньше y
x > y     // больше
x <= y   // меньше или равно
x >= y   // больше или равно
x == y   // равно
x != y    // не равно
```

Результат выполнения этих операторов: «истина» это «1» (точнее «не ноль»), «ложно» это «0». Значения, хранимые в переменных (в регистрах) x и y, не изменяются!

Логические операции:

В результате логической операции вы получаете не число, а логическое значение «истина» или «ложь».

||	«ИЛИ»
&&	«И»
!	«НЕ»
!(истина)	дает «ложь»
!(ложь)	дает «истина»

Для логических операций && и || берутся результаты выражений слева и справа от знака операции, преобразованные в «истину» или «ложь», и определяется логический результат операции. Компилятор результат «истина» превращает в 1, а «ложь» в 0.

4.2. Операторы сдвига

Оператор сдвига битов имеет следующий синтаксис:

C = (a << b), вернее a << b,

где a – это то число, двоичное представление которого нужно сдвинуть, а b показывает, на сколько бит нужно его сдвинуть.

При этом возможна потеря значения, хранящегося в a (т.е. не всегда возможно восстановить из C то, что было в a).

Рассмотрим пример 1:

```
C = (9 << 3);
```

Число 9 в двоичном коде будет выглядеть как 0000 1001, а после сдвига влево на 3 бита получим C = 0100 1000.

Что может использоваться для арифметического умножения на 2/4/8/16/32…

Т.е. C = 0100 1000 = 72

Если 72 / 8, то получим начальное число 9.

Аналогично существует и сдвиг вправо, используемый так же как деление на 2/4/8/16/32....

Рассмотрим пример 2

```
char C;
…
C = ((0xFF << 2) >> 2);
```

Сначала выполнится действие во внутренних скобках (0xFF – это 255 в шестнадцатеричном коде), из 11111111 получится 11111100, потом произойдет сдвиг вправо и получим C = 00111111. Как видим, здесь две взаимно обратные операции привели к другому числу, т. к. мы потеряли два бита. Этого не произошло бы, если бы переменная C была типа int, т. к. int занимает 16 бит.

Рассмотрим еще два битовых оператора, широко применяющиеся при программировании МК (Таблица 8). Это оператор «побитовое и» (&) и «побитовое или» (|).

Таблица 8 — Операторы сдвига языка Си

Действие	**Результат**	**Описание**
C = 0;	// C = 00000000	Установить в 0
C = (1 << 5) | (1 << 2) | (1 << 0);	// C = 00100101	Установить в единицу 0, 2,5 биты
C |= (1 << 3);	// C = 00101101	Добавить единицу к 3-ему биту
C &= (0xF0 >> 2);	// C = 00101100	Отчистить 2 младших (правых) бита
C = (C & 4) | 3;	// C = 00000111	Выполнить логическую операцию

4.3. Команды условных переходов Си

Выделим команды условий и рассмотрим их подробно:

- if(){}else{};
- while(){};
- for(;;){};
- switch(){};
- goto

4.3.1.Команды if(){}else{};

Идеальная конструкция, если вам нужно выполнить какую-то часть программы при наличии каких-либо условий:

```
if (выражение)
{
/* делать этот код, если выражение «истина» — т.е. резуль-
```

```
тат его вычисления не ноль */
 }
else
      { /* делать этот код, если выражение «ложь» — т.е. резуль-
тат его вычисления равен нулю */
};
```

else { } это не обязательный элемент конструкции:

Возможна простая запись:

```
      if (выражение)
      { /* делать этот код, если выражение «истина» — т.е. ре-
зультат его вычисления не ноль */
        };
```

4.3.2.Команда while(){};

Условный цикл — используйте, если вам нужно выполнять какой-то код программы, пока выполняется (существует, справедливо, не ноль) некоторое условие:

```
      while (выражение)
      { /* делать этот код, если выражение «истина» — т.е. ре-
зультат его вычисления не ноль. Пока выполняется этот код, выражение не проверяется на истинность. После выполнения кода происходит переход к строке while, чтобы снова проверять истинность выражения */
        };
```

Цикл while имеет вариант do — while, при котором код в { } выполняется по меньшей мере один раз независимо от истинности условия в скобках:

```
      do { /* сделать этот код один раз, затем, если выражение
есть «истина» — т.е. результат его вычисления не ноль — опять делать код с начала, и так до тех пор, пока выражение «истина» */ }
while (выражение);
```

4.3.3.Команда for(;;){};

Этот цикл позволяет выполнить часть программы нужное число раз:

```
char i; /* объявление переменной для for. Это обычная переменная и, значит, может иметь любое допустимое имя по вашему желанию */
for (i=5;i<20;i+=4)
{ /* код цикла for. i=5 — это начальное выражение. Число 5, просто для примера, может быть таким, как позволяет объявление переменной i, в нашем случае от 0 до 255. i<20 — контрольное выражение. Может быть с разными операторами отношения, важно лишь, чтобы по ходу цикла оно становилось когда-то «ложью» — иначе цикл «зациклится», т.е. никогда не кончится. i+=4 — счетчик. Обычно это i++, т.е. к переменной добавляется 1 каждый «прогон» цикла. Но может быть таким, какое вам требуется, важно лишь достижение когда-либо условия, оговоренного выше! Иначе цикл станет бесконечным. Код цикла for будет первый раз выполнен для i=5, затем по выражению i+=4 i станет 9. Теперь будет проверено контрольное выражение i<20, и так как 9<20, код цикла for будет выполнен еще раз. Так будет происходить до тех пор, пока контрольное выражение «истинно». Когда оно станет «ложно», цикл for закончится и программа пойдет дальше. */
};
```

Начальным условием может быть любое допустимое в Си выражение, результатом которого является целое число. Контрольное выражение определяет, до каких пор будет выполняться цикл. Счетчик показывает, как изменяется начальное выражение перед каждым новым выполнением цикла.

Циклы for(;;) и while() часто используют вот так:

```
while(1);
for (;;);
/* Так написанные эти циклы означают: МК выполнять эту строчку, пока есть питание, нет сброса и нет прерывания. Когда возникает прерывание, программа переходит на обработчик прерывания и (если в обработчике нет перехода в другое место программы) по завершении кода обработчика опять возвращается в такой цикл */
```

4.3.4.Команда switch(){};

Оператор множественного выбора, позволяет сделать выбор из нескольких вариантов.

```
switch (выражение)
{
case 7:
/* этот код будет выполняться, если результат вычисления выражения равен числу 7. На этом работа оператора switch закончится */
break;
case -28:
/* этот код будет выполняться, если результат вычисления выражения равен отрицательному числу -28. На этом работа оператора switch закончится */
break;
case 'G':
/* этот код будет выполняться, если результат вычисления выражения равен числу, соответствующему символу G в таблице ASCII. На этом работа оператора switch закончится */
break;
default:
/* этот код будет выполняться, если результат вычисления выражения не равен ни 7, ни -28, ни 'G'. А так же после выполнения кода, не имеющего в конце  break. На этом работа оператора switch закончится */
};
 /* switch закончен — выполняется дальнейший код программы */
```

case может быть столько, сколько вам нужно. Чтобы программа работала быстрее, старайтесь наиболее вероятные варианты располагать выше!

default — не обязателен. Его можно расположить и не в конце.

break; — если его не использовать, то, найдя нужный вариант, программа будет выполнять и следующие ниже условия case.

4.3.5.Команда goto

оператор безусловного (немедленного) перехода.

```
mesto_5: /* сюда мы попадем после выполнения строки программы goto mesto_5   */
goto mesto_1; /* перейти в то место программы, где в начале строки написано mesto_1:     */
goto mesto_5; /* перейти в то место программы, где в начале строки написано mesto_5:     */
mesto_1: /* сюда мы попадем после выполнения строки программы goto mesto_1   */
```

goto существует наверно во всех языках и в ассемблере в том числе. Используйте его с осторожностью! Например: если вы покинете функцию-обработчик прерывания по goto, не завершив ее, то не произойдет автоматического включения прерываний глобально — т.е. не установится бит I в регистре SREG. Этот бит устанавливается автоматически после полного выполнения функции обработки прерывания и «естественного» выхода из неё.

4.4. Структура программы на языке Си

Типовая компоновка программы на языке С выглядит следующим образом:

```
# include //подключение файлов

Объявление глобальных переменных //могут отсутствовать

Фунцкции //могут отстствовать
 {
 }

Прерывания //могут отсутствовать
 {
 }

int main()  //обязательная запись
{
```

```
инициализация;

 Главный БЕСКОНЕЧНЫЙ цикл.
   {
   … собственно программа …
   }
}
```

Рассмотрим более подробно структуру программы:

1)заголовок (// …/*….*/) – надпись, позволяющая разобраться кем и когда написана программа;

2) включение необходимых внешних файлов (#include);

3) ваши макросы/определения для удобства работы (#define);

4)объявление глобальных переменных (глобальные переменные объявляются вне какой-либо функции, т.е. не после фигурной скобки {, доступны в любом месте программы, значит, можно читать их значения и присваивать им значения там, где требуется);

5) описание функций (ISR) – обработчиков прерываний;

6) описание других функций, используемых в программе;

7) функция main () (это единственный обязательный пункт).

Функция имеет { "тело" } в фигурных скобках. Тело — это код на Си, определяющий то, что делает функция. Знак ; после функции не ставится.

Программа на Си начинает работу с функции main(), по необходимости из main() вызываются другие функции программы, по завершении работы функции программа возвращается в main() в то место, откуда функция была вызвана.

```
main()
{
... какой то код программы ...
вызов функции_1; //программа перейдет в функцию_1
строка программы; // будет выполняться после возврата
... какой то код программы ...
}
```

Функции могут вызываться не только из main(), но и из других функций. Кроме того, описанный выше ход программы может нарушаться прерываниями.

Приведем пример программы на Си с описанной выше структурой (текст в рамке).

/* **Заголовок программы**
Он оформляется как комментарий и обычно содержит:
– информацию о названии, назначении, версии и авторе программы;
– краткое описание алгоритма программы;
– пояснения о назначении выводов МК;
– другие сведения, которые вы считаете полезным указать.
*/
// Комментарий после двух косых черт пишут в одну строку!

// **Включение внешних файлов**

#include <mega16.h> /* перед компиляцией, препроцессор компилятора вставит вместо этой строчки содержимое (текст) заголовочного файла mega16.h — этот файл содержит перечень регистров, имеющихся в МК ATmega16, и соответствие их названий их физическим адресам в МК. Посмотрите его содержание, вызвав CVAVR\inc\mega16.h */

//delay functions
#include <delay.h>
/* перед компиляцией, препроцессор компилятора вставит вместо этой строчки текст «хидера» delay.h — этот файл содержит функции для создания пауз в программе.
Теперь, чтобы сделать паузу, вам нужно лишь написать:
delay_us(N); // сделать паузу N (число) мкс
delay_ms(x); // сделать паузу x мс
x — может быть переменная или число от 0 до 65535 (тип unsigned int), например delay_ms(peremennaya)*/

// **Определения пользователя**
// AD7896 control signals PORTB bit allocation

```
#define MY_CONSTANT 0x15
#define ADC_BUSY PINB.0

/* после этих двух строк, перед компиляцией, препроцессор компилятора заменит в тексте программы MY_CONSTANT на 0x15 и ADC_BUSY на PINB.0. Таким образом, вместо того, чтобы помнить, что вывод какой то микросхемы подключен к ножке PB0, вы можете проверять значение осмысленного понятия ADC_BUSY — «АЦП занят»

    #define — Это удобно, но вовсе не обязательно. */
```

4.5. Объявление переменных

Перед использованием переменной в программе на Си её необходимо объявить, т.е. указать компилятору, какой тип данных она может хранить и как она называется.

Формат объявления переменной таков:
[<storage modifier>] <type definition> <identifier>;

[<storage modifier>] — необязательный элемент,
он нужен только в некоторых случаях и может быть:
extern — если переменная объявляется во внешнем файле, например в хидере delay.h, приведенном выше;

volatile — ставьте, если нужно предотвратить возможность повреждения содержимого переменной в прерывании и не позволить компилятору попытаться выкинуть её при оптимизации кода.

Пример:
volatile unsigned char x;

static — если переменная локальная, т.е. объявлена в какой-либо функции и должна сохранять свое значение до следующего вызова этой функции.

eeprom — разместить переменную в EEPROM. Значение таких переменных сохраняется при выключении питания и при перезагрузке МК.

Пример:

eeprom unsigned int x;

Если это первая переменная в EEPROM, то её младший байт будет помещен в ячейку 1 EEPROM, а старший в ячейку 2. Необходимо помнить, что запись в EEPROM длительный процесс — 8500 тактов процессора.

Глобальные переменные объявляются до появления в тексте программы какой-либо функции. Глобальные переменные доступны в любой функции программы.

Локальные переменные объявляются в самом начале функций, т.е. сразу после фигурной скобки { . Локальные переменные доступны только в той функции, где они объявлены!

<type definition> — тип данных, которые может хранить переменная.

Наиболе часто используемые типы данных:

unsigned char — хранит числа от 0 до 255 (байт);
unsigned int — хранит числа от 0 до 65535 (слово == 2 байта);
unsigned long int — хранит от 0 до 4294967295
(двойное слово == 4 байта).

Вместо **unsigned char** можно писать просто **char**, так как компилятор по умолчанию считает **char** беззнаковым байтом. А если вам нужен знаковый байт, то объявляйте его так:

signed char imya_peremennoi;

<identifier> — имя переменной — некоторый набор символов по вашему желанию, но не образующий зарезервированные слова языка Си. Выше был уже пример идентификатора — имени переменной: imya_peremennoi.

Желательно давать осмысленные имена переменным и функциям, напоминающие вам об их назначении. Принято использовать маленькие буквы, а для отличия имен переменных от названия функций имена переменных можно, например, начинать с буквы, а названия функций (кроме main, конечно) с двух символов подчеркивания.

Например, так: moya_peremennaya , __vasha_funkziya.

Глобальные переменные, а также локальные с модификатором static — при старте и рестарте программы равны 0, если вы не присвоили им (например, оператором =) иное значение при их объявлении или по ходу программы.

Вот несколько примеров объявления переменных:

unsigned char my_peremen = 34;
unsigned int big_peremen = 34034;

Пример массива, содержащего три числа или элемента массива.

char mas[3]={11,22,33};

Нумерация элементов начинается с 0, т.е. элементы данного массива называются mas[0], mas[1], mas[2] и в них хранятся деся-тичные числа 11, 22 и 33.

Где-то в программе вы можете написать: mas[1] = 120;

Теперь в mas[1] будет храниться число 120. Можно не присваивать значений элементам массива при объявлении, но только при объявлении вы можете присвоить значения всем элементам массива сразу. Потом это можно будет сделать только индивидуально для каждого элемента.

Строковая переменная, или массив, содержащий строку символов.

char stroka[6]="Hello"; /* Символов (букв) между кавычками 5 , но указан размер строки 6. Дело в том, что строки символов должны заканчиваться десятичным числом 0. Не путайте его с символом '0', которому соответствует десятичное число 48 по таблице ASCII, которая устанавливает каждому числу определенный символ */

Например:

Элемент строки stroka[1] содержит число 101, которому по таблице ASCII соответствует символ 'e'.

Элемент stroka[4] содержит число 111, которому соответствует символ 'o'.

Элемент stroka[5] содержит число 0, которому соответствует символ 'NUL', его еще обозначают вот так '\0'.

Строковая переменная может быть «распечатана» или выведена в USART МК вот так: printf("%s\n", stroka);

flash и **const** ставятся перед объявлением констант, неизменяемых данных, хранящихся во flash-памяти программ. Они позволяют использовать не занятую программой память МК. Обычно для хранения строковых данных — различных информационных сообщений либо чисел и массивов чисел.

Примеры:

flash int integer_constant=1234+5;
flash char char_constant=’a’;
flash long long_int_constant1=99L;
flash long long_int_constant2=0x10000000;
flash int integer_array1[ ]={1,2,3};
flash int integer_array2[10]={1,2};
flash char string_constant1[ ]=”This is a string constant”;
const char string_constant2[ ]=”This is also a string constant”.

4.6. Описание функций-обработчиков прерываний

Функция «обработчик прерывания» может быть названа вами произвольно, как и любая функция, кроме main. При каком прерывании ее вызывать, компилятор узнает из строчки ISR[ADC_VECT]. По первому зарезервированному слову — ISR — он узнаёт, что речь идет об обработчике прерывания, а номер вектора прерывания (адрес, куда физически, внутри МК, перескочит программа при возникновении прерывания) будет подставлен вместо ADC_VECT препроцессором компилятора перед компиляцией — этот номер указан в подключенном нами ранее заголовочном файле («хидере») описания «железа». МК — mega16.h — это число, сопоставленное слову ADC_VECT.

```
// Прерывание
ISR [ADC_VECT]
 {
PORTB=(unsigned char) ~(ADCW>>2);
/* отобразить горящими светодиодами, подключенными
```

от + питания МК через резисторы 560 Ом к ножкам порта B, старшие 8 бит результата аналого-цифрового преобразования.

Сделаем паузу 127 мс, чтобы в реальном устройстве можно было увидеть переключение светодиодов */
delay_ms(127);

/* В реальных программах старайтесь не делать пауз в прерываниях! Обработчик прерывания должен быть как можно короче и быстрее */

// начать новое АЦ-преобразование
ADCSRA|=0x40;

} // закрывающая скобка обработчика прерывания

Пример установки флага в регистре ADCSRA:

ADCSRA|=0x40; /* результат поразрядного ИЛИ с маской 01000000 поместить обратно в регистр ADCSRA, т.е. установить бит 6. Обратите внимание на необходимость ставить в конце выражений точку с запятой — не забывайте! */

// **Функции, используемые в программе**

/* их может быть столько, сколько вам нужно. У нас будет одна, кроме main и обработчика прерывания. Это будет функция, в которой описано начальное конфигурирование МК в соответствии с поставленной задачей. Удобно над функцией сделать заголовок, подробно поясняющий назначение функции!*/

(void)__init_mk(void) {
/* В начале любой функции объявляются локальные переменные — если, конечно, они вам нужны */

/* void — означает пусто. Перед названием функции — void — означает, что функция не возвращает никакого значения. А в скобках после названия означает, что при вызове в функцию не передаются никакие значения. */

// инициализация Port_B

```
DDRB=0xFF;  // все ножки сделать выходами
PORTB=0xFF; // вывести на все ножки «1»

    /* настройка АЦП производится записью определенного числа в регистр ADCSRA.
    Нам нужно:
    – включить модуль АЦП;
    – установить допустимую частоту тактирования АЦП при частоте кварца 3.69 МГц. Мы выберем коэффициент деления 64 — это даст частоту такта для процессов в АЦП  57.656 КГц;
    – включить прерывание по завершению АЦ-преобразования.
    По ДШ для этого нужно записать в регистр ADCSRA число 1000 1110  или 0x8E  */
    // ADC initialization w Oscillator=3.69MHz
// ADC Clock frequency: 57.656 kHz
// ADC Interrupts: On
ADCSRA=0x8E;

    /* Теперь выбираем вход АЦП ADC0 (ножка PA0) и внешнее опорное напряжение (это напряжение, код АЦП которого будет 1023) с ножки AREF. Смотрим, что нужно записать для этого в регистр мультиплексора (выбора входа) АЦП ADMUX  */

    // Нужно записать 0 (он там по умолчанию)
ADMUX=0;
    /* Разрешаем глобально все прерывания, разрешенные индивидуально. Вы, наверно, поняли, что индивидуально мы разрешили лишь прерывание по завершении АЦП — вот оно-то и сможет возникать у нас.  */
#asm("sei")
    } // скобка закрывающая для функции __init_mk()
```

Так делаются вставки ассемблерных инструкций:

#asm(«инструкция на ассемблере»). Обратите внимание — точки с запятой нет. На Си можно управлять всеми программно

изменяемыми битами в регистрах МК, но часто используются такие строки:

```
#asm("sei") // Разрешить ГЛОБАЛЬНО все прерывания
#asm("cli") // Запретить ГЛОБАЛЬНО все прерывания
#asm("nop") // Пауза в 1 такт процессора
#asm("wdr") // Сбросить сторожевой таймер
```

```
/*  Главная функция main() — обязательная!
Главная функция — программа начинает выполняться с нее */
    void main(void){
/* В начале любой функции объявляются (если нужны)
ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ */
    __init_mk(); /*Вызываем функцию инициализации, настройки аппаратуры МК. Выполнив ее, программа вернется сюда
и будет выполнять следующую строку */
    // запускаем первое АЦ преобразование
ADCSRA|=0x40;
    // бесконечный цикл в ожидании прерываний
while(1);}
/* Программа будет крутиться на этой строчке, постоянно проверяя, истинно ли условие в скобках после while, а так как там константа 1 — то условие будет истинно всегда!*/
 // функция main закончена
```

Теперь программа будет работать так. По завершении цикла АЦП будет возникать прерывание и программа будет перескакивать в функцию «обработчик прерывания» **adc_isr**().

При этом будут автоматически запрещены все прерывания. В конце **adc_isr**() запускается новое АЦ-преобразование, и при выходе из обработчика прерывания снова разрешаются глобально прерывания, а программа возвращается опять в бесконечный цикл **while**(1).

4.7. Тематические задания

- Приведите математические и логические операции присвоения.
- Какие существуют операторы сдвига.

- Приведите пример команды if(){}else{}.
- Приведите пример команды while(){}.
- Приведите пример команды for(;;){}.
- Приведите пример команды switch(){}.
- Приведите пример команды goto.
- Типовая структура программы на языке Си.
- Описание функций-обработчиков прерываний.

5. ЗАГРУЗКА ПРОГРАММЫ В МИКРОКОНТРОЛЛЕР

После того, как программа написана в любом компиляторе, она компилируется в бинарный файл (или файл формата intel HEX) и прошивается («заливается») в микроконтроллер посредством программатора.

Внешне программатор представляет адаптер, который соединяет компьютер с одной стороны и плату с микроконтроллером с другой стороны. Также потребуется прошивающая программа, которая по специальному протоколу скопирует данные в микроконтроллер.

Под разные семейства контроллеров существуют свои программаторы. Существуют «универсальные», позволяющие прошивать одним большую часть существующих устройств. Для программирования микроконтроллеров фирмы AVR потребуется программатор умеющий программировать именно данные микросхемы. Если потребуется откомпилировать и прошить программу, например в PIC16, TMS320…, то потребуется свой специфический программатор.

Существуют разные способы прошивки микроконтроллера. Полный список представлен в документации на конкретный микроконтроллер. Наиболее часто встречающиеся:

- Внутрисхемное программирование (ISP)
- Прошивка через JTAG
- Параллельное высоковольтное программирование
- С использованием загрузчика

Программатор необходим для выставления частоты работы микроконтроллера, а так же позволяет установить пароль на за-

пись/стирание программы или отключить возможность перепрошивки. Данные настройки используются, если необходимо исключить возможность копирования разработанного устройства другими.

В ATmega с завода включен внутренний RC-генератор на частоте 1 МГц (уточните это по ДШ и его возможные частоты). Если вам нужна другая частота или нужно включить внешний кварцевый или керамический резонатор — вам нужно запрограммировать некоторые фьюзы по таблицам из ДШ. Незапрограммированный фьюз — 1, запрограммированный — 0.

Внутрисхемное программирование (ISP)

Самый популярный способ прошивать современные контроллеры. Внутрисхемным данный метод называется потому, что микроконтроллер в этот момент находится в схеме целевого устройства. Для нужд программатора в этом случае выделяется несколько выводов контроллера (обычно 3..5 в зависимости от контроллера).

К этим выводам ISP подключается прошивающий шнур программатора и происходит копирование («заливка») прошивки (Рис. 3). После чего шлейф отключается, и контроллер начинает работу. Посмотреть состояние выполнения программы, регистров, флагов в большинстве устройств не удастся.

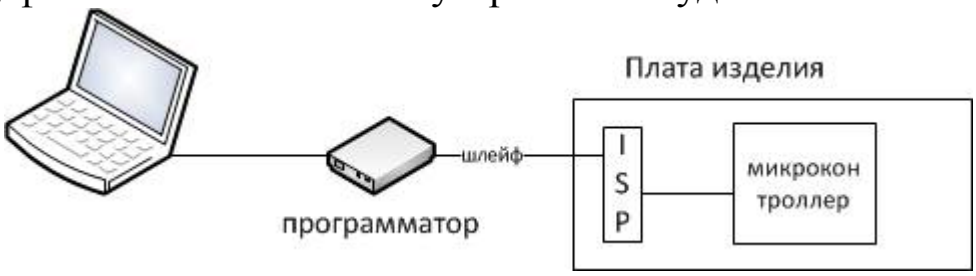


Рис. 3 – Внутрисхемный ISP метод программирования

Фирменный простейший программатор AVRISP mkII фирмы AVR, позволяет прошивать через USB выход компьютера и стоит около 10$ (Рис. 4).

Рис. 4 — Программатор AVRISP mkII

В AVR устройствах прошивка «заливается» по интерфейсу SPI и для работы программатора нужно четыре линии (отдельные ножки микрокотроллера) и питание (достаточно только земли, чтобы уравнять потенциалы земель программатора и устройства):

- MISO — данные идущие от контроллера (Master-Input/Slave-Output)
- MOSI — данные идущие в контроллер (Master-Output/Slave-Input)
- SCK — тактовые импульсы интерфейса SPI
- RESET — сигналом на RESET программатор вводит контроллер в режим программирования
- GND — земля

Сам же разъем внутрисхемного программирования может выглядеть по разному (Рис. 5, Рис. 6) и обычно представляет собой всего лишь несколько штырьков. Лишь бы на него было удобно надеть разъем.

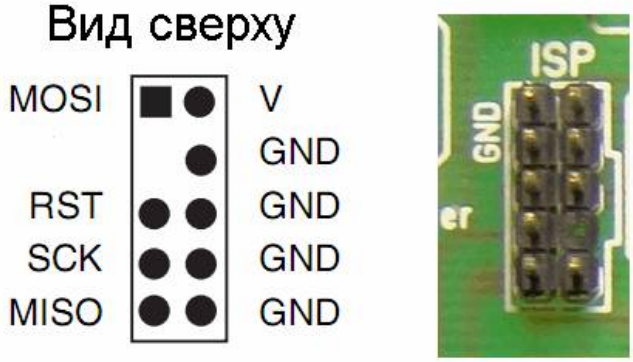


Рис. 5 - Наиболее популярный стандарт из 10 пинов.

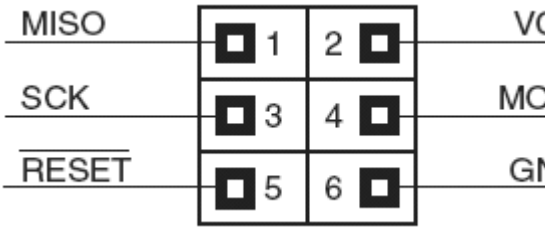


Рис. 6 — Миниатюрный 6 контактный разъем для ISP

Для внутрисхемной прошивки контроллеров AVR существует не один десяток разнообразных программаторов. Отличаются они в первую очередь по скорости работы и типу подключения к компьютеру (COM/LPT/USB). И цене. Самые простые дешевые программаторы могут работать только под Windows XP, возможна зависимость от частоты процессора.

Внутрисхемное программирование, несмотря на все его удобства, имеет ряд ограничений.

Микроконтроллер должен быть запущен, иначе он не сможет ответить на сигнал программатора. Поэтому если неправильно выставить биты конфигурации (FUSE), например, переключить на внешний кварцевый резонатор, а сам кварц не поставить, то контроллер не сможет запуститься и прошить его внутрисхемно будет уже нельзя. По крайней мере до тех пор пока МК не будет запущен. Также в битах конфигурации можно отключить режим внутрисхемной прошивки или перенастроить вывод RESET в обычный порт ввода-вывода (это справедливо для малых МК, у которых RESET совмещен с портом). Такое действо тоже отключат программирование по ISP.

Прошивка через JTAG

JTAG это отладочный интерфейс, работающий по принципу ISP. Он помимо прошивки дополнительно позволяет пошагово выполнять программу прямо в кристалле. Просматривать состояние флагов, регистров и др.. К сожалению JTAG доступен далеко не во всех микроконтроллерах, только в старших моделях в 40ногих микроконтроллерах. Начиная с Atmega16. Дополнительным минусом является бОльшая цена программатора, в сравнении с простым ISP.

Компания AVR выпускает фирменный комплект JTAG ICEII для работы с микроконтроллерами по JTAG (Рис. 7). Также есть первая модель JTAG ICE. Минусом является цена от 20 т.р.

Рис. 7 — Фирменный комплект программатора JTAG ICEII

Параллельное высоковольтное программирование

Обычно применяется на поточном производстве при массовой (сотни штук) прошивке чипов в программаторе перед запайкой их в устройство.

Параллельное программирование во много раз быстрей последовательного (ISP), но требует подачи на RESET напряжения в 12 вольт. А также для параллельной зашивки требуется уже не 3 линии данных, а восемь + линии управления. Для программирования в этом режиме микроконтроллер вставляется в панельку программатора, а после прошивки переставляется в целевое устройство.

Достоинством является возможность изменение любых FUSE бит и, если был отключен режим ISP. Известные параллельные программаторы: HVProg от ElmChan, Paraprog, DerHammer. А также есть универсальные: TurboProg 6, BeeProg, ChipProg++, Fiton…

Прошивка через Bootloader

Многим известно, что большинство устройств бытовой электроники позволяют обновить прошивку, загрузив ее с сайта про-

изводителя – видеокамеры, фотоаппараты, различны плееры, USB 3G модемы, Wi-Fi роутеры и т.п.

Многие микроконтроллеры фирмы AVR имеют возможность самопрошивки. Для включения Загрузчика нужно изначально, любым указанным выше способом, зашить специальную программу — bootloader. Дальше для перешивки программатор не нужен. Достаточно выполнить сброс микроконтроллера и подать ему специальный сигнал. После чего он входит в режим программирования и через обычный последовательный интерфейс в него «заливается» прошивка, например через USB, диск, карту памяти. Возможности зависят от настроенного и загруженное bootloader`a. Таким образом потребуется как минимум два файла с прошивкой готового изделия: загрузчика и основной программы.

5.1. Тематические задания

- Что представляет собой внутрисхемное программирование (ISP)?
- Для чего променяет прошивка через JTAG и что для этого необходимо?
- Особенности параллельного высоковольтного программирования.
- Что такое Bootloader?

6. МИКРОКОНТРОЛЛЕР ATMEGA16. ОСНОВНЫЕ ХАРАКТЕРИСТИКИ, РЕГИСТРЫ

AVR RISC-архитектура - архитектура высокой производительности и малого потребления;

– система команд содержит 130 инструкций, большинство которых выполняется за один машинный цикл;

– единый 16-разрядный формат команд;

– производительность 16 MIPS на частоте 16 Мгц;

– наличие аппаратного умножителя;

– Память, программирование:

o 16 Кбайт Flash ПЗУ программ, с возможностью до 1000 циклов стирания/записи;

o 512 байт ЭСППЗУ (EEPROM) данных, с возможностью до 100000 циклов стирания/записи;

o 1 Кбайт оперативной памяти (SRAM);

o возможность программирования непосредственно в целевой системе через последовательные интерфейсы SPI и JTAG;

o возможность самопрограммирования (bootloader);

– Встроенная периферия:

o два 8-разрядных таймера/счетчика с предварительным делителем частоты и режимом сравнения;

o один 16-разрядный таймер/счетчик с предварительным делителем частоты, режимом сравнения и режимом внешнего события;

o один сторожевой таймер WDT;

o четыре канала генерации выходных ШИМ-сигналов;

o один аналоговый компаратор;

o один 8-канальный 10-разрядный АЦП как с несимметричными, так и с дифференциальными входами;

o один полнодуплексный универсальный синхронный/асинхронный приемопередатчик USART;

o один последовательный синхронный интерфейс SPI, используемый также для программирования Flash-памяти программ;

o один последовательный двухпроводный интерфейс TWI (аналог I2C)

o 32 программируемые линии ввода/вывода с уровнями ТТЛ; на эти линии выведена также поддержка периферийных функций;

– возможность внутрисхемной отладки в соответствии со стандартом IEEE 1149.1 (JTAG);

– различные способы синхронизации: встроенный RC-генератор с внутренней и внешней задающей RC-цепочкой или с внешним резонатором (пьезокерамическим или кварцевым); внешний сигнал синхронизации;

– 6 режимов пониженного энергопотребления (Idle, ADC Noise Reduction, Power-save, Power-down, Standby и Extended Standby);

– детектор снижения напряжения питания (BOD);

– 21 источник прерываний (внутренних и внешних);

– многоуровневая система прерываний, поддержка очереди прерываний;

– напряжения питания 2.7 … 5.5 В.

Полная документация на контроллер представлена в [6].

6.1. Описание выводов микроконтроллера AVR ATmega16(L)

Микроконтроллер Atmega16 (Рис. 8) имеет 4 порта ввода-вывода - port A (PA7 - PA0), port B (PB7 - PB0), port C (PC7 - PC0) и port D (PD7 - PD0). Порты являются двунаправленными портами ввода-вывода с внутренними подтягивающими резисторами, устанавливаемыми для каждого бита в отдельности (Рис. 9).

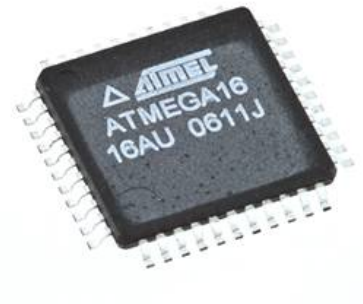


Рис. 8 — Внешний вид микроконтроллера в TQFP корпусе

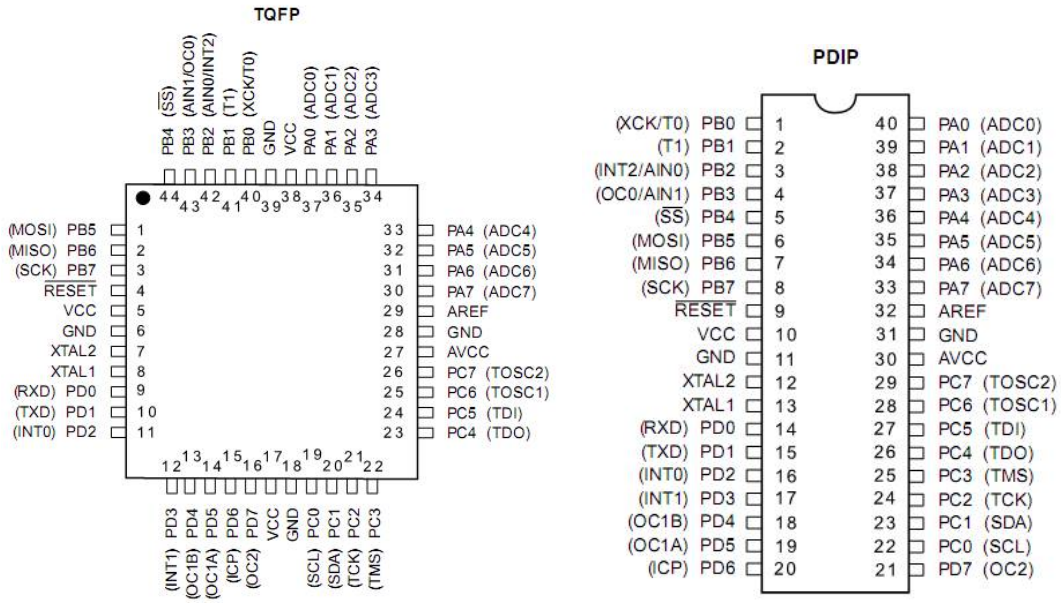


Рис. 9 — Расположение выводов ATmega16:

Порты ввода-вывода также могут выполнять специальные функции для различных устройств ввода-вывода (Таблица 9):

– Port A (PA7 - PA0) может использоваться как восьми разрядный аналоговый вход для АЦП. В этом случае AVCC является контактом питания АЦП. Если АЦП не используется, то этот вывод должен быть соединен с источником питания напрямую, а в случае использования АЦП, подключение к источнику питания осуществляется через НЧ фильтр.

– Port B может использовать

o PB0(T0) и PB1(T1) - два входа таймеров,

o PB2(AIN0) и PB3(AIN1) - два входа аналоговых компараторов,

o PB4(SS), PB5(MOSI), PB6(MISO), PB7(SCK) - выводы для подключения интерфейса SPI.

– Port C может использовать

o PC0(SCL), PC1(SDA) - выводы интерфейса I2C,

o PC5(TDI), PC4(TDO), PC3(TMS), PC2(TCK) - выводы для подключения интерфейса JTAG.

– Port D может использовать

o PD0(RXD), PD1(TXD) - выводы интерфейса UART

o (RS232), PD2(INT0), PD3(INT1) - два входа внешних прерываний.

– Специальные выводы микроконтроллера Atmega16:

- AREF - опорное напряжение АЦП.
- RESET - вывод сброса с активным низким уровнем напряжения,
- XTAL1 - вход для подключения кварцевого резонатора,
- XTAL2 - выход для подключения кварцевого резонатора,
- VCC и GND - подключение источника питания и общего провода.

Таблица 9 — Описание выводов микроконтроллера AVR ATmega16(L)

Обозначение	**Номер вывода**	**Описание**	**DIP**
XTAL1	13	I	Вход тактового генератора
XTAL2	12	O	Выход тактового генератора
$\overline{\text{RESET}}$	9	I	Вход сброса
AREF	32	P	Вход опорного напряжения для АЦП
AGND	31	P	Общий вывод (аналоговый)
AVCC	30	P	Вывод источника питания АЦП
GND	11	P	Общий вывод
VCC	10	P	Вывод источника питания
PA0 (ADC0) – – PA7 (ADC7)	40 – – 33	I/O	A0 – A7 (Вход канала 0–7 АЦП)
PB0 (T0/XCK)	1	I/O	B0 (Вход внешнего тактового сигнала таймера/счетчика T0 / Вход/выход тактового сигнала USART)
PB1 (T1)	2	I/O	B1 (Вход внешнего тактового сигнала таймера/счетчика T1)
PB2 (AIN0/INT2)	3	I/O	B2 (Положительный вход компаратора / Внешнее прерывание)
PB3 (AIN1/OC0)	4	I/O	B3 (Отрицательный вход компаратора / Выход таймера/счетчика T0 (режимы Compare, PWM))
PB4 ($\overline{\text{SS}}$)	5	I/O	B4 (Выбор Slave-устройства на шине SPI)
PB5 (MOSI)	6	I/O	B5 (Выход (Master) или вход (Slave) данных модуля SPI)
PB6 (MISO)	7	I/O	B6 (Вход (Master) или выход (Slave) данных модуля SPI)
PB7 (SCK)	8	I/O	B7 (Выход (Master) или вход (Slave) тактового сигнала модуля SPI)
PC0 (SCL)	22	I/O	C0 (Тактовый сигнал модуля TWI)
PC1 (SDA)	23	I/O	C1 (Линия данных модуля TWI)
PC2 (TCK)	24	I/O	C2 (Тактовый сигнал JTAG)
PC3 (TMS)	25	I/O	C3 (Выбор режима JTAG)
PC4 (TDO)	26	I/O	C4 (Выход данныхJTAG)
PC5 (TDI)	27	I/O	C5 (Вход данныхJTAG)
PC6 (TOSC1)	28	I/O	C6 (Выход для подключения резонатора к таймеру/счетчику T2)
PC7 (TOSC2)	29	I/O	C7 (Вход для подключения резонатора к тайме-

			ру/счетчику T2)
PD0 (RXD)	14	I/O	D0 (Вход USART)
PD1 (TXD)	15	I/O	D1 (Выход USART)
PD2 (INT0)	16	I/O	D2 (Вход внешнего прерывания)
PD3 (INT1)	17	I/O	D3 (Вход внешнего прерывания)
PD4 (OC1B)	18	I/O	D4 (Выход B таймера/счетчика T1 (режимы Compare, PWM))
PD5 (OC1A)	19	I/O	D5 (Выход A таймера/счетчика T1 (режимы Compare, PWM))
PD6 (ICP)	20	I/O	D6 (Вход захвата таймера/счетчика T1 (режим Capture))
PD7 (OC2)	21	I/O	D7 (Выход таймера/счетчика T2 (режимы Compare, PWM))

6.2. Порты ввода-вывода

Основное назначение портов (Таблица 9):

– через порты ввода/вывода микроконтроллер получает сигналы (будь то аналоговые или цифровые) от внешних устройств (для их последующей обработки)

– управляет или обменивается данными с внешними устройствами

– сигнализирует о проделанной работе и т.д.

6.2.1. Структура порта/вывода

Вывод микроконтроллера, это отдельное внутреннее устройство. За каждым выводом обычно закреплено по несколько функций, число которых зависит от того, сколько периферийных устройств имеется в микроконтроллере и подключено к данному выводу. Однако, несмотря на относительно сложную конструкцию, управление выводом (в режиме общего назначения) осуществляется всего лишь через три регистра.

Каждый порт (PortA, PortB, PortC, PortD) состоит из 8 выводов. Название вывода в микроконтроллере формируется из инициала слова Port (“**P**“), после чего идет имя порта (к примеру “**B**“) и порядковый номер вывода в этом порте (к примеру “**0**“). В документации можно встретить и такое обозначение : “Pxn”, где “x” это имя порта, а “n” это номер вывода.

Схема функциональная вывода (Рис. 10) включает в себя диоды **D1** и **D2**, предназначены для защиты вывода от напряже-

ний, превышающих напряжение питания (электростатика (ESD) и т.п.), паразитная емкость вывода **Cpin** , управляемый подтягивающий резистор Rpu на питание и входной ключ.

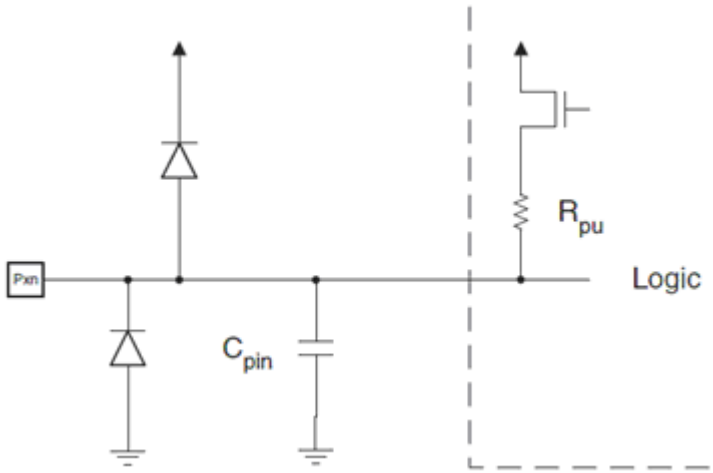


Рис. 10 — Блок-схема вывода (ножки) AVR микроконтроллера из документации

Управление, настройка, контроль и другие всевозможные операции с выводами AVR микроконтроллеров осуществляется всего лишь через три регистра :

- DDRx
- PORTx
- PINx

где “x” это имя порта (A,B,C,D) которому принадлежит вывод. Поскольку в состав одного порта входят восемь выводов, то управлять/настраивать/контролировать и т.д. можно восемью выводами одновременно. Нумерация начинается с нулевого. Каждый n-ый бит этих регистров управляет/настраивает/контролирует и т.д. n-ым выводом данного порта микроконтроллера.

6.2.2. Регистры управления

DDR (Data Direction Register) – Регистр направления (Рис. 11)

Bit	7	6	5	4	3	2	1	0	
	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Рис. 11 —Регистр направления выводов (порт B)

Используется для настройки направления работы каждого порта (A,B,C,D) или отдельного вывода из этого порта.

Регистр DDRB предназначен для настройки выводов порта B на вход или на выход, а отдельно взятый бит этого регистра, соответственно, отвечает за настройку отдельно взятого вывода этого порта. Отдельный бит DDB3, в реггистре DDRB, отвечает за настройку вывода номер 3 порта B (нумерация с 0).

Для настройки направления порта необходимо:

DDRxn = "**1**" – вывод "n" порта "x" настроен на **выход**

DDRxn = "**0**" – вывод "n" порта "x" настроен на **вход**

Примеры записи на языке Си:

```
DDRB = 0b10000001; // в двоичном виде
DDRB = 0x81; // в шестнадцатеричном
DDRB = 129; // в десятичном
DDRB = (1<<7) | (1<<0); // побитовая настройка
DDRB = (1<<DDB7) | (1<<DDB0); // побитовая настройка
```

PORTx – Регистр состояния (Рис. 12)

Bit	7	6	5	4	3	2	1	0	
	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Рис. 12 —Регистр PORTB

Регистр PORTx, управляет состоянием выводов порта "x". В зависимости от выбранного направления работы (вход или выход) порта или отдельно взятого вывода (при помощи регистра DDRx), значение записанное в регистр PORTx того же порта, может присваивать выводу различные состояния.

Таблица 10 — Состояние выводов микроконтроллера AVR ATmega16(L)

DDxn	PORTxn	PUD (in SFIOR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if ext. pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output Low (Sink)
1	1	X	Output	No	Output High (Source)

Таблица 10 представляет возможные состояния вывода, в зависимости от значений записанных в соответствующие биты регистров DDxn и PORTxn. Когда вывод настроен как **выход** (DDxn = "**1**"), то любое значение записанное в соответствующий бит регистра PORTx, поступает на выход:

PORTxn = "**1**" – состояние вывода соответствует **лог. единице** (Output High)

PORTxn = "**0**" – состояние вывода соответствует **лог. нулю** (Output Low)

т.е. состояние вывода (когда он настроен на выход) может изменяться только между лог. единицей и лог. нулем. В этом режиме, резистор Pull-up (Rpu на Рис. 10) отключен и никак не влияет на состояние вывода (No Pull-up).

Когда вывод настроен как **вход** (DDxn = "**0**",), то значение записанное в соответствующий бит регистра PORTx, либо подключает подтягивающий резистор Rpu (pull-up resistor) к выводу, либо отключает его. В этом режиме, текущее состояние выводов, можно прочитать при помощи регистра PINx этого же порта.

PORTxn = "**1**" – подключить **подтягивающий резистор** к выводу

PORTxn = "**0**" – отключить подтягивающий резистор, высокий импеданс (**Hi-Z**)

Для формирования на выходе 0 или 1 нужно написать:

```
DDRB |= 0x01; // первый вывод делаем выходным
PORTB |= (1<<PB0); //выставить 1 на выводе нужной ножки
PORTB &=~ (1<<PB0); // выставить 0 на выводе
```

PINx – Регистр текущего состояния вывода (Рис. 13**)**

Bit	7	6	5	4	3	2	1	0	
	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Рис. 13 — Регистр PINB

Для того чтобы прочитать текущее состояние любого вывода или всего порта, используется регистр PINx, где "x" имя порта. Для чтения текущего состояния вывода (например, с кнопки) нужно написать команды:

```
Unsigned char pin_value; // переменная для хранения состоя-
```

```
ний
pin_value = PINB; // читаем состояние порта B
```

6.3. Таймеры счетчики

Микроконтроллеры ATmega16 оснащены тремя таймерами/счетчиками общего назначения - **двумя** 8-разрядными T/C0 и T/C2 (максимальный счет до 256) и **одним** 16-разрядным T/C1 (максимальное число 65536).

Таймеры/счетчики 0 и 2- модули многофункционального одноканального 8-разрядного таймера/счетчика с аппаратным выходом для генерации ШИМ-сигнала и встроенным асинхронным опциональным тактовым генератором.

16-разрядный таймер/счетчик 1 предназначен для точного задания временных интервалов, генерации прямоугольных импульсов и измерения временных характеристик импульсных сигналов.

Управление работой таймера/счетчика 1 осуществляется с помощью регистров (Таблица 11). Настройка таймеров 0 или 2 происходит в своих регистрах, аналогично рассмотренному для таймера 1.

Таблица 11 — Описание регистров управления таймера/счетчика 1

Регистр	Краткое описание
TCNT1	Timer/Counter1 - регистр, содержащий текущее значение таймера счетчика 1.
TCCR1A	Timer/Counter1 Control Register A - регистр задания режимов таймера/счетчика 1
TCCR1B	Timer/Counter1 Control Register B - регистр задания режимов таймера/счетчика 1
OCR1A	Timer/Counter Output1 Compare Register A - выходной регистр компаратора A
OCR1B	Timer/Counter Output1 Compare Register B - выходной регистр компаратора B
ICR1	Timer/Counter1 Input Capture Register1 - входной регистр защелки 1-го таймера
TIMSK	Timer Interrupt Mask Register - регистр маски прерываний
TIFR	Timer Interrupt Flag Register - регистр флагов прерыва-

	ний

6.3.1. Регистры управления

16-битные регистры OCR1A и OCR1B служат для задания значения, при достижении которого в режиме счёта происходит их сравнение, и таймер/счетчик генерирует соответствующие прерывания по совпадению.

TCNT1H и TCNT1L – Timer/Counter1 (Рис. 14)

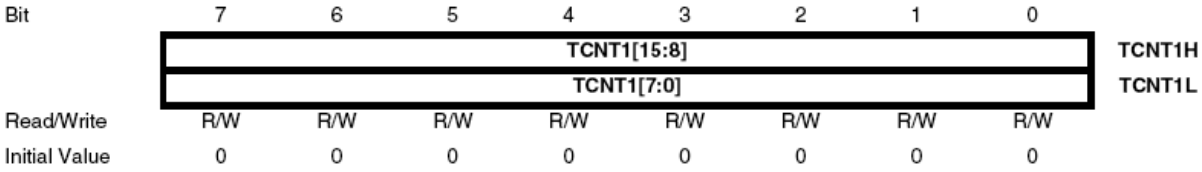


Рис. 14 — Регистр TCNT1H и TCNT1L

16-битный регистры TCNT1 содержит текущее значение таймера счетчика 1. Состоит из старшего TCNT1H (Timer counter 1 High byte) и младшего регистраTCNT1L (Low byte). Для копирования регистра TCNT1 при побайтной работе используется регистр ICR1.

TCCR1A - Регистр задания режимов таймера/счетчика1 (Рис. 15)

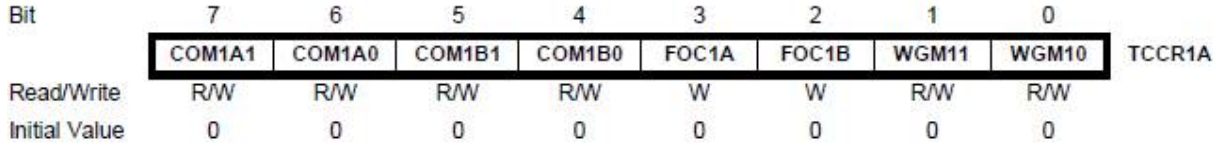


Рис. 15 — Регистр TCCR1A

TCCR1A и TCCR1B - Регистры задания режимов таймера/счетчика 1 определяют режимы работы таймера/счетчика 1 (Таблица 12).

- Bit 7…4 – COM1A1,COM1A0, COM1B1 и COM1B0 изменяют режим формирования выходного сигнала OC1A и OC1B.
- Bit 3…2 – FOC1A, FOC1B принудительной установки результатов сравнения каналов A и B.
- Bit 1…0 – WGM11 и WGM10 в регистре TCCR1A и биты WGM12 и WGM13 в регистре TCCR1B и служат для задания режима работы таймера/счетчика 1.

Таблица 12 — Режимы работы таймера/счетчика1

WGM13	WGM12 (CTC12)	WGM11 (pwm11)	WGM10 (pwm10)	Режим работы	Верхний предел счета	Обновление OCR1x	Установка Флага TOV1 на:
0	0	0	0	Нормальный	0xFFFF	сразу после записи	МАКС
0	0	0	1	8-разр. ШИМ ФК	0x00FF	на вершине счета	нижнем пределе
0	0	1	0	9-разр. ШИМ ФК	0x01FF	на вершине счета	нижнем пределе
0	0	1	1	10-разр. ШИМ ФК	0x03FF	на вершине счета	нижнем пределе
0	1	0	0	СТС	OCRnA	сразу после записи	МАКС
0	1	0	1	8-разр. быстрая ШИМ	0x00FF	на вершине счета	на вершине счета
0	1	1	0	9-разр. быстрая ШИМ	0x01FF	на вершине счета	на вершине счета
0	1	1	1	10-разр. быстрая ШИМ	0x03FF	на вершине счета	на вершине счета
	0	0	0	ШИМ ФЧК	ICRn	на нижнем	нижнем
						Пределе	пределе
	0	0	1	ШИМ ФЧК	OCRnA	на нижнем	нижнем
						Пределе	пределе
	0	1	0	ШИМ ФК	ICRn	на вершине счета	нижнем пределе
	0	1	1	ШИМ ФК	OCRnA	на вершине счета	нижнем пределе
	1	0	0	СТС	ICRn	сразу после записи	МАКС.
	1	0	1	(резерв)	-	-	-
	1	1	0	Быстрая ШИМ	ICRn	на вершине счета	на вершине счета
	1	1	1	Быстрая ШИМ	OCRnA	на вершине счета	на вершине счета

TCCR1B – Timer/Counter1 Control Register B (Рис. 16)

Bit	7	6	5	4	3	2	1	0	
	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Рис. 16 — Регистр TCCR1B

• Bit 7 – ICNC1: Input Capture Noise Canceler. Установка этого бита в лог. 1 активирует входной подавитель шума, при этом будет фильтроваться входной сигнал Input Capture Pin (ICP1). Функция фильтрации требует 4 последовательных одинаковых значений, поступивших на вывод ICP1, чтобы было зарегистрировано изменение уровня сигнала. Таким образом, захват входных импульсов (Input Capture) будет задержан на 4 такта генератора микроконтроллера, когда возможность фильтрации разрешена.

• Bit 6 – ICES1: Input Capture Edge Select. Этот бит выбирает тип среза (фронт или спад) на входе ICP1, который вызовет событие захвата импульса. Когда в ICES1 записан лог. 0, то спад (от-

рицательный срез) вызовет срабатывание триггера, и когда в ICES1 записан лог. 1, срабатывание триггера вызовет уже фронт (положительный срез) сигнала.

Когда срабатывает триггер захвата события по входу в соответствии с установкой ICES1, значение счетчика (TCNT1, регистры TCNT1H и TCNT1L) копируется в регистр захвата Input Capture Register (ICR1). Событие также вызовет установку флага Input Capture Flag (ICF1), и это может использоваться для срабатывания прерывания Input Capture Interrupt, если оно разрешено.

• Bit 2:0 – CS12:10: Clock Select. Эти 3 бита задают источник тактового сигнала для счетчика (Таблица 13).

Таблица 13 — Источники тактового сигнала TI

CS12	CS11	CS10	Описание
0	0	0	Источник тактов не задан (таймер/счетчик остановлен).
0	0	1	$clk_{I/O}$ (без делителя частоты)
0	1	0	$clk_{I/O}$ / 8 (с выхода делителя)
0	1	1	$clk_{I/O}$ / 64 (с выхода делителя)
1	0	0	$clk_{I/O}$ / 256 (с выхода делителя)
1	0	1	$clk_{I/O}$ / 1024 (с выхода делителя)
1	1	0	Внешний тактовый сигнал, поданный на вход T1. Тактирование происходит по срезу (спаду) уровня сигнала.
1	1	1	Внешний тактовый сигнал, поданный на вход T1. Тактирование происходит по фронту (нарастанию) уровня сигнала.

Для подсчета импульсов на входе T1 можно выбрать последние 2 варианта в таблице. Если для подсчета выбрана ножка T1, Импульсы будут подсчитываться даже тогда, когда порт T1 настроен как выход. Эта возможность позволяет программно управлять счетом.

ICR1H и ICR1L – Input Capture Register 1 (Рис. 17)

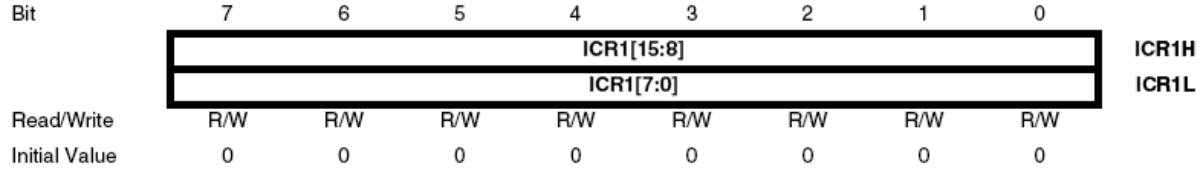


Рис. 17 — Регистр ICR1H и ICR1L

Два регистра составляют 16-битный регистр ICR1. Событие захвата по входу (Input Capture, на выводе ICP1 или опционально на выходе аналогового компаратора) вызывает обновление содержимого ICR1 содержимым счетчика (TCNT1) . Когда ICR1 используется как значение TOP (см. описание битов WGM13:0, размещенных в регистрах TCCR1A и TCCR1B), вывод ICP1 бу-

дет отключен, и следовательно функция захвата Input Capture будет запрещена.

TIMSK – Timer/Counter Interrupt Mask Register (Рис. 18)

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Рис. 18 — Регистр TIMSK

• Bit 5 – TICIE1: Timer/Counter1, Input Capture Interrupt Enable. Когда этот бит установлен в 1, и установлен флаг I в регистре статуса SREG (прерывания разрешены глобально), то разрешено прерывание захвата таймера/счетчика 1 (Timer/Counter1 Input Capture Interrupt). При срабатывании прерывания (когда произошло событие захвата и установился флаг ICF1 в регистре TIFR) будет вызвана подпрограмма обработчика по соответствующему вектору.

• Bit 2 – TOIE1: Timer/Counter1, Overflow Interrupt Enable. Когда этот бит установлен в 1, и установлен флаг I в регистре статуса SREG (прерывания разрешены глобально), то разрешено прерывание переполнения таймера/счетчика 1 (Timer/Counter1 Overflow Interrupt). При срабатывании прерывания (когда произошло переполнение и установился флаг TOV1 в регистре TIFR) будет вызвана подпрограмма обработчика по соответствующему вектору.

TIFR – Timer/Counter Interrupt Flag Register (Рис. 19)

Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Рис. 19 — Регистр TIFR

Регистр флагов прерывания таймеров/счетчиков TIFR устанавливает биты прерывания всех трех (0,1,2) таймеров микроконтроллера и имеет следующие биты:

- Bit 0,1 – отвечают за таймер/счетчик 0 и эквивалентен TOV1, OCF1A, OCF1B
- Bit 2 - TOV1 -При переполнении регистра TCNT1 таймер/счетчик 1 устанавливает флаг TOV1 (Overflow) - переполне-

ние таймера/счетчика 1. Установка этого флага зависит от установки битов WGM13:10. В режимах нормального счета и при очистке таймера на сравнении (Clear Timer on Compare, CTC) флаг TOV1 будет установлен, когда таймер переполнится. TOV1 автоматически очищается, когда вызывается обработчик прерывания по вектору Input Capture Interrupt. Альтернативно TOV1 может быть очищен записью в этот бит лог. 1.

- Bit 3 - OCF1B -При совпадении значений в регистрах TCNT1 и OCR1B формируются флаг OCF1B (Output Compare B Match).
- Bit 4 - OCF1A - При совпадении значений в регистрах TCNT1 и OCR1A формируются флаг OCF1A (Output Compare A Match).
- Bit 5 - ICF1 - При захвате входного значения формируется флаг ICF1 (Input Capture).
- Bit 6,7 - отвечают за таймер/счетчик 2 и эквивалентен TOV1, OCF1A, OCF1B

Вызов соответствующих прерываний зависит от значения соответствующих битов регистра TIMSK и флага I в Status регистре процессора.

6.3.2. Режимы работы

Под режимом работы 16-разр. таймера понимается его алгоритм счета и поведение связанного с ним выхода формирователя импульсов, что определяется комбинацией бит, задающих режим работы таймера (WGMn3-0) и режим формирования выходного сигнала (COMnx1:0).

Нормальный режим работы (простого счета)

Самым простым режимом работы является нормальный режим (WGMn3-0 = 0b0000). В данном режиме счетчик работает как суммирующий (инкрементирующий), при этом сброс счетчика не выполняется. Переполнение счетчика происходит при переходе через максимальное 16-разр. значение (0xFFFF) к нижнему пределу счета (0x0000). В нормальном режиме работы флаг переполнения таймера-счетчика TOVn будет установлен на том же такте синхронизации, когда TCNTn примет нулевое значение.

Режим сброса таймера при совпадении (CTC)

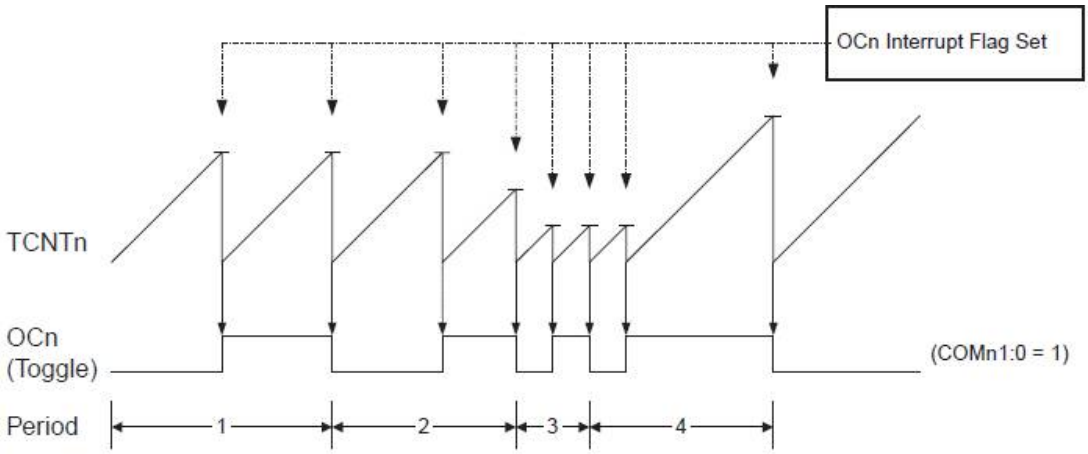


Рис. 20 — Временная диаграмма для режима CTC

В режиме CTC (WGM01, WGM00 =0b10) регистр OCR0 используется для задания разрешающей способности счетчика (Рис. 20). Если задан режим CTC и значение счетчика (TCNT0) совпадает со значением регистра OCR0, то счетчик обнуляется (TCNT0=0). Таким образом, OCR0 задает вершину счета счетчика, а, следовательно, и его разрешающую способность. В данном режиме обеспечивается более широкий диапазон регулировки частоты генерируемых прямоугольных импульсов. Он также упрощает работу счетчика внешних событий. Счетчик (TCNTn) инкрементирует свое состояние до тех пор, пока не возникнет совпадение со значением OCRnA или ICRn, а затем счетчик (TCNTn) сбрасывается.

По достижении верхнего предела счета может генерироваться прерывание с помощью флагов OCFnA или ICFn, соответствующим используемым регистрам для задания верхнего предела счета.

Для генерации сигнала на ножке контроллера в режиме CTC выход OCnA может использоваться для изменения логического уровня при каждом совпадении, для чего необходимо задать режим переключения (COMnA1, COMnA0 = 0b01). Значение OCnA будет присутствовать на выводе порта, только если для данного вывода задано выходное направление. Максимальная частота генерируемого сигнала равна fOC0 = fclk_I/O/2, если OCRnA = 0x0000. Для других значений OCRn частоту генерируемого сигнала можно определить по формуле:

$$f_{OCn} = \frac{f_{clk\_I/O}}{2 \cdot N \cdot (1 + OCRn)}$$

где переменная N задает коэффициент деления предделителя (1, 8, 32, 64, 128, 256 или 1024).

Режим быстрой ШИМ

Режим быстрой широтно-импульсной модуляции (ШИМ) (WGMn3-0 = 0b0101, 0b0110, 0b0111, 0b1110, 0b1111) предназначен для генерации ШИМ-импульсов повышенной частоты. В отличие от других режимов работы в этом используется однонаправленная работа счетчика. Счет выполняется в направлении от нижнего к верхнему пределу счета (пилообразный счет – «пила»).

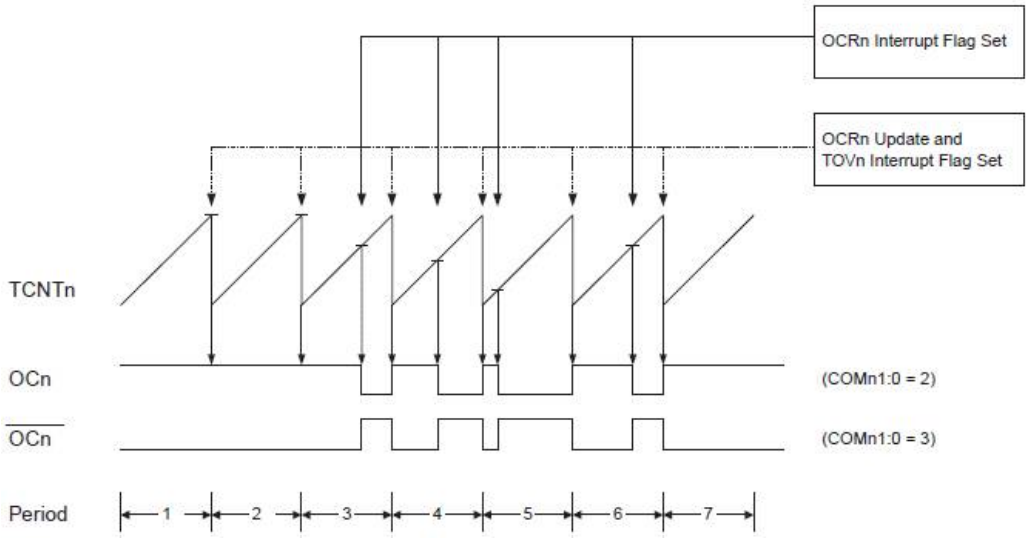


Рис. 21 — Временная диаграмма для режима быстрой ШИМ

На рисунке (Рис. 21) показан режим быстрой ШИМ, когда для задания верхнего предела используется регистр OCRnA или ICRn. Значение TCNTn на временной диаграмме показано в виде графика функции для иллюстрации однонаправленности счета. На диаграмме показаны как инвертированный, так и неинвертированный ШИМ-выходы, формируемые на ножке микроконтроллера. Короткой горизонтальной линией показаны точки на графике TCNTn, где совпадают значения OCRnx и TCNTnx. Флаг прерывания OCnx устанавливается при возникновении совпадении. Флаг переполнения таймера-счетчика (TOVn) устанавливается всякий раз, когда счетчик достигает верхнего предела. Дополнительно тем же тактовым импульсом вместе с флагом TOVn

могут установиться флаги OCnA или ICFn, если для задания верхнего предела используется регистр OCRnA или ICRn, соответственно.

Если задан неинвертирующий режим выхода, то при совпадении TCNTn и OCRnx сигнал OCnx устанавливается, а на верхнем пределе счета сбрасывается. Если задан инвертирующий режим, то выход OCnx сбрасывается при совпадении и устанавливается на верхнем пределе счета. Фактическое значение OCnx можно наблюдать на выводе порта, если для него задано выходное направление (DDR_OCnx).

Разрешающая способность ШИМ может быть фиксированной 8, 9 или 10 разрядов или задаваться регистром ICRn или OCRnA, но не менее 2 разрядов (ICRn или OCRnA = 0x0003) и не более 16 разрядов (ICRn или OCRnA = 0xFFFF). Разрешающая способность ШИМ при заданном значении верхнего предела (ВП) вычисляется следующим образом:

$$f_{OCnPWM} = \frac{f_{clk\_I/O}}{N \cdot 256}$$

где переменная N задает коэффициент деления предделителя (1, 8, 32, 64, 128, 256 или 1024).

Если требуется генерация меандра (прямоугольные импульсы со скважностью 2 или заполнением 50%) высокой частоты, то необходимо использовать режим быстрой ШИМ с установкой бит COMnA1:0 = 0b01, которая вызывает переключение (инвертирование) логического уровня на выходе OCnA при каждом совпадении. Данное применимо, только если OCRnA используется для задания верхнего предела (WGMn3-0 =0b1111).

Режим широтно-импульсной модуляции с фазовой коррекцией

Режим широтно-импульсной модуляции с фазовой коррекцией (ШИМ ФК) (WGMn3-0 = 0b0001, 0b010, 0b0011, 0b1010 или 0b1011) предназначен для генерации ШИМ сигнала с фазовой коррекцией и высокой разрешающей способностью (Рис. 22).

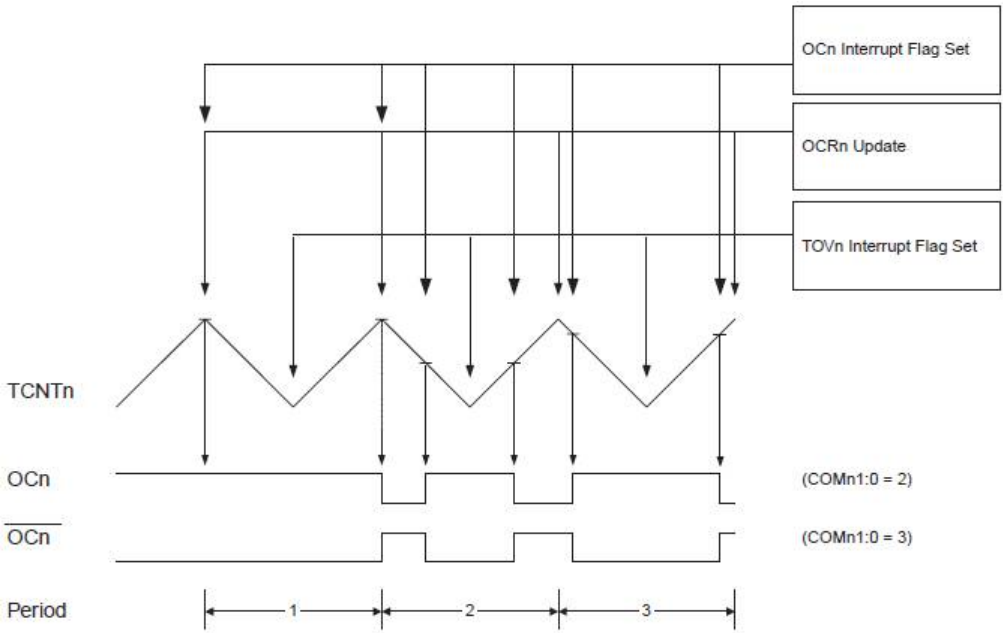


Рис. 22 — Временная диаграмма для режима ШИМ ФК

Режим ШИМ ФК основан на двунаправленной работе таймера-счетчика. Счетчик циклически выполняет счет в направлении от нижнего предела (0x0000) до верхнего предела, а затем обратно от верхнего предела к нижнему пределу, формируя треугольный счет.

На рисунке (Рис. 22) показан режим ШИМ ФК с использованием регистра OCRnA или ICRn для задания верхнего предела. Состояние TCNTn представлено в виде графика функции для иллюстрации двунаправленности счета. На рисунке представлены, как неинвертированный, так и инвертированный ШИМ-выход. Короткие горизонтальные линии указывают точки на графике изменения TCNTn, где возникает совпадение со значением OCRnx. Флаг прерывания OCnx устанавливается при возникновении совпадения.

Если задан неинвертирующий режим выхода формирователя импульсов, то выход OCnx сбрасывается/устанавливается при совпадении значений TCNTn и OCRnx во время прямого/обратного счета. Если задан инвертирующий режим выхода, то, наоборот, во время прямого счета происходит установка, а во время обратного – сброс выхода OCnx. Фактическое значение OCnx можно наблюдать на выводе порта, если в регистре направления данных для данного вывода порта задано выходное направление (DDR_OCnx).

Разрешающая способность ШИМ в данном режиме может быть либо фиксированной (8, 9 или 10 разрядов) либо задаваться с помощью регистра ICRn или OCRnA. Минимальная разрешающая способность равна 2-м разрядам (ICRn или OCRnA = 0x0003), а максимальная -16-ти разрядам (ICRn или OCRnA =0xFFFF). Если задан верхний предел, то разрешающая способность ШИМ в данном режиме определяется следующим образом:

$$f_{OCnPCPWM} = \frac{f_{clk\_I/O}}{N \cdot 510}$$

Флаг переполнения таймера-счетчика (TOVn) устанавливается всякий раз, когда счетчик достигает нижнего предела. Если для задания верхнего предела используется регистр OCRnA или ICRn, то, соответственно устанавливается флаг OCnA или ICFn тем же тактовым импульсом, на котором произошло обновление регистра OCRnx из буферного регистра (на вершине счета).

Режим широтно-импульсной модуляции с фазовой и частотной коррекцией

Основное отличие между режимами ШИМ ФК и ШИМ ФЧК состоит в моменте обновления регистра OCRnx из буферного регистра OCRnx.

Режим широтно-импульсной модуляции с фазовой и частотной коррекцией (ШИМ ФЧК) (WGMn3-0 = 0b1000 или 0b1001) предназначен для генерации ШИМ-импульсов высокой разрешающей способности с фазовой и частотной коррекцией. Также как и режим ШИМ ФК режим ШИМ ФЧК основан на двунаправленной работе счетчика. Счетчик циклически считает от нижнего предела (0x0000) до верхнего предела, а затем обратно от верхнего предела к нижнему пределу. Если задан неинвертирующий режим ШИМ, то выход OCnx сбрасывается, если возникает совпадение между TCNTn и OCRnx во время прямого счета, и устанавливается, если возникает совпадение во время обратного счета. В инвертирующем режиме работа инверсная.

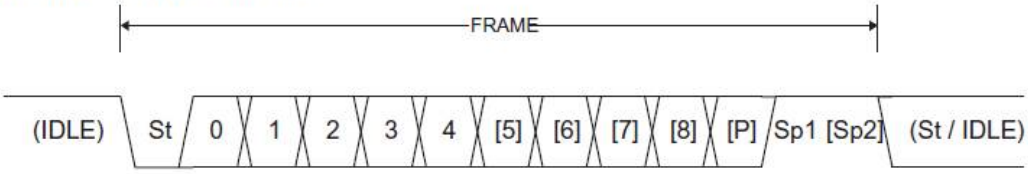
6.4. Модуль UART

В микроконтроллере для работы с модулем USART используются 6 регистров :

- управляющий регистр UCSRA ,
- управляющий регистр UCSRB ,
- управляющий регистр UCSRC,
- регистры скорости передачи UBRRL и UBRRH,
- регистр данных UDR.

Рассмотрим формат кадра, представленный в документации на контроллер (Рис. 23).

Figure 19-4. Frame Formats



St	Start bit, always low. **Старт бит, всегда 0**
(n)	Data bits (0 to 8). **Разряды данных (0...8)**
P	Parity bit. Can be odd or even. **Бит четности**
Sp	Stop bit, always high. **Стоп бит, всегда 1**
IDLE	No transfers on the communication line (RxD or TxD). An IDLE line must be high. **Нет обмена по RxD или TxD (должна быть 1)**

Рис. 23 — Формат кадра UART модуля

Под кадром в данном случае понимается совокупность одного слова данных и сопутствующей информации. Кадр начинается со старт-бита, за которым следует младший разряд слова данных. После старшего разряда слова данных следует один или два стоп-бита. Если включена схема формирования бита четности, он включается между старшим разрядом слова данных и первым стоп-битом. Формат кадра определяется разрядом UCSZ2 регистра UCSRB и разрядами UCSZ1, UCSZ0 регистра UCSRC. Выбор количества стоп-битов (в USART) осуществляется с помощью разряда USBS регистра UCSRC. Разряды UPM1:UPM0 регистра UCSRC определяют функционирование схемы контроля четности модулей USART .

Управляющий регистр UCSRA (Рис. 24)

Bit	7	6	5	4	3	2	1	0
	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM
Read/Write	R	R/W	R	R	R	R	R/W	R/W
Initial Value	0	0	1	0	0	0	0	0

Рис. 24 — Регистр UCSRA

- Bit 7 - RXC - Флаг завершения приема. Флаг устанавливается в «1» при наличии непрочитанных данных в буфере приемника (регистр данных UDR). Сбрасывается флаг аппаратно после опустошения буфера.
- Bit 6- TXC - Флаг завершения передачи. Флаг устанавливается в «1» после передачи всех разрядов посылки из сдвигового регистра передатчика, при условии, что в регистр данных UDR не было загружено нового значения. Флаг сбрасывается аппаратно при выполнении подпрограммы обработки прерывания или программно, записью в него лог. 1.
- Bit 5 - UDRE - Флаг опустошения регистра данных. Данный флаг устанавливается в «1» при пустом буфере передатчика (после пересылки байта из регистра данных UDR в сдвиговый регистр передатчика). Установленный флаг означает, что в регистр данных можно загружать новое значение. Если разряд UDRIE регистра UCR (UCSRB) установлен, генерируется запрос на прерывание «регистр данных пуст». Флаг сбрасывается аппаратно, при записи в регистр данных
- Bit 4 - FE - Флаг ошибки кадрирования. Флаг устанавливается в «1» при обнаружении ошибки кадрирования, т. е. если первый стопбит принятой посылки равен «0». Флаг сбрасывается при приеме стоп бита, равного «1».
- Bit 3 - DOR - Флаг переполнения. В USART флаг устанавливается в «1», если в момент обнаружения нового стартбита в сдвиговом регистре приемника находится последнее принятое слово, а буфер приемника полон (два значения). В UART флаг устанавливается в «1», если новый кадр будет помещен в сдвиговый регистр приемника до того, как из регистра данных будет считано предыдущее слово. Флаг сбрасывается при пересылке принятых данных из сдвигового регистра приемника в буфер.
- Bit 2 - PE - Флаг ошибки контроля четности. Флаг устанавливается в «1», если в данных, находящихся в буфере приемника, выявлена ошибка контроля четности. При отключенном контроле четности этот разряд постоянно читается как «0»
- Bit 1 - U2X - Удвоение скорости обмена. Если этот разряд установлен в «1», коэффициент деления предделителя контрол-

лера скорости передачи уменьшается с 16 до 8, удваивая тем самым скорость асинхронного обмена по последовательному каналу. В USART разряд U2X используется только при асинхронном режиме работы. В синхронном режиме он должен быть сброшен.

- Bit 0 - MPCM - Режим мультипроцессорного обмена. Разряд MPCM используется в режиме мультипроцессорного обмена. Если он установлен в «1», ведомый микроконтроллер ожидает приема кадра, содержащего адрес. Кадры, не содержащие адреса устройства, игнорируются.

Управляющий регистр UCSRB (Рис. 25)

Bit	7	6	5	4	3	2	1	0	
	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8	UCSRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Рис. 25 — Регистр UCSRB

- Bit 7 - RXCIE - Разрешение прерывания по завершению приема. Если данный разряд установлен в «1», то при установке флага RXC регистра UCSRA генерируется прерывание.
- Bit 6 - TXCIE - Разрешение прерывания по завершению передачи. Если данный разряд установлен в «1», то при установке флага TXC регистра UCSRA генерируется прерывание.
- Bit 5 - UDRIE - Разрешение прерывания при очистке регистра данных UART. Если данный разряд установлен в «1», то при установке флага UDRE в регистра UCSRA генерируется прерывание.
- Bit 4 - RXEN - Разрешение приема. При установке этого разряда в «1» разрешается работа приемника USART/UART и переопределяется функционирование вывода RXD.
- Bit 3 - TXEN - Разрешение передачи. При установке этого разряда в «1» разрешается работа передатчика UART и переопределяется функционирование вывода TXD. 4
- Bit 2 - UCSZ2 - Формат посылок. Этот разряд используется для задания размера слов данных, передаваемых по последовательному каналу. В модулях USART он используется совместно с разрядами UCSZ1 и UCSZ0 регистра UCSRC.

- Bit 1 - RXB8 - 8 й разряд принимаемых данных. При использовании 9 разрядных слов данных этот разряд содержит значение старшего разряда принятого слова. В случае USART содержимое этого разряда должно быть считано до прочтения регистра данных UDR.
- Bit 0 - TXB8 - 8 й разряд передаваемых данных. При использовании 9 разрядных слов данных, содержимое этого разряда является старшим разрядом передаваемого слова. Требуемое значение должно быть занесено в этот разряд до загрузки байта данных в регистр UDR.

Управляющий регистр UCSRC (Рис. 26)

Bit	7	6	5	4	3	2	1	0	
	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL	UCSRC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	1	0	0	0	0	1	1	0	

Рис. 26 — Регистр UCSRC

- Bit 7 - URSEL - Выбор регистра. Этот разряд определяет, в какой из регистров модуля производится запись. Если разряд установлен в «1», обращение производится к регистру UCSRC. Если же разряд сброшен в «0», обращение производится к регистру UBRRH.
- Bit 6 - UMSEL - Режим работы USART. Если разряд сброшен в «0», модуль USART работает в асинхронном режиме. Если разряд установлен в «1», то модуль USART работает в синхронном режиме.
- Bit 5,4 - UPM1 и UPM0 - Режим работы схемы контроля и формирования четности. Эти разряды определяют функционирование схем контроля и формирования четности.
- Bit 3 - USBS - Количество стопбитов. Этот разряд определяет количество стоп битов, посылаемых передатчиком. Если разряд сброшен в «0», передатчик посылает 1 стоп бит, если установлен в «1», то 2 стоп бита. Для приемника содержимое этого разряда безразлично.
- Bit 2,1 - UCSZ1 и UCSZ0 - Формат посылок. Совместно с разрядом UCSZ2 эти разряды определяют количество разрядов данных в посылках (размер слова).

- Bit 0 - UCPOL - Полярность тактового сигнала. Значение этого разряда определяет момент выдачи и считывания данных на выводах модуля. Разряд используется только при работе в синхронном режиме. При работе в асинхронном режиме он должен быть сброшен в «0».

Регистры скорости передачи UBRRL и UBRRH (Рис. 27)

Bit	15	14	13	12	11	10	9	8	
	URSEL	–	–	–	UBRR[11:8]				UBRRH
	UBRR[7:0]								UBRRL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Рис. 27 — Регистр UBRRL и UBRRH

- Bit 15 - USEL - Выбор регистра. Этот разряд определяет, в какой из регистров модуля производится запись. Если разряд установлен в «1», обращение производится к регистру UCSRC. Если же разряд сброшен в «0», обращение производится к регистру UBRRH.
- Bit 11-0 UBRR[11..0] - Значение скорости передачи в бодах.

Установка скорости передачи в асинхронном режиме.

При работе в асинхронном режиме скорость обмена определяется не только содержимым регистра UBRR, но и состоянием разряда U2X регистра UCSRA. Если этот разряд установлен в «1», коэффициент деления предделителя уменьшается в два раза, а скорость обмена соответственно удваивается. При работе в синхронном режиме этот разряд должен быть сброшен.

Скорость обмена определяется следующими формулами:

Асинхронный Нормальный режим (U2X=0)	$BAUD = \frac{f_{OSC}}{16(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{16BAU}$
Асинхронный Удвоенная	$BAUD = \frac{f_{OSC}}{8(UBRR+1)}$	$UBRR = \frac{f_{OSC}}{8BAU}$

скорость(U2X=1)

Синхронный режим $BAUD = \frac{f_{OSC}}{2(UBRR+1)}$ | $UBRR = \frac{f_{OSC}}{2BAUD}$

Например, для скорости 9600бит/сек, при встроенном 8МГц осцилляторе в нормальном режиме в регистр UBRR нужно записать число:

UBRR=8000000/(16*9600)-1=51.0833 , округляем = 51

,где BAUD — скорость передачи в бодах,

f_{OSC} — тактовая частота микроконтроллера,

UBRR – число в регистре контроллера скорости передачи (0…4095), которое обеспечит требуемую скорость.

Полученное значение нужно прописать в 8-битные регистры **UBRRL** и **UBRRH -** в первый младшие 8 бит скорости, во второй старшие 2 бита. В примере число 51 размещаем в младшем регистре, в старшем пишется 0.

Для инициализации и/или отправки нужно написать:

```
// Пример инициализации
void USART_Init( unsigned int baud )
{
/* Set baud rate */
UBRRH = (unsigned char)(baud>>8);
UBRRL = (unsigned char)baud;
/* Enable receiver and transmitter */
UCSRB = (1<<RXEN)|(1<<TXEN);
/* Set frame format: 8data, 2stop bit */
UCSRC = (1<<URSEL)|(1<<USBS)|(3<<UCSZ0);
}
```

```
//Пример отправки 5-8 бит
void USART_Transmit( unsigned char data )
{
/* Wait for empty transmit buffer */
while ( !( UCSRA & (1<<UDRE)) )
;
```

```
//Прием 5-8 бит
unsigned char USART_Receive( void )
{
/* Wait for data to be received */
while ( !(UCSRA & (1<<RXC)) )
;
/* Get and return received data
```

/* Put data into buffer, sends the data */ UDR = data; }	from buffer */ return UDR; }

//отправка 9 бит
void USART_Transmit(unsigned int data)
{
/* Wait for empty transmit buffer */
while (!(UCSRA & (1<<UDRE))))
;
/* Copy 9th bit to TXB8 */
UCSRB &= ~(1<<TXB8);
if (data & 0x0100)
UCSRB |= (1<<TXB8);
/* Put data into buffer, sends the data */
UDR = data;
}
	//Прием 9 бит
	unsigned int USART_Receive(void)
	{
	unsigned char status, resh, resl;
	/* Wait for data to be received */
	while (!(UCSRA & (1<<RXC)))
	;
	/* Get status and 9th bit, then data */
	/* from buffer */
	status = UCSRA;
	resh = UCSRB;
	resl = UDR;
	/* If error, return -1 */
	if (status & (1<<FE)|(1<<DOR)|(1<<PE))
	return -1;
	/* Filter the 9th bit, then return */
	resh = (resh >> 1) & 0x01;
	return ((resh << 8) | resl);
	}

6.5. Модуль АЦП Аналого-цифровой преобразователь - (Analog to Digital Converter)

Служит для преобразования аналогового сигнала в дискретный код (Рис. 28)

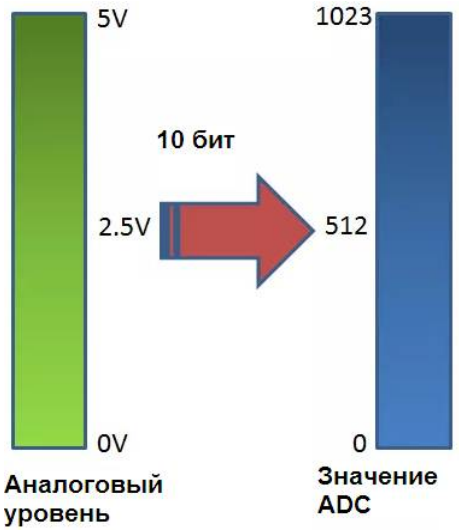


Рис. 28 — Отношение уровня напряжения и цифрового кода

Основные характеристики:

- 10-разрядное разрешение
- Интегральная нелинейность 0.5 мл. разр.
- Абсолютная погрешность ±2 мл. разр.
- Время преобразования 13 - 260 мкс.
- Частота преобразования до 15 тыс. преобр. в сек. при максимальном разрешении
- 8 мультиплексированных однополярных каналов (входов)
- 7 дифференциальных каналов (входов)
- 2 дифференциальных канала (входа) с подключаемым усилением на 10 и 200
- Представление результата с левосторонним или правосторонним выравниванием в 16-разр. слове
- Диапазон входного напряжения ADC 0…VCC
- Выборочный внутренний ИОН (Reference Voltage) на 2.56 В
- Режимы одиночного преобразования и автоматического перезапуска
- Прерывание по завершении преобразования ADC
- Механизм подавления шумов в режиме сна

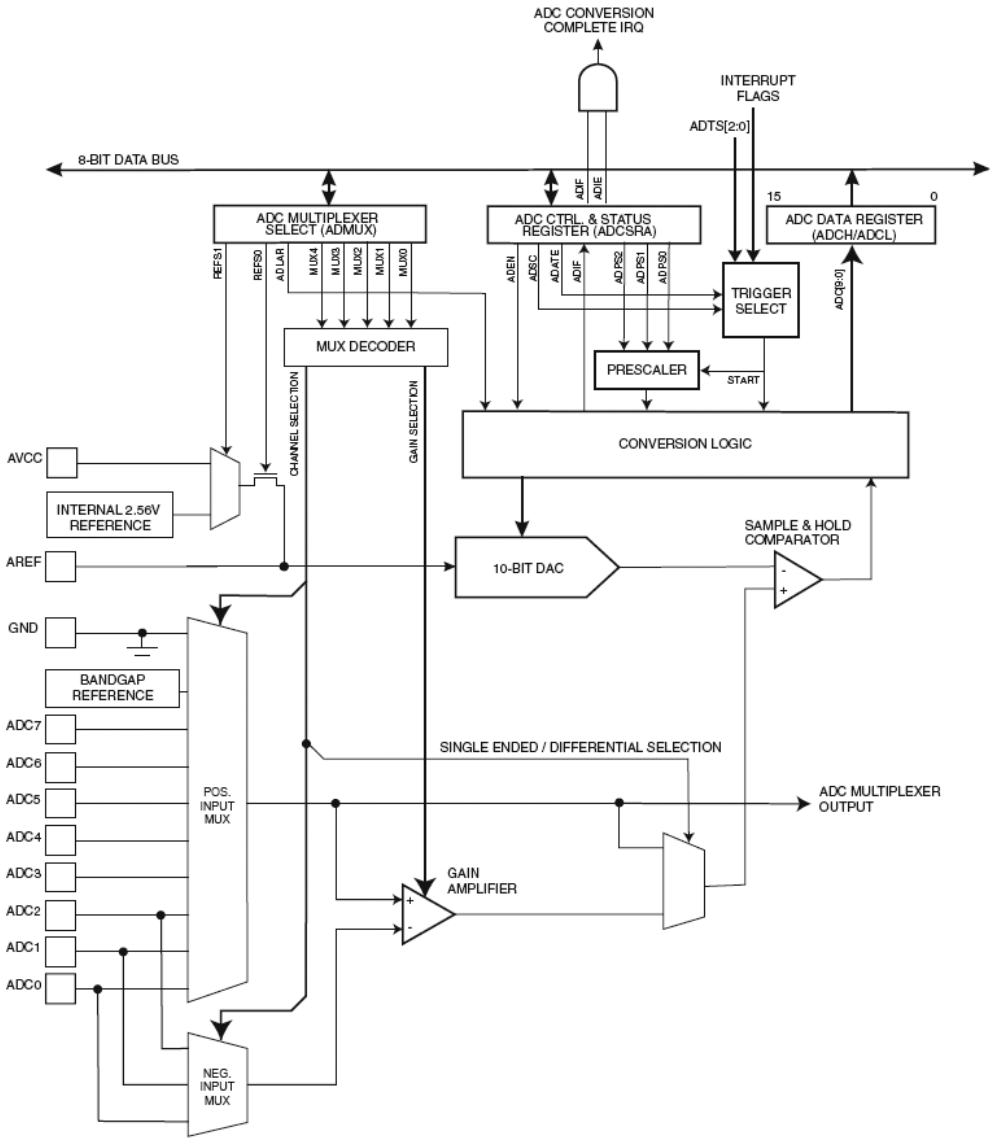


Рис. 29 — Блок-схема аналого-цифрового преобразователя

Микроконтроллеры ATmega оснащены 10-разрядным ADC последовательного приближения. ADC подсоединен к 8-канальному аналоговому мультиплексору, позволяющему использовать любой вывод порта A в качестве входа ADC. ADC содержит усилитель выборки/хранения, удерживающий напряжение входа ADC во время преобразования на неизменном уровне (Рис. 29). Для питания ADC используются два отдельных вывода: AVCC и AGND. Вывод AGND должен быть подсоединен к GND.

Управление происходит четырьмя регистрами:

– ADMUX (ADC Multiplexer Select Register) - Регистр выбора мультиплексора ADC

– ADCSRA (ADC Control and Status Register) - Регистр управления и состояния ADC

– ADC (ADCL и ADCH) - Регистры данных ADC

– SFIOR - Регистр специальных функций ввода-вывода

ADMUX (ADC Multiplexer Select Register) - Регистр выбора мультиплексора ADC (Рис. 30)

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Рис. 30 — Регистр ADMUX

- Bit 7:6 – REFS1:0: Биты выбора источника опорного напряжения. Данные биты определяют, какое напряжение будет использоваться в качестве опорного для ADC (Таблица 14). Внутренний ИОН можно не использовать, если к выводу AREF подключен внешний опорный источник.

Таблица 14 — Выбор опорного источника ADC

REFS1	REFS0	Источник опорного напряжения ADC
0	0	Внешний источник, подключенный к AREF, внутренний VREF отключен
0	1	AVCC с внешним конденсатором на выводе AREF
1	0	Зарезервировано
1	1	Внутренний источник опорного напряжения 2.56В с внешним конденсатором на выводе AREF

- Bit 5 – ADLAR: Бит управления представлением результата преобразования. Бит ADLAR влияет на представление результата преобразования в паре регистров результата преобразования ADC. Если ADLAR = 1, то результат преобразования будет иметь левосторонний формат, в противном случае - правосторонний.
- Bit 4:0 – MUX4:0: Биты выбора аналогового канала и коэффициента усиления. Данные биты определяют какие из имеющихся аналоговых входов подключаются к ADC. Кроме того, с

их помощью можно выбрать коэффициент усиления для дифференциальных каналов (Таблица 15).

Таблица 15 — Выбор входного канала и коэффициента усиления

MUX4..0	Недифференциальный вход	Неинвертирующий дифференциальный вход	Инвертирующий дифференциальный вход	Коэф. Усил.,GAIN (Ку)
00000	ADC0	Дифференциальных входов и усиления нет		
00001	ADC1			
00010	ADC2			
00011	ADC3			
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC7			
01000	Недифференциальных входов нет	ADC0	ADC0	10x
01001		ADC1	ADC0	10x
01010(1)		ADC0	ADC0	200x
01011(1)		ADC1	ADC0	200x
01100		ADC2	ADC2	10x
01101		ADC3	ADC2	10x
01110(1)		ADC2	ADC2	200x
01111(1)		ADC3	ADC2	200x
10000		ADC0	ADC1	1x
10001		ADC1	ADC1	1x
10010		ADC2	ADC1	1x
10011		ADC3	ADC1	1x
10100		ADC4	ADC1	1x
10101		ADC5	ADC1	1x
10110		ADC6	ADC1	1x
10111		ADC7	ADC1	1x
11000		ADC0	ADC2	1x
11001		ADC1	ADC2	1x
11010		ADC2	ADC2	1x
11011		ADC3	ADC2	1x
11100		ADC4	ADC2	1x
11101		ADC5	ADC2	1x
11110	1.22 В (VBG)	Дифференциальных входов и усиления нет		
11111	0 В (GND)			

ADCSRA (ADC Control and Status Register) - Регистр управления и состояния ADC (Рис. 31)

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Рис. 31 — Регистр ADCSRA

- Bit 7 – ADEN: Разрешение работы ADC. Запись в данный бит лог. 1 разрешает работу ADC. Если в данный бит записать лог. 0, то ADC отключается, даже если он находился в процессе преобразования.
- Bit 6 – ADSC: Запуск преобразования ADC. В режиме одиночного преобразования установка данного бита инициирует старт каждого преобразования. В режиме автоматического перезапуска установкой этого бита инициируется только первое преобразование, а все остальные выполняются автоматически.
- Bit 5 – ADATE: Включение режима автоматического запуска ADC. Если в данный бит записать лог. 1, то ADC перейдет в режим автоматического перезапуска. В этом режиме ADC автоматически запускает преобразование по положительному фронту выбранного запускающего сигнала. Выбор источника запуска происходит битами ADTS регистра SFIOR.
- Bit 4 – ADIF: Флаг прерывания ADC. Данный флаг устанавливается после завершения преобразования ADC и обновления регистров выходных данных (ADCH:ADCL). Если установлены биты ADIE и I (регистр SREG), то происходит вызов обработчика прерывания по завершении преобразования.
- Bit 3 – ADIE: Разрешение прерывания ADC. Когда этот бит установлен в лог. 1, и установлен бит I в регистре SREG, разрешается прерывание по завершении преобразования ADC.
- Bit 2:0 – ADPS2:0: Биты управления предделителем ADC. Данные биты (Таблица 16) определяют на какое значение тактовая частота ЦПУ будет отличаться от частоты входной синхронизации ADC.

Таблица 16 — Управление предделителем (прескалером) ADC

ADPS2	ADPS1	ADPS0	Коэффициент деления
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

ADC (ADCL и ADCH) - Регистры данных ADC (Рис. 32, Рис. 33)

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	ADC9	ADC8	ADCH
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Рис. 32 — Регистр ADCL и ADCH для ADLAR = 0

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Рис. 33 — Регистр ADCL и ADCH для ADLAR = 1

По завершении преобразования результат помещается в этих двух регистрах. При использовании дифференциального режима преобразования результат представляется в коде двоичного дополнения.

Формат представления результата (левостороннее выравнивание или правостороннее выравнивание) зависит от состояния бита ADLAR в регистре ADMUX. Левосторонний формат представления результата удобно использовать, если достаточно 8 разрядов. В этом случае 8-разрядный результат хранится в регистре ADCH и, следовательно, чтение регистра ADCL можно не выполнять. При правостороннем формате необходимо сначала считать ADCL, а затем ADCH.

SFIOR - Регистр специальных функций ввода-вывода (Рис. 34)

Bit	7	6	5	4	3	2	1	0	
	ADTS2	ADTS1	ADTS0	–	ACME	PUD	PSR2	PSR10	SFIOR
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Рис. 34 — Регистр SFIOR

Bit 7:5 – ADTS2:0: Биты выбора источника автоматического запуска ADC. Если ADATE в регистре ADCSRA записан в лог. 1, значения бит ADTS определяют, какой будет использоваться источник для запуска преобразования. Если ADATE в регистре ADCSRA записан в лог. 0, значения бит ADTS никакого значения не имеют (Таблица 17). Преобразование будет запускаться по положительному фронту (нарастание сигнала) выбранного сигнала.

Таблица 17 — Источник запуска преобразования ADC

ADTS2	**ADTS1**	**ADTS0**	**Источник запуска преобразования ADC**
0	0	0	Постоянное преобразование (Free Running mode)
0	0	1	Аналоговый компаратор
0	1	0	Внешний запрос на прерывание 0
0	1	1	Timer/Counter0 Compare Match
1	0	0	Timer/Counter0 Overflow
1	0	1	Timer/Counter1 Compare Match B
1	1	0	Timer/Counter1 Overflow
1	1	1	Timer/Counter1 Capture Event

Bit 4 – Res: Зарезервированный бит

6.6. Прерывания

Прерывание (Interrupt) – сигнал, сообщающий процессору о наступлении какого-либо события. При этом выполнение текущей последовательности команд приостанавливается и управление передаётся процедуре обработки прерывания, соответствующая данному событию, после чего исполнение кода продолжается ровно с того места где он был прерван (возвращение управления). Таким образом, они освобождают процессор от периодического непрерывного опроса флагов.

Процедура обработки прерывания (Interrupt Service Routine) – это ни что иное как функция/подпрограмма, которую следует выполнить при возникновении определенного события.

У каждого периферийного устройства, что входит в состав AVR микроконтроллеров, есть как минимум один источник прерывания (Interrupt source). Ко всем этим прерываниям следует причислить и прерывание сброса – Reset Interrupt, предназначение которого отличается от всех остальных.

За каждым прерыванием, строго закреплен вектор (ссылка) указывающий на процедуру обработки прерывания (Interrupt service routine). Все векторы прерываний заранее прошиты на заводе и располагаются в самом начале памяти программ и вместе формируют “таблицу векторов прерываний” (Interrupt vectors table) (Таблица 18).

Таблица 18 — Таблица векторов прерываний

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	TIMER2 COMP	Timer/Counter2 Compare Match
5	$008	TIMER2 OVF	Timer/Counter2 Overflow
6	$00A	TIMER1 CAPT	Timer/Counter1 Capture Event
7	$00C	TIMER1 COMPA	Timer/Counter1 Compare Match A
8	$00E	TIMER1 COMPB	Timer/Counter1 Compare Match B
9	$010	TIMER1 OVF	Timer/Counter1 Overflow
10	$012	TIMER0 OVF	Timer/Counter0 Overflow
11	$014	SPI, STC	Serial Transfer Complete
12	$016	USART, RXC	USART, Rx Complete
13	$018	USART, UDRE	USART Data Register Empty
14	$01A	USART, TXC	USART, Tx Complete
15	$01C	ADC	ADC Conversion Complete
16	$01E	EE_RDY	EEPROM Ready
17	$020	ANA_COMP	Analog Comparator
18	$022	TWI	Two-wire Serial Interface
19	$024	INT2	External Interrupt Request 2
20	$026	TIMER0 COMP	Timer/Counter0 Compare Match
21	$028	SPM_RDY	Store Program Memory Ready

Каждому прерыванию соответствует определенный “бит активации прерывания” (Interrupt Enable bit) в каждом отдельном регистре для конкретного модуля (UART, АЦП, Таймер…). Таким образом, чтобы использовать определенное прерывание, следует записать в его “бит активации прерывания” – лог. единицу.

Далее, независимо от того активировали или нет определенные прерывания, микроконтроллер не начнет обработку этих прерываний, пока в “бит всеобщего разрешения прерываний” (Global Interrupt Enable bit в регистре состояния SREG) не будет записана лог. единица. Также, чтобы запретить все прерывания (на неопределенное время), в бит всеобщего разрешения прерываний следует записать – лог. нуль.

Флаг глобального разрешения прерываний помогает тогда, когда требуется выполнить ответственный по времени участок кода и вместо запретов каждого по отдельности прерывания (для UART, АЦП, Таймера..) достаточно выставить один глобальный запрет, после выполнения разрешить.

Прерывание Reset (перезагрузки), в отличие от всех остальных, нельзя запретить. При подаче питания процессор сразу переходит на вектор прерывания Reset (сброс), расположенный по адресу 0x0000, а после уже на основную программу main(). Такие прерывания еще называют Non-maskable interrupts.

У каждого прерывания есть строго определенный приоритет. Приоритет прерывания зависит от его расположения в “таблице векторов прерываний”. Чем меньше номер вектора в таблице, тем выше приоритет прерывания. То есть, самый высокий приоритет имеет прерывание сброса (Reset interrupt), которое располагается первой в таблице, а соответственно и в памяти программ. Внешнее прерывание INT0, идущее следом за прерыванием Reset в “таблице векторов прерываний”, имеет приоритет меньше чем у Reset, но выше чем у всех остальных прерываний и т.д.

Пример ассемблерного кода atmega16 векторов прерываний

```
Address Labels Code Comments
$000 jmp RESET ; Reset Handler
$002 jmp EXT_INT0 ; IRQ0 Handler
$004 jmp EXT_INT1 ; IRQ1 Handler
$006 jmp TIM2_COMP ; Timer2 Compare Handler
$008 jmp TIM2_OVF ; Timer2 Overflow Handler
$00A jmp TIM1_CAPT ; Timer1 Capture Handler
$00C jmp TIM1_COMPA ; Timer1 CompareA Handler
$00E jmp TIM1_COMPB ; Timer1 CompareB Handler
```

```
$010 jmp TIM1_OVF ; Timer1 Overflow Handler
$012 jmp TIM0_OVF ; Timer0 Overflow Handler
$014 jmp SPI_STC ; SPI Transfer Complete Handler
$016 jmp USART_RXC ; USART RX Complete Handler
$018 jmp USART_UDRE ; UDR Empty Handler
$01A jmp USART_TXC ; USART TX Complete Handler
$01C jmp ADC ; ADC Conversion Complete Handler
$01E jmp EE_RDY ; EEPROM Ready Handler
$020 jmp ANA_COMP ; Analog Comparator Handler
$022 jmp TWSI ; Two-wire Serial Interface Handler
$024 jmp EXT_INT2 ; IRQ2 Handler
$026 jmp TIM0_COMP ; Timer0 Compare Handler
$028 jmp SPM_RDY ; Store Program Memory Ready Handler;
$02A RESET: ldi r16,high(RAMEND) ; Main program start
$02B out SPH,r16 ; Set Stack Pointer to top of RAM
$02C ldi r16,low(RAMEND)
$02D out SPL,r16
$02E sei ; Enable interrupts
$02F <instr> xxx
```

6.7. Тематические задания

- ✓ Как настраиваются порты общего назначения?
- ✓ Сколько таймеров в AtMega16? Какие существуют режимы работы в таймерах?
- ✓ Как настроить модуль UART на прием и передачу? Как узнать, что данные приняты/отправлены?
- ✓ Где хранится результат оцифровки данных? Как узнать, что оцифровка закончилась?
- ✓ Приведите таблицу векторов прерываний.

ЛИТЕРАТУРА

1. Шарапов А.В. Цифровые и микропроцессорные устройства: Учеб. пособие. — Томск: ТМЦ ДО, 2003. — 166 с.
2. Русанов В.В., ШевелёвМ.Ю. Микропроцессорные устройства и системы: Учебное пособие для вузов. – Томск: Томский государственный университет систем управления и радиоэлектроники, 2007. – 184с.
3. Кривченко И.В. Микроконтроллеры общего назначения для встраиваемых приложений производства Atmel Corp. / И.В. Кривченко // Электронные компоненты. — 2002. — №5. — С. 69–73.
4. Гребнев В.В. Микроконтроллеры семейства AVR фирмы Atmel. — М.: ИП «Радиософт», 2002.
5. Официальный сайт фирмы Atmel [электронный ресурс]. Режим доступа: http://www.atmel.com .
6. Документация на контроллер atmega16 [электронный ресурс]. Режим доступа: http://www.atmel.com/images/2466s.pdf
7. Евстифеев А.В. Микроконтроллеры AVR семейств Tiny и Mega фирмы ATMEL. — М.: Издательский дом «Додэка-XXI», 2005. — 560 с.
8. Микроконтроллеры AVR — самоучитель начинающим с нуля: Краткий курс [электронный ресурс]. Режим доступа: http://www.123avr.com.

ГЛОСАРИЙ

- *Микропроцессорная техника (МПТ)* – это комплекс технических и программных средств для построения различных микропроцессорных устройств и систем.
- *Микропроцессорная система (МПС)* – это функционально законченное изделие, состоящее из одного или нескольких устройств, главным образом микропроцессорных.
- *Микропроцессорное устройство (МПУ)* представляет собой функционально и конструктивно законченное изделие, состоящее из нескольких микросхем, в состав которых входит микропроцессор. Оно предназначено для выполнения определенного набора функций: получение, обработка, передача, преобразование информации и управление.
- *Микропроцессор (МП)* – это программно-управляемое устройство, построенное, как правило, на одной БИС и осуществляющее процесс управления и цифровой обработки информации.
- *Микроконтроллер* – это однокристальная ЭВМ, структурная схема которой содержит все функциональные узлы, необходимые для обеспечения автономной работы в качестве управляющего или вычислительного устройства.
- *Семейство микроконтроллеров* - это микроконтроллеры, имеющие общую архитектуру и структуру и объединенные общей системой команд.
- *Архитектура микроконтроллера* – это комплекс его аппаратных и программных средств, предоставляемых пользователю.
- *Структура микропроцессора* определяет состав и взаимодействие основных устройств и блоков, размещенных на его кристалле.

ЗАКЛЮЧЕНИЕ

Данное учебное пособие ориентировано на самостоятельное знакомство с базовым функционалом микроконтроллеров. Ориентировано для студентов вузов радиоэлектронного профиля и инженеров-проектировщиков средств и систем автоматики и промышленной электроники.

Пособие включает в себя программную модель, систему команд и характеристики периферийных устройств микроконтроллеров AVR фирмы Atmel семейства Mega 16. Приводятся основные регистры микроконтроллера и примеры инициализации периферийных устройств с упором на язык Си.

После прочтения пособия и выполнения курса лабораторных и практических занятий студент приобретает базовый опыт в проектировании, разработки и отладки несложных микропроцессорных устройств. Делается акцент на изучение базового функционала распространенного микроконтроллера MEGA16, который используется в качестве примера часто используемых решений в других микропроцессорах. При нехватке производительности или функционала изученного микроконтроллера студент без больших сложностей может выбрать самый современный кристалл из рассмотренной линейки Mega и перенести свою программу на новое современное ядро.

Документация на изучаемый контроллер ATMega16 составляет около 350 листов и включает описание всех регистов. Документация на последние современные многоядерные микроконтроллеры, имеющие около 8 Ethernet портов, десятка шин UART и I2c, поддержку последних типов памяти DDR, составляет более 10 000 листов на английском языке. Проблемой будет являться изучение таких сложных кристаллов без изучения базовых принципов на простых решениях типа AtMega16.

Дальнейшей рекомендацией студентам к изучению микропроцессорных устройств и систем является выбор контроллеров на ядрах ARM и изучение их функциональных особенностей как наиболее перспективные с оглядкой на будущее.

На момент написания руководства последней версией компилятора AVR является Atmel Studio 6.2, в которой выполняется курс самостоятельных работ.