**ОГЛАВЛЕНИЕ**

[*ВВЕДЕНИЕ* 4](#_Toc405838211)

[Тема № 1. Общее устройство микроконтроллеров AVR 5](#_Toc405838212)

[1. Краткие сведения о МК AVR 5](#_Toc405838213)

[2. Основные семейства МК AVR 5](#_Toc405838214)

[3. Структурная схема МК AVR 6](#_Toc405838215)

[4. Основные периферийные устройства 9](#_Toc405838216)

[**Задания для самостоятельной работы** 11](#_Toc405838217)

[Тема № 2. Тактирование,сброс и подключение мк avr 12](#_Toc405838218)

[1. Способы тактирования МК 12](#_Toc405838219)

[2. Сброс МК 14](#_Toc405838220)

[3. Питание МК 14](#_Toc405838221)

[**Задания для самостоятельной работы** 16](#_Toc405838222)

[Тема № 3. Общие принципы программирования МК 17](#_Toc405838223)

[1. Atmel Studio 17](#_Toc405838224)

[2. Команды инструкции и нотации AVR-ассемблера, прерывания 21](#_Toc405838225)

[3. Прерывания 25](#_Toc405838226)

[4. Структура AVR программ 27](#_Toc405838227)

[5. Отладка и компиляция простейшей AVR программ 28](#_Toc405838228)

[6. Конфигурационные биты (Fuse биты) 30](#_Toc405838229)

[**Задания для самостоятельной работы** 31](#_Toc405838230)

[Тема №4. Система команд AVR 32](#_Toc405838231)

[1. Команды условных переходов и регистр SREG 32](#_Toc405838232)

[2. Команды пересылки данных 35](#_Toc405838233)

[3. Выполнение типовых процедур на ассемблере 39](#_Toc405838234)

[**Задания для самостоятельной работы** 44](#_Toc405838235)

[Тема №5. Порты ввода/вывода 45](#_Toc405838236)

[1. Общие сведения о портах ввода/вывода 45](#_Toc405838237)

[2. Режимы работы портов ввода/вывода 47](#_Toc405838238)

[3. Подключение к портам ввода/вывода внешних устройств 48](#_Toc405838239)

[*3.1 Подключение кнопки и светодиода.* 48](#_Toc405838240)

[*3.2 Подключение семисегментного индикатора.* 54](#_Toc405838241)

[4. Внешние прерывания 58](#_Toc405838242)

[**Задания для самостоятельной работы** 58](#_Toc405838243)

[Тема № 6. Программирование таймероВ 59](#_Toc405838244)

[1. Общие сведения о таймерах и режимы их работы 59](#_Toc405838245)

[2. Особенности программирования таймеров 65](#_Toc405838246)

[3. Простейший секундомер 67](#_Toc405838247)

[4. Таймеры в режиме ШИМ 71](#_Toc405838248)

[**Задания для самостоятельной работы** 73](#_Toc405838249)

[Тема № 7. Передача и прием данных UART 74](#_Toc405838250)

[1. Краткое описание протокола 74](#_Toc405838251)

[2. Реализация UART в МК AVR 75](#_Toc405838252)

[Тема №8. Аналоговый компаратор и АЦП 78](#_Toc405838253)

[Тема № 9. Использование EEPROM 79](#_Toc405838254)

[Тема № 10. Динамическая индикация и опрос матрицы кнопок 80](#_Toc405838255)

[1. Динамическая индикация 80](#_Toc405838256)

[2. Опрос матричной клавиатуры 85](#_Toc405838257)

[**Задания для самостоятельной работы** 90](#_Toc405838258)

[Тема № 11. Управление двигателями 91](#_Toc405838259)

[***1. Шаговые двигатели*** 91](#_Toc405838260)

[*1.1 Разновидности шаговых двигателей и способы управления ими* 91](#_Toc405838261)

[*1.2 Программная реализация полушагового режима работы* 94](#_Toc405838262)

[***2. Управление сервоприводом*** 96](#_Toc405838263)

[*2.1 Способы управления сервоприводами* 96](#_Toc405838264)

[*2.2 Программная реализация управления сервоприводом* 98](#_Toc405838265)

[**Задания для самостоятельной работы** 102](#_Toc405838266)

[ЗАКЛЮЧЕНИЕ 103](#_Toc405838267)

[СПИСОК ЛИТЕРАТУРЫ 104](#_Toc405838268)

# *ВВЕДЕНИЕ*

# Тема № 1. Общее устройство микроконтроллеров AVR

*Вопросы, рассматриваемые в данной теме*:

1. Семейства AVR

2. Структурная схема МК

3. Типы памяти, используемые в МК

4. Основные периферийные устройства

**Краткие теоретические сведения**

## 1. Краткие сведения о МК AVR

Микроконтроллеры (МК) AVR представляют собой достаточно несложные в изучение 8-разрядные RISK МК, это связано не только с тем, что набор команд RISK МК сам по себе прост, но и с тем, что данные МК уже достаточно давно на рынке. Вследствие чего они достаточно хорошо описаны и документированы, в том числе и на русском языке. Так же для МК данной архитектуры любителями было создано множество устройств, схемы и программный код, которых можно найти в сети Internet, что тоже в свою очередь может помочь в освоении МК.

Кроме простоты в изучении и хорошей документированности данные МК легкодоступны для покупки, легко прошиваются и обладают преемственной архитектурой, то есть при смене МК, ранее написанный код нуждается лишь в небольшой доработке. *Далее в работе, если не указано иное под МК подразумевается МК архитектуры AVR.*

## 2. Основные семейства МК AVR

Существуют четыре основных семейства МК:

1. МК семейства Classic (AT90S*xxx*), данное семейство МК уже не выпускается, а самая популярная модель AT90S2313, была заменена на ATtiny 2313.
2. МК семейства Tiny (ATtiny*xx*), самые простые с точки зрения внутреннего устройства МК, в основном размещаются в корпусах с 8-20 выводами, имеют небольшую память программ объемом 1-8 Кб, аппаратный стек и используются в простых и дешевых устройствах, а также как вспомогательные контроллеры.
3. МК семейства Mega (Mega*xx*) обладают большим объемом памяти программ (8-256Кб), и корпусами с 28-100 выводами, по сравнению с МК семейства Tiny имеют более богатую периферию и вследствие этого большую функциональность. Применение МК Mega хватает для решения практически всех не узкоспециализированных задач.
4. МК семейства XMega характеризуются повышенным быстродействием, еще большими объемами памяти программ и другими улучшениями. В связи с появлением более дешевых 32-разрядных МК архитектур ARM-Cortex M3 с теми же функциональными возможностями, семейство XMega практически не используется, поэтому *в данной работе не рассматриваются.*

Кроме вышеперечисленных семейств существуют специализированные МК основанные на ядре AVR, например, для работы с ЖК-дисплеями, USB-интерфейсом, CAN-интерфейсом, аккумуляторами. Это не значит, что данный функционал нельзя реализовать на стандартных МК, но зачастую использование специализированного МК ускоряет и упрощает разработку целевого устройства.

## 3. Структурная схема МК AVR

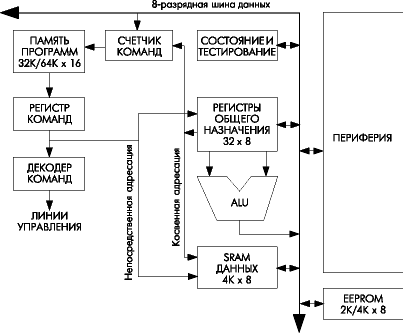
Рассмотрим структурную схему МК (рис. 1.1) и кратко разберем основной функционал входящих в ядро блоков:

*1. Регистры общего назначения (регистровый файл)*. Ядро МК содержит 32 8-рязрядных регистра общего назначения, которые связанны непосредственно с арифметико-логическим устройством (ALU), это дает возможность напрямую обращаться к регистрам МК для выполнения над ними различных операций (например, сложения, сравнения и т.п.) напрямую, без переноса значения в аккумулятор. РОН входящие в состав МК можно разделить на три группы:

Младшие R0…R15 – не могут быть использованы в операциях непосредственно с операндом (SBR, CBR, LDI и другие), это особенность архитектуры.

Старшие R16…R31– работают со всеми командами без исключения.

Индексные R26…R31 – помимо использования как обычные регистры, могут объединяться в регистровые пары X(R26:R27), Y(R28:R29), Z(R30:R31) использующиеся в виде указателей для работы с памятью.



*Рис. 1.1.* Общая структурная схема МК

*2. Арифметико-логическое устройство (ALU).* Используется для обработки всей математики в МК, доступны команды сложения, вычитания, сравнения, сдвигов, логических операций, переносов, умножения. Команда деления отсутствует. Так как ALU напрямую подключен к РОН поддерживаются арифметические, логические операции и битовые операции, как между регистрами, так и между регистром и константой.

*3.* *Счетчик команд и декодер команд.* По умолчанию при старте МК значение программного счетчика указывает на адрес первой команды в памяти программ ($0000). Микроконтроллер выбирает из адреса два байта (код команды и аргумент) и передает их на выполнение в декодер команд. Счетчик команд инкрементируется, команда выполняется, и происходит выборка очередной команды.

Если же встречается команда перехода, в счетчик команд загружается адрес, указанный в команде (абсолютный переход), либо его значение увеличивается не на единицу, а на необходимое значение (команды проверки-пропуска) и в следующем такте команда будет взята с нового адреса и выполнена.

Далее рассмотрим память, в структуре AVR имеются три разновидности:

*1. Память программ (Flash-память).* Данный тип памяти, как и любая другая flash-память имеет страничную организацию, причем размер страницы зависит от модели и составляет от 64 до 256 байт. Программирование страницы памяти возможно только целиком.

Память программ можно считать построенной из отдельных ячеек – слов по два байта каждое. Страничная организация памяти программ обусловлена тем, что подавляющее большинство команд в AVR имеют длину ровно 2 байта. Исключение составляют некоторые команды переходов (JMP, CALL) и команды работы с памятью (LDS) оперирующие с 16 и более разрядными адресами, длина таких команд составляет 4 байта. Такие команды используются в моделях МК с памятью программ свыше 8 Кб.

*2. Память данных (ОЗУ, SRAM).* К данному типу памяти в отличие от вышерассмотренной памяти программ адресация происходит линейно, без какого-либо деления на страницы или сегменты. Многие младшие МК семейства Tiny памяти данных не имеют, ограничиваясь наличием РОН. Многие старшие модели семейства Mega позволяют подключить дополнительную память объемом до 64 Кб с параллельным интерфейсом.

*Рис 1.2*. Адресное пространство SRAM

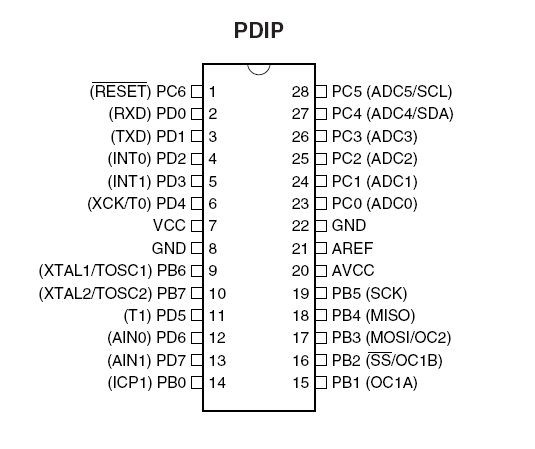
Разделение адресного пространства памяти SRAM представлено на рисунке 1.2.

В связи с тем, что операции чтения/записи одинаково работают со всеми адресами доступного пространства памяти SRAM, нужно быть внимательным, так как вместо SRAM можно попасть в один из РОН или регистр ввода/вывода (РВВ), например, команда загрузки значения регистра R0 в регистр R16 (MOV R0, R16) равносильна записи регистра R16 по нулевому адресу (STS $0000, R16). Такой прямой способ запись значения в регистр не очень удобен, так как занимает два такта вместо одного, но иногда это позволяет обойти ограничения записи в некоторые регистры.

*3.Энергонезависимая память (EEPROM).* Данный тип памяти используется для хранения констант и данных при отключении питания, ее объем обычно варьируется в пределах от 64 байт у младших моделей до 4 Кб у старших. Одно из отличий от Flash-памяти при аналогичной страничной организации (до 4Кб страница) это возможность выборочного программирования побайтно. Чтение из памяти осуществляется в течение одного машинного цикла, а вот процесс записи проходит более медленно (от 2 до ~4 мс), время записи зависит от встроенного в МК тактового RC-генератора, частота которого не стабильна, и зависит от напряжения питания.

## 4. Основные периферийные устройства

Кратко рассмотрим основные периферийные устройства (более подробно с практическими примерами наиболее используемые устройства будут рассмотрены в следующих темах), входящие в состав МК AVR, наиболее популярные из них, такие как таймеры, порт UART, аналоговый компаратор и сторожевой таймер имеется практически во всех моделях МК. Несмотря на то, что при написании программы не обязательно использовать все устройство, при выборе оптимального МК стоит учитывать, что даже незадействованное устройство потребляет энергию. Адресация всех периферийных устройств происходит через РВВ, так как прямая модификация регистров РВВ ограничена, то перенос значений РВВ в РОН и обратно осуществляется с помощью команд IN и OUT.

*1. Порты ввода/вывода.* Используются для обмена данными с внешними устройствами (от кнопки и лампочки до ЖКИ), в различных моделях может быть от 1 до 7 портов, номинально каждый порт 8 разрядный, то есть на один порт приходится 8 линий ввода/вывода, но в некоторых МК количество линий на порт может быть и меньше. Порты обозначаются буквами A, B, C и далее. Так же для сокращения контактов корпуса выводы МК, соответствующие портам ввода вывода, несут дополнительную функцию. Например, на рисунке 1.3 представлен МК AtMega8 в корпусе PDIP у которого 3 порта ввода/вывода (A, B, C), причем один порт имеет только 7 линий ввода вывода, и все порты МК несут альтернативный функционал, например, линия 3 порта D является так же линией внешнего прерывания.

*Рис.1.3.* МК AtMega8

*2. Таймеры-счетчики.* Данные периферийные устройства имеют очень широкую область применения от подсчета времени до работы в режимах ШИМ. Базовая частота таймеров равняется частоте МК, но может быть поделена на 8, 64, 256 или 1024, так же возможно тактирование таймеров от внешнего сигнала. Таймеры бывают двух разновидностей 8 и 16 разрядными, причем 8-разрядные таймеры имеют номера 0 и 2, а 16 разрядные 1 и 3. Один из 8-разрядных счетчиков в большинстве МК может работать в асинхронном режиме от отдельного тактового генератора, что позволяет, например, реализовать часы. На рисунке 1.3. таймеры представлены выводами T0, T1, OC1A, OC1B и другими.

Так же во всех без исключения МК существует сторожевой таймер (Watchdog) предназначенный для вывода МК из режимов энергосбережения или перезапуска МК в случае его зависания.

*3. Аналогово-цифровой преобразователь (АЦП).* Входит в большинство моделей МК, обычно число каналов равно 8, разрядность АЦП 10 бит. Служит для преобразования аналоговых сигналов в цифровые, может работать как в режиме одиночного, так и непрерывного преобразования. На рисунке 1.3. АЦП представлен выводами ADC0-ADC5.

Кроме вышеперечисленных устройств в МК аппаратно реализованы протоколы последовательной передачи данных, такие как *UART, SPI, TWI (I2C), USI.* Основное преимущество последовательных портов перед параллельными – снижение числа соединений, а также более стабильная работа на высоких скоростях. Стоит сказать, что практически любой порт как последовательный, так и параллельный можно имитировать программно, например, можно реализовать аппаратно не поддерживаемый МК AVR протокол 1-Wire.

**Задания для самостоятельной работы**

# Тема № 2. Тактирование,сброс и подключение мк avr

*Вопросы, рассматриваемые в данной теме*:

1. Способы тактирования МК

2. Сброс МК

3. Подключение питания к МК

**Краткие теоретические сведения**

## 1. Способы тактирования МК

В большинстве случаев, когда не требуется высокая стабильность частоты работы контроллера (например, в системе не используется таймеры для отсчета точных временных интервалов) используется встроенный RC-генератор, способный работать на частотах приблизительно равных 1, 2, 4 и 8 МГц.

Если же для работы устройства требуется отличная от этих величин частота, но требований к стабильности частоты так же нет, то можно подключить внешнюю RC цепочку (рис.2.1, в). В такой схеме емкость С1 не должна быть менее 22 пФ, а резистор выбирается из диапазона 3.3-100кОм, с учетом того что частота генератора определяется по формуле F=2/3RC. Так же МК позволяет вместо конденсатора C1 подключить встроенный конденсатор емкостью 36 пФ, записав 0 в конфигурационную ячейку CKPOT.



*Рис.2.1.* Способы тактирования МК

Если же устройство требовательно к стабильности тактовой частоты (например, используется протокол UART, или протокол, основанный на выдержке четких интервалов времени, такой как 1-Wire), то одним из решений будет подключение внешнего кварцевого резонатора к соответствующим выводам (рис.2.1, а). Емкость конденсаторов С1 и С2 должна составлять 15-22пФ.

Вторым решением будет являться тактирование МК от внешнего генератора (рис. 2.1, б), такой режим тактирования удобен в случаях, если необходимо синхронизировать МК с внешними компонентами, или запустить МК на нестандартной очень точной частоте тактирование.

По умолчанию МК семейств Tiny и Mega настроены на частоту работы от встроенного тактового генератора на частоте 1 МГц (CSEL=0001) , следовательно, для других режимов работы необходимо перенастроить конфигурационные биты МК в соответствии с таблицей 2.1.

*Таблица 2.1.* Установки конфигурационных ячеек CKSEL в зависимости от режимов тактирования

|  |  |  |
| --- | --- | --- |
| CKSEL3…0 | Источник тактирования | Частота |
| 0000 | Внешняя частота | 0…16 МГц |
| 0001 | Встроенный RC-генератор | 1 МГц |
| 0010 | Встроенный RC-генератор | 2 МГц |
| 0011 | Встроенный RC-генератор | 3 МГц |
| 0100 | Встроенный RC-генератор | 8 МГц |
| 0101 | Внешняя RC-цепочка | <0.9 МГц |
| 0110 | Внешняя RC-цепочка | 0.9…3.0 МГц |
| 0111 | Внешняя RC-цепочка | 3.0…8.0 МГц |
| 1000 | Внешняя RC-цепочка | 8.0…12 МГц |
| 1001 | Низкочастотный резонатор | 32.768 кГц |
| 101x | Кварцевый резонатор | 0.4…0.9 МГц |
| 110x | Кварцевый резонатор | 0.9…3.0 МГц |
| 111x | Кварцевый резонатор | 3.0…8.0 МГц |
| 1xxx (CKPOT=0) | Кварцевый резонатор | >1.0 МГц |

При конфигурировании МК всегда необходимо быть очень внимательным в связи с тем, что, неправильно выставив биты МК его нельзя будет перепрограммировать до установки выбранного источника частоты.

Более подробно о конфигурационных битах так же называемых FUSE битами будет рассказано далее.

## 2. Сброс МК

Сброс (RESET) это установка МК в начальный режим работы, при этом РВВ устанавливаются в состояние по умолчанию, а РОН могут принимать случайные значения, поэтому используемые в РОН переменные всегда необходимо инициализировать значениями.

Источниками сброса могут быть следующие события:

1. Включение МК

2. Аппаратный сброс, то есть подачи сигнала низкого уровня на вход /RESET.

3. Срабатывание сторожевого таймера.

Во многих МК на выводе /RESET присутствует подтягивающий резистор номиналом (100-500кОм) на котором могут наводиться помехи, приводящие к непредсказуемым сбросам МК, для избегания этого рекомендуется подключение внешнего резистора величиной 2-5кОм от вывода /RESET к напряжению питания. Так же для более надежной работы МК рекомендуется установить конденсатор 0.1-0.5 мкФ от вывода /RESET на «землю» (рис.2.2).

*Рис.2.2.* Подключение вывода /Reset

## 3. Питание МК

МК AVR работают от напряжения питания в пределах 1.8-5В в зависимости от серии и модели. Напряжение питания обычно обозначается как Vcc, земля или минус питания GND, перед подключением МК к источнику питания всегда рекомендуется уточнить допустимый диапазон питания в документации на МК. Так стоит обратить внимание на то, что возможность работы МК на определённых частотах зависит от величины питание, чем она больше, тем на больших частотах сможет работать МК. Если в МК больше чем один вывод Vcc и GND, то необходимо запитать их все. Так же в большинстве МК присутствуют выводы AGND и AVCC, аналоговая земля и аналоговое питание соответственно которые питают АЦП контроллера, таким образом, если использовать АЦП не планируется, то его можно запитать тем же напряжением что и Vcc и GND, в противном случае АЦП желательно запитать через дополнительные фильтры (например, дроссель), чтобы помехи, возникающие в основной цепи не влияли на результаты измерений. Полная схема подключения МК к питанию приведена на рисунке 2.3.



*Рис.2.3.* Подключение МК к питанию

Как видно из рисунка в дополнение к вышесказанному были добавлены два конденсатора в цепь питания, один из них емкостью 47мкФ сглаживает броски напряжения, конденсатор емкостью 100нФ сглаживает краткие импульсные помехи в шине питания, вызванные работой цифровых схем. Так же через конденсатор емкостью 100нФ подключен вход AREF МК, это вход опорного напряжения МК относительно которого свои значения будет рассчитывать АЦП, использование конденсатора несколько улучшит качество опорного напряжения.

**Задания для самостоятельной работы**

# Тема № 3. Общие принципы программирования МК

*Вопросы, рассматриваемые в данной теме*:

1. Atmel Studio

2. Команды инструкции и нотации AVR-ассемблера

3. Прерывания

4. Структура AVR программ

5. Регистр SREG, Отладка и компиляция AVR программ

6. Конфигурационные биты (Fuse биты)

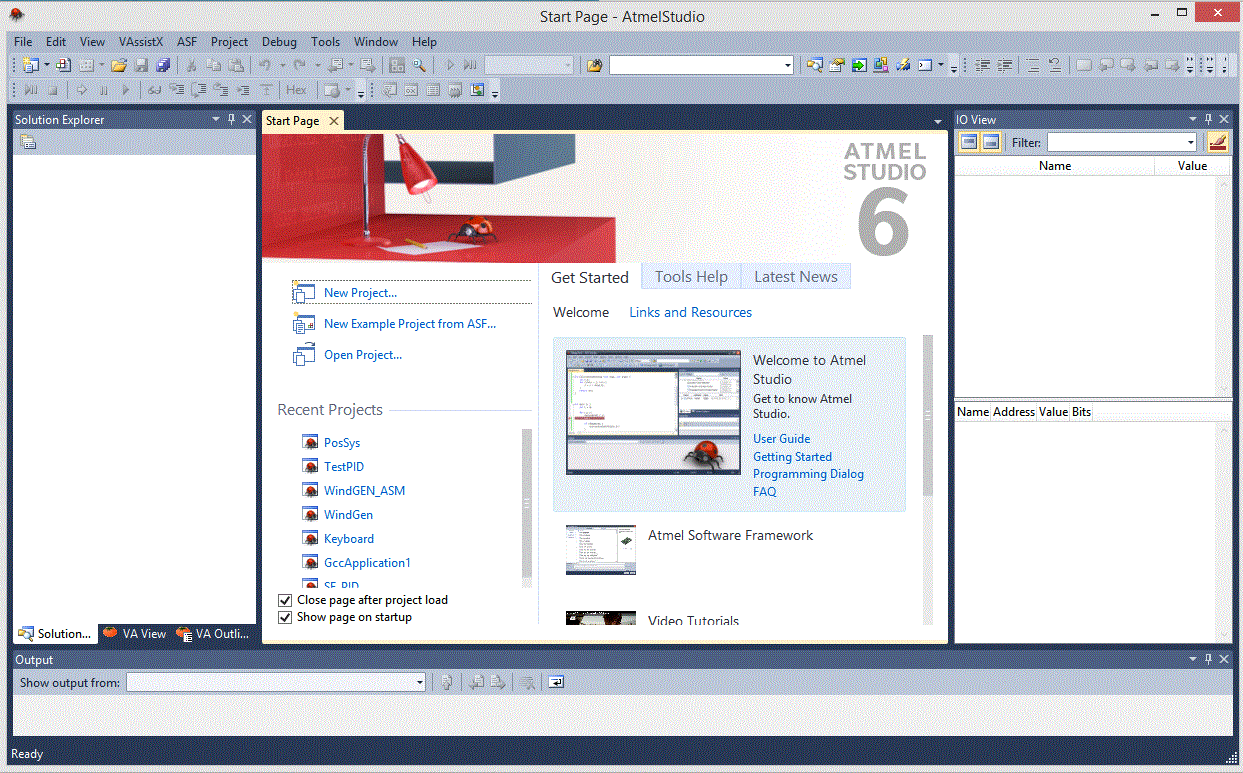
**Краткие теоретические сведения**

## 1. Atmel Studio

При написании каких-либо программ всегда встает вопрос в выборе языка, на котором программа будет писаться. При изучении микроконтроллеров обычно используется язык ассемблера, как известно ассемблер — это не универсальный язык программирования, это просто несколько правил, по которым последовательность команд процессора, записанных в мнемоническом виде, может объединяться в программу. Эта последовательность команд затем компилируется с помощью собственно ассемблера, который переводит мнемоники команд в форму пригодную для записи в МК. Одной из причин, по которой свои первые программы под МК рекомендуется писать на ассемблере является то, что это позволяет более подробно разобраться в его архитектуре, и в дальнейшем при написании сложных программ на языках высокого уровня будет полное понимание как что работает, что зачастую позволяет найти ошибки в программе и написать архитектурно более «правильный код». Еще одна причина, это простота ассемблера, его фирменное описание занимает всего несколько страниц, а команды понятны и легко запоминаются. Ассемблера обладает одним большим недостатком: программы получаются большими, и не всегда удобочитаемыми, так же многие математические операции, такие как деление, операции с 16 разрядными числами отдаются на откуп программиста, а не компилятора и встроенных библиотек высокого уровня.

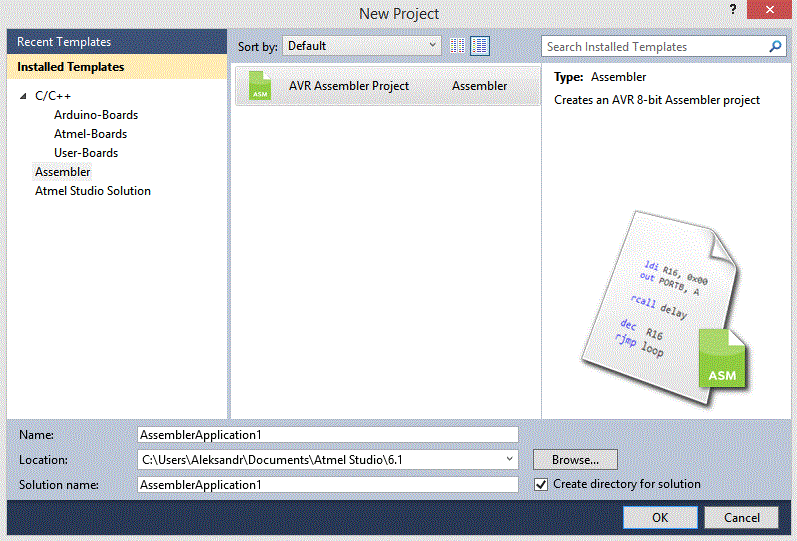
Как было сказано выше, для того чтобы получить последовательность команд и данных в двоичном формате допустимую для загрузки в МК, написанную программу нужно скомпилировать. Написание, отладка и компиляция программ под МК AVR обычно выполняются в среде разработки Atmel Studio (ранее AVR Studio) последнюю версию которого можно скачать на официальном сайте. Данный программный продукт позволяет создавать программы под все МК производства фирмы Atmel как на языке ассемблера, так и на C.

Рассмотрим создание простейшего проекта программы на языке ассемблера. При запуске Atmel Studio появляется окно (рис.3.1):



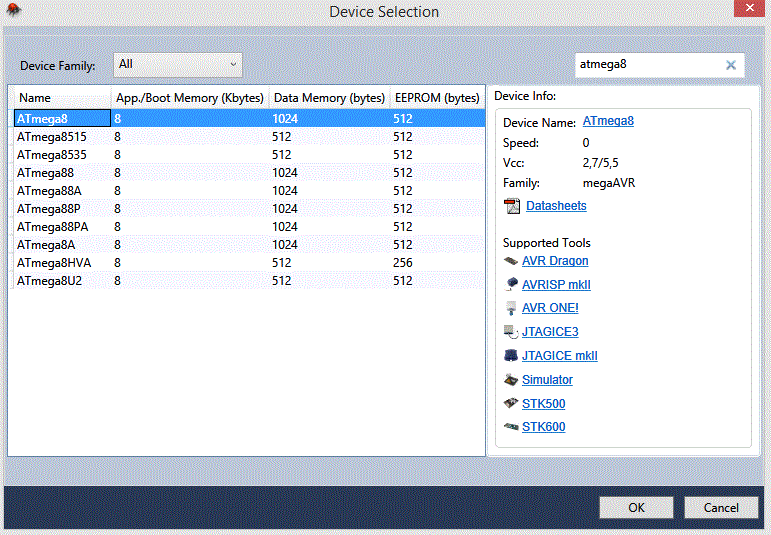
*Рис.3.1.* Главное окно Atmel Studio

Стартовое окно сразу предлагает создать новый проект, открыть старый, и показывает недавно открывавшиеся проекты. Создадим новый проект, нажав кнопку New Project (Так же новый проект можно создать через меню File-New-Project). Откроется окно нового проекта (рис.3.2.). В котором предлагается на выбор создание проекта на языках С/С++ с выбором под какую готовую платформу пишется код, или Ассемблер, а также предлагается дать имя проекту, и выбрать его местоположение (не рекомендуется названия и использовать пути к проекту содержащие кириллицу, проект может не компилироваться). После выбора необходимых параметров (Assembler), для перехода к следующему окну создания проекта щелкаем OK.



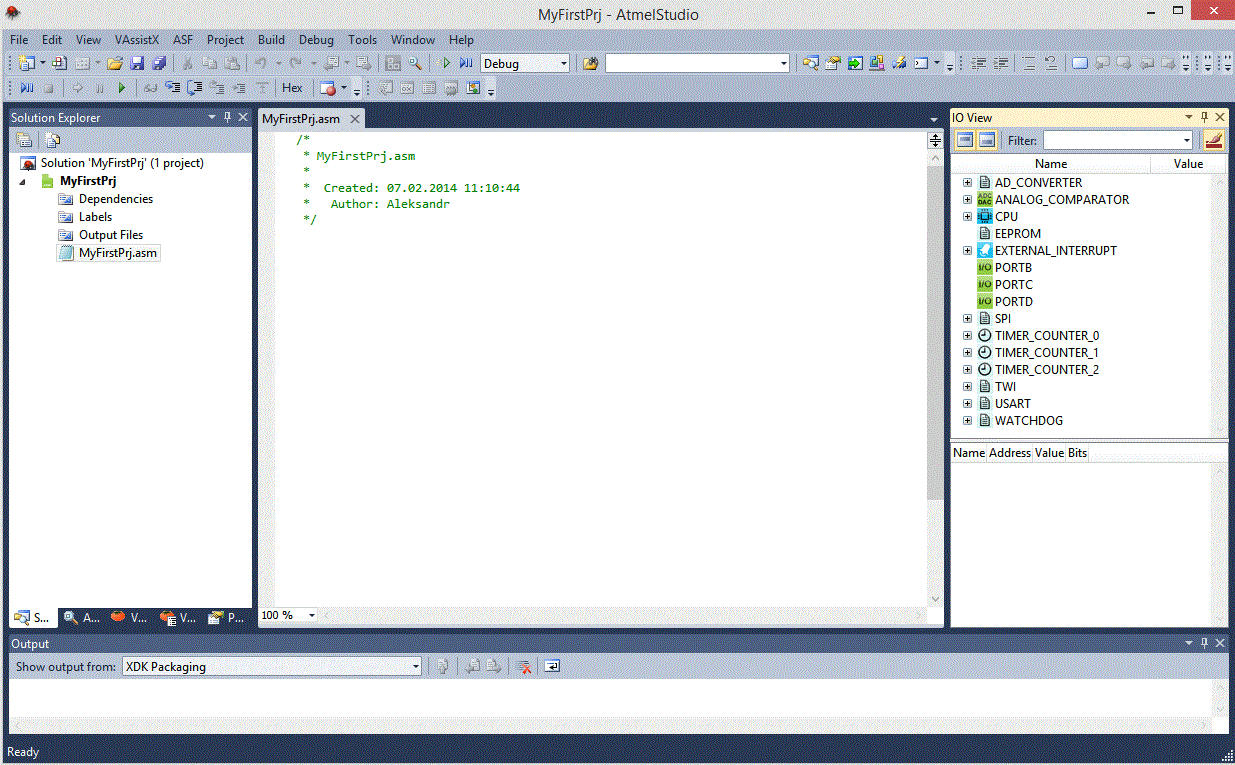
*Рис.3.2.* Окно нового проекта

Следующее окно (рис.3.3) – окно выбора МК, в этом окне можно отсортировать семейства (выпадающий список Device Family), так и найти нужный МК в строке поиска, для примера выбран МК AtMega8. В разделе со списком МК даны их базовые характеристики: объем памяти программ, данных и EEPROM. В разделе Device Info приведены: имя МК, его напряжение питания, тип семейства, дана ссылка на техническое описание МК (Datasheet), а также представлены поддерживаемые данным МК отладчики. Программный отладчик, используемый в Atmel Studio, называется Simulator. Выбранный МК является избыточным для решения большинства простых задач, но очень удобен для примеров, так как содержит всю необходимую периферию. После выбора МК для перехода к следующему окну необходимо нажать кнопкуOK.



*Рис.3.3.* Окно выбора МК

Откроется окно среды разработки (рис.3.4), в котором можно приступать к написанию программы. Рассмотрим его основные составляющие, слева располагается Solution Explorer, путеводитель по вашему проекту, в разделе Dependencies можно увидеть все дополнительно подключенные к проекту библиотеки, в разделе Labels - метки переходов, в Output Files – получаемые после компиляции выходные файлы, и последним пунктом идет файл .asm с названием созданного проекта. Слева находится панель I/O View в которой можно увидеть абсолютно все устройства МК, а также их регистры с описанием за что какой отвечает (если оно отсутствует включить его можно из меню Debug-Windows- I/O View), данную панель удобно использовать не только при отладке, но и при написании программы для того чтобы посмотреть название регистра и его назначения не открывая справочник. Снизу находится панель выходных данных, в которой при компиляции программы буду появляться ошибки, предупреждения, отчеты об успешной компиляции и другие сообщения.



*Рис.3.4.* Окно среды разработки

Среда Atmel Studio очень гибко и удобно настраивается, и ее первоначальный вид может немного отличаться от представленного на рисунке. Так же внешний вид среды при написании программы и ее отладке тоже несколько отличаются.

## 2. Команды инструкции и нотации AVR-ассемблера, прерывания

Перед тем как перейти непосредственно к написанию программы рассмотрим некоторые базовые понятия ассемблера, особенности записи команд, а также разберем, что такое прерывания и как проходит их обработка.

Начнем с особенности записи команд в ассемблере, сначала идет команды, которая может быть двух-, трех- или четырехбуквенная, затем через пробел (или через несколько, количество пробелов не влияет на исполнение команды) идут операнды. Есть команды, не имеющие операндов (LPM, RET), другие команды имеют только один операнд (DEC R16). В командах, которым необходимо два операнда используется «прямая польская запись», то есть сначала указывается приемник, затем источник. Операнды разделяются запятой. Например, команда ADD R16, R17 сложить значения указанных регистров и результат операции поместит в регистр R16. Каждая команда должна занимать отдельную строку, кроме команды, строка может содержать метки и комментарии (выделяются символом ; или //). Регистр букв ассемблером не различается и значения не имеет, то есть команда RJMP LABEL и rjmp label идентичны.

Далее о метках, команда из примера выше означает, что будет совершен относительный переход на метку LABEL, в коде программы метка, на которую будет совершаться переход обозначается как произвольно придуманное имя с двоеточием на конце (LABEL:), после метки может быть расположена какая-либо инструкция.

Во многие команды, работающие с константами можно включать числа и выражения, все числа по умолчанию считаются десятичными, за исключением чисел с ведущим нулем (06), которым обозначаются восьмеричные числа. Двоичные числа записываются в виде: 0b00101100, шестнадцатеричные числа могут быть записаны как 0xAA, так и $AA.

Стоит обратить внимание на то, что адреса в программе являются числовыми константами и запись вида RJMP LABEL+1 допустима и будет интерпретирована как перейти на следующую команду после той, на которую ссылается метка.

Кроме операции сложения с константами можно выполнять любые математические и логические операции, одной из часто употребляемых является операция сдвига влево (<<), данная используется для установки определённых именованных битов регистров (для удобства все биты РВВ имеют имена), вместе с операцией побитовое «ИЛИ» (|), в одном регистре можно установить сразу несколько битов, например:

LDI R16, (1<<CS00)|(1<<CS01)

OUT TCCR0, R16

То есть сначала в один из регистров МК (напрямую записывать константы в РВВ нельзя) записывается необходимые значения регистров в данном примере отвечающие за предделитель таймера 0, то есть биты CS00 и CS01 устанавливаются 1, и копируются в конфигурационный регистр таймера 0.Так происходит настройка всех устройств МК.

Часто в программах кроме команд используются различные директивы компилятора, самые распространённые из них это: .def, .equ, #include. Первая директива используется для именования регистров:

.def temp=R16 ;теперь R16 является переменной temp

Хорошим тоном считается именовать используемые РОН, так как это упрощает чтение и понимание логики программы.

Директива .equ применяется для именования констант:

.equ min\_value = 5 ; Минимальное значение = 5

.equ max\_value = 10 ; Максимальное значение = 10

Именование и описание констант, используемых в программе очень важная возможность, которой не стоит пренебрегать, позволяет избегать «волшебных чисел», констант в командах которые не говорят о том, что это за константа совершенно ничего. Так же с помощью директивы .equ можно определять довольно сложные выражения, так например, чтобы вручную не рассчитывать значения констант UART для скорости обмена можно использовать такую конструкцию:

.equ XTAL = 8000000 ;Частота МК

.equ baudrate = 9600 ;Необходимая скорость передачи

.equ bauddivider = XTAL/(16\*baudrate)-1 ;Расчет константы

Помимо вышеперечисленных способов использования .equ, эта директива часто используется для обозначения переменных, которые располагаются в области SRAM, для этого вместо десятичной константы определяется константа в шестнадцатеричном формате равная адресу переменной в памяти.

Изначально ассемблер не знает об именовании регистров, он знает только их физические адреса в памяти, для того чтобы связать мнемонические обозначения с адресами необходимо подключить специальный .inc файл в котором именуются все адреса регистров, данный файл для каждого МК отличается, для его подключения служит директива #include:

#include "m8def.inc"

Данная директива подключает файл с определениями всех констант и адресов для МК AtMega8. #include позволяет подключить абсолютно любой файл, например, какую-либо библиотеку. Файлы библиотек необходимо подключать очень внимательно, так как компилятор вставляет все содержимое файла в месте использования директивы, и, если вставить библиотеку, которая содержит команды, таблица векторов прерываний будет фактически затерта, что может привести к неработоспособности программы. Во избежание этого такую библиотеку необходимо вставить не вначале программы, а после таблицы векторов прерываний.

Дополнительно рассмотрим директивы сегментов памяти .DSEG, .CSEG, .ESEG, они обозначают начало сегмента памяти SRAM, сегмент кода и сегмент EEPROM. Эти директивы важны при создании массивов в памяти МК с помощью директивы .db, они однозначно указывают в каком типе памяти должны размещаться константы, если не указать тип памяти, константы будут сохранены в память программ. Например:

N\_mask:

.db 1, 3, 5, 7, 9

Здесь в память программ (так как не указан тип памяти), по адресу, обозначенному меткой N\_mask, сохранен набор целочисленных констант.

Директива .db работает только с памятью программ и EEPROM, для размещения данных в SRAM используется директива .byte которая указывает число резервируемых в памяти байт .

Еще одна часто используема директива - .org позволяет задать компилятору начальный адрес в пределах сегментов кода, данных и EEPROM-памяти, то есть тот адрес, с которого пойдет запись в выбранный тип памяти.

## 3. Прерывания

Перед рассмотрением структуры программы на языке ассемблера, разберем вопрос использования прерываний. Работа МК это, прежде всего ожидание и обработка событий. Таким образом, аппаратные прерывания являются важной частью МК, которая позволяет автоматически определять наступление какого-либо события, останавливать выполнения основной программы и переходить непосредственно к коду прерывания.

Это происходит следующим образом: при возникновении условия прерывания (например, таймер переполнен) МК выставляет бит в регистре прерываний таймера, если прерывания разрешены контроллер, автоматически вычисляет адрес соответствующего вектора прерывания и переходит к обработке прерывания. Перед переходом к прерыванию МК запрещает все прерывания во избежание срабатывания прерывания во время исполнения текущего, а также сохраняет содержимое счетчика команд в стеке для того чтобы после завершения прерывания продолжить выполнение программы с места где исполнение программы было прервано. По возвращению из прерывания выполнение других прерываний автоматически разрешается. Так как для работы прерываний необходим стек, в моделях МК в которых отсутствует аппаратный стек его необходимо инициализировать сразу после начала выполнения программы:

//Инициализация стека: (обязательно во всех МК с программным стеком AtMega)

LDI R16, LOW(RAMEND) //Загрузка указателя стека в конец SRAM

OUT SPL, R16

LDI R16, HIGH(RAMEND) //Загрузка указателя стека в конец SRAM

OUT SPH, R16

Рассмотрим другие важные моменты, связанные с прерываниями:

1. По умолчанию прерывания запрещены.

2. При возникновении прерывания содержимое регистра флагов SREG не сохраняется, поэтому если он важен для корректного выполнения основной программы, его необходимо сохранить в стек.

3. Кроме глобального разрешения прерываний, каждое прерывание должно быть разрешено индивидуально в специальном регистре прерываний имеющегося у каждого периферийного устройства способного вызывать прерывания.

4. Вложенные прерывания допустимы, но необходимо учитывать размер стека, так например аппаратный стек в МК Atiny имеет всего три уровня, и при вызове более трех подпрограмм (в том числе и прерываний), контроллер не сможет «вспомнить» по какому адресу он должен продолжить выполнение программы.

5. Если во время прерывания или сразу по выходу из прерывания сработает другое прерывание, переход к выполнению прерывания произойдет только после выполнения одной команды основной программы.

6. Во время выполнения атомарных процедур, то есть тех процедур, прерывание которых может вызвать некорректность работы программы необходимо запрещать прерывание. Примером такой операции может служить запись в EEPROM.

Каждый МК имеет свою индивидуальную таблицу прерываний, которую можно найти в техническом описании МК, так таблица прерываний для МК AtMega8 выглядит следующим образом:

.ORG INT0addr ; External Interrupt Request 0

RETI

.ORG INT1addr ; External Interrupt Request 1

RETI

.ORG OC2addr ; Timer/Counter2 Compare Match

RETI

.ORG OVF2addr ; Timer/Counter2 Overflow

RETI

.ORG ICP1addr ; Timer/Counter1 Capture Event

RETI

.ORG OC1Aaddr ; Timer/Counter1 Compare Match A

RETI

.ORG OC1Baddr ; Timer/Counter1 Compare Match B

RETI

.ORG OVF1addr ; Timer/Counter1 Overflow

RETI

.ORG OVF0addr ; Timer/Counter0 Overflow

RETI

.ORG SPIaddr ; Serial Transfer Complete

RETI

.ORG URXCaddr ; USART, Rx Complete

RETI

.ORG UDREaddr ; USART Data Register Empty

RETI

.ORG UTXCaddr ; USART, Tx Complete

RETI

.ORG ADCCaddr ; ADC Conversion Complete

RETI

.ORG ERDYaddr ; EEPROM Ready

RETI

.ORG ACIaddr ; Analog Comparator

RETI

.ORG TWIaddr ; 2-wire Serial Interface

RETI

.ORG SPMRaddr ; Store Program Memory Ready

RETI

.ORG INT\_VECTORS\_SIZE

*Листинг 3.1.* Таблица векторов прерываний AtMega8

Рассмотрим ее подробней, с помощью директивы .org определяется физический адрес каждого прерывания в МК, команда RETI выступает заглушкой на случай самопроизвольного срабатывания прерывания, в программе для реализации прерывания необходимо заменить RETI на RJMP LABEL, где LABEL это метка обработчика прерывания, выход из которого необходимо совершать по команде RETI. Более подробно реализация прерываний будет рассмотрена далее на примерах.

## 4. Структура AVR программ

Каждая программа, написанная для МК написанная на языке ассемблера, включает в себя несколько обязательных элементов (указаны в порядке расположения в программе):

1. Директивы препроцессора, в которой подключаются, необходимы библиотеки, определяющие тип МК.

2. Именование переменных и определение констант.

3. Сегмент SRAM, где выделяется память под переменные.

4. Таблица векторов прерываний, за которой обычно располагают обработчики прерываний.

5. Сегмент кода, и собственно программа, исполнение каждой программы начинается с нулевого адреса, где всегда располагается одна и та же команда безусловного перехода RJMP RESET. Естественно данная метка (именование метки RESET общеупотребимо, но называться она может как угодно) должна быть дальше указана в программе, в любом удобном месте после таблицы векторов прерываний. Обычно по метке RESET располагают инициализацию стека, настройку периферии, портов ввода/вывода.

6. В каждой программе должен присутствовать бесконечный цикл, он может быть основан как на метке RESET, так и на любой другой.

7. В конце программы обычно располагается сегмент данных EEPROM.

С учетом вышеперечисленных требований структура программы (таблица прерываний указана не полностью) будет выглядеть следующим образом:

//Директивы препроцессора

#include "m8def.inc"

//Определение переменных (директива .def)

//Определение констант (директива .equ)

//Сегмент ОЗУ (RAM)

.DSEG

//Сегмент кода (Flash)

.CSEG

.ORG 0x0000

RJMP RESET

//Таблица векторов прерываний:

.ORG INT0addr ; External Interrupt Request 0

RETI

…

.ORG SPMRaddr ; Store Program Memory Ready

RETI

.org INT\_VECTORS\_SIZE

//Конец таблицы векторов прерываний

//Сегмент обработчиков прерываний

//Конец сегмента обработчиков прерываний

RESET:

//Инициализация стека: (обязательно во всех МК с программным стеком AtMega)

LDI R16, LOW(RAMEND) //Загрузка указателя стека в конец SRAM

OUT SPL, R16

LDI R16, HIGH(RAMEND) //Загрузка указателя стека в конец SRAM

OUT SPH, R16

MAIN:

RJMP MAIN

//Сегмент энергонезависимой памяти (EEPROM)

.ESEG

*Листинг 3.2.* Структура программы

## 5. Отладка и компиляция простейшей AVR программ

Может возникнуть вопрос, в случае если же прерывания не используются, можно ли начать выполнение программы с нулевого адреса? Конечно можно, рассмотрим такую простейшую программу, а также процесс ее компиляции и отладки.

//Директивы препроцессора

#include "m8def.inc"

//Сегмент кода (Flash)

RJMP RESET

RESET:

LDI R16, 12 //Загрузим в регистр константу

LDI R17, 18 //Загрузим в регистр константу

ADD R16,R17 //Сложим значения в регистрах, результат в R16

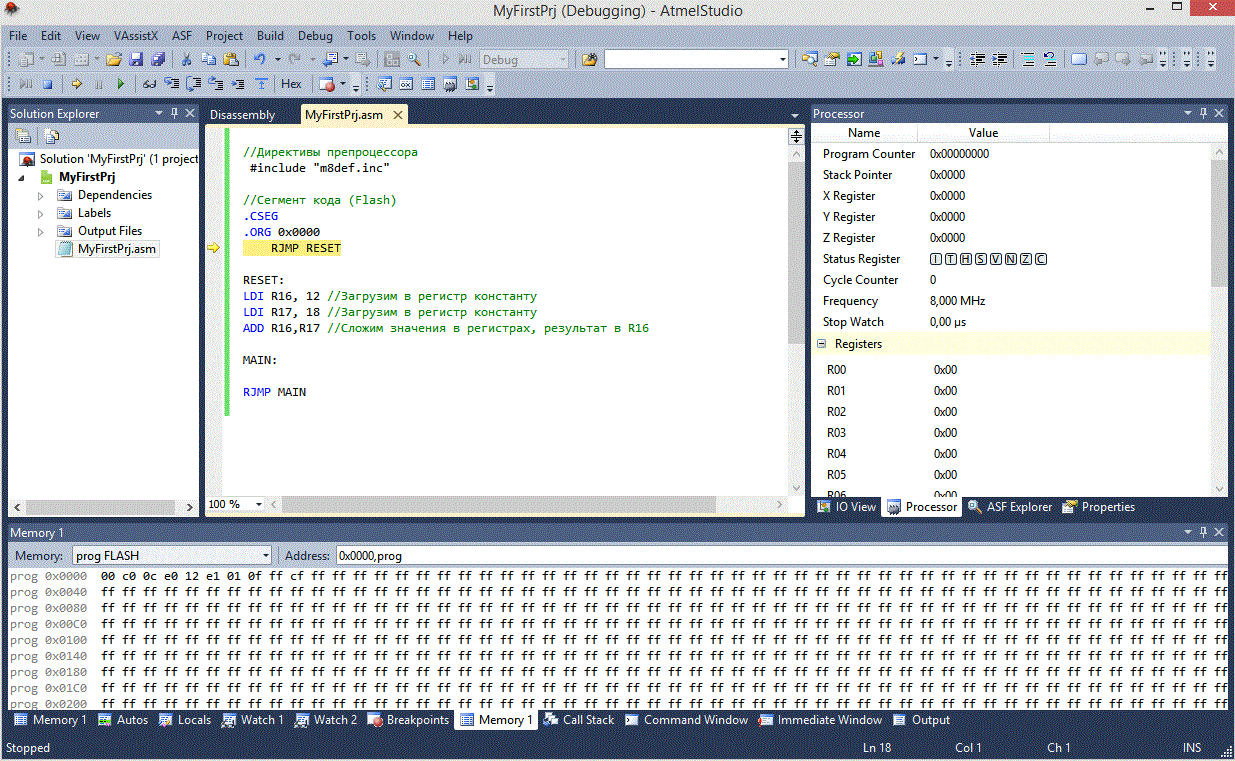
MAIN:

RJMP MAIN

*Листинг 3.3.* Простейшая программа

Как видно из листинга 3.3 в данной программе отсутствует таблица векторов прерывания, и инициализация стека, таким образом использовать прерывания и вызов функций командой CALL в такой программе нельзя. Также не именуются никакие переменные и константы, присутствует только обязательный переход с нулевого адреса и бесконечный цикл. Процесс работы программы прост, в каждый регистр загружается константа, затем значения, находящиеся в регистрах, складываются, и результат помещается в первый из них, рассмотрим процесс отладки программы. Для перехода к отладке необходимо в меню Debug выбрать пункт Start debugging and break, при первом запуске отладчика может появиться окно с выбором типа отладчика, необходимо выбрать Simulator. При запуске отладчика вид рабочей среды видоизменяется, рассмотрим его (рис.3.5). В окне с кодом появился указатель, который указывает текущее место выполнения программы. На панели справа кроме вкладки I/O VIEW появилась вкладка Processor, в которой можно увидеть текущее состояние контроллера, значение счетчика команд, указателя стека, регистровых пар, регистра SREG, количество тактовых циклов, частоту МК, время реальной работы программы и все РОН. Переход к следующей команде осуществляется нажатием кнопки F11. Так же важным моментом является то, что можно самостоятельно менять значения битов в РВВ и РОН щелкая по ним мышкой.

Посмотрим, как выполняется программа, изначально указатель стоит по нулевому адресу на команде перехода RJMP, затем происходит переход по метке RESET: и в регистры R16 и R17 записываются значения констант, в окне Processor изменяются соответствующие значения регистров, далее происходит сложение регистров и результирующее значение помещается в регистр R16.



*Рис.3.5.* Окно отладчика

Так же внизу окна программы можно увидеть карты памяти МК, в выпадающем меню можно выбрать различные сегменты памяти, flash-память, сегмент памяти регистров и другие.

Компиляция программы выполняется из меню Build командой Build solution, результат компиляции в окно Output, и в случае успешной компиляции включает в себя пути компиляции, таблицу с указанием используемой памяти, сведения о количестве ошибок и предупреждений. В случае наличия ошибок и предупреждений, будет выведен их код и краткое описание с указанием на строку с ошибкой.

## 6. Конфигурационные биты (Fuse биты)

Fuse биты или конфигурационные биты МК необходимы для «жесткого» конфигурирования МК, данная операция производится при прошивке устройства и требует внимательности, так как неверно выставленные Fuse биты могут привести микроконтроллер в состояние полной неработоспособности. Для разных моделей МК набор конфигурационных битов практически одинаков. Рассмотрим типовое состояние ячеек, заданное производителем. По умолчанию любой МК тактируется от внутренней RC-цепи, таким образом для базовой работы МК наличие внешней цепи тактирования не требуется. Для работы от другого источника тактового сигнала как было сказано выше необходимо изменить соответствующие конфигурационные биты (CKSEL0-3).

Рассмотрим другие достаточно часто используемые ячейки:

BOOT\_ - ячейки настройки загрузки МК, позволяют изменить начальный адрес программы, адрес вектора прерывания, выбирать тип запуска МК.

Ячейки BODEN и BODLEVEL, разрешают работу схемы сброса контролера при падении питания ниже допустимого порога (схема BOD) и позволяют установить этот самый порог соответственно. Верный выбор значений предохранит МК от выполнения команд при недостаточном питание, которое может привести к «странностям» в работе устройства.

SPIEN – разрешает или запрещает последовательное программирование по SPI.

EESAVE – разрешает или запрещает обнуление EEPROM при программировании памяти программ.

В случае же если были выставлены неверные значения и МК не реагирует ни на какие внешние воздействия, можно переустановить Fuse-биты с помощью параллельного программатора.

**Задания для самостоятельной работы**

# Тема №4. Система команд AVR

*Вопросы, рассматриваемые в данной теме*:

1. Команды условных переходов и регистр SREG

2. Команды пересылки данных

3. Выполнение типовых процедур на ассемблере

4. Битовые операции на ассемблере

**Краткие теоретические сведения**

## 1. Команды условных переходов и регистр SREG

Практически во всех микроконтроллерах различных архитектур присутствует регистр состояния, в архитектуре AVR он называется SREG, о нем не раз упоминалось в предыдущих разделах, рассмотрим его подробней.

Название расшифровывается как Status Register, этот регистр состоит из 8 независимых битов-флагов, которые могут принимать значения 0 или 1 в зависимости от операций, выполняемых контроллером. По значениям флагов в этом регистре можно судить о том, что произошло с процессором и определить результаты некоторых операций, например, если флаг Z установлен в значение 1, значит, в предыдущей математической операции результатом был 0. Биты регистра состояния приведены в таблице 4.1.

*Таблица 4.1.* Биты регистра SREG

|  |  |  |  |
| --- | --- | --- | --- |
| № бита | Обозначение | Наименование | Описание |
| 7 | I | Общее разрешение прерываний | Для разрешения прерываний данный флаг должен быть установлен в 1 (команда SEI), данный флаг сбрасывается аппаратно после входа в прерывание, и автоматически устанавливается после выхода из прерывания по команде RETI |
| 6 | T | Хранение копируемого бита | Используется в качестве источника или приемника бита при применении команд BLD (bit load) и BST (bit store) |
| 5 | H | Флаг половинного переноса | Устанавливается в случае если по результатам математической операции был совершен перенос бита из младшей половины байта в старшую или заем бита из старшей половины в младшую |
| 4 | S | Флаг знака | Устанавливается если результат математической операции меньше 0 |
| 3 | V | Флаг переполнения дополнительного кода | Устанавливается в 1 при переполнении разрядной сетки знакового результата (числа представленного в виде с дополнительным кодом) |
| 2 | N | Флаг отрицательного значения | Устанавливается в 1 если старший разряд результата равен 1 |
| 1 | Z | Флаг нуля | Устанавливается в 1 если результат операции равен нулю. |
| 0 | C | Флаг переноса | Устанавливается в 1, в если результате выполнения операции произошел выход за границы байта |

Кроме автоматической установки каждый флаг можно устанавливать и сбрасывать с помощью команды SE\* и CL\* соответственно, где на месте \* подставляется буквенное обозначение флага.

На базе этого регистра состояния работают такие важные команды, как команды условных переходов. Разберем эти команды на примере: представим, что у нас есть два неких значения, снятые с датчика и в зависимости от их разности МК должен выполнять различные действия, в данном случае устанавливать значение равное значению регистра R17 (Листинг 4.1).

//Директивы препроцессора

#include "m8def.inc"

//Сегмент кода (Flash)

RJMP RESET

RESET:

LDI R16, 12 //Загрузим в регистр константу

LDI R17, 18 //Загрузим в регистр константу

MAIN:

CP R16,R17 //Сравним значения в регистрах

BRNE NOT\_EQUAL

RJMP MAIN

NOT\_EQUAL:

MOV R16, R17 // Скопируем в R16 значение R17

RJMP MAIN

*Листинг 4.1.* Пример использования команды условного перехода (BRNE)

Посмотрим, как изменяются значения бит в регистре SREG при работе команды сравнения (CP) в случае если первый операнд команды меньше второго. Запустив отладчик можно увидеть, что биты, устанавливаются во флагах S, N, C, а это значит, что операция сравнения фактически вычитает из первого значения второе и, не меняя значения в регистрах, выставляет флаги в регистре SREG. Далее выполняется команда условного перехода BRNE (brunch not equal – перейти если не равно) которая проверяет наличие вставленного флага Z, и, если он не выставлен, совершается переход на указанную метку (NOT\_EQUAL). По данной метке командой MOV копируется значение регистра R17 в регистр R16 и происходит переход в основной бесконечный цикл, где снова происходит операция сравнения, по итогам которой выставляется флаг Z, и команда перехода естественно не выполняется.

Существует множество команд условных переходов, которые можно посмотреть в справочнике команд, самые распространённые из них это переходы по условиям равенства (BREQ), неравенства (BRNE), а также команды перейти, если значение больше или равно (BRSH) и перейти, если значение меньше (BRLO). В дополнение к ним существуют команды типа BRxS и BRxC, которые в зависимости от того установлен (S) или сброшен (C) флаг x регистра SREG совершают переход на метку.

Одной из особенностей команд переходов BR\*\* является то, что эти команды являются командами относительного перехода и их разрядности может не хватить, чтобы перепрыгнуть на метку, которая находится достаточно далеко в адресном пространстве, если это произойдет, компилятор выдаст ошибку «Relative branch out of reach», что делать в этом случае? Использовать промежуточные пустые переходы.

Кроме команд типа «перейти, если» существует еще один тип команд переходов: проверить и пропустить. Такие команды работаю по принципу, проверяем условие, если оно верно – пропускаем следующую команду. Существует две пары команд данного типа — это команды SBRC/SBRS, данные команды работают с РОН, и команды SBIC/SBIS которые проверяют биты РВВ. Рассмотрим пример команды работающей с РОН:

//Директивы препроцессора

#include "m8def.inc"

//Сегмент кода (Flash)

RJMP RESET

RESET:

LDI R16, 0b00000000 //Загрузим в регистр константу

MAIN:

SBRC R16, 4 //Если 4 бит в регистре не установлен

RJMP MAIN //Пропускаем следующую команду

LDI R16, 0b00010000 //Установим четвертый бит в R16

RJMP MAIN

*Листинг 4.2.* Пример использования команды условного перехода (SBRC)

В данной программе в регистр R16 записано значение равное нулю, затем в основном цикле проверяется, установлен ли бит 4 в регистр, если нет, то следующая команда пропускается, и программа выполняется дальше.

Команды условных переходов очень важны при написании программ на ассемблере, с помощью них могут быть реализованы все типовые конструкции языков высокого уровня, такие как if…then и им подобные.

## 2. Команды пересылки данных

Теперь рассмотрим другой важный класс команд, команды, которые переносят данные из одной области памяти в другую. В архитектуре AVR для записи и переноса данных между разными типами памяти существуют разные команды. Команды, работающие с РОН уже знакомы, так как были использованы ранее, это LDI и MOV, их отличие состоит в том, что первая команда в РОН загружает непосредственно константу, а вторая копирует значение одного регистра в другой.

Следующий класс команд служит для работы с РВВ, это OUT и IN, первая команда позволяет записать в РВВ значение, вторая соответственно считать его.

Команды LD и ST, позволяют загружать и читать значения из памяти SRAM. В любом случае при записи и чтении SRAM используются регистровые пары X,Y и Z. Существует несколько режимов обращения к SRAM: чтение одной ячейки, и режимы с преддекрементом и постинкрементом служащие для чтения/записи фрагмента данных. Рассмотри основной порядок действий при чтении из памяти одной ячейки используя в качестве указателя на нее регистровую пару Z. Для чтения из памяти используются регистровые пары по причине того, что если бы использовался только один 8 разрядный регистр, то нельзя было бы адресовать памяти больше чем 28, тогда же как в МК памяти SRAM больше. Чтение из ячейки памяти с заданным адресом SAMPLE, изменение и запись нового значения выполняются так:

//Директивы препроцессора

#include "m8def.inc"

//Сегмент данных (SRAM)

.dseg

SAMPLE: .BYTE 1 //Выделим в SRAM 1 байт по адресу SAMPLE

.cseg

//Сегмент кода (Flash)

RJMP RESET

RESET:

LDI ZH, HIGH (SAMPLE) //Установим старший разряд регистра-указателя

LDI ZL, LOW (SAMPLE) //Установим младший разряд регистра-указателя

//Таким образом регистровая пара Z указывает на первую ячейку выделенной

//памяти SAMLE

LD R16,Z //Загрузим значение в регистр

LDI R16, 0xFF //Модифицируем его

ST Z, R16 //Сохраним модифицированное значение обратно в память

MAIN: //Бесконечный цикл

RJMP MAIN

*Листинг 4.3.* Пример чтения и записи в SRAM

Запустим программу (листинг 4.3) в режиме отладки и исследуем ее работу. В сегменте данных программы директивой .BYTE выделяется один байт памяти, который располагается по адресу SAMPLE (имя естественно может быть любое, фактически это метка), затем происходит установка указателя на данный сегмент памяти в регистровую пару Z, таким образом можно увидеть, что в один из регистров этой пары записывается значение адреса (R30=0x60), из этого можно сделать вывод что чтение и запись будет происходить именно по этому адресу в SRAM, откроем вкладку Memory 1отладчика и выберем из выпадающего списка память – data IRAM, и посмотрим, что находится в необходимой нам ячейке памяти (0x00). Далее происходит загрузка этого значения в регистр (LD), его изменение (LDI) и сохранение по адресу в памяти (ST), теперь посмотрев значение ячейки SRAM можно увидеть в нем новое значение.

При работе с памятью данных нужно внимательно следить за двумя вещами, первое, это не выходить за границы существующей памяти, второе: если в программе используются прерывания и вызовы процедур, нужно помнить, что последние адреса памяти занимаются под стек.

В тех случаях, когда надо считать и записать массив данных используются команды:

ST -Z, Rx //запись в ячейку с адресом Z-1 значения из регистра x,

//после выполнения команды Z-1

ST Z+, Rx //запись в ячейку с адресом Z значения из регистра x,

//после выполнения команды Z

Команды чтения выглядят аналогично:

LD Rx, -Z //чтение из ячейки с адресом Z-1 значения из регистра x,

//после выполнения команды Z-1

LD RX, Z+ //чтение из ячейки с адресом Z значения из регистра x,

//после выполнения команды Z

Используя простейший цикл можно, например, записать в выделенную область памяти массив чисел от 0 до 16:

//Директивы препроцессора

#include "m8def.inc"

//Определение переменных (директива .def)

.def TMP = R16 //Переменная для хранения временных значений

//Определение констант (директива .equ)

.equ MAX\_VALUE=16 //Количество чисел которые мы ходим записать

//Сегмент данных (SRAM)

.dseg

SAMPLE: .BYTE MAX\_VALUE //Выделим в SRAM MAX\_VALUE байт по адресу SAMPLE

.cseg

//Сегмент кода (Flash)

RJMP RESET

RESET:

LDI ZH, HIGH (SAMPLE) //Установим старший разряд регистра-указателя

LDI ZL, LOW (SAMPLE) //Установим младший разряд регистра-указателя

//Таким образом регистровая пара Z указывает на первую ячейку выделенной

//памяти SAMLE

CLR TMP //Отчистим регистр TMP (а вдруг не 0)

LOOP: //Начало цикла записи

ST Z+, TMP //Запишем значение в память и увеличим указатель на 1

INC TMP //Увеличим число во временном регистре на 1

CPI TMP,MAX\_VALUE //Проверим условие, дошли до максимального числа?

BRNE LOOP //Если нет, то повторяем, если да выходим из цикла

MAIN: //Бесконечный цикл

RJMP MAIN

*Листинг 4.4.* Запись блока данных в SRAM

Рассмотрим подробнее код программы и листинга 4.4, сразу необходимо обратить внимание в этой программе определяются переменные и константы директивами .DEF и .EQU, это удобно и делает код более наглядным, далее, так же как и в предыдущем примере выделяется определенный объем памяти, и по вектору RESET происходит переход к выполнению непосредственно программы, в которой устанавливается указатель на нужный сегмент памяти, обнуляется регистр (CLR) и начинается цикл записи с постинкрементом. В цикле сначала записывается новое значение в память (для первой итерации 0, для второй 1 и так далее), затем значение регистра, хранящего в себе число, которое копируется в память увеличивается на 1 (INC) и происходит проверка условия достижения границы выделенной области памяти, если достигли, выходим из цикла, если нет, переходим по метке LOOP и продолжаем выполнение цикла. При отладке этой программы можно увидеть, как изменяются значения регистра-указателя и как в цикле память наполняется значениями.

Кроме сохранения каких-либо значений в SRAM архитектура МК AVR позволяет создавать массивы констант в памяти программ, это может быть полезно для вывода каких-либо значений, которые изначально известны и не могут изменяться в процессе выполнения программы, например, масок цифр семисегментных дисплеев. Для чтения данных из памяти программ используется команда LPM, которая считывает указанный байт из памяти программ и помещает результат в регистр R0. Рассмотрим пример использование команды:

//Директивы препроцессора

#include "m8def.inc"

//Определение переменных (директива .def)

.def TMP = R16 //Переменная для хранения временных значений

//Определение констант (директива .equ)

//Сегмент данных (SRAM)

.dseg

.cseg

//Сегмент кода (Flash)

RJMP RESET

RESET:

LDI ZH, HIGH (N\_MASK\*2) //Установим старший разряд регистра-указателя

LDI ZL, LOW (N\_MASK\*2) //Установим младший разряд регистра-указателя

//Для адресации к памяти программ может быть использована только регистровая пара Z

//Так как Flash память имеет двухбайтовую организацию, необходим множитель 2

LDI TMP,3

ADD ZL,TMP //Установим указатель на 3 элемент массива N\_MASK

LPM //Выгрузим значение в регистр R0

MAIN: //Бесконечный цикл

RJMP MAIN

N\_MASK: //Массив значений

.db 1, 3, 5, 7, 9, 11

*Листинг 4.5.* Чтение из памяти программ

Основное отличие при чтении из памяти программ заключается в том, что только регистровая пара Z может выступать указателем и то что организация памяти программ страничная (двухбайтовая) вследствие этого при инициализации указателя появляется умножение на 2.

## 3. Выполнение типовых процедур на ассемблере

Рассмотрим реализацию таких типичных для языков высокого уровня конструкций как условная конструкция if…then…else, оператор выбора case, цикл while. При рассмотрении конструкций, все полезные операции заменены на команды NOP.

Начнем рассмотрение с конструкции if…then…else. Для задания условий в блоке if используются команды сравнения (CP,CPI) и команды условного перехода (BR\*\*). Важно понимать, что команды сравнения фактически вычисляют разность двух значений и выставляют соответствующие результату флаги в регистр SREG.

Для примера рассмотрим реализацию условия в виде строго неравенства A>B:

|  |  |
| --- | --- |
| //IF...THEN...ELSE (строгое неравенство)  CP R16, R17 //Сравним два значения A(R16) и B(R17)  //Значения в регистрах равны, переход на ELSE  BREQ ELSE\_ACTION  //Если R16>R17, то флаг C регистра SREG не установится  BRCS ELSE\_ACTION  IF\_ACTION: //if  NOP  NOP  RJMP NEXT\_ACTION  ELSE\_ACTION: //else  NOP  NOP  NEXT\_ACTION:  NOP  NOP | if(A>B)  {  IF\_ACTION  }  else  {  ELSE\_ACTION  }  NEXT\_ACTION |

*Листинг 4.6.* Реализация конструкции if(A>B)…else на языке ассемблера

Рассмотрим задание условий подробнее. Так как у нас задано строгое условие, то одной командой условного перехода обойтись не удастся (в отличии от условия «больше либо равно»). Однозначно сказать что A>B, можно только в случае соблюдения двух условий: их разность не равна 0 (флаг нуля Z регистра SREG не установлен), и в случае если разность двух значений больше 0 (то есть флаг переноса C регистра SREG не установлен). За первое условие отвечает команда BREQ (перейти, если равно), за второе BRCS (перейти, если флаг C регистра SREG установлен), в случае выполнения одного из условий происходит переход по метке ELSE\_ACTION, в ином случае выполняется условие по метке IF\_ACTION.

Таким образом комбинируя различные условия, можно реализовать проверку на условие практически любой сложности.

Теперь рассмотрим простой оператор выбора switch-case:

|  |  |
| --- | --- |
| //Оператор выбора SWITCH-CASE  CPI R16, CONST1 //Выполнено первое условие?  BREQ CASE1 //Переходим по метке  CPI R16, CONST2 //Второе?  BREQ CASE2 //Переходим по метке  DEFAULT\_CASE: //Ни одно условие не выполнено  NOP  NOP  RJMP NEXT\_ACTION  CASE1: //Условие 1  NOP  NOP  RJMP NEXT\_ACTION  CASE2: //Условие 2  NOP  NOP  RJMP NEXT\_ACTION  NEXT\_ACTION:  NOP  NOP | switch (TMP)  {  case CONST1:  action CASE1  break;  case CONST2:  action CASE2  break;  default:  action DEFAULT\_CASE  break;  }  NEXT\_ACTION |

*Листинг 4.7.* Реализация конструкции switch-case на языке ассемблера

Рассмотрим листинг 4.7: определение к какому блоку case перейти выполняется с помощью сравнения (CPI) переменной, хранящейся в каком-либо из регистров и константы. В случае равенства этих значений и происходит переход по необходимой метке. Если ни одно из условий не верно, то выполняется условие по умолчанию DEFAULT\_CASE, и выполнение конструкции завершается переходом к метке NEXT\_ACTION.

Теперь посмотрим, как реализовать цикл с предусловием типа while, использование данного типа цикла пригодится в дальнейшем, например, для реализации программной задержки:

|  |  |
| --- | --- |
| //Цикл с предусловием WHILE  CLR CNT //Отчищаем значение счетчика  WHILE:  INC CNT //Увеличиваем счетчик на 1  //Досчитали до необходимого значения?  CPI CNT, MAX\_VAL  BREQ NEXT\_ACTION //Если да, переход по метке  WHILE\_ACTION: //Нет? Выполняем к/л операции  NOP  RJMP WHILE //И возвращаемся в начало цикла  NEXT\_ACTION:  NOP | while(CNT<MAX\_VAL)  {  WHILE\_ACTION;  }  NEXT\_ACTION |

*Листинг 4.8.* Реализация цикла с предусловием while на языке ассемблера

Сначала для правильной работы цикла обнуляется счетчик CNT, а затем по метке WHILE: начинается выполнение цикла, увеличивается на 1 значение счетчика и проверяется условие выхода из цикла (значение в счетчики CNT равно константе MAX\_VAL), если условие выполняется, то происходит переход по метке NEXT\_ACTION и выполнение цикла завершается, в ином случае выполняются операции из тела цикла по метке WHILE\_ACTION: и происходит возврат в начало цикла (метка WHILE:).

## 4. Битовые операции на ассемблере

Битовые операции являются одними из самых распространённых операций при программировании МК, и большая часть таких операций идет через битовые маски. Ранее уже была рассмотрена операция сдвига влево (<<), операция побитовое «ИЛИ» (|), а также конструкция:

LDI R16, (1<<CS00)|(1<<CS01)

OUT TCCR0, R16

позволяющая установить определенные биты в регистре. Основной недостаток такой операции, то что так в регистр установятся только указанные биты, все остальные биты регистра будут обнулены. Разберем пример установки и сброса битов регистра без изменения состояния регистра. Допустим изначально необходимо на базе таймера (TIM1) настроить ШИМ на выход COM1A, а затем последовательно (не одновременно) подключить выход COM1B, и отключить выход COM1A.

С первоначальной установкой битов (листинг 4.9а) в пустой регистр никаких проблем нет, вышеуказанным способом используем операцию сдвига и побитовый «ИЛИ».

//ЧАСТЬ А

//Настройка таймера TIM1 для работы в режиме ШИМ

//Настроим ШИМ на сброс при обнулении, выход COM1A

LDI TMP, (1<<COM1A1)|(1<<COM1A0)|(1<<WGM10)

OUT TCCR1A, TMP

LDI TMP, (1<<WGM12)

OUT TCCR1B, TMP

*Листинг 4.9а.* Установка и сброс битов в регистре

Чтобы подключить выход COM1B и не обнулить ранее установленные биты регистра (листинг 4.9б) сначала нужно прочитать старое значение из регистра, а затем с помощью команды ORI (логическое «ИЛИ» с константой, наложить на старое значение новую маску.

//ЧАСТЬ Б

//Необходимо дополнительно подключить выход COM1B

//При этом в регистре настройки TCCR1A уже установлены

//Другие настроечные биты

IN TMP, TCCR1A //Читаем значение из регистра

ORI TMP, (1<<COM1B1)|(1<<COM1B0)//Накладываем на считанное значение маску (лог. ИЛИ)

OUT TCCR1A, TMP //Выводим новое значение в регистр

*Листинг 4.9б.* Установка и сброс битов в регистре

Сбросить биты и отключить выход COM1A можно так же используя битовую маску, но уже с операцией логического «И». Для удобства чтения кода (листинг 4.9в), в конструкции сброса используется установка битов в 1, а маска инвертируется операцией побитого «НЕ» (~).

//ЧАСТЬ В

//Необходимо дополнительно отключить выход COM1A

//При этом в регистре настройки TCCR1A уже установлены

//Другие настроечные биты

IN TMP, TCCR1A //Читаем значение из регистра

ANDI TMP, ~(1<<COM1A1|1<<COM1A0)//Накладываем на считанное значение инвертированную маску (лог. И)

OUT TCCR1A, TMP //Выводим новое значение в регистр

*Листинг 4.9в.* Установка и сброс битов в регистре

В дополнение к операциям сбросов и установки бит, достаточно часто используются операции сдвигов, такие операции бывают двух типов без переноса (LSR и LSL) и через перенос (ROL и ROR), отличие этих операций в том, что при сдвиге без переноса в зависимости от направления байт будет заполняться нулями, а при сдвиге с переносом бит уходящий за край будет попадать во флаг C регистра SREG и выходить с другого конца байта. Одно использование команды дает один шаг влево или вправо. Операции сдвига часто используется если необходимо разделить или умножить какое-либо число, на число, являющееся степенью двойки. Разберем на примере: разделим 2496 на 8.

//Директивы препроцессора

#include "m8def.inc"

//Определение переменных (директива .def)

.def CNT = R17 //Счетчик

.def DATA\_H = R19 //Старший разряд числа

.def DATA\_L = R20 //Младший разряд числа

//Определение констант (директива .equ)

.equ DIV\_NUM = 3 //Степень числа 2, чтобы получить 8

MAIN:

LDI DATA\_H,0x09 //Загрузим значение 2496 (0x09C0)

LDI DATA\_L,0xC0

CLR CNT //Отчистим регистр счетчик

DIV8: //Операция деления на 8

LSR DATA\_H //Сдвинули старший разряд вправо

ROR DATA\_L //Сдвинули младший разряд вправо с переносом

INC CNT //Увеличили счетчик

CPI CNT,DIV\_NUM //Проверили сместили значение необходимое число раз

BRNE DIV8 //Нет? Повторяем.

RJMP MAIN

*Листинг 4.10.* Деление чисел с помощью команд сдвига

После определения переменных и констант, загрузим значение, которое необходимо разделить в регистры, так как значение выход за 8-битный предел, для его хранения используется два регистра (DATA\_H и DATA\_L), в которых соответственно сохраняются старший и младший разряды числа. Затем следует сама операция деления на 8, так как 8 является результатом возведения 2 в 3 степень, то для деления необходимо три итерации сдвига, что и реализовано в функции DIV8, сначала старший байт (DATA\_H) сдвигает вправо без переноса, затем младший (DATA\_L) так же сдвигается вправо, но уже с переносом, после происходит проверка количества сдвигов, если три сдвига проведено, программа начинает выполняться заново, иначе запускается новая итерация сдвига.

Умножение выполняется аналогичным образом, только разряды смещаются не вправо, а влево.

**Задания для самостоятельной работы**

# Тема №5. Порты ввода/вывода

*Вопросы, рассматриваемые в данной теме*:

1. Общие сведения о портах ввода/вывода

2. Режимы работы портов ввода/вывода

3. Подключение к портам ввода/вывода внешних устройств и их программирование.

3.1. Подключение светодиода.

3.2 Подключение кнопки.

3.3 Подключение семисегментного индикатора.

3.4 Подключение матричной клавиатуры.

4. Внешние прерывания

**Краткие теоретические сведения**

## 1. Общие сведения о портах ввода/вывода

Для взаимодействия с «внешним миром» в каждом МК предусмотрен набор портов ввода/вывода, в зависимости от модели МК может различаться как количество портов, так и их разрядность (номинально порты 8-разрядные). Обозначаются порты ввода/вывода латинскими заглавными буквами A, B, C и так далее, причем не обязательно по порядку, то есть в МК может присутствовать только порт B и D.

Так же для сокращения общих физических выводов МК порты ввода/вывода часто сочетают в себе и дополнительные функции. В документации на распиновке МК дополнительная функциональность вывода указывается в скобках. На этот момент стоит особо обратить внимание при проектировании микроэлектронного устройства и при возможности стараться не задействовать одну линию МК для обоих типов функций. Все линии портов автономны друг от друга и могут управляться как группами, так и независимо. При включении МК порты по умолчанию работают на вход в состоянии с высоким импедансом (Hi-Z), дополнительные функции выводов отключены.

Рассмотрим упрощенное устройство линии порта ввода/вывода:



*Рис. 5.1.* Упрощенное устройство линии порта ввода/вывода

Диоды VD1 и VD2 на входе МК защищают линии МК от перенапряжения, например, если поданное на ножку МК напряжение будет выше напряжения питания, то откроется диод VD1 и напряжение будет подано на шину питания, в случае если на ввод попадет отрицательное напряжение, то оно через диод VD2 будет стравлено на землю. Но как показывает практика диоды стоят маломощные и помогают только от помех.

Далее из рисунка 5.1 видно, что в каждой линии порта есть три ключа управления (полевых транзистора) каждый со своим условием переключения, в случае выполнения условия ключ замыкается. Как можно легко понять из рисунка за состояние порта отвечают три регистра DDRx, PORTx, PINx. Рассмотрим каждый из этих регистров, а так возможные комбинации значений в этих регистрах:

DDRx – регистр направления порта, очевидно что в каждый момент времени, порт может работать либо в режиме входа, либо в режиме выхода. При DDRx=0 вывод работает как вход, при 1 как выход.

PORTx – регистр управляющих состоянием вывода, его функциональность зависит от того в какое значение выставлен регистр DDRx, в случае если DDRx=1 (линия порта настроена на выход), то значение в регистре PORTx определяет состояние вывода, при значении PORTx=1 на выводе будет логическая 1, в случае PORTx=0 на выходи соответственно будет логический 0.

Если DDRx=0 (линия порта настроена на вход), то возможны два режима работы вывода при PORTx=1 это режим с подтяжкой вывода к питанию (PullUp), при PORTx=0 то вывод находится в высокоимпедансном состоянии (Hi-Z), подробнее эти два режима работы будут рассмотрены ниже.

PINx – регистр чтения состояния порта, доступен только для чтения. В данном регистре содержится информация о текущем логическом уровне на выводах порта. То есть для того чтобы узнать какой сигнал в текущий момент на входе линии МК необходимо прочесть соответствующий разряд регистра PINx.

Если внимательно посмотреть на логические условия к ключам выводов МК, то можно заметить наличие еще одного регистра PUD (PullUp Disable), данный регистр запрещает подтяжку для всех портов, по умолчанию установлен в 0.

## 2. Режимы работы портов ввода/вывода

Режим выхода (DDRx=1) используется для управления внешними устройствами, если необходимо выдать на выход логическую 1, то в соответственный бит регистр PORTx записываем 1, если 0, то соответственно записываем 0.

Вход Hi-Z ((DDRx=0, PORTx=0), режим по умолчанию, все ключи разомкнуты, при этом сопротивления порта очень велико и его принято считать бесконечным, физически линия никуда не подключена и не может ни на что влиять. Кроме того, что это самый безопасный режим для работы МК он так же может использоваться в следующих случаях:

1. *Прослушивание шины данных.* Так как в этом режиме работы МК не оказывает никакого влияние на внешние устройства, но при этом постоянно опрашивает себя и записывает состояние в регистр PINx.
2. *Генератор случайных чисел.* Если вход оставить висеть в воздухе, то напряжение будет изменяться на нем в зависимости от внешних наводок, что позволяет сгенерировать более-менее случайную последовательность чисел. В бытовых условиях основная наводка — это наводка от сети 220В, в следствии этого в регистре PIN будет меняться 0 и 1 с частотой около 50 Гц.

Вход PullUp (DDRx=0, PORTx=1), вход с подтяжкой к шине питания через резистор 100кОм, таким образом линия, висящая в воздухе, переходит в состоянии логической единицы. Данный режим работы позволяет не допустить случайного изменения значений на линиях портов МК при его работе. В случае же появления на входе логического нуля (линия замкнута на землю каким-то внешним воздействием, например, нажатие кнопки) резистор не сможет удержать подтяжку к шине питания и на входе появиться логический 0. Так же с целью снижения энергопотребления в этот режим рекомендуется устанавливать все неиспользуемые порты ввода/вывода, так как в таком режиме работы от каждой помехи МК не придется переключать значение в регистре PINx.

## 3. Подключение к портам ввода/вывода внешних устройств

### *3.1 Подключение кнопки и светодиода.*

Начнем рассмотрение подключения внешних оборудования к портам ввода/вывода МК с простейших устройств таких как кнопки и светодиоды.

Самый простой способ (рис. 5.2а) подключения кнопки (SB1) — это подключить кнопку напрямую к порту МК и при инициализации настроить линии порта МК на вход с подтяжкой (PullUp). Недостаток такого способа в том, что встроенный подтягивающий резистор достаточно «слаб», а значит помехоустойчивость системы значительно снизится. Второй способ (рис. 5.2б) – подключение параллельно внутреннему внешнего подтягивающего резистора (R1) номиналом 1-10кОм, что значительно улучшит помехоустойчивость системы.



*Рис. 5.2.* Схемы подключение кнопки к МК

Основной момент, на который надо обратить внимание при подключении светодиодов это необходимость использования ограничительного резистора. Для расчета номинала ограничительного резистора (Rог) необходимо знать две характеристики:

1. Максимальный прямой ток светодиода и максимальный ток которым можно нагрузить линии порта МК.
2. Падение напряжение на светодиоде.

Обе характеристики можно посмотреть в документации на светодиод и МК. Для примера рассчитаем номинал ограничивающего резистора для красного светодиода с падением напряжения 1.8В и током около 20мА (данный ток для линии МК допустим). Очевидно, что при если на выводах МК напряжение будет равняться 5В, то на резисторе должно падать напряжение порядка 3.2В. Таким образом найдем сопротивление ограничительного резистора по закону Ома:

*,*

В стандартном значении сопротивлений такое значение имеется, но для большей надежности рекомендуется взять следующее значение из ряда, яркость светодиода практически не уменьшиться. Если же значение в стандартном ряду отсутствует, то необходимо всегда выбирать резистор с сопротивлением больше чем рассчитанное. Схема подключения светодиода к линии порта МК представлена на рисунке:



*Рис. 5.3.* Схема подключения светодиода к МК

Рассмотрим пример программы, управляющей линейкой из 8 светодиодов с помощью двух кнопок UP и DOWN, при первом нажатии кнопки UP появляется бегущая дорожка из зажжённых светодиодов вверх, при повторном нажатии дорожка гаснет, кнопка DOWN работает аналогично, за исключением того то что линейка светодиодов зажигается в противоположном направлении. При разборе исходного кода программы опустим раннее рассмотренные конструкции, такие как определение переменных, таблицы векторов прерываний, инициализацию стека и тому подобное. Рассмотрим общую логику программы, инициализацию портов ввода/вывода и решение таких проблем как дребезг контактов и необходимость задержки при переключении на новую группу светодиодов.

Для удобства и более легкого восприятия кода (листинг 5.1) сначала определим переменные и константы, которые будут использоваться в дальнейшем.

//Определение переменных (директива .def)

.def TMP = R16 //Временная переменная

.def CNT = R17 //Счетчик

.def CNT2 = R18 //Дополнительный счетчик

.def SB\_FLAG = R19 //Флаг нажатия кнопки

//Определение констант (директива .equ)

.equ LED\_MSK = 0b00010001

.equ FST\_PUSH = 1

*Листинг 5.1.* Инициализация констант и переменных

Константа LED\_MSK отвечает за движение горящих светодиодов по линейки, и изменяя ее можно добиться разной конфигурации движения. Определение первый ли раз нажата кнопка происходит благодаря константе FST\_PUSH использование которой в программе далее будет рассмотрено подробнее.

После определения переменных, векторов прерываний, и инициализации стека, необходимо настроить порты ввода/вывода на необходимый режим работы (листинг 5.2), а так как напрямую в порт ввода/вывода ничего записать нельзя, то для этого используется дополнительный регистр, в нашем случает это регистр R16 которому присвоено имя TMP. Команда LDI загружает константу во временный регистр TMP, а затем с помощью команды OUT, значение из регистра TMP загружается в один из управляющих регистров порта ввода/вывода, в случае линейки светодиодов, порт B полностью настраивается на выход (в регистр DDRB записывается значение 0xFF), для кнопок, так как порт по умолчанию настроен на вход, к необходимым линиям порта (0,1) просто подключаются встроенные резисторы подтяжки.

//Настройка портов ввода/вывода

//DDRx - направление работы линии порта x (1-выход, 0-вход)

//PORTx - Значение уровня на линии порта x (1-высокий, 0-низкий)

// если порт x настроен как вход (1-PullUp)

//PINx - Уровень сигнала на линии порта x (Только для чтения)

//Подключение линейки светодиодов (PORTB) - все линии на выход

LDI TMP, 0xFF

OUT DDRB, TMP

//Подключение кнопок (PORTD)-линии 0,1 - вход с подтяжкой

LDI TMP, 0b00000011

OUT PORTD, TMP

*Листинг 5.2.* Инициализация портов ввода/вывода

После инициализации необходимых устройств начинается выполнение основного бесконечного цикла (MAIN:):

MAIN:

//Проверяем нажата ли кнопка "ВВЕРХ"?

SBIS PIND,0

RJMP SB\_UP

//Нажата ли кнопка "ВНИЗ"?

SBIS PIND,1

RJMP SB\_DOWN

RJMP MAIN

*Листинг 5.3.* Основной цикл программы

В данном цикле с помощью команды проверки-пропуска (SBIS), определяется значение из необходимого регистра (в нашем случает это регистр состояния порта PIND и линии 0,1) и в случае если разряд РВВ установлен, то следующая команда пропускается (регистр адресов PC+2), и опрашивается состояние следующей кнопки. Может возникнуть вопрос, почему кнопка считается не нажатой если на линии порта высокий уровень? Ответ прост, ранее, при инициализации эти линии порта во избежание ложных срабатываний были подтянуты через встроенные резисторы к напряжению питания МК.

В случае же если одна из кнопок нажата пропуска команды не происходит. И по команде относительного перехода RJMP происходит переход к обработчику одной из кнопок.

Для примера рассмотрим обработчик кнопки ВВЕРХ (SB\_UP):

//Нажата кнопка вверх

SB\_UP:

SBIS PIND,0 //Кнопка опущена?

RJMP SB\_UP //Нет, ждем отпускания, иначе продолжаем выполнять программу

INC SB\_FLAG //Установим флаг нажатия кнопки

CPI SB\_FLAG,FST\_PUSH //Кнопка нажата первый раз?

BRNE CLR\_MASK //Нет? Надо остановиться и перейти к обнулению линейки

LDI TMP, LED\_MSK //Иначе запишем во временный регистр маску движения светоидов

UP\_CYCLE: //Цикл движения вверх (фактически вправо)

ROR TMP //Сдвигаем маску вправо

RCALL DELAYnS //Для наглядности вводим маленькую задержку (на частоте 1МГц достаточно)

OUT PORTB,TMP //Выводим маску в порт

//Проверяем нажимались ли кнопки, если да обрабатываем нажатие и останавливаем вывод

SBIS PIND,0

RJMP SB\_UP

SBIS PIND,1

RJMP SB\_DOWN

//Если нет, продолжаем вывод в цикле

RJMP UP\_CYCLE

*Листинг 5.4.* Обработка нажатия кнопки вверх

Для борьбы с дребезгом контактов существует несколько приемов, в данной программе используется прием (листинг 5.4), при котором нажатие кнопки начинает обрабатывается не сразу после нажатия на кнопку, а только после ее отпускания, то есть до момента отпускания кнопки происходит процедура аналогичная определению нажата ли кнопка. После отпускания кнопки собственно и начинается обработка события «кнопка нажата». Так как в условии задания указано что при повторном нажатии на кнопку устройство должно останавливать свою работу, введем в программу флаг нажатия кнопки SB\_FLAG (смотри инициализацию переменных и констант). С помощью команды инкрементирования (INC) значение флага увеличивается на единицу и далее сравнивается (CPI) с константой FST\_PUSH, в случае если значения равны (то есть кнопка нажата первый раз), команда условного перехода (BRNE) не срабатывает и продолжается дальнейшее выполнение программы. Во временный регистр TMP командой LDI записывается маска движения светодиодов. И запускается цикл вывода (UP\_CYCLE:) значений маски на порты ввода/вывода МК. Для того чтобы дорожка двигалась вверх воспользуемся командой логического сдвига вправо через перенос (ROR), а для того чтобы человеческий глаз успевал замечать движение дорожки введем подпрограмму небольшой программной задержки (RCALL DELAYnS), после задержки маска выводится в порт и светодиоды зажигаются (OUT PORTB,TMP). Так же в цикле вывода постоянно опрашиваются кнопки, и в случае нажатия любой из кнопок вывод на линейку светодиодов останавливается.

Теперь рассмотрим подпрограмму задержки (DELAYnS:) и функцию отчистки флага кнопки и отключения вывода значений на линейку:

//Отчистка флага кнопки и обнуление показаний на линейке

CLR\_MASK:

CLR SB\_FLAG //Устанавливаем флаг в 0

OUT PORTB,SB\_FLAG //Выводим на линейку 0

RJMP MAIN

//Задержка

DELAYnS:

LDI CNT,255 //Записываем в счетчик 255

LDI CNT2,255 //Записываем в дополнительный счетчик 255

DELAY\_CYCLE:

DEC CNT //Уменьшаем

BRNE DELAY\_CYCLE //Дошли до 0?

DELAY2\_CYCLE:

LDI CNT,255 //Загружаем в счетчик новое значение

DEC CNT2 //Уменьшаем на единицу дополнительный счетчик

BRNE DELAY\_CYCLE //Дошли до нуля? Если нет возвращаемся к первому циклу

RET //Выходим

*Листинг 5.5.* Функция отчистки и подпрограмма задержки

В функции отчистки (CLR\_MASK:) командой CLR обнуляется значения флага нажатия кнопки, а затем это же значение выводится в порт гася все светодиоды.

Программная задержка реализована в виде подпрограммы с двумя вложенными циклами внутри (DELAY\_CYCLE и DELAY\_CYCLE2). Что позволяет на частоте 1МГц реализовать задержку которая позволяет наблюдать движение зажигающихся светодиодов по линейке.

Сначала в подпрограмме задержки (листинг 5.5) оба счетчика CNT и CNT2 выставляются в максимальное значение (255), затем в первом цикле подпрограммы (DELAY\_CYCLE) уменьшаем значение счетчика CNT командой декремента (DEC) и проверяем командой условного перехода (BRNE) 0 ли в регистре счетчика, если да, то переходим к выполнению второго цикла (DELAY2\_CYCLE), в котором снова восстанавливаем значение в счетчике CNT (LDI CNT,255) и уменьшив значение второго счетчика на 1, и если значение счетчика CNT2 не равно 0 то возвращаемся к выполнению первого цикла. Если же равно, то значит подпрограмма задержки отработала и можно возвращаться (RET) к дальнейшему выполнению обработки кнопки. Таким образом на 8-разрядном МК был реализован 16-разрядный программный счетчик, который если рассчитать точное значение задержки можно использовать и для отсчета каких-то более точных промежутков времени. Основной недостаток (кроме точности отсчета) такого подхода это то что во время выполнения отсчета МК никакими другими полезными операциями фактически не занимается.

В следующем разделе рассмотрим подключение семисегментного дисплея и работы с константами, записанными в памяти программ (Flash-памяти).

### *3.2 Подключение семисегментного индикатора.*

Рассмотрим светодиодные (LED) семисегментные индикаторы, фактически они являются группой светодиодов, подключенных и расположенных в определенном порядке. Данные индикаторы могут быть как с общим анодом, в этом случае вывод плюса питания общий, а зажигаются сегменты коммутацией их к нулю питания, так и с общим катодом (общий отрицательный вывод). Так же существуют семисегментные индикаторы со встроенным дешифратором что существенно упрощает вывод на них цифр, но ничего кроме цифр вывести на них нельзя. Здесь мы рассмотрим классические семисегментные индикаторы. Изображение на индикаторах строиться с помощью зажигания определенных светодиодов, таким образом вывести на индикатор можно как цифры, так и большую часть алфавита. Обозначение выводов семисегментных индикаторов, и таблица формирования цифр можно посмотреть в документации, но зачастую они соответствуют представленным на рисунке 5.3:



*Рис. 5.3.* Обозначение выводов семисегментных индикаторов,

и таблица формирования цифр

Схема подключения индикаторов аналогична подключению обычных светодиодов. Но при этом при подключении LED-индикаторов, состоящих из большого числа светодиодов стоить учитывать ограничение суммарного тока через порты МК (порядка 140мА в зависимости от модели), то есть сколько угодно индикаторов напрямую к портам ввода/вывода МК не подключишь. В случае же если требуется подключить большее чем способен запитать МК количество устройств, часто используются транзисторные ключи.

Рассмотрим пример управление семисегментным индикатором решив простейшую задачу: при включении устройства и нажатии на кнопку на дисплей выводятся последовательно числа от 0 до 9, при следующем нажатии снова выводится ноль и так далее. Самым очевидном способом решения этой задачи является вывод из регистра в порт маски необходимой в данный момент цифры. Основной недостаток такого способа в том, что такую операцию тяжело обрабатывать в цикле, плюс постоянно будет присутствовать операции чтения-записи. Более оптимальным способом является размещение массива масок чисел в памяти, а так как маски чисел константы лучше всего будет разместить их во flash-памяти МК. Кроме уменьшения количества операций чтения-записи при выполнении программы размещение массива констант во flash-памяти позволяет обращаться к данным как к обычному массиву.

При рассмотрении кода опустим подробное описание инициализации портов ввода/вывода, обработку нажатия кнопки и другие процедуры, рассмотренные ранее. Начнем с инициализации констант и переменных, которые будут использоваться далее.

//Определение переменных (директива .def)

.def TMP = R16 //Временная переменная

.def L\_Counter = R17 //Счетчик символов

//Определение констант (директива .equ)

.equ END\_STRING = 10

*Листинг 5.6.* Инициализация констант и переменных

Константа END\_STRING необходима для того чтобы определить, что на дисплее выведено максимальное значение и при следующем нажатии необходимо сбросить значение на дисплее в 0.

Так как константы необходимо разместить во flash памяти, то директива их размещения (.db) размещается в сегменте кода (.CSEG) сразу за программным кодом перед сегментом EEPROM (.ESEG). Согласно таблице формирования цифр (рис. 5.3) создадим массив констант в памяти программ МК:

//Маска цифр для вывода на семисегментный дисплей (расположен во FLASH памяти)

N\_mask:

.db 0b00111111, 0b00000110, 0b01011011, 0b01001111, 0b01100110, 0b01101101, 0b01111101, 0b00000111, 0b01111111, 0b01101111

*Листинг 5.7.* Массив констант в памяти программ МК

Для обращения к конкретным элементам массива данных расположенных в памяти программ необходимо установить указатель на начало массива, а так как память программ имеет страничную организацию, для корректной адресации к данным младший и старший разряд регистра указателя умножаются на 2 (листинг 5.7). Так же стоит обратить внимание на то что данные в памяти размещаются последовательно друг за другом и что для адресации к памяти программ может быть использована только регистровая пара Z.

MAIN:

//Установим указатель на начало массива с символами

LDI ZH, HIGH (N\_MASK\*2) //Установим старший разряд регистра-указателя

LDI ZL, LOW (N\_MASK\*2) //Установим младший разряд регистра-указателя

//Для адресации к памяти программ может быть использована только регистровая пара Z

//Так как Flash память имеет двухбайтовую организацию, необходим множитель 2

*Листинг 5.8.* Установка указателя на начало массива данных.

Данная конструкция (листинг 5.8) располагается в начале главного бесконечного цикла MAIN: то есть указатель будет переинициализироваться каждый раз при его запуске цикла, зачем это сделано будет понятно далее.

Продолжим рассмотрения исходного кода программы (листинг 5.9), после обработки нажатия кнопки обязательно идет проверка, достигли ли мы максимального значения (константа END\_STRING), если достигли, что проверяется командой проверки-пропуска BRNE, то значение переменной L\_Counter обнуляется и запускается функция вывода символа на дисплей (STR\_OUT:). Рассмотрим эту функцию подробно. Сначала установим указатель на адрес в памяти по которому находится необходимое нам число. По умолчанию указатель стоит в начале массива где находится число 0. Таким образом если необходимо вывести на дисплей число 5, то к значению адреса расположенного в регистровой паре Z необходимо прибавить число 5. В программе же прибавляется (ADD) значение с именем L\_Counter. Далее командой обращения к памяти LPM происходит обращения к данным находящимся по адресу Z и данные автоматически помещаются в регистр R0. Далее полученное значение выводится на дисплей (OUT) и значение счетчика L\_COUNTER увеличивается на единицу (INC). Далее происходит переход к основному бесконечному циклу MAIN: указатель устанавливается на начало массива и устройство ожидает повторного нажатия кнопки.

//Нажата кнопка

SB\_PUSH:

SBIS PIND,0 //Кнопка опущена?

RJMP SB\_PUSH //Нет, ждем отпускания, иначе продолжаем выполнять программу

CPI L\_COUNTER, END\_STRING //Проверяем, дошли до конца строки?

BRNE STR\_OUT //Нет, выводи следующий символ

CLR L\_Counter //Да, обнуляем счетчик

STR\_OUT:

ADD ZL,L\_Counter //Сместим указатель на элемент массива символов на величину равную номеру требуемого к выводу символа

LPM //Достанем символ из адреса, на который указывает указатель

OUT PORTC,R0 //Выведем его на дисплей

INC L\_Counter //Увеличим номер выводимого символа на единицу

RJMP MAIN

*Листинг 5.9.* Функция чтения из памяти программ и вывода на дисплей.

Теперь должно быть понятно почему при переходе к основному бесконечному циклу указатель устанавливается на начало массива, ведь если этого не сделать, то при следующем смещении указателя на необходимое значение мы получим не требуемый адрес, а предыдущий плюс указанное при новой итерации смещение.

В заключении можно сказать что, хотя управление одним дисплеем не представляет особых сложностей, то при подключении достаточного большого количество светодиодных индикаторов схема подключения оказывается достаточно громоздкой, а при нехватке портов ввода/вывода приходится применять такие методы как динамическая индикация (будет рассмотрена далее). В таких случаях более рационально использование ЖК индикаторов.

## 4. Внешние прерывания

Часто для индикации какого-либо внешнего события могут быть использованы внешние прерывания. В каждом МК за внешние прерывания отвечают определенные выводы портов ввода/вывода называющиеся INTx, где x – номер по порядку. Так, например, у МК AtMega8 две линии внешних прерываний INT0 и INT1 привязаны к линиям PD2 и PD3 соответственно.

Для разрешения внешних прерываний существует специальный регистр настройки GICR, в котором установка бита INTx, разрешает прерывание х, а сброс запрещает.

Внешнее прерывание может происходит по одному из условий задаваемом битами ISC01 и ISC00 конфигурационного регистра MCUCR. Основные условия срабатывания прерывания сведены в таблицу 5.1:

Таблица 5.1. Условия генерации внешнего прерывания (INTx)

|  |  |  |
| --- | --- | --- |
| ISCx1 | ISCx0 | Условие генерации прерывания |
| 0 | 0 | По низкому уровню |
| 0 | 1 | По любому изменению логического уровня |
| 1 | 0 | По спадающему фронту |
| 1 | 1 | По нарастающему фронту |

А разница между уровнями сигнала представлена на рисунке 5.4.



*Рис. 5.4.* Разница между фронтом и уровнем сигнала

Перепишем программу из раздела 3.2 обрабатывая нажатие кнопки как внешнее прерывание. Опустим ранее рассмотренные фрагменты программы и разберем конфигурацию прерывания (листинг 5.10):

//Настройка внешнего прерывания INT0 (для обработки нажатия на кнопку)

LDI TMP, 1<<ISC01 //Прерывание по спадающему фронту

OUT MCUCR, TMP

LDI TMP, 1<<INT0 //Разрешение прерывания

OUT GIMSK,TMP

SEI //Не забываем вообще разрешить прерывания

*Листинг 5.10.* Настройка прерывания

В листинге 5.10 выполняется конфигурация прерывания INT0 по спадающему фронту (1<<ISC01), разрешение этого прерывания (1<<INT0), а также глобальное разрешение прерываний (SEI).

Вывод на дисплей выполняется аналогично примеру, рассмотренному в разделе 3.2, только в обработчике прерывания INT0.

**Задания для самостоятельной работы**

# Тема № 6. Программирование таймероВ

*Вопросы, рассматриваемые в данной теме*:

1. Общие сведения о таймерах и режимы их работы

2. Особенности программирования таймеров

3. Простейший секундомер

4. Таймеры в режиме ШИМ

**Краткие теоретические сведения**

## 1. Общие сведения о таймерах и режимы их работы

Любой таймер в МК является простым двоичным счетчиком, и в зависимости от разрядности может считать 0 до 255 в случае 8 –разрядного таймера и от 0 до 65535 в случае 16-разрядного таймера. Именно такой разрядности таймеры присутствуют в МК AVR. В зависимости от модели может присутствовать разное количество таймеров, но 8-разрядные таймеры всегда имеют четные номера (0,2), а 16-разрядные – нечетные (1,3). В чем же основные отличая от обычного программного таймера счетчика? Самое главное отличие — это возможность аппаратного таймера считать независимо от работы процессора и генерировать прерывания по различным событиям. Так же стоить отметить возможность работы таймеров в асинхронном режиме, в режиме подсчета импульсов и генерации различных ШИМ сигналов. В данной главе будут рассмотрены только основные режимы работы и конфигурационные регистры таймеров, так как полному описание режимов работы таймеров с примерами и особенностями использования можно посвятить отдельную книгу.

Рассмотрение таймеров начнем со способов их тактирования. Тактироваться таймер может как от внутреннего генератора, так и от счетного входа (выводы T0, T1), при тактировании от счетного входа, в зависимости от его настройки, может считаться либо задний, либо задний фронт импульсов поступающего на вход сигнала. Основное условие для работы таймера при тактировании от счетного входа это то что частота тактового сигнала не должна превышать частоту МК, иначе МК не будет успевать обрабатывать импульсы.

Так же один из таймеров может работать в асинхронном режиме, обычно это таймер 2, в этом случае таймер считает импульсы от отдельного кварцевого генератора подключаемого к выводам МК TOSC1 и TOSC2. Благодаря этому режиму можно легко организовать часы использовав для тактирования обычный часовой кварц на 32768Гц. Так же асинхронный таймер может продолжать работать в режимах глубокого энергосбережения МК и выводить из таких режимов МК по прерыванию.

Для удобства пользования таймерами в МК предусмотрен предделитель. Делить частоту можно на 8, 32, 64, 128, 256, 1024. Предделитель удобно использовать для отсчета достаточно больших промежутков времени, ведь если единственный источник тактовых сигналов — это тактовый генератор процессора, работающий на частоте 8МГц, то даже 16-разрядный таймер будет переполняться очень быстро, в случае же использования предделителя на 1024 работа с таймером станет гораздо удобнее. З настройки предделителя обычно отвечают биты CSx2..CSx0 регистра TCCRx(в некоторых таймерах регистр TCCRx фактически 16 разрядный и представляет собой два регистра TCCRxA и TCCRxB и в целом является основным регистром настройки таймера) где х это номер таймера. В зависимости от типа таймера и модели МК настройка предделителя может отличаться и ее необходимо уточнить в документации на МК. Режимы работы предделителя для таймеров МК AtMega8 представлены в таблице:

Таблица 6.1. Режимы работы предделителя таймеров МК

|  |  |
| --- | --- |
| CS02…CS00 | Режим работы |
| 000 | Таймер остановлен |
| 001 | Предделитель 1 (выключен), таймер считает тактовые импульсы |
| 010 | Предделитель 8 |
| 011 | Предделитель 64 |
| 100 | Предделитель 256 |
| 101 | Предделитель 1024 |
| 110 | Импульсы с вывода T0, при переходе с 1 на 0 |
| 111 | Импульсы с вывода T0, при переходе с 0 на 1 |

Все результаты отсчетов таймеров помещаются в счетный регистр TCNTx в случае 8-разрядного таймера, или TCNTxH и TCNTxL в случае 16-разрядного таймера где x – номер таймера. Особенности их чтения и записи рассмотрим позже.

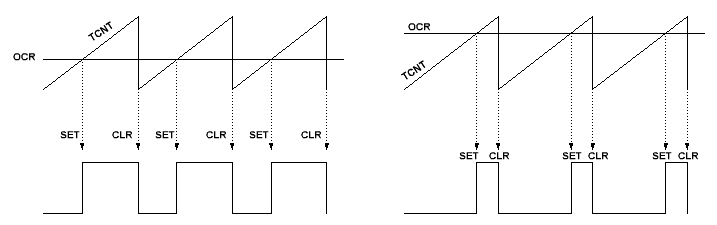
Так же каждый таймер может генерировать множество прерываний, за прерывания в таймерах отвечают регистры TIMSK и TIFR, регистр TIMSK, это регистр масок, то есть все его биты локально разрешают какие-либо прерывания от таймера (пока прерывание явно не разрешено, даже при глобальном разрешении прерываний, оно не сработает). Регистр TIFR это непосредственно флаговый регистр, при срабатывании прерывания в него устанавливается флаг о том, что условие прерывания выполнены, и когда программа уходит по вектору прерываний флаг сбрасывается. В случае же если прерывания запрещены глобально (CLI), то флаг так и будет стоять и прерывание сработает при их повторном разрешении (SEI), для того чтобы этого не произошло флаг необходимо сбросить вручную, для этого в нужный бит регистр TIFR необходимо записать 1. Все основные биты отвечающие за прерывания приведены в таблицы:

Таблица 6.2. Биты разрешения прерываний таймеров

|  |  |  |
| --- | --- | --- |
| Имя бита | Тип прерывания | Таймеры |
| TOVx | Переполнение таймера | T0,T1,T2 |
| ICFx | Захват входа | T1 |
| OCFxA | По совпадению А | T1 |
| OCFxB | По совпадению B | T1 |
| OCFx | По совпадению данных OCRx | T2 |

Кроме вышеперечисленных режимов таймеры могут работать в режиме ШИМ. Данный режим реализован на базе регистра сравнения таймера OCRxx, принцип его работы заключается в том, что когда значение в счетном регистре таймера TCNTx достигает значения находящегося в регистре сравнения то, возникает аппаратное событие: прерывание по совпадению или изменение состояния вывода OCxy, где х – номер таймера, y регистр сравнения (A или B).

Допустим, что мы настроили ШИМ так, что когда значение в счетном регистре (TCNTx) больше чем в регистре сравнения (OCRxy), то на выходе OCxy высокий уровень сигнала, иначе низкий. Таким образом таймер будет считать, как ему и положено, а при переполнении сбрасываться и продолжать отсчет заново, при этом на выходе OCxy будут появляться импульсы, скважность которых будет зависеть от значения в регистре сравнения (OCRxy). Временная диаграмма такого режима работы представлена на рисунке 6.1.



*Рис. 6.1.* Временная диаграмма работы ШИМ

Рассчитаем частоту ШИМ, для 8-разрядного таймера и МК работающего на максимальной для внутреннего тактового генератора частоте 8МГц:

Большую частоту ШИМ можно получить, увеличив частоту МК, то есть подключив внешний кварц или другой источник тактового сигнала более высокой частоты. Так же существует возможность повысить разрядность ШИМ (если это позволяет разрядность таймера), но это вместе с повышением дискретности выходного аналогового сигнала, резко снижается частота ШИМ.

Рассмотрим типы ШИМ аппаратно реализованные в МК:

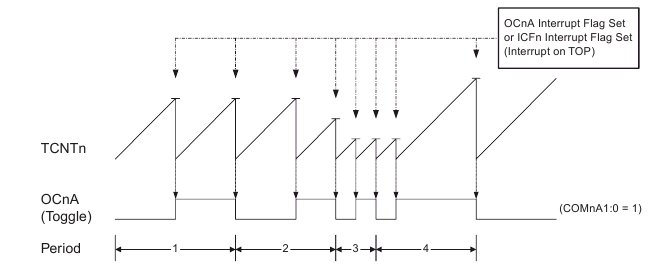
1. Fast PWM (рис.6.2а) – режим в котором счетчик считает от 0 до 255, после переполнения автоматически сбрасывается и продолжает счет. Когда значение в счетчике TCNTx становится равным значению лежащем в регистре сравнения OCRxy тогда соответствующий ему вывод OCxy сбрасывается в 0, при обнулении счетчика на выводе устанавливается 1.

2. Phase correct PWM (рис.6.2б) - ШИМ с корректировкой фазы, отличается от предыдущего режима в способе отсчета. Данный тип ШИМ сначала считает от 0 до 255, а затем в обратном направлении от 255 до 0, выход OCxy при первом совпадении сбрасывается, при втором устанавливается. Но так как период возрастает, то частота ШИМ падает в два раза. При таком режиме ШИМ при изменении скважности не изменяется угол фазового сдвига между двумя ШИМ сигналами, то есть центры импульсов в разных каналах и разной скважности будут совпадать.



*Рис. 6.2.* Временные диаграммы режимов работы ШИМ

3. Clear timer on compare - CTC Mode (рис. 6.3) – сброс при сравнении, в этом режиме таймер работает не от 0 до максимального значения, а от 0 до значения записанного в регистре сравнения и после достижения этого значения сразу сбрасывается. Как результат на выходе образуются выходы одинаковой скважности, но разной частоты.



*Рис. 6.3.* Временная диаграмма работы ШИМ (режим CTC)

Данный режим часто используется, когда необходимо с помощью таймера отсчитывать периоды и генерировать прерывания.

Рассмотрим на примере таймера 1 (TIM1) основные биты TCCR1A и TCCR1B, используемые для настройки ШИМ:

За поведение выводов сравнения OC1A и OC1B отвечают биты COM1A1:COM1A0 и COM1B1:COM1B0 соответственно. Режимы работы выходов приведены в таблице:

Таблица 6.3. Режимы работы выходов OC1A и OC1B

|  |  |  |
| --- | --- | --- |
| COMxx1 | COMxx0 | Режим работы выхода |
| 0 | 0 | Вывод не задействован |
| 0 | 1 | Поведение зависит от режима, заданного в битах WGM, и различается для разных МК |
| 1 | 0 | Сброс при совпадении (прямой счет) |
| 1 | 1 | Сброс при обнулении (обратный счет) |

Биты WGM11, WGM10, WGM12, WGM13 задают режим работы генератора:

Таблица 6.4. Режим работы ШИМ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| WGM13 | WGM12  (CTC1) | WGM11  (PWM11) | WGM10  (PWM10) | Режим работы |
| 0 | 0 | 0 | 0 | Обычный режим работы |
| 0 | 0 | 0 | 1 | PWM, Phase Correct, 8-bit |
| 0 | 0 | 1 | 0 | PWM, Phase Correct, 9-bit |
| 0 | 1 | 0 | 0 | PWM, Phase Correct, 10-bit |
| 0 | 1 | 0 | 0 | CTC (сброс при сравнении) |
| 0 | 1 | 0 | 1 | Fast PWM, 8-bit |
| 0 | 1 | 1 | 0 | Fast PWM, 9-bit |
| 0 | 1 | 1 | 1 | Fast PWM, 10-bit |

В таблице 6.4 приведены только основные режимы работы ШИМ, полную таблицу можно найти в документации на конкретный МК

После настройки режима ШИМ, для его запуска необходимо только запустить таймер. Пример программы для работы с ШИМ будет рассмотрен далее.

## 2. Особенности программирования таймеров

Начнем рассмотрение особенностей программирования таймеров со счетного регистра TCNT. Для 8-разрядного таймера в случае если в этот регистр нужно записать число проблем не возникает, используется обычная команда вывода значения РОН в РВВ (OUT). В случае же использования 16-разрядного таймера счетный регистр состоит из двух регистров TCNTxH и TCNTxL. А так как таймер считает независимо от процессора, то в промежутке между чтением или записью «половинок» счетного регистра один из них может измениться, например, считаем содержимое младшего байта счетного регистра (TCNTxL) которое в текущий момент времени имеет значение 0xFF, затем считаем содержимое старшего байта (TCNTxH), в это время состояние таймера становится на единицу больше и младший байт обнуляется, в итоге вместо реального значения лежащего в регистрах (0x0100) мы прочтем неверное значение (0x01FF), что совершенно не одно и тоже. При записи числа в счетный регистр может происходить аналогичная ситуация. Поэтому во избежание такой ситуации в16-разрядных таймерах предусмотрен специальный механизм чтения и записи счетного регистра таймера. Рассмотрим его на примере записи в счетный регистр:

//Запись в 16-разрядный регистр TCNT

CLI //Для атомарности операции запретим прерывания

OUT TCNTxH, Rx //Первым пишется старший байт

OUT TCNTxL, Rx //Вторым младший

SEI //Операция завершена, прерывания можно разрешить

*Листинг 6.1.* Запись в 16-разрядный счетный регистр TCNT

Так как запись в счетный регистр по сути является операцией атомарной то срабатывание прерывание в процессе записи может привести к тому что после записи в регистре окажется фактически «мусор», поэтому прерывания запрещаются на глобальном уровне, далее согласно документации, сначала записывается значение в старший байт регистра TCNT, а затем в младший. Порядок записи менять нельзя. Процедура чтения аналогично, за исключение того что при чтении сначала читается младший байт, а затем старший:

//Чтение из 16-разрядного регистра TCNT

CLI //Для атомарности операции запретим прерывания

IN Rx, TCNTxL //Первым читается младший байт

IN Rx, TCNTxH //Вторым старший

SEI //Операция завершена, прерывания можно разрешить

*Листинг 6.2.* Чтение из 16-разрядного счетного регистра TCNT

При запуске таймеров несмотря на то что формально все РВВ устанавливаются в 0 при включении питания, фактически для получения точного первого отсчета рекомендуется регистрTCNT обнулить перед первым запуском в работу (не забывая о порядке записи в него).

Зачем писать значения в счетный регистр таймеров? Ответ прост, возможность записи позволяет начинать отсчёт не с нуля и регулировать интервалы счета.

Следующее что стоит учитывать при программировании таймеров это особенности предделителя. Особенность это заключается в том, что при запуске таймера с предделителем установленным, например, на 1024, то первый отсчет не обязательно придется через 1024 импульса. Это зависит от того в каком состоянии предделитель находился в предыдущий момент времени (то есть до какого значения он досчитал), предделитель один и работает независимо от того включен таймер или нет. Таким образом для точности отсчета предделитель нужно сбрасывать, но стоит учитывать, что предделитель один для всех таймеров и при сбросе его значения может сбиться выдержка до следующего отсчета у других работающих в настоящий момент времени таймеров. Например, может возникнуть ситуация, когда один таймер работает на выводе предделителя 1:64, а второй на 1:1024, и для точности работы первого таймера предделитель сбрасывается в цикле чаще чем один раз в 1024 такта, таким образом второй таймер работать не будет, хотя и должен согласно всем настройкам.

Для сброса предделителя необходимо записать бит PSRx регистра SFIOR. Бит PSRx будет сброшен автоматически на следующем такте МК:

//Очистка предделителя таймеров

LDI R16, (1<<PSR2) //Запишем значение в регистр R16

OUT SFIOR, R16 //Выведем значение в РВВ

*Листинг 6.3.* Сброс предделителя таймеров

## 3. Простейший секундомер

Рассмотри программу простейшего секундомера, который фактически умеет считать секунды от 0 до 9, результат выводится на семисегментный дисплей с дешифратором, а запуск или остановка производится нажатием на соответствующую кнопку. Как обычно в процессе разбора программного кода будет опущено подробное описание всех ранее рассмотренных моментов. Подробно будут рассмотрены собственно сама настройка таймера и использование прерываний.

Кратко ход программы можно описать так, по нажатию на кнопку запускается таймер, при его переполнении срабатывает прерывание, в обработчики прерывания значение секунд увеличивается на 1, и выводится на семисегментный дисплей, МК ожидает следующего срабатывания прерывания или нажатия на кнопку.

Для удобства программирования определим переменные и константы, которые будут использоваться в программе:

//Определение переменных (директива .def)

.def TMP = R16 //Временная переменная

.def CNT = R17 //Счетчик

.def NUM = R18 //Значение выводимое на дисплей

//Определение констант (директива .equ)

.equ MAX\_V = 245 //Значение регистра счетчика для таймера(до 1 секунды)

.equ MAX\_NUM = 10 //Значение для того чтобы вовремя обнулить значение на дисплее

*Листинг 6.4.* Определение констант и переменных

Рассмотрим используемые константы (листинг 6.4), первая MAX\_V, указывает сколько раз должен переполнится таймер для того чтобы прошла одна секунда, это значение рассчитано из соображений что при тактовой частоте МК 4МГц и коэффициенте предделителя 1:64, частота работы таймера будет равна 62500Гц, таким образом 8-разрядный таймер будет переполняться порядка 62500/256=244,1 раз в секунду.

Вторая константа MAX\_NUM используется для ограничения вывода символов на дисплей, то есть для перехода после 9 к 0.

Так как в программе будет использоваться прерывание, посмотрим, как это указывается в векторе прерываний:

.ORG OVF1addr ; Timer/Counter1 Overflow

RETI

.ORG OVF0addr ; Timer/Counter0 Overflow

RJMP TIM0OVF

.ORG SPIaddr ; Serial Transfer Complete

RETI

.ORG URXCaddr ; USART, Rx Complete

RETI

*Листинг 6.5.* Фрагмент вектора прерываний.

Как видно из листинга 6.5, для того чтобы задать обработчик прерывания, в векторе прерываний для необходимого прерывания команда RETI заменятся на команду перехода RJMP с указанием метки, имя метки может быть любым, в нашем случае метка носит говорящее название TIM0OVF. Сам обработчик прерывания рассмотрим позже.

После инициализации стека и настройки портов ввода/вывода, перейдем к настройке таймера. Используем 8-разрядный таймер (TIM0).

//Настройка таймера TIM0

LDI TMP, (1<<TOIE0) //Разрешим прерывания по переполнению таймера

//(по умолчанию запрещены)

OUT TIMSK, TMP

LDI CNT,MAX\_V //Запишем в регистр-счетчик значение для задержки в 1 сек

SEI //Разрешим прерывания

*Листинг 6.6.* Настройка таймера

Так как нам необходимо прерывание по переполнению таймера, а каждое прерывание должно быть разрешено явно, то установим бит разрешения прерывания по переполнению TOIE0 регистра TIMSK в 1. Затем загрузим в регистр CNT значение константы, отвечающей за количество переполнений таймера. И так как в программе будут использоваться прерывания глобально разрешим их командой SEI. Особо стоит обратить внимание на то, что таймер еще не запущен, так как настройки предделителя не выставлены (листинг 6.6).

Далее начинается выполнение основного бесконечного цикла программы, в которой ожидается нажатие одной из кнопок. При нажатии одной из кнопок (СТАРТ или СТОП) происходит переход к обработчику кнопки. Рассмотрим их:

//Нажата кнопка "Старт"

SB\_START:

SBIS PIND,0 //Кнопка опущена?

RJMP SB\_START //Нет, ждем отпускания, иначе продолжаем выполнять программу

//Таймеры запускаются выставлением значения в предделителе (биты CSxx регистра TCCR0)

LDI TMP, (1<<CS00)|(1<<CS01) // Предделитель (/64)

OUT TCCR0,TMP // Записываем значение в регистр

RJMP MAIN

*Листинг 6.7.* Обработчик кнопки СТАРТ

При обработке кнопки СТАРТ (листинг 6.7) Запускается таймер TIM0. Запуск производится установкой бит CSxx в регистр настройки таймера TCCRx, в нашем случае необходимо запустить таймер с предделителем 1:64, согласно таблице 6.1 для этого необходимо установить биты CS00 и CS01 в 1, сразу после установки этих бит в регистр TCCR0 таймер запускается и работает независимо от работы МК, обработка нажатия кнопки завершается, и программа переходит к выполнению бесконечного цикла, ожидая следующего нажатия кнопки или прерывания.

Обработка кнопки СТОП происходит аналогичным образом (листинг 6.8), за исключением того, что таймер выключается, остановка счета таймера производится обнулением битов регистра TCCR0 отвечающих за предделитель, то есть фактически проводится операция обратная запуску таймера.

SB\_STOP:

SBIS PIND,0 //Кнопка опущена?

RJMP SB\_STOP //Нет, ждем отпускания, иначе продолжаем выполнять программу

//Останавливаем таймер обнуляя биты предделителя

LDI TMP, (0<<CS00)|(0<<CS01) // Предделитель обнуляем

OUT TCCR0,TMP // Записываем значение в регистр

RJMP MAIN

*Листинг 6.8.* Обработчик кнопки СТОП

Каждый раз при переполнении таймера в регистре прерываний таймера (TIFR) выставляется флаг прерывания, и если прерывания разрешены, то обычный ход программы прерывается и начинается обработка прерывания. При срабатывании прерывания автоматически сохраняется в стек адрес возврата, а также глобально запрещаются все прерывания. При выходе из прерывания по команде RETI, глобальный запрет на прерывания автоматически сбрасывается. **Регистр SREG автоматически при возникновении прерывания не сохраняется.**

Рассмотрим обработчик прерывания подробно:

//Сегмент обработчиков прерываний

TIM0OVF:

DEC CNT //Уменьшаем счетчик

BREQ DELAY1S //Счетчик равен 0? (переполнение произошло 256 раз)

RETI //Нет, ждем

DELAY1S:

LDI CNT,MAX\_V //Запишем в регистр-счетчик новое значение для задержки в 1 сек

INC NUM //Увеличим число которое собираемся выводить

CPI NUM,MAX\_NUM //Дошли до предельного значения?

BREQ ZERO //Да, обнуляем

OUT PORTC,NUM //Нет, выводим на дисплей

RETI

ZERO:

CLR NUM //Обнуляем значение

OUT PORTC,NUM //Выводим на дисплей

RETI

*Листинг 6.9.* Обработчик прерывания по переполнению таймера.

Первым делом уменьшаем счетчик количества переполнений таймера, и проверяем равен ли он 0, в случае если нет, то выходим из прерывания по команде RETI и продолжаем выполнение программы до следующего прерывания. В ином случае (CNT=0), таймер переполнился необходимое количество раз (секунда отсчитана) и происходит переход к метке DELAY1S: в которой счетчик CNT устанавливается в значение MAX\_V, затем командой инкремента увеличивается на единицу значение числа, которое будет выводится на дисплей, а так же происходит проверка нового числа, на максимальное значение, если максимальное значение не достигнуто, выводит число на дисплей (OUT), если достигнуто, то перейдя по метке (ZERO:) значение числа обнуляется и выводится на дисплей. После вывода числа по команде RETI обработка прерывания из листинга 6.9 заканчивается и продолжается выполнение программы с адреса возврата, взятого из стека.

Как можно было обратить внимание в данной программе в порт напрямую выводится двоичное число, без какой-либо обработки. Это возможно так как предполагается что в устройстве используется семисегментный дисплей с дешифратором. Такой дисплей имеет 4 управляющих вывода, и автоматически преобразуется двоичный код в символы от 1 до F, для того чтобы получить на таком дисплее 0, нужно просто подать на него питание.

## 4. Таймеры в режиме ШИМ

Для примера настроим таймер (TIM1) для работы в режиме ШИМ, на два выхода OC1A и OC1B c длительностью импульсов 1/2 и 1/3 периода соответственно. Сначала разберем настройку таймера в режиме Fast PWM 8-bit, а затем в режиме Phase correct PWM 8-bit.

Как ранее было сказано главный настроечный регистр таймера это TCCR, установим режим работы ШИМ согласно таблицам 6.3 и 6.4. (листинг 6.10) Установим сброс при обнулении (обратный счет ШИМ) установив биты COM1A0, COM1A1, COM1B0, COM1B1. За выбор режима работы ШИМ отвечают биты WGM, для задания Fast PWM 8-bit установим биты WGM10, WGM12. А затем запустим таймер включив предделитель (1<<CS10). При записи битов в настроечный регистр стоит обратить внимания на то что в 16-разрядном таймере их два, TCCR1A и TCCR1B, какие биты в них есть можно посмотреть либо в документации на МК, либо в окне отладчика AVR Studio. Далее соблюдая порядок записи в 16-разрядные регистры сравнения (сначала старший, затем младший байт), запишем значения для необходимой скважности:

///Настройка таймера TIM1 для работы в режиме ШИМ (FAST PWM 8-bit)

LDI TMP, 2<<COM1A0|1<<WGM10|0<<WGM11|2<<COM1B0 //Настроим ШИМ на сброс при обнулении

OUT TCCR1A, TMP

LDI TMP, 1<<WGM12|1<<CS10|0<<WGM13

OUT TCCR1B, TMP

//Запишем необходимые значения в регистры сравнения (не забывая про порядок записи!)

LDI TMP,0

OUT OCR1AH,TMP

LDI TMP,128

OUT OCR1AL,TMP

LDI TMP,0

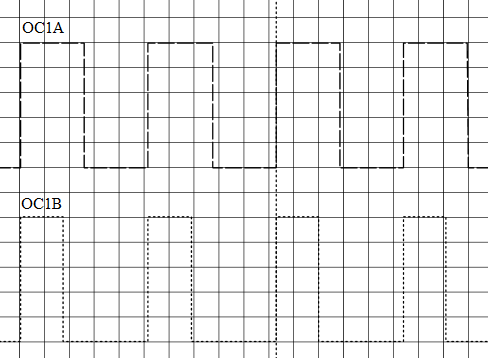
OUT OCR1BH,TMP

LDI TMP,85

OUT OCR1BL,TMP

*Листинг 6.10.* Настройка TIM1 на работу в режиме Fast PWM 8-bit

Прошив МК, осциллографом можно проверить работу ШИМ подключившись к ножкам OC1A и OC1B. (рисунок 67)



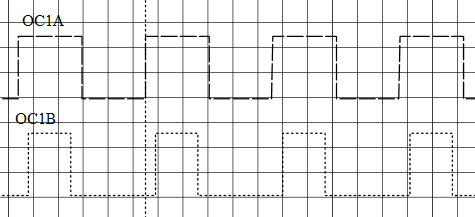
*Рис. 6.4.* Работа ШИМ в режиме Fast PWM 8-bit

Из рисунка 6.4 видно, что импульсы идут с заданной длительностью, а также происходит сдвиг фаз, который должен отсутствовать в режиме Phase correct PWM 8-bit. Изменим программу для работы в этом режиме сбросив бит WGM12 регистра TCCR1B:

LDI TMP, 0<<WGM12|1<<CS10|0<<WGM13

OUT TCCR1B, TMP

После перепрошивки МК можно будет увидеть следующую картинку:



*Рис. 6.5.* Работа ШИМ в режиме Phase correct PWM 8-bit

Из которой видно, что сдвига фаз не происходит. Такой режим работы часто используется для создания многофазных ШИМ сигналов, ведь так как сигналы с разных каналов и разной скважности будут совпадать по фазе, угол фазового сдвига между двумя сигналами не будет сбиваться.

Практическое применение ШИМ рассмотрим в главе 11 на примере управления сервоприводом.

**Задания для самостоятельной работы**

# Тема № 7. Передача и прием данных UART

*Вопросы, рассматриваемые в данной теме*:

1. Краткое описание протокола

2. Реализация UART в МК AVR

3. Программа приема-передачи данных через UART

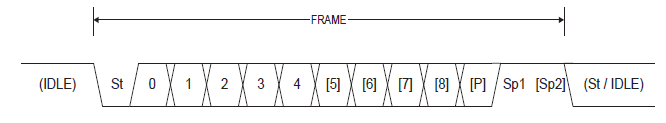
4. Сопряжение МК и ПК

**Краткие теоретические сведения**

## 1. Краткое описание протокола

Большинство современных МК содержат в себе модули асинхронного приёмопередатчика(UART) или асинхронно-синхронного (USART). Так как в работе эти два передатчика ничем кроме наименования нескольких регистров не отличаются, в дальнейшем чтобы избежать путаницы будем называть этот модуль UART.

Протокол работы передатчика и формат кадра достаточно прост (рис. 7.1):



*Рис.7.1.* Протокол работы и формат кадра UART

St – Старт-бит, всегда низкий уровень

(n) – Биты данных (с 0 по 8)

P – Бит четности

Sp – Стоп-бит, всегда высокий уровень

[] – настраиваемые параметры

IDLE – Ожидание на линии высокий уровень, прием/передача отсутствуют)

Изначально линия находиться в состоянии IDLE, то есть прием/передача отсутствует, и линии подтянута к высокому уровню. Для того чтобы начать прием/передачу, передатчик просаживает линии к низкому уровню, то есть отправляет в линии старт-бит (St), определив, что линии просела, приемник выдерживает интервал T1, а затем считывает первый и последующие биты с интервалом времени T2. Дальше работа протокола зависит от его настройки, для контроля передачи данных, после всех битов, данных может стоять бит четности (P), за которым обязательно следует один стоп бит(Sp1), либо для большей надежности два стоп-бита (Sp1 и Sp2). После стопового бита (битов), прием передача заканчивается, и начинаете или передача нового кадра со стартового бита (St) и протокол переходит в режим IDLE.

Естественно все настройки протокола должны быть произведены до начала приема/передачи. Самым популярным вариантом настройки является – 8 бит данных, 1 старт-бит, 1 стоп-бит, проверка четности отсутствует.

Подключение двух устройств по протоколу UART между собой (рис. 7.2) выполняется соединением передатчика одного устройство с приемником другого.



*Рис 7.2.* Подключение двух устройств по USART между собой

Кстати по такому же протоколу работает COM порт компьютера, вопрос сопряжения МК и ПК будет рассмотрен далее.

## 2. Реализация UART в МК AVR

UART в МК AVR реализован аппаратно, поэтому вручную следить за состоянием линии, форматом кадра, и управляющими битами не нужно. В целом постоянно следить за состоянием UART не нужно, так как все его обслуживание (прием и отправка данных) можно повесить на прерывания.

Для принятых и передаваемых данных существует регистр UDR (UART Data Register), физически это два разных регистра, (один для приема, а второй для передачи) имеющие один адрес. То есть для того чтобы передать байт, необходимо записать его в регистр UDR, а для того что бы принять, его необходимо считать из регистра.

О том, что байт поступил в регистр UDR можно определить по срабатыванию прерывания, которое вызывается каждый раз, когда приемник заканчивает получать новый байт. Так как передача идет довольно медленно, и для приема и передачи доступен только один общий регистр UDR, перед записью в него нужно убедиться, что он свободен и готов к приему нового байта, для этого существует бит UDRE, который кроме того, что говорит о готовности к записи в буфер, вызывает аппаратное прерывание по опустошению буфера.

Теперь рассмотрим настройку UART. Все основные параметры хранятся в регистрах конфигурации (UCSRA, UCSRB, UCSRC), а скорость работы приемопередатчика задается записью значений в регистровую пару UBBRH:UBBRL.

Подробнее рассмотрим основные биты регистров конфигурации:

1. *Регистр UCSRA:*

Биты RXC и TXC – флаги завершения приема или передачи соответственно, бит RXC устанавливается, когда принятый байт попадает в UDR, бит TXC устанавливается после отправки байта и опустошения UDR. Биты RCX и TXC.генерируют прерывание по завершению приема (Rx Complete) или передачи (Tx Complete).

Бит UDRE указывает на то что регистр UDR приемника пуст и в него можно записывать новое значение. Сбрасывается аппаратно, генерирует порывание UDR пуст (Data Register Empty).

Бит U2X – бит удвоения скорости при работе в асинхронном режиме.

Биты FE, OR, PE – биты ошибки кадрирования, переполнения буфера и контроля четности соответственно.

1. *Регистр UCSRB:*

Биты разрешения приема (RXEN) и передачи (TXEN).

Биты разрешения прерываний по завершению приема (RXCIE), передачи (TXCIE) и опустошению буфера передачи (UDRIE).

Бит UCSZ2 совместно с битами UCSZ0 и UCSZ1 находящимися в регистре UCSRC, задают формат передаваемых данных.

1. *Регистр UCSRC:*

Так как регистры UCSRC и UBRRH находятся в одном адресном пространстве, того чтобы записать в регистр UCSRC в старшем бите URSEL должна быть записана единица, в противном случае запись будет идти в регистр UBRRH.

Остальные биты задают различные параметры протокола, такие как число стоп-битов, контроль четности, формат посылки, при необходимости их можно посмотреть в документации на МК. Режим по умолчанию – 1 старт-бит, 1 стоп-бит, проверка четности отсутствует. Формат посылки равный 8 битам можно задать, записав в биты UCSZ0 и UCSZ1 единицу.

Скорость обмена и значение необходимое для записи в UBRR соответствующее режиму работы может быть вычислено по формулам из таблицы 7.1:

*Таблица 7.1.* Вычисление значений для необходимой скорости обмена

|  |  |  |
| --- | --- | --- |
| Режим работы | Формула расчета скорости передачи | Расчет значения регистра UBRR (для требуемой скорости) |
| Асинхронный режим работы (U2X=0) |  |  |
| Асинхронный режим работы с удвоенной скоростью (U2X=1) |  |  |
| Синхронный режим работы |  |  |

где, BAUD – скорость передачи (бит в секунду), fOSC – частота МК

UBRR – значение в регистрах UBRRH и UBRRL (0-4095)

## 3. Программа приема-передачи данных через UART

Рассмотрим программу, выводящую по протоколу UART в терминал запрос на ввод значения от 0 до 9, и при вводе значения выводящее его на индикатор. Для вывода строки символов в порт будет использоваться прерывание по наличию в регистре UDR данных для передачи (UD\_OK) и прерывание по окончанию приема (RX\_OK).

Рассмотрение программы начнем с используемых переменных и констант:

//Определение переменных (директива .def)

.def TMP = R16 //Временная переменная

//Определение констант (директива .equ)

.equ XTAL = 4000000 //Частота контроллера

//Расчет константы для требуемой частоты UART

.equ baudrate = 9600 //Необходимая частота UART

.equ baud\_const = XTAL/(16\*baudrate)-1 //Константа для частоты

//Сообщение для вывода на дисплей, хранится во Flash-памяти

ServiceMsg: .db "ENTER NUMBER (0-9):",0x0D, 0,0

*Листинг 7.1.* Инициализация констант и переменных

В листинге 5.5 инициализируется только одна переменная TMP, а из констант автоматически в зависимости от требуемых частот МК (XTAL) и передачи данных (baudrate) рассчитывается значение константы для записи в регистр UBRR. Строковый массив (ServiceMsg) располагается во flash-памяти (0x0D – переход на новую строку, 0 – конец массива).

Инициализируем UART (листинг 7.2), задав частоту приемопередатчика записав ранее рассчитанную константу baud\_const в регистр UBRR, разрешив прерывания по приему и передаче (но запретив прерывание по наличию данных в регистре UDR), и настроив параметры кадра (8 бит, без проверки четности):

//Инициализация UART:

LDI TMP, LOW(baud\_const) //Регистр частоты

OUT UBRRL,TMP

LDI TMP, HIGH(baud\_const) //Регистр частоты

OUT UBRRH,TMP

LDI TMP, (1<<RXEN)|(1<<TXEN)|(1<<RXCIE)|(1<<TXCIE)|(0<<UDRIE) //Прерывания разрешены, прием-передача разрешен

OUT UCSRB,TMP

LDI TMP, (1<<URSEL)|(1<<UCSZ0)|(1<<UCSZ1) //Формат кадра – 8 бит, без проверки четности

OUT UCSRC, TMP

*Листинг 7.2.* Инициализация UART

Для вывода строки в терминал (листинг 7.3) установим указатель на массив содержащий требуемую строку, и разрешим прерывание по наличию данных в регистре UDR:

MAIN:

LDI ZL, LOW (ServiceMsg\*2) //Установим указатель на область памяти с сообщением

LDI ZH, HIGH(ServiceMsg\*2)

H\_MSG:

//Разрешим прерывание при наличии данных для передачи

LDI TMP,(1<<RXEN)|(1<<TXEN)|(1<<RXCIE)|(1<<TXCIE)|(1<<UDRIE)

OUT UCSRB, TMP

BREQ STOP //Если данные для передачи закончились, останавливаем работу

RJMP H\_MSG //Иначе выводим следующий символ

STOP:

RJMP STOP

*Листинг 7.3.* Вывод строки в терминал

Вывод будет посимвольно продолжаться в прерывании (листинг 7.4) до окончания строки (символ 0 в массиве), и далее программа, перейдя по метке STOP уйдет в бесконечный цикл ожидая приема данных.

UD\_OK: //В регистре UDR есть данные для передачи

LPM TMP, Z+1

CPI TMP,0

BREQ STOP\_TX

OUT UDR, TMP

RETI

STOP\_TX:

//Запретим прерывание при наличии данных для передачи

LDI TMP,(1<<RXEN)|(1<<TXEN)|(1<<RXCIE)|(1<<TXCIE)|(0<<UDRIE)

OUT UCSRB, TMP

SEZ //Установим 1 в бит Z регистра SREG

RETI

*Листинг 7.4.* Прерывание по наличию бита в UDR

Подробнее рассмотрим прерывание UD\_OK (листинг 7.4), в нем в регистр TMP считывается значение, лежащее по адресу, на который указывает указатель Z, и значение Z инкрементируется, затем проверяется условие конца массива, и, если оно выполнено, прерывание по наличию данных в регистре UDR запрещается. В противном случае символ из массива выводится в терминал.

Прием значений из терминала организован с помощью прерывания прием завершен (листинг 7.5), в котором полученный байт сохраняется в регистр TMP и выводится на дисплей:

RX\_OK: //Прием завершен

IN TMP,UDR //Забираем число из UDR

OUT PORTC,TMP //Выводим в порт

RETI

*Листинг 7.5.* Прием данных

## 4. Сопряжение МК и ПК

Один из самых простых способов сопряжения МК и ПК является соединение их через COM-порт согласовав уровни протоколов UART и RS-232.

В UART используется положительная логика, при которой логическая единица соответствует высокому уровню сигнала, и логический ноль – низкому. В RS-232 биты передаются разнополярными уровнями напряжения, при этом инвертированными относительно уровней UART, так логическая единица соответствует отрицательному уровню (-3…-12В), логический ноль положительному уровню (0…12В).

Таким образом сначала нужно согласовать физические параметры протоколов, это можно сделать как использовав готовые решение на базе микросхем типа MAX232, MAX202, ADM202 и других, так и сделав собственный преобразователь-инвертор напряжения. Схемы таких преобразователей нетрудно найти в Интернете. В случае отсутствия COM-порта хорошо зарекомендовали себя преобразователи RS-232/USB на базе микросхем FTDI.

Согласовав физические параметры протокола, нужно также согласовать программную часть, то есть установить одинаковые формат кадра и скорость в каждом приемо-передатчике.

Такой вариант сопряжения с ПК является наиболее простым, так как сопряжение с ПК по другим протоколам зачастую требует более дорогостоящих преобразователей интерфейсов, и зачастую более сложную программную часть.

**Задания для самостоятельной работы**

# Тема №8. Аналоговый компаратор и АЦП

*Вопросы, рассматриваемые в данной теме*:

1. Использование аналогового компаратора

2. Использование АЦП

3. Методы повышения точности АЦП

**Краткие теоретические сведения**

## 1. Использование аналогового компаратора

В МК AVR существует два основных способов работы с аналоговыми сигналами, это использование аналогового компаратора и 10-разрядного многоканального АЦП. Данные периферийные устройства входят во все МК AVR. В данной теме опустим подробности аналогово-цифровых преобразований, и разберем практические вопросы использования аналогового компаратора и АЦП.

Начнем с аналогового компаратора. Аналоговый компаратор сравнивает две величины напряжения на своих входах (положительном и отрицательном) и в зависимости от их соотношения устанавливает свой выход в одно из логических состояний. В МК AVR эти входы называются AIN0 (положительный) и AIN1 (отрицательный). Согласно документации, ошибка аналогового компаратора AtMega8 составляет не более 40мВ, а время отклика не более 0.5 мкс. Напряжение питания на его входах не может быть больше чем напряжение питания МК.

Управление аналоговым компаратор осуществляется регистром ACSR, рассмотрим его биты:

Бит ACD управляет включением компаратора, причем его нулевое состояние означает что компаратор включен (режим по умолчанию), поэтому если в схеме компаратор не используется, и вопрос энергопотребления важен, компаратор необходимо отключать.

Бит ACBG отвечает за подключение к положительному входу компаратора (AIN0) источника опорного напряжения величиной 1.22В, данный источник довольно неточен и погрешность его значений может достигать до 0.1В.

Бит ACIE отвечает за разрешение прерывания при смене состояния выхода компаратора.

Биты ASIS1 и ASIS0 отвечают за событие вызывающее прерывание, все события приведены в таблице 8.1:

*Таблица 8.1.* События вызывающие прерывания

|  |  |  |
| --- | --- | --- |
| ASIS1 | ASIS0 | Событие |
| 0 | 0 | Прерывание по любому перепаду уровней (по умолчанию) |
| 1 | 0 | Прерывание по низкому уровню на выходе |
| 1 | 1 | Прерывание по высокому уровню на выходе |

Бит ACO указывает на текущее состояние выхода компаратора и в любой момент может быть прочитан.

С учетом вышесказанного рассмотрим простую программу: на положительный вход компаратора (AIN0) подается опорное напряжение равное 3.3В, на отрицательный вход (AIN1) какое-то напряжение в диапазоне от 0 до 5 вольт, необходимо в случае снижения напряжения ниже опорного уровня подавать сигнал зажигая светодиод.

Инициализируем компаратор:

//Настроим аналоговый компаратор

LDI TMP, (1<<ACIE)

OUT ACSR,TMP

*Листинг 8.1.* Инициализация аналогового компаратора

Так как компаратор по умолчанию запущен, разрешим его прерывания (листинг 8.1) установив бит ACIE регистра ACSR, так как биты ASIS1:ASIS0 не изменялись то прерывание будет срабатывать при любом перепаде уровня на выходе компаратора.

Пока напряжение на входе AIN1, выше чем опорное напряжение на входе AIN0, выход компаратора (бит ACO) не установлен, как только напряжение на входе AIN1 упадет ниже опорного напряжение, бит ACO установится и сработает прерывание:

ACI\_R:

SBIS ACSR,ACO //Если бит ACO установлен, зажигаем LED

RJMP RES\_LED //Иначе гасим

SBI PORTB,0

RETI

RES\_LED:

CBI PORTB,0

RETI

*Листинг 8.2.* Прерывание аналогового компаратора

В прерывании (листинг 8.2) проверяется установлен ли бит ACO, если да, то напряжение на входе AIN1 ниже опорного, и необходимо зажечь светодиод, в противном случае, бит ACO сброшен и светодиод необходимо потушить.

Другие неочевидные способы использования компаратора такие как реализация интегрирующего АЦП на компараторе или подключение компаратора ко входу захвата таймера TIM1 для формирования входных импульсов и измерения длительности временных интервалов или подсчета событий можно найти в документации и специализированной литературе посвященной аналоговым измерениям.

## 2. Использование АЦП

АЦП в МК AVR может работать в двух основных режимах, в режиме непрерывного преобразования, когда после окончания преобразования сразу начинается следующее, и в режиме одиночного преобразования, когда для начала преобразования необходимо подавать команду на запуск.

Рассмотрим основные управляющие и конфигурационные биты АЦП разделив их для удобства на группы.

1. *Включение, запуск и выбор режима преобразования АЦП:* включение АЦП производится установкой бита ADEN конфигурационного бита ADCSR(ADCSRA), а выбор режима преобразования с помощью бита ADFR(ADATE), при установке которого АЦП конфигурируется режим непрерывного преобразования. Если выбран режим запуска по умолчанию (смотри следующий пункт), то преобразование запускается записью бита ADCS. В непрерывном режиме преобразования, бит ADCS запустит первое преобразования, а остальные будут происходить автоматически. В режиме однократного преобразования, установка бита ADCS запускает однократное преобразование.
2. *Выбор режима запуска преобразования*: возможность выбора режима запуска преобразования есть не во всех МК, там, где она есть выбор режима осуществляется установкой битов ADTS2…0 регистра SFIOR согласно таблице:

*Таблица 8.2.* Режим запуска преобразования

|  |  |  |  |
| --- | --- | --- | --- |
| ADTS2 | ADTS1 | ADTS0 | Режим запуска преобразования |
| 0 | 0 | 0 | Преобразования в непрерывном режиме или по ручному запуску (по умолчанию) |
| 0 | 0 | 1 | Запуск АЦП от аналогового компаратора |
| 0 | 1 | 0 | Запуск от внешнего прерывания INT0 |
| 0 | 1 | 1 | По совпадению таймера T0 |
| 1 | 0 | 0 | По переполнению таймера T0 |
| 1 | 0 | 1 | По совпадению таймера T1 |
| 1 | 1 | 0 | По переполнению таймера T1 |
| 1 | 1 | 1 | «Захват» таймера T1 |

1. *Тактовая частота АЦП:* согласно документации, оптимальная скорость преобразования лежит в диапазоне 50-200кГц, и выбирается установкой значений в биты предделителя ADPS2…0 регистра ADCSR(ADCSRA). Коэффициенты устанавливаются по степеням двойки, то есть все нули в трех битах соответствуют коэффициенту деления 2, все единицы 128.
2. *Выбор каналов и режимов их взаимодействия:* сигнал в АЦП подается через мультиплексор, а количество каналов зависит от типа МК. Выбор входа с которого будет сниматься аналоговый сигнал осуществляется битами MUX3…MUX0 регистра ADMUX. Записанное в эти регистры число определяет выбранный канал. Кроме того, существует несколько служебных наборов битов, некоторые из которых будут рассмотрены далее.
3. *Выбор источника опорного напряжения:* битв REFS1…REFS0 позволяют выбрать источник опорного напряжения согласно таблице:

Таблица 8.3. Выбор источника опорного напряжения

|  |  |  |
| --- | --- | --- |
| REFS1 | REFS0 | Источник опорного напряжения |
| 0 | 0 | Внешний источник (AREF) (по умолчанию) |
| 0 | 1 | AVCC |
| 1 | 1 | Встроенный 2.56В |

Напряжение внешнего источника питания, подключенного к выводу AREF должно лежать в пределах от 2В до напряжения питания аналоговой части AVcc, при этом напряжение питания аналоговой части не должно отличаться от напряжения питания цифровой части на величину большую чем 0.3В.

Для вариантов использования AVcc и встроенного источника опорного напряжения рекомендуется к ножке AREF подключить фильтрующий конденсатор. Так же стоит отметить что встроенный источник опорного напряжения имеет достаточно неточен и может иметь значение от 2.4 до 2.7В.

Кроме источника опорного напряжения величиной 2.56В можно подключить источник опорного напряжения величиной 1.22В установив биты MUX3…0=1110.

1. *Прерывания и результат преобразования:* бит ADIE регистра ADCSR(ADCSRA) разрешает прерывание по окончанию преобразования, в котором можно сохранить результат преобразования и провести над ним какую-либо математическую обработку.

Результат преобразования автоматически записывается в регистровую пару ADCH:ADCL, из которых фактически используются только 10 битов. При этом результат можно выровнять как по правому краю (режим по умолчанию), тогда два старших бита результата будут в ADCH, остальные в ADCL, так и по левому (установив бит ADLAR регистра ADMUX):

|  |  |  |  |
| --- | --- | --- | --- |
| Выравнивание по правому краю | | Выравнивание по левому краю | |
| ADCH | ADCL | ADCH | ADCL |
| [x][x][x][x][x][x][9][8]:[7][6][5][4][3][2][1][0] | | [9][8][7][6][5][4][3][2]:[1][0][x][x][x][x][x][x] | |

Учитывая то что зачастую в младших разрядах только мусор и шумы, имеет смысл делать выравнивание и забирать весь результат из регистра ADCH, то есть АЦП будет фактически работать в 8-битном режиме. Если же используются все разряды, то необходимо соблюдать порядок чтения этих регистров, сначала младший (ADCL) потом старший (ADCH), иначе можно получить значения состоящее в котором старший байт будет из текущего измерения, а младший, например, из следующего.

Кроме вышеописанных режимов АЦП может работать в дифференциальном режиме. В этом режиме АЦП измеряет разность напряжение между двумя каналами с последующим умножением на коэффициент усиления.

Пример использования АЦП рассмотрим на программе, которая будет снимать значение с 0 канала АЦП, преобразовывать полученное значение и выводить его целую часть на семисегментный дисплей.

По традиции начнем рассмотрение программы с определения переменных и констант:

//Определение переменных (директива .def)

.def TMP =R16 //временная переменная

.def ADC\_Res =R17 //результат АЦ преобразования

//Определение констант (директива .equ)

.equ ADC\_Const = 5

//Сегмент ОЗУ (RAM)

*Листинг 8.3.* Определение переменных и констант

Результат преобразования будет хранится в переменной ADC\_Res, а для временных нужд воспользуемся TMP. Константу преобразования АЦП рассчитаем из тех условий что при диапазоне измерений от 0 до 5В и разрядностью АЦП шаг измерений будет равен 0.02В. Для того чтобы работать с вещественными числами в МК (например, умножать) необходимо это число «загнать» в диапазон целых чисел, умножив его на число степень двойки, например, на 28 и округлив до целого, таким образом умножив 0.02 на 28 округлив получим константу (ADC\_Const) равную 5.

Чтобы перейти к реальному значению, нужно будет умножить получено значение с АЦП на константу ADC\_Const и разделить на степень двойки сдвигом вправо.

Настроим АЦП (листинг 8.4) на работу в режиме непрерывных измерений с предделителем 64, что при частоте работы МК равной 4МГц, даст частоту работы АЦП 62.5КГц, что укладывается в диапазон. Данные будем считывать с 0 канала, за эталонное напряжение примем напряжение питания МК. Результат выровняем по левому краю. Разрешим прерывания, включим АЦП и дадим старт преобразованию:

//Настройка АЦП

//ADLAR-Выравнивание по левому краю

//REFS- Выбор источника опорного напряжения

//MUX- Выбор канала АЦП

LDI Temp, (1<<ADLAR|1<<REFS0) //Устанавливаем нужные нам биты в 1

OUT ADMUX,TMP //Записываем значение в управляющий регистр

//Инициализируем АЦП

//ADEN-Включить АЦП

//ADSC-Начать преобразование

//ADFR-Режим непрерывных измерений

//ADIE-Разрешение прерывания

//ADPS-Предделитель тактовой частоты АЦП

LDI TMP, (1<<ADEN|1<<ADSC|1<<ADFR|1<<ADIE|1<<ADPS0|1<<ADPS2)

OUT ADCSRA, TMP

*Листинг 8.4.* Инициализация АЦП

Вся дальнейшая работа программы будет заключаться получение результата преобразования и его обработке в прерывании по окончанию преобразования (ADC\_END):

//Прерывание по окончанию АЦ преобразования

ADC\_END:

LDI TMP,ADC\_Const //Запишем константу для получения реального значения с АЦП

IN ADC\_Res,ADCH //Перенесем полученное значение АЦП в РОН

MUL ADC\_Res,TMP //Умножим на константу (результат в R0-R1)

MOV ADC\_Res,R1 //Сдвиг на 8 разрядов вправо это фактически перенос старшего разряда в младший)

OUT PORTB, ADC\_Res //Выведем полученное значение

RETI

*Листинг 8.5.* Прерывание по окончанию преобразования

Здесь (листинг 8.5) сначала во временную переменную TMP записывается константа ADC\_Const, для дальнейшего математического преобразования, затем результат преобразования сохраняется в ADC\_Res, и умножается на ADC\_Const, далее, чтобы получить реально значение, нужно полученный результат разделить на 8, что фактически есть перенос старшего разряда в младший, считаем старший разряд результата умножения (R1) и выведем его на дисплей.

Стоит заметить, что ввиду всех сокращений, точность выводимого на экран значения не очень велика.

## 3. Методы повышения точности АЦП

Рассмотрим основные методы повышения точности аналогово-цифрового преобразования, их можно разделить на программные и аппаратные методы повышения точности. Рассмотрим их

1. Если скорость измерения значительно превышает скорость измерения сигнала, то можно снимать не одно значение, а группу с последующим усреднением результата. Так же для эффективного подавления шумов и помех нужно чтобы частота выборки была ниже частоты различных паразитных колебаний, например, наводок от сети, примерно в 2-3 раза.
2. Использование специального режима работы АЦП – ADC Noise Reduction, который останавливает все схемы МК за исключением асинхронного таймера и «сторожевого пса». Неудобство данного способа заключается в том, что при каждом измерении МК «засыпает» не менее чем на 20 тактов, поэтому при достаточно частых измерениях это может негативно сказаться на других функциях, выполняемых МК.
3. Использование высокоточного источника опорного напряжения, например, ADR420 или REF195 либо завести напряжение на AVcc через дроссель (рисунок 2.3).
4. Фильтрация питания МК, установкой керамический конденсаторов емкостью порядка 0.1uF между Vcc и GND (рисунок 2.3), как можно ближе к выводам МК, это позволит гасить помехи, возникающие при переключении логических уровней внутри МК.
5. Так же можно организовать как можно более сплошную аналоговую землю, окружающую всю аналоговую часть и не содержащую в себе замкнутых контуров. С цифровой землей ее следует соединить в одной точке как можно дальше от точных цепей.
6. Использование фильтров для сигналов, у которых известна частота, что позволит отфильтровывать все что не входит в требуемы диапазон.

**Задания для самостоятельной работы**

# Тема № 9. Использование EEPROM

*Вопросы, рассматриваемые в данной теме*:

1. Энергонезависимая память данных EEPROM

2. Сохранность данных в EEPROM

3. Запись и чтение EEPROM

**Краткие теоретические сведения**

## 1. Энергонезависимая память данных EEPROM

Как было отмечено ранее (в главе 3), практически все МК AVR имеют встроенную память EEPROM для хранения констант и данных при отключении питания. Рассмотрим основные моменты важные при программировании данного типа памяти.

Одной из особенностей EEPROM является то что, запись в память протекает значительно медленнее чем чтение из памяти. И так как процесс записи зависит от встроенного RC-генератора то длительность цикла записи одного байта в EEPROM не определена и может занимать в среднем от 2 до 4 мс. С точки зрения МК время порядка 2 мс огромно, и за него можно успеть выполнить несколько тысяч команд, поэтому к процессу записи нужно подходить внимательно и следить за его целостностью.

Так же содержимое памяти EEPROM очень чувствительно к снижению питающего напряжения, то есть при снижении питания ниже порога стабильной работы, но недостаточного для того чтобы МК «отключился» начнется выполнение случайных команд, в том числе может быть выполнена команда записи в EEPROM что с большой вероятностью испортит ее содержимое. Как с этим бороться рассмотрим далее.

Кроме записи в EEPROM во время работы МК, можно осуществить запись в память и при прошивке контроллера. Но такая возможность требуется не часто, так как в основном в EEPROM хранят какие-либо пользовательские установки, константы вычисляемые при работе устройства и другие данные которые необходимо сохранить в памяти устройства при выключении питания.

## 2. Сохранность данных в EEPROM

Разберем подробнее какие меры можно принять для сохранности данных в EEPROM, существует два метода, использование встроенных средств МК и подключение дополнительного монитора питания.

Для начала рассмотрим первый способ –использование встроенного супервизора питания - схемы BOD (по умолчанию выключена), которая позволяет при снижении напряжения ниже допустимого порога сбрасывать МК и возвращать его в рабочее состояние после восстановления напряжения. Настройка режима работы BOD выполняется с помощью fuse-битов BODLEVEL2…0 в соответствии с таблицей 9.1, а его включение битом BODEN.

*Таблица 9.1.* Режимы работы BOD

|  |  |  |  |
| --- | --- | --- | --- |
| BODLEVEL2 | BODLEVEL1 | BODLEVEL0 | Режим работы |
| 1 | 1 | 1 | Схема BOD выключена  (значение по умолчанию) |
| 1 | 0 | 1 | Включает BOD с порогом срабатывания 2.7В |
| 1 | 0 | 0 | Включает BOD с порогом срабатывания 4В |

Очевидно, что чем меньше разница между напряжением питания и порогом срабатывания, тем надежнее будет работа МК, но также стоит учитывать, что при питании от батарей можно привести схему в неработоспособное состояние в момент, когда батареи еще не исчерпали свой ресурс.

Стандартное время срабатывания встроенной схемы BOD составляет порядка 4мс, а максимальное время включение может быть около 68мс, что не всегда перекрывает дребезг, возникающий при снижении питания.

Даная схема помогает защитить данные EEPROM, но в некоторых случаях ее все же недостаточно. Тогда имеет смысл установить в схему внешний монитор питания, например, MC34064 имеет порог срабатывания 4,6В и обладает малым собственным потреблением, порядка 0,5мА. При снижении напряжения его время срабатывания составляет 200нс, а время обратного включения занимает доли секунд, что предотвращает срабатывания при дребезге и позволяет добиться большей сохранности данных в памяти.

## 3. Запись и чтение EEPROM

Рассмотрим запись и чтение EEPROM, на примере программы позволяющей кнопкой (SB\_UP) выбрать одно значение от 1 до 9, выводимое на один из дисплеев и сохранить кнопкой (SB\_SAVE) его, и при нажатии на кнопку (SB\_LOAD) вывести на второй дисплей, причем оба дисплея, семисегментные, с дешифратором и подключенные к одному порту МК.

Как обычно рассмотрение программы начнем инициализации констант и переменных:

//Определение переменных (директива .def)

.def TMP = R16 //Временная переменная

.def CNT\_U = R17 //Счетчик первого дисплея

.def CNT\_L = R18 //Счетчик второго дисплея

.def EE\_AddrH = R19 //Адрес для записи (старший)

.def EE\_AddrL = R20 //Адрес для записи (младший)

//Определение констант (директива .equ)

.equ CNT\_MAX =10 //Для ограничение вывода на дисплей значений (0-9)

.equ D1\_MASK =0b00001111

.equ D2\_MASK =0b11110000

*Листинг 9.1.* Процедура записи в EEPROM

В листинге 9.1 сначала определяются три переменных: обычная временная TMP, и две переменных счетчика CNT\_U и CNT\_L, отвечающие за значения, чисел, которые будут выводится на дисплеи. Для задания адреса EEPROM в который будет производится запись определенны две переменные EE\_AddrL и EE\_AddrH, младший и старший байты адреса соответственно.

Константами определенны: число при достижение которого значение на дисплее сбрасывается в 0 (CNT\_MAX) и две маски (D1\_MASK, D2\_MASK), использующиеся для того чтобы не портить значение в одном из дисплеев при записи в другой, ведь они подключены к одному порту МК.

Далее разберем процедуру записи:

//Процедура записи EEPROM

EE\_WRITE:

SBIC EECR,EEWE //Ожидаем готовность памяти к записи

RJMP EE\_WRITE //То есть установку флага EEWE

OUT EEARL, EE\_AddrL //Загружаем адрес необходимой ячейки (0)

OUT EEARH, EE\_AddrH //(старший и младший)

OUT EEDR, CNT\_U //и сами данные

SBI EECR, EEMWE //Устанавливаем бит предохранитель

SBI EECR, EEWE //Записываем байт

RET

*Листинг 9.2.* Процедура записи в EEPROM

Так как запись в EEPROM процесс достаточно медленный то сначала необходимо дождаться готовности памяти к записи, то есть сброса (автоматического) флага записи EEWE регистра EECR. Далее в регистры EEARL и EEARH, записывается адрес ячейки нужной для записи, а в регистр EEDR сами данные которые необходимо сохранить. Затем в регистр EECR устанавливается специальный бит предохранитель EEMWE, после установки которого для записи нужного значения в память необходимо в течении четырех тактов необходимо установить бит EEWE регистра EECR. Если не успеть это сделать, то бит EEWE сбросится и его необходимо будет выставить снова. Это сделано для защиты от случайной записи в EEPROM случаи которой рассматривались выше.

Чтение происходит практически аналогичным образом:

//Процедура чтения EEPROM

EE\_READ:

SBIC EECR, EEWE //Ждем окончания предыдущей записи

RJMP EE\_READ

OUT EEARL, EE\_AddrL //Выставляем адрес

OUT EEARH, EE\_AddrH

SBI EECR,EERE //Выставляем бит чтения

IN CNT\_L, EEDR //Читаем бит из ячейки памяти

RET

*Листинг 9.2.* Процедура чтения EEPROM

Ожидается готовность памяти (листинг 9.2), выбирается адрес, в котором лежат нужные данные, затем выставляется бит чтения EERE регистра EECR, и данные забираются из регистра EEDR в любой из регистров общего назначения.

Кроме вышеперечисленного стоит отметить что из-за наличия бита предохранителя EEMWE процедура записи в EEPROM процесс атомарный, и очевидно не должен ничем прерываться, поэтому если в программе используются прерывания, на время записи их необходимо запретить. Так же если в программе используется сторожевой таймер (по умолчанию выключен), то в цикле ожидания готовности памяти, необходимо сбрасывать его командой WDR.

Кратко рассмотрим другие части программы, в обработчике нажатия кнопки SB\_SAVE вызывается процедура записи в память EE\_WRITE. Обработчик нажатия кнопки SB\_LOAD рассмотрим подробнее, так как в нем во второй дисплей загружаются данные так чтобы не испортить значение в первом:

SB\_LOAD:

SBIS PINB,1

RJMP SB\_LOAD

RCALL EE\_READ //Вызовем функцию чтения

DEC CNT\_L //Уменьшаем так как в память записывается значение на 1 больше.

SWAP CNT\_L //Переставляем тетрады (дисплеи на одном порту)

IN TMP, PORTD //Сохраняем текущее значение порта (чтобы не испортить)

ANDI TMP, D1\_MASK //Маскируем

OR TMP, CNT\_L //Обновляем значение

OUT PORTD,TMP //И выводим

RJMP MAIN

*Листинг 9.3.* Загрузка и вывод данных на дисплей

В листинге 9.3 за стандартным ожиданием отпускания кнопки, следует вызов процедуры чтения из памяти, в котором значение из EEPROM копируется в CNT\_L, и уменьшается на единицу (по причине того, что в память записываться значение на единицу больше реального), затем командой SWAP меняются тетрады байта, так как значение лежит в младшей тетраде, а дисплей подключен к старшей. Далее текущее значение битов PORTB копируется во временный регистр TMP, на который для сохранности значения первого дисплея с помощью операции логического «И» накладывается маска D1\_MASK, и операцией логическое «ИЛИ» устанавливается новое значение для второго дисплея.

Разберем пример (таблица 9.2): допустим, что на первом дисплее (D1) – выведено число 5 (0b00001001), а на втором дисплее (D2) выведено число 4 (0b10000000) которое должно быть обновлено на число 3 (0b00110000):

*Таблица 9.2.* Логические преобразования при выводе на дисплеи

|  |  |  |  |
| --- | --- | --- | --- |
| Операция | Старшая  тетрада (D2) | Младшая  тетрада (D1) | Регистр/  маска |
| Загрузка из памяти числа 3 | 0000 | 0011 | CNT\_L |
| Перестановка тетрад (SWAP) | 0011 | 0000 | CNT\_L |
| Сохраним значение PORTB в TMP | 1000 | 1001 | TMP |
| Логическое «И» с маской D1\_MASK | 0000 | 1111 | D1\_MASK |
| Результат: | 0000 | 1001 | TMP |
| Логическое «ИЛИ» с CNT\_L | 0011 | 0000 | CNT\_L |
| Результат: | 0011 | 1001 | CNT\_L |

То есть после всех операций в регистре CNT\_L оказалось обновленное значение второго дисплея, и неизменное значение первого дисплея.

Обработчик кнопки выбора числа (SB\_UP), в целом аналогичен обработчику кнопки SB\_LOAD, за исключением того, что в нем нет чтения из памяти, и есть проверка на максимальное значение в регистре.

Так же кроме рассмотренного метода записи EEPROM во всех современных моделях МК существует специальное прерывание, которое генерируется по окончанию очередного цикла записи. Данное прерывание удобно использовать при записи достаточно больших массивов данных, время записи которых длится порядка секунд. Данный способ организации записи заметно сложнее, и даже официальная документация для небольших объемов данных рекомендует использовать вышеприведенные методы.

**Задания для самостоятельной работы**

# Тема № 10. Динамическая индикация и опрос матрицы кнопок

*Вопросы, рассматриваемые в данной теме*:

1. Динамическая индикация

2. Опрос матричной клавиатуры

**Краткие теоретические сведения**

## 1. Динамическая индикация

Теперь после изучения принципов работы портов ввода/вывода, а также режимов работы таймеров, можно рассмотреть такие практические задачи как вывод информации на группу семисегментных дисплеев с помощью динамической индикации, и опрос клавиатуры или матрицы кнопок.

Разберем способ подключения группы семисегментных дисплеев (с общим анодом) к МК, и реализуем на базе 4-х семисегментных дисплеев секундомер, считающий от 0 до 1000 и управляющийся кнопки START и STOP.

Смысл динамической индикации заключается в том, что выводы дисплея, отвечающие за включение сегментов, объединяются в общую шину (шину данных) и подключаются к одному порту МК, а общие аноды (катоды), далее будем называть их управляющими выводами, соответственно подключаются к другим линиям портов ввода/вывода МК, таким образом если с достаточно большой частотой переключать дисплеи и выводить на них данные создастся эффект постоянно горящего изображения. И в отличии от раздельного подключения дисплеев будет задействовано всего 12 выводов МК (8 – шина данных, 4 – управляющие выводы).

Рассмотри подключение группы из двух дисплеев (с общим анодом) к МК (рис. 10.1):



*Рис. 10.1.* Схема подключения группы семисегментных дисплеев

Шина данных подключается аналогично выводам семисегментного дисплея через ограничительные резисторы (R1-R8) номиналом 470 Ом, а выводы управления в зависимости от выбранных дисплеев подключаются либо напрямую (если хватает мощности МК), либо через транзисторы (Q1, Q2). Выбор типа транзистора зависит от типа дисплея, если используется дисплей с общим анодом (включение сегментов дисплея производится низким уровнем), то используется транзисторы PNP типа, в случае общего катода, включение сегментов дисплея производится высоким уровнем и соответственно используется транзисторы NPN типа. Кроме раздельного подключения нескольких дисплеев существуют сборки из двух, четырех и более дисплеев в которых шина данных объединена внутри корпуса, что упрощает проектирование печатной платы.

Кратко принцип работы программы будет заключатся в следующем: отсчет времени производится благодаря таймеру TIM1, значения для вывода на дисплей будут обрабатываться в его прерывании при совпадении CTC (TIM1M\_A), а вывод динамической индикации будет осуществляться в прерывании по переполнению(TIM0OVF) таймера TIM0. Маска для вывода значений на дисплей хранится во flash-памяти и используется аналогично ранее рассмотренному случаю вывода чисел на обычный семисементный дисплей.

Перейдем к рассмотрению вышеуказанной программы. Для удобства зададим константы и переменные (листинг 10.1):

//Определение переменных (директива .def)

.def TMP = R16 //Временная переменная

.def CNT\_1N = R17 //Число для загрузки из памяти (1 разряд)

.def CNT\_2N = R18 //Число для загрузки из памяти (2 разряд)

.def CNT\_3N = R19 //Число для загрузки из памяти (3 разряд)

.def CNT\_4N = R20 //Число для загрузки из памяти (4 разряд)

.def COUNTER = R22 //Выбор текущего выводимого разряда на индикаторе

.def MASK = R21 //Маска для вывода на дисплей (из FLASH)

//Определение констант (директива .equ)

.equ IND\_MSK = 0b00000001 //Маска управляющего сигнала

.equ MAX\_CNT = 10 //Для определения переполнения разряда

*Листинг 10.1.* Настройка портов ввода/вывода МК

Переменные CNT\_xN будут использоваться для хранения, числа которое должно быть выведено на дисплей x, COUNTER используется для определения на какой из дисплеев необходимо вывести значение, MASK для считанной из памяти маски числа (аналогично обычному семисегментному дисплею). Константа IND\_MSK, для указания на линию порта управляющего выбором дисплеев, MAX\_CNT, для определения переполнения.

Далее настроим порты ввода вывода МК, с учетом того что используются дисплеи с общим анодом (листинг 10.2):

//Подключение 7 сегментных дисплеев (общий анод, включение низким уровнем)

LDI TMP,0xFF

OUT DDRD,TMP //8 бит данных

OUT PORTD,TMP

LDI TMP,0x0F //4 управляющих бита (биты выбора активного дисплея)

OUT DDRC,TMP

*Листинг 10.2.* Настройка портов ввода/вывода МК

Порт D используется для подключения шины данных дисплеев и полностью конфигурируется как выход, на его линиях выставляется высокий уровень (все разряды выключены). 4 бита порта C к которым подключены управляющие линии дисплеев, так же устанавливаются на выход.

Рассмотрим настройку таймеров и сам принцип динамической индикации (листинг 10.3):

//Настройка таймеров TIM0, TIM1

LDI TMP, 1<<TOIE0|1<<OCIE1A //Разрешим прерывание TIM0 переполнению

OUT TIMSK, TMP //и TIM1 по совпадению с A

LDI TMP, 1<<CS00|1<<CS01 //Запустим таймер динамической индикации

OUT TCCR0,TMP

LDI TMP, 1<<WGM12 //Режим работы таймера TIM1, сброс при достижении значения

OUT TCCR1B, TMP

//Запись в 16-разрядный регистр TCNT

LDI TMP, HIGH(62500)

OUT OCR1AH, TMP //Первым пишется старший байт

LDI TMP, LOW(62500)

OUT OCR1AL, TMP //Вторым младший

*Листинг 10.3.* Настройка таймеров МК

Существует два очевидных варианта переключения с определенной частотой, это программные счетчики (низкая точность) и аппаратный таймер. Принцип работы переключения в случае аппаратного таймера строится на том, что при каждом его переполнении срабатывает прерывание (TIM0OVF) и в его обработчике выводится следующий сегмент дисплея, использование прерывания гарантирует обновления дисплея через равные промежутки времени. Например, если МК работает на частоте 4 МГц, а 8-разрядный таймер (TIM0) подключен через предделитель 64, то прерывание будет срабатывать с частотой примерно равной 245 Гц, а весь дисплей обновляться с частотой примерно равно 60Гц, что достаточно, чтобы человеческий глаз не замечал переключения.

Принцип отсчета времени для секундомеров был рассмотрен ранее, здесь же для отсчета одной секунды используем 16-разрядный таймер (TIM1), в режиме работы CTC (сброс при совпадении). Рассчитаем значение, при котором таймер должен сбрасываться, при частоте работы МК равной 4 МГц, и предделителе 64, частота переполнения 16-разрядного МК составит 0.9537 Гц, таким образом, чтобы считаться с частотой 1 Гц, необходимо сбрасывать таймер при достижении значения в счетчике равном 62500.

Можно обратить внимание что в листинге 10.3 таймер отсчета времени TIM1 в отличии от таймера динамической индикации TIM0 не запускается, это связано с тем что запускается он по нажатию кнопки START.

Теперь рассмотрим прерывания, начнем с прерывания по переполнению таймера TIM0 отвечающего за динамическую индикацию (TIM0OVF) (листинг 10.4):

TIM0OVF:

//Установим указатель на начало массива с символами

LDI ZH,HIGH (N\_mask\*2)

LDI ZL,LOW (N\_mask\*2)

CPI COUNTER,0

BREQ DISP1

...

CPI COUNTER,3

BREQ DISP4

DISP1:

INC COUNTER //Увеличим счетчик, для того чтобы в следующей итерации

//обновить другую часть дисплея

OUT PORTC,MASK //Выберем необходимый дисплей

//Вывод на дисплей символа из памяти по адресу(ZL+CNT\_xN)

ADD ZL,CNT\_1N //Прибавим к указателю на Flash нужное значение

LPM //Достанем его из памяти (регистр R0)

OUT PORTD,R0 //Выведем

RETI

DISP2: //Аналогично

INC COUNTER

LSL MASK //Сместим маску влево

OUT PORTC,MASK

ADD ZL, CNT\_2N

LPM

OUT PORTD,R0

RETI

...

DISP4: //Аналогично

LSL MASK

OUT PORTC,MASK

ADD ZL, CNT\_4N

LPM

OUT PORTD,R0

//Отчистим счетчик итераций и вернем маску к начальному значению

CLR COUNTER

LDI MASK,IND\_MSK

RETI

*Листинг 10.4.* Прерывание TIM0OVF

Листинг приводится в сокращенном виде, так как фактически состоит из повторяющихся операций. Сначала устанавливается указатель на начало массива в котором лежат маски значений, далее определяется на какой дисплей необходимо вывести значение, и происходит переход к необходимой метке DISPx, где если необходимо увеличивается счетчик для вывода при следующем прерывании на следующий дисплей (либо этот счетчик обнуляется для вывода на первый дисплей) и смещается (или сбрасывается) маска выбора управляющего вывода. Так же по каждой из меток (аналогично ранее рассмотренному выводу на семисегментный дисплей) выводится нужное число.

Прерывание TIM1 для отсчета временных интервалов практически аналогично прерыванию для ранее рассмотренного секундомера, за исключением того, что в нем обрабатываются четыре разряда, а не два разряда, поэтому рассматриваться подробно не будет.

Использовать метод динамической индикации можно как на большем, так и на меньшем числе дисплеев, а также на различных светодиодных матрицах, ограничиваясь только доступным количеством линий ввода/вывода МК. А если использовать 2 сдвиговых регистра HC595, то количество необходимых выводов для рассматриваемой задачи можно сократить до 4.

## 2. Опрос матричной клавиатуры

Зачастую для опроса большого количества кнопок выводов может не хватить, в таком случае можно использовать матричную клавиатуру (рис. 10.2). Такую клавиатуру можно найти как в готовом виде, так и собрать самостоятельно, подключая кнопки по схеме, представленной на рисунке 10.2б.



*Рис. 10.2.* Схема подключения матричной клавиатуры (а) и ее кнопок (б)

Принцип опроса матричной клавиатуры (рис. 10.2а) схож с принципом динамической индикации и заключается в том, что в МК выделяется два порта, сканирующий (C1-C3) и считывающий (P1-P4). Сканирующий порт в режиме выхода (через диоды VD1-VD3 служащие для защиты от короткого замыкания между линиями строк и столбцов) подключается к столбцам и подтягивается к напряжению питания. Хотя логика работы опроса клавиатуры не допускает такой возможности, на период отладки установка диодов рекомендуется. Считывающий порт работает в режиме входа с подтяжкой (Pull-Up), для этого могут использоваться как внешние резисторы (R1-R3), так и встроенные Pull-Up резисторы МК.

Затем в одну из линий сканирующего порта (например, C1) выводится низкий уровень сигнала (0), что придавливает весь столбец к земле. В это время поочередно считываются сразу все значения из считывающего порта (P1-P4), и если ни одна из кнопок не нажата, то на всех линиях считывающего порта будет высокий уровень сигнала (1), в противном случае нажатая кнопка замкнет вывод С1 на соответствующую линию считывающего порта изменив на нем сигнал с высокого уровня на низкий. Таким образом будут получены однозначные координаты нажатой кнопки. Например, если нажата кнопка 3, то на линии P1 будет низкий уровень сигнала (0).

Далее по аналогичному принципу сканируются оставшиеся порты столбцов (C1-C3). Зная принцип сканирования матрицы кнопок можно составить таблицу соответствия считываемых кодов с реальными значениями. Другой способ составление таблицы, использованный в данном случае, вывод поочередно масок в неиспользуемый порт и их последующая запись в таблицу:

*Таблица 10.1*. Соответствия маски и реальных значений

|  |  |  |
| --- | --- | --- |
| Значение маски (MSK) | | Реальное значение |
| Двоичное | Шестнадцатеричное |
| 0b11000001 | 0xC1 | 1 |
| 0b10100001 | 0xA1 | 2 |
| 0b01100001 | 0x61 | 3 |
| 0b11000010 | 0xC2 | 4 |
| 0b10100010 | 0xA2 | 5 |
| 0b01100010 | 0x62 | 6 |
| 0b11000011 | 0xC3 | 7 |
| 0b10100011 | 0xA3 | 8 |
| 0b01100011 | 0x63 | 9 |
| 0b11000100 | 0xC4 | \* |
| 0b10100100 | 0xA4 | 0 |
| 0b01100100 | 0x64 | # |

Рассмотрим программную реализацию вышеописанного метода опроса матричной клавиатуры на примере устройства считывающего нажатую кнопку клавиатуры и выводящую ее значение на цифровой индикатор.

Для начала (листинг 10.5) определим переменные и константы, а также во flash-памяти создадим массив соответствия маски и реальных значений для вывода на цифровой индикатор:

//Определение переменных (директива .def)

.def TMP = R16 //Временная переменная

.def CNT = R17 //Счетчик

.def MSK = R18 //Маска и результат

//Определение констант (директива .equ)

.equ KEYMSK = 0b11011111 //Маска для снятия значения со столбцов

.equ SCANMSK = 0b11100000 //Маска для того чтобы не "портить" оставшиеся линии порта

//Таблица соответствия

Code\_table:

.db 0xC1,0x01 //1

.db 0xA1,0x02 //2

.db 0x61,0x03 //3

.db 0xC2,0x04 //4

.db 0xA2,0x05 //5

.db 0x62,0x06 //6

.db 0xC3,0x07 //7

.db 0xA3,0x08 //8

.db 0x63,0x09 //9

.db 0xC4,0x0A //\*(a)

.db 0xA4,0x00 //0

.db 0x64,0x0B //#(b)

.db 0xFF,0 //Конец массива

*Листинг 10.5.* Определение констант и переменных

В данной программе используется три переменных: временная (TMP), счетчик столбцов (CNT), и переменная для записи считаной маски (MSK). Для удобства переключений столбцов используется маска KEYMASK, в которой 0 указывает на опрашиваемый столбец, и для перехода к следующему столбцу сдвигается. Маска SCANMSK используется, для того чтобы при опросе клавиатуры не изменять значения других линий ввода/вывода которые могут использоваться для других целей. Массив Code\_table содержит таблицу соответствий между считаной маской и реальными значениями.

Основная часть программы состоит из двух частей, вызывающихся последовательно, сначала вызывается функция опроса клавиатуры, а затем функция преобразования и вывода считаного значения на дисплей.

Сначала опустив повторяющиеся операции рассмотрим функцию опроса клавиатуры:

//Функция сканирования клавиатуры

KEYB\_INIT:

LDI CNT,3 //Выставляем счетчик проходов (по количеству столбцов)

LDI MSK, KEYMSK //Загружаем маску опроса

KEYB\_SCAN:

IN TMP, PORTB //Во временный регистр записываем значения опрашиваемого порта (чтобы не "испортить")

ORI TMP, SCANMSK //Выставляем в 1 все линии опроса

AND TMP, MSK //Логическим И выделяем опрашиваемую линию

OUT PORTB, TMP //Выводим в маску в порт

NOP //Задержка для того чтобы сигналы

NOP //точно установились

SBIS PIND,0 //Нажата кнопка в 1 ряду?

RJMP SB1

...

SBIS PIND,3 //Нажата кнопка в 4 ряду?

RJMP SB4

ROL MSK //Сдвигаем опрашиваемый бит влево

DEC CNT //Уменьшаем счетчик циклов

BRNE KEYB\_SCAN //Если не равен 0, продолжаем опрос

CLR MSK //Иначе "чистим" значение маски

RET //Возвращаемся в основной цикл

//Нажата кнопка в 1 ряду

SB1:

ANDI MSK,SCANMSK //Формируем сканированное значение, выделяя значащие биты

ORI MSK, 0x01 //Кнопка нажата в первом ряду

RET

...

//Нажата кнопка в 4 ряду

SB4:

ANDI MSK,SCANMSK //Формируем сканированное значение, выделяя значащие биты

ORI MSK, 0x04 //Кнопка нажата в первом ряду

RET

*Листинг 10.6.* Функция опроса клавиатуры

В листинге 10.6 по метке KEI\_INIT происходит инициализация опроса в которой выставляется по количеству столбцов счетчик проходов (CNT), а так же в регистр MSK загружается маска опроса KEYMASK. После инициализации начинается сам опрос (KEY\_SCAN), в котором сначала во временный регистр TMP сохраняется значения из порта B, затем линии опроса выставляется в 1, логическим «И» выделяется опрашиваемая линия, и полученное значение выводится в порт. Потом после некоторой задержки необходимой для того чтобы все линии порта точно установились в нужный уровень, начинается опрос считывающего порта. Если ни одна из кнопок не нажата, маска MSK опроса сдвигается влево, а счетчик CNT уменьшается. Если еще не все столбцы опрошены, то опрос продолжается, если опрос завершен, то маска обнуляется, и функция опроса завершает свое выполнение. В случае если кнопка нажата командой проверки-пропуска (SBIS) происходит переход по метке SBx (x –номер ряда), в которой из координат столбца и строки окончательно формируется маска MSK указывающая на нажатую кнопку.

После формирования значения выполнения функции прекращается. И начинается выполнение функции перевода координат в реальное значение:

//Инициализация функции декодирования

FIND\_SYMBOL\_INIT:

CLR TMP //На всякий случай отчищаем временный регистр

LDI ZH,HIGH(Code\_table\*2) //Загружаем указатель на массив

LDI ZL,LOW(Code\_table\*2)

//Непосредственно поиск символа

FIND\_SYMBOL:

LPM TMP,Z+ //Считаем значение по адресу Z и инкрементируем адрес(Z+)

//Теперь указатель указывает на реальное значение

CPI TMP,0xFF//Конец массива?

BREQ MAIN //Да, выходим из функции

CP MSK,TMP //Значение найдено в массиве?

BREQ DISPLAY\_OUT //Выводим

LPM TMP, Z+ //Если нет, то снова увеличиваем адрес чтобы пропустить

//адрес в котором лежит реальное значение и продолжаем поиск

RJMP FIND\_SYMBOL

DISPLAY\_OUT:

LPM TMP,Z //Загружаем значение из массива

OUT PORTC,TMP //Выводим его

RJMP MAIN

*Листинг 10.7.* Функция декодирования значения

Рассмотрим подробно листинг 10.7 Инициализации функции включает в себя, обнуление временной переменной TMP и установку указателя на массив соответствия значений Code\_table. Далее по метке FIND\_SYMBOL идет поиск соответствия полученной маски и реального значения, для этого в TMP загружается значение, лежащее по адресу Z, указатель инкрементируется и теперь указывает на реальное значение. Если конец массива не достигнут (TMP≠0xFF), то значение, считанное из массива, сравнивается с маской MSK, и в том случае если они равны, то по метке DISPLAY\_OUT реальное значение загружается в TMP из массива и выводится в порт. В ином случае, значение указателя снова инкрементируется и начинается новая итерация поиска.

Данный алгоритм работы с матричной клавиатурой с небольшими изменениями также может быть использован для опроса клавиатур других размерностей.

**Задания для самостоятельной работы**

# Тема № 11. Управление двигателями

*Вопросы, рассматриваемые в данной теме*:

1. Шаговые двигатели

1.1 Разновидности шаговых двигателей и способы управления ими

1.2 Программная реализация полушагового режима работы

2. Сервоприводы

2.1 Способы управления сервоприводами

2.2 Программная реализация управления сервоприводом

**Краткие теоретические сведения**

***1. Шаговые двигатели***

### *1.1 Разновидности шаговых двигателей и способы управления ими*

Шаговый двигатель — это электромеханическое устройство способное преобразовывать сигнал управления в угловое перемещение ротора с последующей фиксацией его в этом положении без устройств обратной связи.

Основными плюсами использование шаговых двигателей являются обеспечение полного момента в режиме остановки в случае если обмотки запитаны, точность их позиционирования без обратной связи, высокая надежность, связанная с отсутствием щеток, возможность регулировать скорость вращения изменяя частоту импульсов. К недостаткам можно отнести, достаточно высокое потребление энергии, возможную потерю контроля положения и относительно сложную схему управления.

Кратко рассмотрим основные типы шаговых двигателей:

*Рис. 11.1.* Типы шаговых двигателей:

а) биполярный, б) униполярный в) четырехобмоточный

Биполярный двигатель (рис 11.1а), имеет одну обмотку в каждой фазе, и для изменения направления магнитного поля должен переполюсовываться драйвером. Всего биполярный двигатель имеет две обмотки и соответственно 4 вывода. Для управления таким двигателем используется мостовой драйвер, либо полумостовой с двухполярным питанием.

В отличии от биполярного двигателя, униполярный (рис 11.1б) в середине каждой обмотки имеет отвод, благодаря этому изменять направление магнитного поля можно простым переключением половинок обмоток. Это позволяет использовать для управления двигателем четыре простых ключа. Обычно такие двигатели имеют 5 или 6 выводов, в зависимости от того объединены ли средние выводы обмоток внутри двигателя.

Иногда униполярные двигатели имеют четыре раздельные обмотки (рис 11.1в), и в зависимости от соединения обмоток такой двигатель может использоваться как биполярный или униполярный. Так же стоит заметить, что биполярные двигатели более мощные и при одних и тех же размерах по сравнению с униполярными обеспечивают больший момент.

С точки зрения программирования МК управления разными типами шаговых двигателей идентично, отличие только в используемом драйвере, так для управления биполярными двигателями часто используется драйвер L293D (рис. 11.2а), а для управления униполярными драйвер LM2003 (рис. 11.2б), который фактически является сборкой мощных транзисторных ключей.

*Рис. 11.2.* Схема подключения шагового двигателя к МК:

а) биполярный, б) униполярный

Существуют три основных способа управления шаговыми двигателями, полношоговый, полушаговый и микрошаговый, кратко рассмотрим каждый из них:

1. Полношаговый режим— это режим в котором в один момент включена только одна фаза, и точки равновесия ротора совпадают с точками равновесия у незапитанного двигателя, основной недостаток этого режима в том, что в один момент времени используется только одна фаза, а вследствие этого уменьшается момент двигателя и количество точек позиционирования. Для реализации такого типа управления на МК, необходимо с линии порта ввода/вывода МК в один момент подавать требуемый уровень сигнала на одну фазу двигателя. Все сигналы управления можно свести в таблицу 11.1, в которой стрелкой указан «север». Так же стоит учитывать, что вид таблицы может отличаться в зависимости от того как подключены обмотки двигателя к МК.

*Таблица 11.1.* Последовательность коммутации фаз (полношаговый режим)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| A | 1 | 0 | 0 | 0 |
| B | 0 | 0 | 1 | 0 |
| C | 0 | 1 | 0 | 0 |
| D | 0 | 0 | 0 | 1 |

1. Полушаговый режим – это режим отличный от полношагового тем, что каждый второй шаг запитаны две фазы, в результате угловое перемещение ротора составляет половину угла шага полношагового режима, что улучшает точность позиционирования.

*Таблица 11.2.* Последовательность коммутации фаз (полушаговый режим)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |
| A | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| B | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| C | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| D | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

Последовательность коммутации фаз (с теми же допущениями что и в полношаговом режиме) для полушагового режима приведена в таблице 11.2.

3. Микрошаговый режим – режим в котором величина шага уменьшается еще больше, благодаря чему это режим обеспечивает еще более точное позиционирование и значительно меньшие вибрации при работе вплоть до нулевой частоты. Принцип работы данного режима в том, что две обмотки двигателя запитываются разными по величине токами в следствие чего точка равновесия ротора смещается. Таким образом ротор можно зафиксировать в любой произвольной позиции. Программная реализация этого метода в настоящей работе рассматриваться не будет.

### *1.2 Программная реализация полушагового режима работы*

Рассмотрим программу управления шагового двигателя в полушаговом режиме, которая реализует возможность вращения шагового двигателя с определенной частотой, как в прямом, так и в обратном направлении с возможностью переключения направления вращения в ходе работы двигателя, а также позволяет остановить его кнопкой STOP.

В программе будут задействован таймер и его прерывание по переполнению для задания частоты вращения двигателя. Последовательность коммутации фаз записана во flash-память по адресу SM\_mask:

SM\_mask:

.db 0b00001110,0b00001100, 0b00001101, 0b00001001, 0b00001011, 0b00000011, 0b00000111, 0b00000110, 0b00001110,0

Рассмотрим переменные и константы введённые для удобства понимания исходного кода и программирования:

//Определение переменных (директива .def)

.def TMP = R16 //Временная переменная

.def CNT = R17 //Счетчик указатель

.def FR\_FLAG = R18 //Флаг направления вращения

//Определение констант (директива .equ)

.equ CNT\_MAX = 8 //Для определения переполнения разряда

.equ F\_FLAG\_EN = 1 //Прямой ход

.equ R\_FLAG\_EN = 2 //Реверс

*Листинг 11.1* Инициализация переменных и констант

Как видно из листинга 11.1, в программе будут использоваться три переменные, временная переменная TMP, переменная CNT в которой будет храниться указатель на конкретный элемент массива последовательности коммутации фаз и переменная FR\_FLAG, которая может принимать значения F\_FLAG\_EN и R\_FLAG\_EN, если двигатель вращается в прямом и обратном направлении соответственно. Константа CNT\_MAX служит для ограничения размера массива SM\_mask.

Подробное рассмотрение настройки периферии, портов/ввода вывода и обработчиков кнопок опустим, отметив что в данной программе фазы двигателя коммутируются низким уровнем с линий портов ввода/вывода МК. Для задания частоты вращения используется таймер TIM0 работающий с предделителем 1024, таким образом прерывание по переполнению (TIM0OVF) будет срабатывать с частотой приблизительно равной 4 Гц (частота МК 1МГц), а частота вращения двигателя с полным шагом 90o составит примерно 0.5 об/с. Такие параметры выбраны для более наглядного представления работы двигателя в симуляторе, и для использования в практических задачах могут быть легко пересчитаны и изменены.

Теперь рассмотрим само прерывание по переполнению (TIM0OVF) в котором и происходит управление коммутацией фаз двигателя:

TIM0OVF:

MOTOR\_CONTROL: //Определение режима работы

CPI FR\_FLAG,R\_FLAG\_EN //Флаг реверса поднят?

BREQ SM\_REVERSE //Да, переходим к SM\_REVERSE

//Прямое вращение двигателя

SM\_FORWARD:

ADD ZL, CNT //Считываем из памяти значение

LPM

OUT PORTD,R0 //Выводим в порт

INC CNT //Увеличиваем

CPI CNT,CNT\_MAX //Достигли максимума?

BREQ RES\_CNT //Сбрасываем счетчик

RETI

//Реверс (аналогично прямому ходу)

SM\_REVERSE:

ADD ZL, CNT

LPM

OUT PORTD,R0

DEC CNT

CPI CNT,0

BREQ RES\_CNTR

CPI CNT,-1 //На случай если вылезли за границу

BREQ RES\_CNTR

RETI

//Сброс счетчика прямого хода

RES\_CNT:

LDI CNT,0

RETI

//Сброс счетчика реверса

RES\_CNTR:

LDI CNT,CNT\_MAX

RETI

Листинг. 11.2 Коммутация фаз двигателя в прерывании

В листинге 11.2 первоначально определяется ход вращения, прямой или обратный, сравнением значения, хранящегося в регистре FR\_FLAG и константой-флагом указывающей на реверс (R\_FR\_FLAG), в зависимости от результата сравнения, происходит переход по одной из меток. В случае вращения в прямом направлении (SM\_FORWARD) сначала из массива достается новая маска коммутации, затем она выводится в порт ввода/вывода, увеличивается и проверяется на адекватность, в случае если при увеличении произошел выход за пределы массива, указатель сбрасывается в 0 по метке RES\_CNT. Реверсивное вращение двигателя обрабатывается аналогично, за исключением того, что массив считывается не с начала в конец, а наоборот и сброс счетчика CNT выполняется не в 0, а в его максимальное значение CNT\_MAX.

***2. Управление сервоприводом***

### *2.1 Способы управления сервоприводами*

Сервопривод — это в общем случае привод с управлением через отрицательную обратную связь, которая позволяет точно управлять параметрами движения. Если рассматривать термин более узко, то это электропривод с обратной связью по положению, что позволяет использовать его в задачах точного позиционирования.

Основные преимущества сервопривода перед шаговым двигателем это:

1. Из-за конструктивных особенностей сервопривод в отличии от шагового двигателя может быть значительно мощнее и быстроходнее.

2. Наличие обратной связи гарантирует более высокую точность.

3. Меньшие затраты энергии в работе.

Из недостатков основных недостатков можно отметить:

1. Более сложная составная конструкция двигателя, включающая в себя, датчик и блок управления.

2. Более сложная программа управления.

3. Более высокая стоимость.

Рассмотрим управление одним из самых распространённых типов сервопривода, привода, удерживающего заданный угол поворота.

Управляется сервопривод импульсами постоянной частоты (обычно 50Гц) и переменной ширины, то есть положение двигателя зависит от длины подаваемых на него импульсов. Для примера (рисунок 11.3) рассмотрим двигатель, работающий в диапазоне от 0o (αmin) до 180o(αmax), причем минимальному положению 0о соответствуют импульсы длительностью 1 мс (Hmin), среднему положению 90о импульсы длительностью 1.5 мс, а максимальному (180о) импульсы длительность 2 мс (Hmax).



*Рис 11.3.* Принцип управления сервоприводом

В реальности как диапазон вращения, так и длины управляющих импульсов могут отличаться, и должны быть взяты из документации, так же стоит заметить, что даже в рамках одной модели сервоприводов может существовать некоторая погрешность и если она превышает максимально возможную, то двигатель необходимо откалибровать, например, экспериментально подобрать именно тот диапазон, который характерен для имеющегося привода.

Для управления сервоприводами используется режим PDM (pulse duration modulation) в котором при постоянной частоте импульсов меняется только их скважность.

### *2.2 Программная реализация управления сервоприводом*

Такой режим работы можно легко организовать на базе одного из режимов ШИМ МК. Данный режим в документации имеет номер 14 и в дальнейшем будем называть его именно так. Суть работы данного режима заключается в том, что счетчик ШИМ будет сбрасываться при достижении определенного значения (ICR1) и соответственно работать с необходимой нам частотой. Длина импульсов будет задаваться в регистре OCR1. То есть мы получим требуем нам тип модуляции (рисунок 11.3). Рассчитаем эти значение с учётом того что частота работы МК 2МГц, а предделитель таймера TIM1 равен 8:

Где fcpu – частота МК, fpwm – требуемая частота ШИМ, N – предделитель, TOP – значение необходимое для записи в регистр ICR1.

Тогда:

Рассчитаем константу, соответствующую углу поворота в 1о:

Тогда длина импульса Hт, соответствующая требуемому углу поворота αтреб, будет равняться:

С учетом того что период работы таймера TIM1 составляет:

Значение, которое необходимо записать в регистр OCR1 будет равно:

Рассчитаем и внесем в таблицу 11.3 значения длин импульсов от 0о до 180о с шагом 45о:

*Таблица 11.3.* Рассчитанные значения OCR1A

|  |  |  |
| --- | --- | --- |
| Требуемый угол | Длина импульса | Значение в OCR1A |
| 0 | 1 | 250 |
| 45 | 1.25 | 312 |
| 90 | 1.5 | 375 |
| 135 | 1.75 | 437 |
| 180 | 2 | 500 |

Разберем программу, которая при нажатии на кнопку поворачивает двигатель на угол 45о, а после достижении максимального значения угла, возвращает двигатель в начальное положение. Подробно разберем две особенности: реализацию нажатия кнопки с помощью прерывания, и настройка требуемого режима ШИМ.

Зададим в программе (листинг 11.3) две переменные TMP и CNT, первую будем использовать для хранения различных временных значений, а во второй хранить текущее положение двигателя. Значения длин импульсов для углов поворота из таблицы 11.3 запишем в явном виде, а значение константы (PWM\_const) для регистра ICR1 рассчитаем в формуле:

//Определение переменных (директива .def)

.def TMP = R16 //Временная переменная

.def CNT = R17 //Счетчик

//Определение констант (директива .equ)

.equ XTAL = 2000000 //Частота контроллера

.equ T1\_PSK = 8 //Предделитель таймера

//Расчет констант для требуемой частоты ШИМ и необходимой скважности импульсов

.equ PWM = 50 //Необходимая частота ШИМ

.equ PWM\_const = (XTAL-T1\_PSK\*PWM)/(T1\_PSK\*PWM) //Константа для частоты

//Расчет констант для определения угла вращения

.equ RP0 = 250 //Вычисляется как длина импульса для угла поворота (в мкс)

.equ RP45 = 312 //разделить на период работы таймера (в мкс)

.equ RP90 = 375 //период работы таймера равен 1/(XTAL/T1\_PSK)

.equ RP135 = 437

.equ RP180 = 500

*Листинг 11.3.* Определение переменных и констант

Настройка (листинг 11.4) портов ввода/вывода и внешнего прерывания для кнопки заключается в настройке линии OC1A на выход (выход ШИМ), и настройки регистров MCUCR и GIMSK, отвечающих за тип внешнего прерывания (1<<ISC01) и разрешения прерывания (1<<INT0) соответственно.

//Настроим линию OC1A на выход

LDI TMP, 0b000000010

OUT DDRB,TMP

//Настройка внешнего прерывания INT0 (для обработки нажатия на кнопку)

LDI TMP, 1<<ISC01 //Прерывание по спадающему фронту

OUT MCUCR, TMP

LDI TMP, 1<<INT0 //Разрешение прерывания

OUT GIMSK,TMP

*Листинг 11.4.* Настройка линии OC1A и внешнего прерывания

Теперь рассмотрим настройку таймера TIM1 в режим ШИМ №14:

//Настройка таймера TIM1 для работы в режиме ШИМ (режим 14, datasheet p.98)

//Данный режим сбрасывает ШИМ при достижении определенного значения что

//позволяет добиться необходимой частоты его работы 50Гц

LDI TMP,HIGH(PWM\_const) //Записываем ранее вычесленное значени

OUT ICR1H,TMP //Не забывая о порядке записи LDI TMP,LOW(PWM\_const)

OUT ICR1L,TMP

LDI TMP, 1<<COM1A1|1<<WGM11 //Настроим ШИМ WGM11,WGM12,WGM13 - режим 14

OUT TCCR1A, TMP //COM1A1 - сброс линии при совпадении (non-inverting mode)

LDI TMP, 1<<WGM12|1<<CS11|1<<WGM13 //CS11 - предделитель 8

OUT TCCR1B, TMP

*Листинг 11.5.* Настройка TIM1 в режим ШИМ №14

В листинге 11.5 видно, что сначала в регистр ICR1 записывается ранее вычисленное значение PWM\_const определяющее частоту ШИМ, а затем согласно документации, устанавливаются нужные биты регистров настройки TCCR1A и TCCR1B:

1. COM1A1 – включает сброс при совпадении, то есть сигнал на линии OC1A изменит уровень с высокого на низкий в момент когда таймер досчитает до значения равного значению в OCR1A, таким образом будет выдержана требуемая длина импульса.
2. WGM11, WGM12, WGM13 – включает режим ШИМ сбрасывающий таймер в том случае, если таймер досчитал до значения равного ICR1, что дает постоянную частоту ШИМ.
3. С11 – включает таймер с предделителем 8.

Вся работа с двигателем проводится в прерывании, возникающем при нажатии кнопки. Рассмотрим его в сокращённом виде (опустив промежуточные положения двигателя):

INT0I:

INC CNT

CPI CNT,2

BREQ ROTATE90

CPI CNT,3

BREQ ROTATE135

CPI CNT,4

BREQ ROTATE180

CPI CNT,5

BREQ ROTATE0

//Поворот в 45

ROTATE45:

//Запишем необходимые значения в регистры сравнения (не забывая про порядок записи!)

LDI TMP,HIGH(RP45)

OUT OCR1AH,TMP

LDI TMP,LOW(RP45)

OUT OCR1AL,TMP

RETI

...

//Поворот в 0

ROTATE0:

//Запишем необходимые значения в регистры сравнения (не забывая про порядок записи!)

LDI TMP,HIGH(RP0)

OUT OCR1AH,TMP

LDI TMP,LOW(RP0)

OUT OCR1AL,TMP

CLR CNT

RETI

*Листинг 11.5.* Управление углом поворота двигателя

Из листинга 11.5 видно, что выбор угла поворота производится на основании значения переменной CNT. После каждого нажатия на кнопку переменная инкрементируется, а затем с помощью команд сравнения и условного перехода происходит переход к метке по которой устанавливается новое значение длины импульсов (OCR1A) однозначно определяющее положение двигателя.

В бесконечном цикле программы не выполняется никаких действий, просто происходит ожидания прерывания.

**Задания для самостоятельной работы**

# ЗАКЛЮЧЕНИЕ

# СПИСОК ЛИТЕРАТУРЫ

ЗАМЕЧАНИЯ

1. Если что-то не работает в симуляторе, совсем не обязательно, что это не будет работать в реальном устройстве, обратное утверждение тоже верно. Подходите к использованию симуляторов разумно.
2. Эта работа в первую очередь о программировании, поэтому многие технические подробности устройств несущественные для программирования опущены, их всегда можно найти в документации на необходимые устройства.
3. Всегда сверяйтесь с документацией на используемый МК, какие-то вещи, использующиеся в предыдущем поколении МК, могли изменится, например, добавиться новые функции или регистры. Или изменится имена регистров.
4. Для программирования МК необходимо знать ассемблер, понимание работы МК здорово упростит жизнь при отладке больших программ, написанных на языке высокого уровня.
5. Не стоит сразу пытаться написать что-то серьезное, сначала поморгайте светодиодом.
6. Схемы устройств сознательно не приводятся, они достаточно просты и их всегда можно посмотрев открыв пример в Proteus или начертив самому.
7. Описание работы программ зачастую идет без указания используемых портов, они очевидны из примеров программ, и схем в Proteus.

СООТВЕТСТВИЕ ГЛАВ И ПРИМЕРОВ ПРОГРАММ

(С КРАТКИМ ОПИСАНИЕМ) И ДРУГИЕ ЗАМЕЧАНИЯ

Первая цифра соответствует главе, вторая цифра соответствует какой по порядку в данной главе рассматривается программа:

5.1. IOPORT - пример работы с портами ввода-вывода (линейка светодиодов и кнопки);

5.2. 7SEG - работа с семисегментным дисплеем и данными во Flash-памяти (вывод на дисплей чисел в сторону увеличения при нажатии кнопки);

5.3. IO\_Interrupt – использование внешних прерываний (для обработки нажатия на кнопку);

6. Timer - пример использования аппаратного таймера и прерываний (одноразрядный секундомер);

7. UART - Пример вывода сообщения в терминал по протоколу UART и отправка данных из терминала в МК;

8.1. A\_comp – пример работы аналогового компаратора, оповещение о падении напряжения ниже допустимого уровня;

8.2. ADC - пример работы АЦП (снятие напряжение с потенциометра);

9. EEPROM – Запись и чтение из EEPROM;

10.1. Dyn\_Indication – Динамическая индикация (секундомер на 4 дисплея);

10.2. Keybord – пример работы с матричной клавиатурой;

11.1. SM\_Control – пример управления шаговым двигателем;

11.2.PWM\_Motor – пример управления сервоприводом;

Программы написаны в AVR STUDIO 6.0, в более раннюю версию могут быть перенесены созданием нового проекта и копированием в него содержимого файла \*.asm. Для симуляции использован Proteus 7.1 SP0.

Список команд и общая информация в удобном виде в файле Asm\_AVR\_rus.pdf.

В комментариях к программам присутствуют грамматические ошибки.

В процессе написания были использованы материалы:

1. Ревич Ю. - Практическое программирование микроконтроллеров Atmel AVR на языке ассемблера.
2. Евстифеев А.В. Микроконтроллеры AVR семейств Tiny и Mega фирмы ATMEL. Электронное издание.
3. Емельянов А.В., Шилин А.Н. Шаговые двигатели: Учебное пособие
4. Вольфганг Трамперт. Измерение, управление и регулирование с помощью AVR микроконтроллеров
5. Сайт: [http://wiki.amperka.ru:сервоприводы](http://wiki.amperka.ru/%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%BE%D1%82%D0%B5%D1%85%D0%BD%D0%B8%D0%BA%D0%B0:%D1%81%D0%B5%D1%80%D0%B2%D0%BE%D0%BF%D1%80%D0%B8%D0%B2%D0%BE%D0%B4%D1%8B)
6. Сайт: <http://easyelectronics.ru/>
7. Сайт: [http://extremeelectronics.co.in](http://extremeelectronics.co.in/avr-tutorials/servo-motor-control-by-using-avr-atmega32-microcontroller/)
8. Документация на МК AVR AtMega 8