МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное
учреждение высшего образования

«ПОВОЛЖСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

ПРИМЕНЕНИЕ МИКРОКОНТРОЛЛЕРОВ В РАДИОТЕХНИЧЕСКИХ И БИОМЕДИЦИНСКИХ СИСТЕМАХ

Учебное пособие

Йошкар-Ола
ПГТУ
2018

УДК 621.391.6:621.396.96(07)
ББК 32.95я73
П 62

Авторы:
А. А. Роженцов, А. А. Баев, Ю. Е. Гарипова, С. А. Охотников

Рецензенты:
Дедов А. Н., канд. техн. наук, доцент кафедры радиотехники и связи, декан радиотехнического факультета ПГТУ
Парсаев Н. В., канд. техн. наук, доцент кафедры прикладной математики и информатики Марийского государственного университета

Печатается по решению
редакционно-издательского совета ПГТУ

П 62 **Применение микроконтроллеров в радиотехнических и биомедицинских системах**: учебное пособие / А. А. Роженцов [и др.]. – Йошкар-Ола: Поволжский государственный технологический университет, 2018. – 172 с.
ISBN 978-5-8158-1992-4

Учебное пособие включает описание шести практических работ и содержит теоретические сведения о микроконтроллерах AVR, а также рекомендации по разработке устройств на их базе.

Для студентов специальности 11.05.01 «Радиоэлектронные системы и комплексы», направлений подготовки 11.04.01 «Радиотехника», 12.03.04, 12.04.04 «Биотехнические системы и технологии», изучающих дисциплины «Цифровые устройства и микропроцессоры», «Микропроцессорные системы», «Проектирование встраиваемых систем на микроконтроллерах», 11.03.02 «Инфокоммуникационные технологии и системы связи», изучающих дисциплину «Системы радиочастотной идентификации».

УДК 621.391.6:621.396.96(07)
ББК 32.95я73

ISBN 978-5-8158-1992-4

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Издание содержит шесть разделов, охватывающих широкий круг вопросов, связанных с разработкой устройств на микроконтроллерах. Первый раздел посвящен получению базовых навыков разработки устройств на микроконтроллерах, второй – изучению стандартных последовательных интерфейсов микроконтроллера, в третьем разделе исследуется аналого-цифровой преобразователь микроконтроллера, в четвертом – изучается возможность применения жидкокристаллических индикаторов в устройствах на микроконтроллерах, в пятом разделе описан подход к использованию интерфейса 1-wire, в шестом разделе изучается подключение RFID считывателя к микроконтроллеру.

Основная *цель учебного пособия* – привить студентам навыки разработки и отладки устройств на микроконтроллерах, в том числе научить их:

– разрабатывать программы для микроконтроллеров в среде AVR Studio;

– выполнять проверку работоспособности и отладку написанных программ в среде Proteus;

– пользоваться справочной информацией по элементной базе, в том числе, на иностранном языке;

– применять стандартные цифровые интерфейсы для обмена данными между различными устройствами;

– использовать различные способы ввода и вывода данных в аналоговой и цифровой формах, методы индикации информации в форме, удобной для человека.

Выполнение практических работ позволяет закрепить и углубить полученные на лекционных занятиях теоретические знания и в дальнейшем применять их в курсовом и дипломном проектировании.

Помочь в усвоении теоретических сведений, систематизации изученного материала и успешном выполнении практических заданий призваны контрольные вопросы по каждой теме и список рекомендуемой литературы, где можно получить дополнительную информацию.

СПИСОК СОКРАЩЕНИЙ

АЦП – аналого-цифровой преобразователь
ДНЛ – дифференциальная нелинейность
ЖКИ – жидкокристаллический индикатор
ИНЛ – интегральная нелинейность
ИОН – источник опорного напряжения
МК – микроконтроллер
ОЗУ – оперативное запоминающее устройство
ПЗУ – постоянное запоминающее устройство
ПО – программное обеспечение
ЦАП – цифро-аналоговый преобразователь
ЦПУ – центральное процессорное устройство
ШИМ – широтно-импульсная модуляция
CRC – (англ. Cyclic Redundancy Check) – циклический избыточный код
EEPROM – (англ. Electrically Erasable Programmable Read-Only Memory) – электрически стираемое перепрограммируемое ПЗУ (ЭСППЗУ)
GPIO – (англ. General Purpose Inputs/Outputs) – выводы общего назначения
IDE – (англ. Integrated Development Environment) – интегрированная среда разработки
PWM – (англ. Pulse Width Modulation) – широтно-импульсная модуляция (ШИМ)
RAM – (англ. Random Access Memory) – запоминающее устройство с произвольной выборкой (ЗУПВ)
ROM – (англ. Read Only Memory) – постоянное запоминающее устройство (ПЗУ)
SPI – (англ. Serial Peripheral Interface) – последовательный периферийный интерфейс
UART – (англ. Universal Asynchronous Receiver-Transmitter) – универсальный асинхронный приёмопередатчик (УАПП)

ВВЕДЕНИЕ

Изделия на базе микроконтроллеров в настоящее время являются одними из наиболее востребованных и распространенных цифровых устройств. Они применяются в приборах бытового назначения, автоэлектронике, системах управления производственными процессами, медицинских приборах, контрольно-измерительной и связной аппаратуре и т.п.

По объему рынка в денежном выражении микроконтроллеры практически не уступают микропроцессорам общего назначения, а в количественном выражении – многократно превосходят их.

В связи со всем вышесказанным овладение навыками разработки устройств на микроконтроллерах является необходимым условием подготовки современного радиоинженера и инженера в области биомедицинской техники.

В настоящее время на рынке микроконтроллеров присутствует несколько крупных производителей.

Среди отечественных разработчиков значительное распространение получила продукция фирмы Atmel. Это можно объяснить доступностью ее изделий, широкой номенклатурой представленных микроконтроллеров, их высокими функциональными характеристиками.

Немаловажным фактором является возможность использования бесплатного программного обеспечения для разработки и отладки программ на этих контроллерах, а также распространенность и относительно невысокая стоимость аппаратных отладочных средств.

В данном учебном пособии, имеющем практическую направленность, упор сделан на разработку схем и программ на базе микроконтроллера ATmega328. При этом особое внимание уделяется вопросам создания и применения устройств, базирующихся на использовании стандартных и широко распространенных цифровых интерфейсов.

Поскольку общие принципы построения большинства котроллеров являются достаточно универсальными, то, благодаря этому, полученные обучающимися навыки могут быть эффективно использо-

ваны при разработке устройств на других типах контроллеров различных производителей.

Особое внимание в настоящем учебном пособии уделено применению отладочных средств на базе пакета Proteus. Поскольку данный пакет обеспечивает возможность анализа работоспособности схемы без предварительного макетирования, студенты могут выполнять значительный объем работ дистанционно или в рамках самостоятельной работы. Это является одной из отличительных особенностей данного учебного пособия.

ОБЩИЕ РЕКОМЕНДАЦИИ К ПОДГОТОВКЕ И ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

Для успешного освоения теоретического материала и выполнения практических работ студенты должны владеть базовыми знаниями в области дискретной математики, информатики и программирования, электроники, цифровых устройств. Также необходимо иметь общее представление о принципах функционирования микропроцессорных систем и о структуре микроконтроллера.

Подготовка к практическим работам

При подготовке к выполнению практических работ необходимо:

– изучить теоретические сведения по каждой теме, приведенные в данном издании;

– изучить используемую элементную базу, опираясь на его техническое описание (datasheet) от производителя;

– получить задание от преподавателя на разработку и моделирование схемы;

– подготовить ***индивидуальный*** конспект отчета, содержащий название и цель работы, необходимые для защиты теоретические сведения, описание принципиальной схемы, текст программы;

– подготовить ответы на теоретические вопросы по каждой теме, приведенные в данном пособии;

– получить у преподавателя допуск к выполнению практической работы.

Выполнение практических работ

Для выполнения практической работы необходимо:

– собрать модель принципиальной схемы в среде Proteus и на отладочной плате;

– разработать программу для микроконтроллера;

– произвести проверку работоспособности и, при необходимости, отладку программы;

– включить в отчет итоговую версию программы и результаты моделирования ее работы в среде Proteus или снимки отладочной платы;

– сделать выводы по результатам проделанной работы.

Критерии оценки знаний

Для получения оценки «удовлетворительно» необходимо выполнить практическую часть работы, знать и уметь объяснить назначение переменных, регистров и алгоритмы работы программы.

Для получения оценки «хорошо» необходимо выполнить задание своего варианта, а также дополнительное задание преподавателя. Продемонстрировать умение свободно использовать справочные материалы. Показать умение работать с отладочной платой и использовать осциллограф для контроля сигналов.

Для получения оценки «отлично» необходимо выполнить дополнительное усложненное задание преподавателя. Уметь свободно использовать справочные материалы. Исходный код должен содержать исчерпывающие комментарии, иметь строгую стилизацию, имена функций и переменных должны полностью отражать выполняемую задачу. Работа исходного кода подтверждается исполнением на отладочной плате с проверкой на осциллографе.

1

Базовые элементы микроконтроллера семейства ATMEGA

Теоретические сведения

Микроконтроллер (МК) – микросхема, предназначенная для управления электронными устройствами. МК представляет собой однокристальную ЭВМ, работающую согласно загруженной в нее программе. В персональной ЭВМ можно выделить такие блоки, как центральный процессор, оперативная память, энергонезависимая память, устройства ввода/вывода информации и некоторые другие. Аналогично МК также содержит эти же компоненты в упрощенном виде.

Существует множество типов контроллеров, каждый из которых оптимально подходит для применения в какой-либо области. Эти контроллеры могут отличаться архитектурой процессорного ядра, объемом и типом встроенной памяти, набором периферии и т. д.

Рассмотрим микроконтроллеры семейства *AVR*. Это микроконтроллеры производства фирмы Atmel, 1996 года разработки, одни из наиболее распространенных 8-битных контроллеров. Термин «8-битный» означает, что вычислительное ядро контроллера работает с данными, размерность которых не превышает 1 байт. Семейство AVR отличается большим разнообразием версий микроконтроллеров, сильно отличающихся по объемам памяти, набору периферийных устройств, потребляемой мощности и прочим параметрам.

Рассмотрим в качестве примера контроллер *ATmega328* и его основные параметры:

- максимальная тактовая частота – до 20 МГц;

- напряжение питания – 1,8…5,5 В;
- объем Flash-памяти – 32 Кбайт;
- объем ОЗУ – 2 Кбайт;
- количество выводов общего назначения – 32.

Контроллер выпускается в различных корпусах, таких как TQFP, PDIP и MLF. Пример расположения выводов контроллера в корпусе TQFP приведен на рисунке 1.

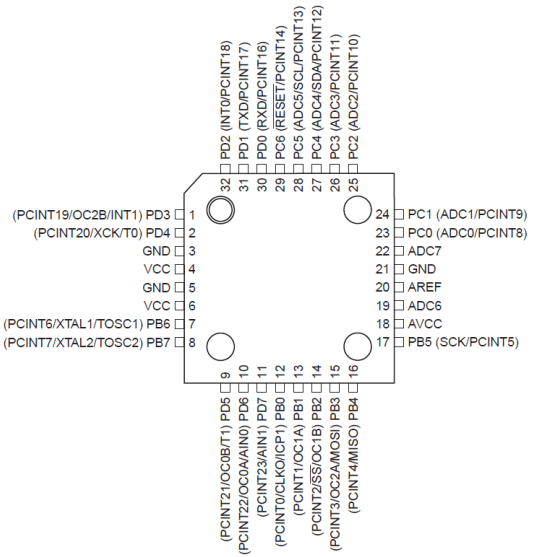


Рисунок 1 – Расположение выводов ATmega328 в TQFP-корпусе

Каждый из приведенных параметров характеризует одно из основных свойств контроллера, по которым разработчик выбирает конкретный контроллер из множества имеющихся на рынке решений. Разумеется, данный список не является исчерпывающим, и в зависимости от ситуации на выбор конкретного контроллера могут влиять и другие факторы.

Максимальная тактовая частота в общем случае может служить характеристикой быстродействия контроллера, поэтому при разработке быстродействующих устройств предпочтение отдается контроллерам с высокой тактовой частотой.

Flash-память (флэш-память) – энергонезависимая память команд контроллера, и ее содержимое сохраняется при отключении питания. В ней хранится программа, согласно которой работает контроллер. Таким образом, объем флэш-памяти позволяет судить о максимальном размере программного кода, который может быть загружен в контроллер.

ОЗУ – энергозависимая память контроллера, ее содержимое стирается при отключении питания, однако время доступа к ней гораздо меньше, чем к Flash, кроме того она гораздо более устойчива к постоянной перезаписи, потому используется контроллером для хранения переменных и прочих изменяемых данных во время работы. Таким образом, размер ОЗУ характеризует объем данных, которыми может одновременно оперировать контроллер.

Выводы общего назначения (GPIO) предназначены для обмена данными между контроллером и внешними устройствами. В общем случае работа любого контроллера сводится к тому, чтобы вовремя установить на соответствующих выводах высокий или низкий уровень напряжения или принять с них данные. Понятие «вовремя» определяется требованиями к устройству, а обеспечивается записанной в память контроллера программой.

Для удобства работы выводы группируются в порты. Микроконтроллер ATmega328 имеет в своем составе 3 порта, обозначаемых буквами B, C, D. В состав каждого порта входит 8 выводов, каждый из которых нумеруется от 0 до 7. Таким образом, каждый вывод имеет свое имя, например PB2 – P (Port) B, вывод 2. Такая группировка позволяет за один такт изменить состояние не одного, а целой группы выводов.

Следует обратить внимание на то, что порт C у ATmega328 неполный – у него нет вывода PC7. Это связано с тем, что микроконтроллеры могут выпускаться в разных корпусах, которые способны отличаться количеством и расположением выводов, однако сам чип микросхемы остается тем же.

Также в состав портов не входят выводы AREF, VCC и GND – эти выводы необходимы для подачи питания на контроллер.

Для работы контроллера необходимо лишь занести в его Flash-память код необходимой программы и подать на него питание. После этого контроллер начинает последовательно читать и выполнять содержащиеся в памяти команды.

Для размещения программного кода в памяти контроллера используются различные вспомогательные устройства, называемые программаторами. Сами программаторы подключаются к персональному компьютеру через USB, LPT или COM порт, а целевой контроллер подключается к программатору. Запись в память контроллера может производиться по интерфейсу SPI, JTAG или UART. Однако запись по UART может быть выполнена только в том случае, если в контроллере уже записана специальная программа-загрузчик, которая запускается при старте контроллера и цель которой заключается в приеме программного кода по UART и размещении его в другой области памяти. Здесь есть некоторое противоречие: чтобы записать программу в контроллер, в нем уже должна быть программа. Данное противоречие разрешается тем, что загрузчик записывается в контроллер на этапе его производства.

Запись по SPI или JTAG лишена такого недостатка, однако программаторы с такими интерфейсами сложнее и дороже. Дополнительным плюсом JTAG интерфейса является то, что он позволяет производить отладку программного кода «в железе», то есть записанная в контроллер программа может быть в любой момент остановлена, пройдена по шагам, можно просмотреть содержимое памяти и регистров контроллера. Такой подход существенно облегчает отладку программного кода. Недостатком является то, что не все контроллеры оснащаются JTAG интерфейсом.

В контроллерах AVR имеется особая область памяти, изменение содержимого которой возможно только при программировании контроллера. Эта область называется *Fuse bits* (фьюз-биты) – область размером 4 байта, отвечающая за начальную глобальную конфигурацию. Эти биты определяют, с каким задающим тактовым генератором (внешним/внутренним) должен работать контроллер, делить частоту генератора на коэффициент или нет, использовать ножку сброса как сброс или как дополнительный порт ввода-вывода, коли-

чество памяти для загрузчика и другое. У каждого контроллера свой набор фьюз-бит. У контроллера ATmega328 имеются следующие фьюз-биты:

CLKDIV8 – бит, включающий предварительное деление частоты кварцевого (или иного имеющегося) тактового генератора на 8. То есть при включенном этом бите и применении кварцевого резонатора на 8 МГц реальная тактовая частота МК составит 1 МГц.

CKOUT – бит, разрешающий вывод тактовой частоты на один из выводов МК (для тактирования других устройств).

RSTDISBL – бит, позволяющий сконфигурировать вывод сброса МК как часть обычного порта ввода-вывода. Этот бит имеется только в тех МК, у которых вывод аппаратного сброса RESET совмещен с одним из портов ввода-вывода. Ошибочная установка этого бита может отключить RESET и сделать невозможной программирование по ISP. «Оживить» МК с установленным RSTDISBL можно только параллельным программатором, причем это относится не ко всем микроконтроллерам.

WDTON – бит, после установки которого сторожевой таймер WDT включается сразу после подачи питания и не может быть отключен программно. Если бит не установлен, то включением и отключением WDT можно управлять программно. Сторожевой таймер (WDT, watchdog timer) – внутренний таймер микроконтроллера, который сбрасывает его при переполнении своего счётного регистра. Служит для автоматического сброса контроллера в случае его зависания.

BOOTRST – бит, определяющий адрес, с которого будет начато исполнение программы после сброса – если бит установлен, то начало программы будет не с адреса 0000h (как обычно), а с адреса области загрузчика *(Boot Loader*).

CKSEL – группа из 4 бит, комбинация которых определяет тип и частоту работающего тактового генератора. Всего возможно до 16 комбинаций, однако не все определены для всех типов МК. Ошибочная установка комбинации этих битов может сделать МК «мертвым» – он не будет работать в схеме без подачи тактового сигнала на ножку внешнего тактирования.

BOOTSZ – группа из двух бит, определяющих размер области памяти программ, выделяемой для загрузчика (*Boot Loader*). Комбинация этих битов, в частности, определяет точку начала исполнения программы после сброса, если установлен бит BOOTRST**.**

Практическая часть

Для отладки программ для контроллеров, не оснащенных JTAG интерфейсом, применяются методы компьютерного моделирования. Разработаны специализированные программные пакеты, которые позволяют смоделировать поведение электрических схем, в том числе и содержащих программируемые микросхемы. Одним из таких пакетов является Proteus. Данный пакет содержит в своем составе набор программ, позволяющих производить создание принципиальных схем, их моделирование, отладку и разработку печатных плат. В этой работе будет использован симулятор ISIS 7, входящий в данный пакет.

Для работы с симулятором необходимо собрать на рабочем поле схему устройства, работа которого будет моделироваться. Первоначально на рабочем поле размещается компонент контроллера. Это делается кликом правой клавиши мыши на свободном месте, в меню выбирается **Place → Component → From libraries**, где ищется компонент под названием ATmega328, после чего он размещается на поле (рис. 2).

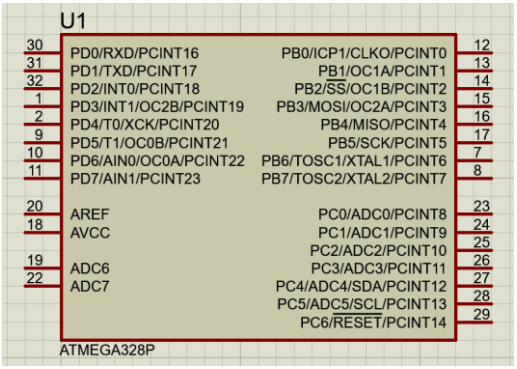


Рисунок 2 – ATmega328 – вид компонента в программном пакете Proteus

Для того чтобы Proteus мог моделировать поведение контроллера, необходимо передать в него hex-файл, содержащий программный код контроллера. В данном файле содержатся двоичные данные в таком виде, что они могут быть записаны и в реальный контроллер.

Для получения данного файла необходимо написать и скомпилировать программу для контроллера. Программа обычно пишется в сторонней среде программирования на языке С или Ассемблер, после чего с помощью компилятора получается необходимый hex-файл.

Одной из популярных интегрированных сред разработки (IDE) является Atmel Studio. Данная IDE включает в себя интегрированный компилятор C/C++, поддерживает инструменты Atmel, совместимые с 8-разрядной AVR архитектурой, обладает удобным редактором кода с подсветкой синтаксиса.

Для того чтобы написать программу для ATmega168 на языке С в программе Atmel Studio, запуститеAtmel Studio, выберите **New project → GCC Executable Project**, не забыв указать имя проекта и директорию (рис. 3).

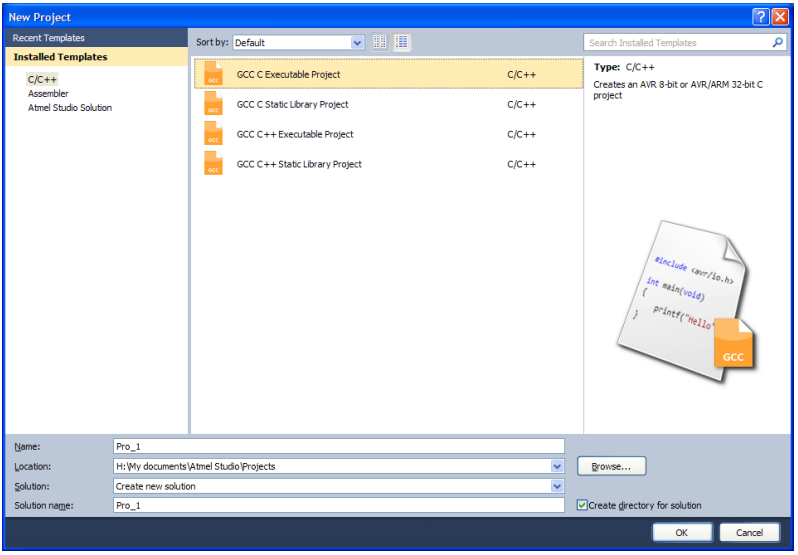


Рисунок 3 – Создание проекта в AtmelStudio

После нажатия **ОК** Atmel Studio предлагает выбрать контроллер, для которого пишется программа, – из списка выбираем ATmega328 (рис. 4).

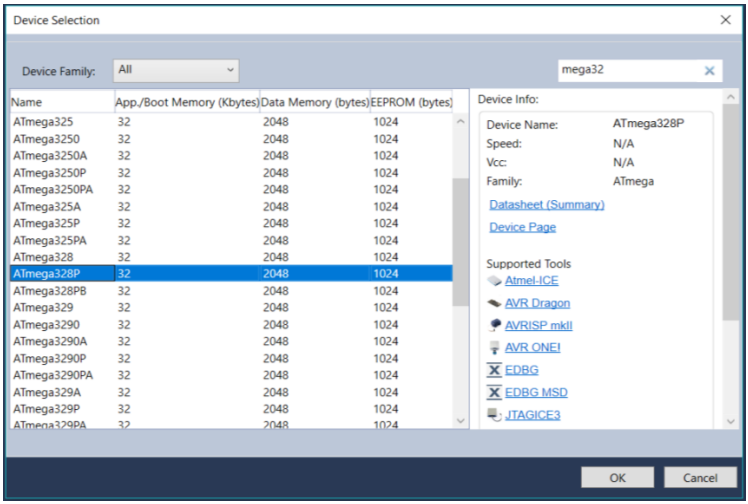


Рисунок 4 – Выбор контроллера

После нажатия **ОК** открывается редактор кода и шаблон программы, состоящий из следующих строк:

```
#include <avr/io.h>

int main(void)
{
    while(1)
    {
     //TODO:: Please write your application code
    }
}
```

Этот шаблон уже можно скомпилировать и записать в контроллер. Однако такая программа не выполняет никаких функций.

Рассмотрим шаблон подробнее. Как и в любой программе на языке С, в коде имеется функция `main`. При старте контроллер начинает выполнение программы с первой команды этой функции.

В приведенном коде единственным наполнением этой функции является бесконечный цикл `while(1)`, который также является стандартным для большинства программ контролеров, так как без него, выполнив содержимое тела `main`, контроллер бы просто остановился за неимением инструкций. Как подсказывает автоматический комментарий, основная часть программы контроллера располагается внутри этого цикла.

Директива `#include <avr/io.h>` подключает к проекту файл `io.h`. Данный файл содержит в себе ссылки на другие файлы, подключаемые автоматически исходя из того, какой контроллер был выбран при создании проекта. В основном в этих файлах регистрам контроллера присваиваются определенные имена, что позволяет программисту с ними работать, иначе любая операция с внутренним содержимым контроллера потребовала бы от человека, занимающегося написанием ПО, прямого указания адресов в памяти контроллера, к которым он обращается. Эти имена являются общепринятыми для конкретного контроллера и приводятся в его техническом описании.

Теперь можно заставить контроллер выполнить какое-либо действие. Пусть к определенному выводу контроллера через токоограничительный резистор подключен светодиод. Тогда, если подать напряжение на этот вывод, светодиод загорится. Для выполнения данного действия необходимо соответствующим образом настроить порты контроллера.

Как уже отмечалось, выводы объединяются в порты, и доступ идет одновременно ко всему порту. С каждым портом связано несколько управляющих регистров, задающих режим его работы. Эти регистры 8-битные, и каждый бит в них связан с соответствующим выводом порта.

DDRx (x – буква названия порта) – данный регистр задает режим работы порта: выводы используются либо как выходы, либо как логические входы. Если в определенном бите данного регистра стоит 1, то соответствующий вывод контроллера работает как логический выход, если 0 – как вход. Например, число $18_{10} = 12_{16} = 10010_2$, записанное в регистр DDRB, настроит выводы PB1 и PB4 как выходы, а все остальные – как входы.

PORTx – функция данного регистра зависит от того, что записано в DDRx. Если соответствующий вывод настроен как выход, то запись 1 в соответствующий бит подаст на этот вывод напряжение. Если вывод настроен на вход, то запись 1 включает подтягивающий к питанию резистор к этому выводу.

PINx – чтение данного регистра позволяет считать информацию с порта, но информативными будут лишь биты, соответствующие выводам, настроенным на вход.

Для моделирования соберем в симуляторе Proteus ISIS следующую схему (рис. 5):

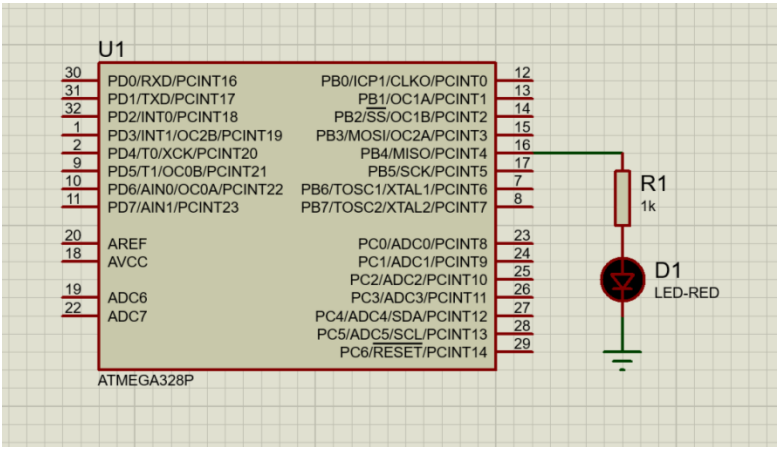


Рисунок 5 – Подключение светодиода

Светодиод подключен к выводу PB4. Исходя из этого, пишем следующий код:

```
#include <avr/io.h>

int main(void)
{
    DDRB = 0b00010000;
    PORTB = 0b00010000;
    while(1){}
}
```

К шаблону добавилось всего 2 строки – в регистре DDRB в бит, соответствующий выводу PB4, записана 1, что настроило вывод как выход, а в регистре PORTB единица в соответствующем бите подала на вывод напряжение. Можно отметить, что эти команды находятся вне цикла while, они будут выполнены только один раз – при старте контроллера. Обычно в этом месте располагают код, назначением которого является первоначальная настройка контроллера.

После того как код написан, необходимо сохранить и скомпилировать проект в Atmel Studio. Это делается из вкладки **Build → Build solution**. В результате в директории проекта появляется директория **Debug**, в которой, среди прочего, находится файл **<название проекта>.hex**. Этот файл и является конечной целью написания программы – в нем содержится написанная программа в виде машинных кодов, понятных контроллеру, и этот файл должен быть помещен в память контроллера.

Поскольку для проверки работы программы используется симулятор Proteus ISIS, то в нем достаточно кликнуть правой кнопкой мыши на символе контроллера → **Правка свойств** и в графе **Program file** указать полученный hex-файл (рис. 6).

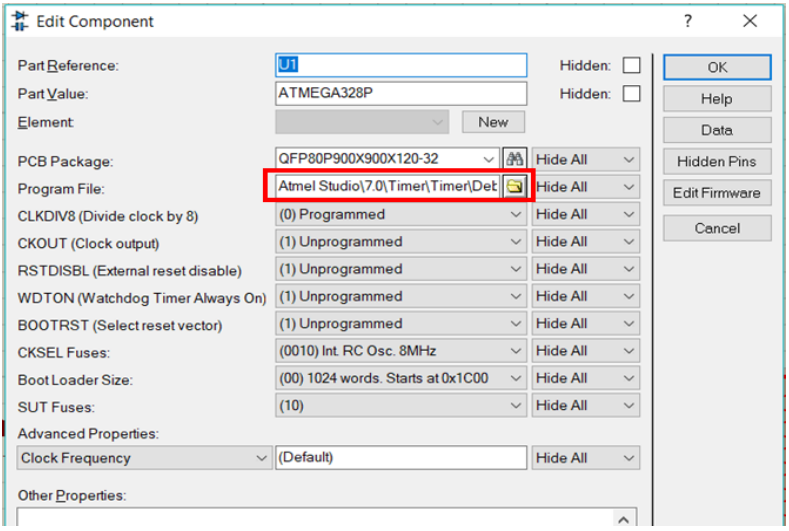


Рисунок 6 – Подключение hex-файла

После этого можно запустить симуляцию, нажав на символ Play в левом нижнем углу экрана. Светодиод должен загореться.

Как видно из текста программы, запись числа в регистры сделана в двоичном коде, однако это совершенно не обязательно. Форма представления данных не важна. Следующие команды идентичны между собой:

```
DDRB = 0b00010000;    // двоичный код
DDRB = 0x10;      // шестнадцатеричный код
DDRB = 16;        // десятичное представление
DDRB = (1 << 4);  // представление в виде сдвига
DDRB = (1 << PB4);    // представление в виде
сдвига
```

Отдельно следует отметить последние два представления. В языке С символами << и >> обозначаются операции битового сдвига влево и вправо соответственно. Так, запись 1<<4 означает сдвиг числа 1 на 4 позиции влево: то есть число 00000001_2 сдвигается 4 раза на 1 бит и получается число 00010000_2.

Во втором представлении используется название вывода PB4. Это наименование определено в файле iomx8.h, который подключен к проекту благодаря io.h. Он содержит строчку:

```
#define PB4 4,
```

которая означает, что при компиляции сочетание PB4 заменяется числом 4.

Подобная замена также может производиться и для других элементов, таких как адреса регистров, битовые маски и т.д. Такой подход очень широко используется при программировании контроллеров, так как не только избавляет программиста от необходимости держать в голове числовые значения, но и делает код более читабельным. Кроме того, отказ от использования конкретных чисел в коде имеет еще одно важное преимущество: он делает код независимым от контроллера, так как при смене контроллера становится достаточно сменить только заголовочные файлы, в которых содержатся конкретные определения используемых символов.

Но чаще всего операции сдвига используются совместно с операторами &= и |=. Эти операции очень важны, так как позволяют

изменять значения конкретных битов в числе. Это важно, когда необходимо изменить, например, значение одного вывода порта, не трогая остальные, при этом их значения могут быть и неизвестны. Например, запись вида

```
PORTB |= 1<<4;
```

может быть расписана как

```
PORTB = PORTB | (1<<4);
```

что означает, что в PORTB будет записано значение PORTB, но побитово сложенное с числом 00010000_2, в результате бит 4 в PORTB станет равным 1, в то время как значение остальных не изменится. По аналогии, чтобы установить конкретный бит в 0 используется запись вида

```
PORTB&= ~(1<<4);
```

Теперь рассмотрим в Proteus ISIS следующую схему (рис. 7):

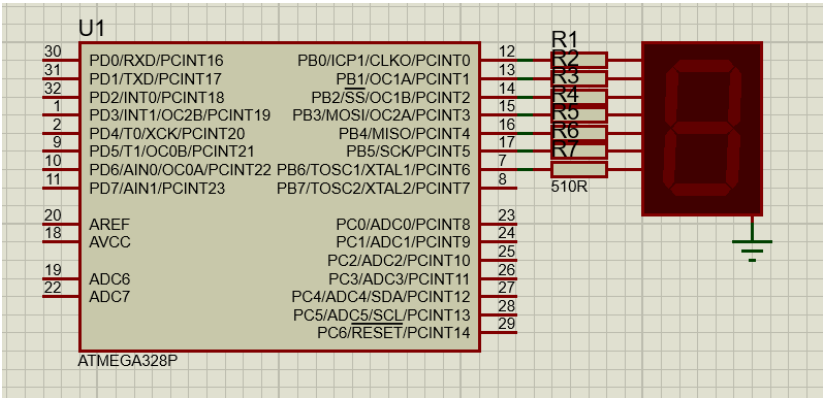


Рисунок 7 – Подключение 7-сегментного индикатора

Здесь к выводам порта B подключен 7-сегментный индикатор. В библиотеках Proteus ISIS этот индикатор располагается в категории **Optoelectronics** под названием **7SEG-COM-CATHODE**. Данный индикатор представляет собой набор из 7 светодиодов, расположенных по известной схеме. Напряжение, поданное на один из выводов индикатора, зажигает соответствующий сегмент индикатора (рис. 8).

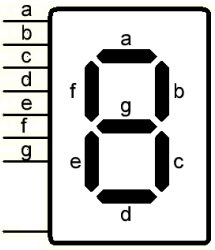


Рисунок 8 – Выводы 7-сегментного индикатора

Приведенная ниже программа заставляет один светящийся сегмент индикатора перемещаться по кругу. Для этого необходимо настроить выводы порта B на выход и выводить на них соответствующий код. Однако так как контроллер при старте, в зависимости от заводских установок, работает на частоте от 1 до 8 МГц, то простое переключение выводов ничего не даст – частота изменения их состояния слишком велика, чтобы ее заметить. Необходимо использовать функцию задержки. Пример кода выглядит следующим образом:

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    DDRB = 0xFF;
    while(1)
    {
        for( int i=0; i<6; i++){
            PORTB = (1<<i);
            _delay_ms(200);
        }
    }
}
```

Здесь используется библиотечная функция `_delay_ms()`, определенная в файле `delay.h`. Для корректной

работы данной функции в проекте должна быть определена тактовая частота работы контроллера.

Технически контроллер может работать на любой тактовой частоте, лишь бы она не была выше указанной в техническом описании, однако на практике есть ряд стандартных частот, которых стараются придерживаться.

Директива `#define F_CPU 1000000UL` определяет тактовую частоту работы контроллера и необходима для работы функции задержки `_delay_ms()`, например: `_delay_ms(200)` – задержка на 200 мс. Необходимо отметить, что установка тактовой частоты таким образом в программе не отменяет необходимости в соответствующей установке фьюз-бит, а если установленные программно и аппаратно частоты будут отличаться, то все производимые в программе расчеты задержек будут выполнены неверно.

В данной программе в порт B выводится 1, сдвинутая на число разрядов, заданное счетчиком цикла i – таким образом зажигаются сегменты от *a* до *f*. Когда цикл `for` кончается, наступает следующая итерация цикла while, и процесс повторяется.

Тот же порт, к которому подключен индикатор, можно использовать одновременно и для подключения кнопки. Для этого схема изменяется следующим образом (рис. 9):

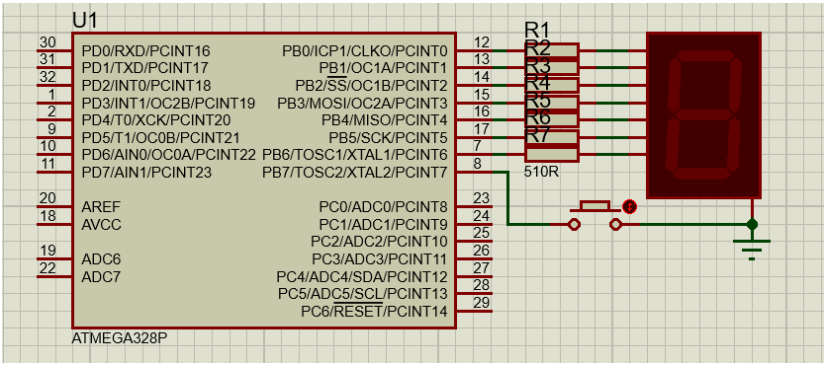


Рисунок 9 – Подключение кнопки

Кнопка подключена к выводу PB7. Для того чтобы использовать ее, необходимо сконфигурировать вывод PB7 как вход и включить внутренний подтягивающий резистор – тогда в обычном состоянии с вывода будет считываться 1, а при нажатой кнопке – 0.

Программа модифицирована таким образом, что при удержании кнопки сегмент двигается в обратном направлении:

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    DDRB = 0xFF & ~(1<<PB7);// PB7 - вход,
остальные - выходы
    PORTB |= (1<<PB7);    // включение подтя-
гивающего резистора
    int button = 0;       // вспомогательная
переменная
    while(1)
    {
        for(int i = 0; i < 6; i++){
            button = PINB & (1<<PB7);     //
чтение состояния PB7
            if(button != 0){
                PORTB = (1<<i);
                _delay_ms(200);
            }else{
                PORTB = (0x20>>i);
                _delay_ms(200);
            }
        }
    }
}
```

Семисегментный индикатор изначально предназначен для индикации цифр. Рассмотрим процесс создания секундомера на 10 секунд – по нажатию кнопки запускается и останавливается счет на индикаторе.

Для индикации цифр создается массив чисел, каждое из которых соответствует набору горящих элементов индикатора, представляющих определенную цифру. Для упрощения процедуры записи в порт кнопка переносится на другой вывод, для того чтобы не было необходимости учитывать ее наличие. Полученная схема устройства представлена на рисунке 10.

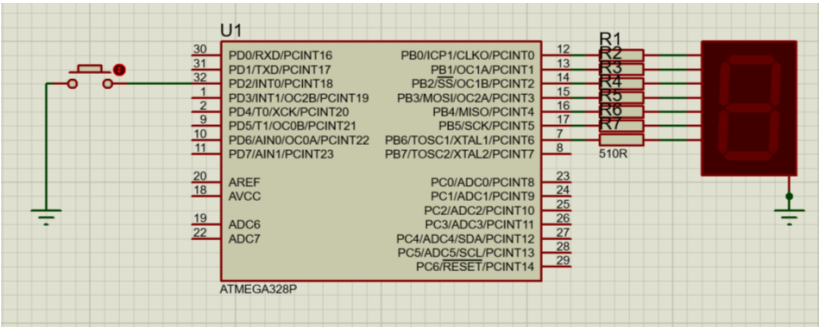


Рисунок 10 – Секундомер

Программа:

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>

uint8_t segments[]={
//   ___GFEDCBA
    0b00111111,   // 0 - A, B, C, D, E, F
    0b00000110,   // 1 - B, C
    0b01011011,   // 2 - A, B, D, E, G
    0b01001111,   // 3 - A, B, C, D, G
    0b01100110,   // 4 - B, C, F, G
    0b01101101,   // 5 - A, C, D, F, G
    0b01111101,   // 6 - A, C, D, E, F, G
    0b00000111,   // 7 - A, B, C
    0b01111111,   // 8 - A, B, C, D, E, F, G
    0b01101111,   // 9 - A, B, C, D, F, G
};
```

```
    int main(void)
    {
        DDRB = 0xFF;        // порт B на выход
        DDRD &= ~(1<<PD2);    // вывод PD2 на вход
        PORTD |= (1<<PD2);    // подтяжка PD2
включена
        int button = 0;
        int switch_state = 0;
        int counter = 0;
        while(1)
        {
            button = PIND & (1<<PD2);    //опрос
кнопки
            if(button == 0){
                while((PIND & (1<<PD2)) ==
0);//ожидание отпускания
                if(switch_state == 0){
                    switch_state = 1;
                } else {
                    switch_state = 0;
                    counter = 0;
                }
            }
            if(switch_state == 0){
                if(counter < 10){
                    PORTB = segments[counter++];
                    _delay_ms(1000);
                } else {
                    counter = 0;
                    PORTB = segments[counter++];
                    _delay_ms(1000);
                }

            }
        }
    }
```

При работе данной программы можно заметить, что кнопка плохо отзывается на нажатия. Дело в том, что опрос кнопки при работающем счете производится раз в секунду, так как ожидается, пока пройдет время задержки. Во многих применениях подобная нечувствительность к срабатываниям неприемлема. В таком случае используется механизм прерываний.

Прерывание – сигнал, сообщающий процессору о наступлении какого-либо события. При этом выполнение текущей последовательности команд приостанавливается и управление передаётся обработчику прерывания, который реагирует на событие и обслуживает его, после чего возвращает управление в прерванный код.

Прерывания могут быть вызваны различными событиями (как внешними для контроллера, так и внутренними). При поступлении сигнала прерывания контроллер приостанавливает работу с текущей программой и сохраняет содержимое регистров ядра в специальной области памяти – в стеке, после чего переходит в область памяти, в которой расположена специальная подпрограмма – обработчик прерывания. По завершении работы обработчика контроллер восстанавливает содержимое регистров и продолжает работу с основной программой с того же места, на котором остановился.

В данной реализации используется прерывание от внешнего источника – по изменению уровня напряжения на выводе с высокого на низкий. Для этого используется вывод PD2, так как альтернативной его функцией является работа в качестве источника внешнего прерывания INT0. Название обработчика данного прерывания – ISR(INT0_vect).

Для использования данного прерывания необходимо особым образом настроить контроллер.

Внешние прерывания настраиваются путем установки соответствующих бит в управляющие регистры контроллера внешних прерываний.

Регистр EICRA содержит контрольные биты для настройки условий генерации прерываний:

Bit	7	6	5	4	3	2	1	0	
(0x69)	–	–	–	–	ISC11	ISC10	ISC01	ISC00	EICRA
Read/write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial value	0	0	0	0	0	0	0	0	

При этом биты ISC11 и ISC10 отвечают за прерывание INT1, а ISC01 и ISC00 за INT0. Данные биты позволяют установить условия, при которых происходит генерация запроса на прерывание. Например, для прерывания INT0 возможны следующие комбинации:

Таблица 1 – **Варианты настройки прерывания INT0**

ISC01	ISC00	Описание
0	0	Низкий логический уровень на INT0 вызывает прерывание
0	1	Любое изменение уровня на INT0 вызывает прерывание
1	0	Прерывание вызывается спадом сигнала на INT0
1	1	Прерывание вызывается фронтом сигнала на INT0

Для INT1 настройки аналогичны.

Регистр EIMSK позволяет включить соответствующее прерывание:

Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	–	–	–	–	–	–	INT1	INT0	EIMSK
Read/write	R	R	R	R	R	R	R/W	R/W	
Initial value	0	0	0	0	0	0	0	0	

Для использования прерывания в программе необходимо выполнить несколько условий:

1) подключить к проекту файл interrupt.h, в котором содержатся названия векторов обработчиков прерываний;

2) найти в описании периферийного устройства бит, установка которого разрешает генерацию запроса на прерывание, и включить его;

3) глобально разрешить использование прерываний в программе.

Для использования вывода PD2 в качестве источника внешнего прерывания модуль INT0 настраивается на генерацию запроса на прерывание по спадающему фронту сигнала (переключение из 1 в 0). Это делается путем записи соответствующих значений в регистры EIMSK и EICRA. Конкретные значения приведены в техническом описании на контроллер.

Также необходимо глобально разрешить прерывания. Это делается короткой командой `sei();`

Также включается подтягивающий резистор на выводе PD2.

В обработчик прерывания выносится переключение состояния таймера и обнуление счетчика при повторном нажатии на кнопку.

Программа имеет следующий вид:

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

uint8_t segments[]={  //0b01111111
//   ___GFEDCBA
    0b00111111,   // 0 - A, B, C, D, E, F
    0b00000110,   // 1 - B, C
    0b01011011,   // 2 - A, B, D, E, G
    0b01001111,   // 3 - A, B, C, D, G
    0b01100110,   // 4 - B, C, F, G
    0b01101101,   // 5 - A, C, D, F, G
    0b01111101,   // 6 - A, C, D, E, F, G
    0b00000111,   // 7 - A, B, C
    0b01111111,   // 8 - A, B, C, D, E, F, G
    0b01101111,   // 9 - A, B, C, D, F, G
};

volatile int button = 0;
volatile int switch_state = 0;
volatile int counter = 0;

ISR(INT0_vect){       // Обработчик прерывания
    if(switch_state == 0){
        switch_state = 1;
    } else {
        switch_state = 0;
        counter = 0;
    }
}
```

```
    int main(void)
    {
        DDRB = 0xFF;
        PORTD |= (1<<PD2);
        EIMSK |= (1<<INT0);  //Включаем INT0
        EICRA |= (1<<ISC01); //Прерывание по спа-
дающему фронту INT0
        sei();                  //Глобальное разре-
шение прерываний
        while(1){
            if(switch_state == 0){
                if(counter < 10){
                    PORTB = segments[counter];
                    counter += 1;
                    _delay_ms(500);
                } else {
                    counter = 0;
                    PORTB = segments[counter];
                    counter += 1;
                    _delay_ms(500);
                }
            }
        }
    }
```

Теперь при включении секундомер начинает считать от нуля до девяти, а при нажатии на кнопку останавливается. При повторном нажатии переменная-счетчик обнуляется, и счет начинается с нуля.

Таким образом, мы рассмотрели основные приемы работы с контроллерами AVR. Важными моментами здесь являются работа с портами, сдвиги и битовые операции, прерывания.

Задания и пояснения к выполнению практической работы

Напишите программу в соответствии с вариантом, предложенным преподавателем.

1 в а р и а н т .

Напишите программу, работающую с кнопкой и двумя 7-сегментными индикаторами, так чтобы при первом нажатии на кнопку запускался секундомер, а при последующих нажатиях он каждый раз обнулялся.

2 в а р и а н т .

Напишите программу, работающую с кнопкой и двумя 7-сегментными индикаторами, которая будет считать число нажатий на кнопку до 15 раз.

Контрольные вопросы

1. Что такое микроконтроллер? Для чего он применяется?
2. Поясните, что отличает микроконтроллер от микропроцессора.
3. Назовите типы памяти, которыми обладает рассматриваемый контроллер ATmega328.
4. Для чего используются программные схемные симуляторы?
5. В любой программе для AVR к проекту подключается файл io.h. Для чего?
6. Для чего нужны фьюз-биты? Как их можно изменить?
7. Что такое прерывание? Что нужно сделать, чтобы использовать определенное прерывание при работе?
8. Назовите регистры управления портами и их функции.
9. Для чего используется запись вида PORTB |= 1<<PB2; или DDRC &= ~1<<PD3?

2

Таймеры и последовательный интерфейс SPI

Теоретические сведения

В разделе №1 было разработано устройство, представляющее собой таймер с одной кнопкой «Пуск / Стоп» с индикацией на одном 7-сегментном индикаторе. Однако такой таймер не имеет практического значения и необходимо увеличивать его разрядность. Увеличение разрядности с сохранением использованного ранее способа управления индикаторами требует применения большого количества выводов микроконтроллера. Один из подходов к решению данной проблемы состоит в поочередном выводе информации на индикаторы. Такой способ называется *динамической индикацией*. Её суть заключается в использовании инерции человеческого зрения. Информация высвечивается на индикаторах по очереди, но с достаточной частотой, чтобы человеческий глаз не замечал мерцания.

Для демонстрации данного способа служит схема, представленная на рисунке 11.

В данной схеме все индикаторы своими выводами, соответствующими определенным сегментам, подключены к порту B контролера, а их катоды подсоединены к выводам порта C. Таким образом, определенный сегмент определенного индикатора загорается только в том случае, когда на соответствующем выводе порта B установлен высокий уровень напряжения, а на выводе порта C, к которому подключен катод данного индикатора, выставлен низкий уровень. Таким образом, ток проходит из порта B, в определенный сегмент и возвращается в порт C, замыкая цепь. Если же на соответствующем

выводе порта C будет высокий уровень напряжения, то индикатор гореть не будет, т.к. выводы светодиодов-сегментов будут находиться под одним потенциалом.

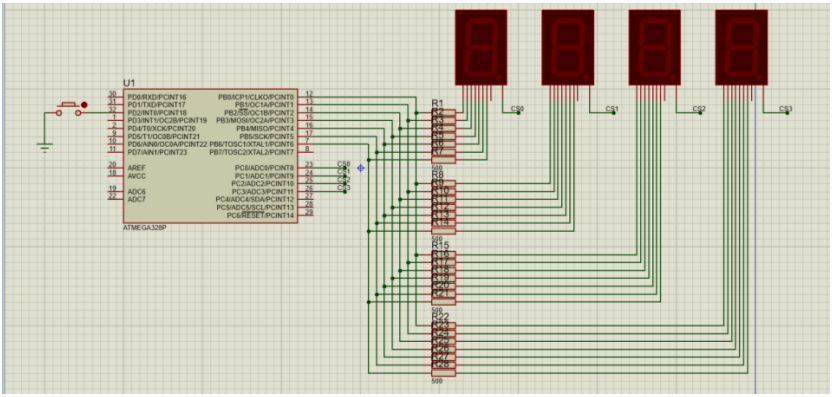


Рисунок 11 – Реализация динамической индикации

Данная схема служит лишь для объяснения принципов динамической индикации, в реальности такая реализация неэффективна или даже невозможна из-за ограничений по протекающему через выводы контроллера току. Поэтому управление катодами индикаторов обычно производится с помощью транзисторов, работающих в ключевом режиме.

Модель индикаторов, использованных в данной схеме, называется **7SEG-MPX1-CC** и располагается в библиотеке **Optoelectronics**. Для них имеется возможность указать время переключения – оно должно быть равно 1 мс.

Для уменьшения количества загружающих схему линий катоды индикаторов подключены к контроллеру с помощью меток – они устанавливаются нажатием на панели инструментов на кнопку **Wire Label Mode**. С помощью этого инструмента проводникам на схеме можно присвоить имена. Проводники, имеющие одно имя, Proteus считает соединенными.

Логика работы таймера остается той же, необходимо лишь адаптировать способ вывода информации на индикаторы. Для этого значение переменной, отвечающей за счет, необходимо перевести в де-

сятичную систему и каждый разряд полученного числа отобразить на соответствующем индикаторе. Для отображения же информации на индикаторах можно использовать встроенный в контроллер аппаратный таймер.

До сих пор измерение отрезков времени производилось с помощью функции задержки, применения которой в серьезных проектах стараются избегать. Она нежелательна, потому что контроллер во время ее выполнения не выполняет полезной работы, а лишь отсчитывает нужное количество тактов. Для избавления контроллера от бесполезной работы используются встроенные в контроллер *таймеры*.

Таймер контроллера представляет собой двоичный счетчик с программируемым делителем частоты и порогом срабатывания. Тактовые импульсы контроллера проходят через делитель частоты и поступают на вход счетчика, изменяя его значение, которое сравнивается с пороговым. Когда значение счетчика сравняется с заданным порогом, вызывается внутреннее прерывание. Таким образом, счет импульсов происходит независимо от хода выполнения остальной программы контроллера, и контроллеру необходимо лишь время от времени реагировать на возникающие прерывания.

Таймеры микроконтроллеров семейства AVR могут работать в нескольких режимах. Разные микроконтроллеры имеют разные наборы режимов для своих таймеров. Для выбора режимов работы существуют специальные регистры – регистры управления таймерами. Для простых таймеров используется один регистр управления. Для более сложных – два регистра. Регистры управления таймером называются TCCRx (где «x» – номер таймера). Например, для таймера T0 используется один регистр с именем TCCR0. Для управления таймером T1 используется два регистра: TCCR1A и TCCR1B. При помощи регистров управления производится не только выбор соответствующего режима, но и более тонкая настройка таймера.

Рассмотрим режимы работы таймеров.

1. Режим «Сброс при совпадении» (CTC)

Для работы в режиме CTC используется специальный регистр – регистр совпадения (сравнения). Если микроконтроллер содержит

несколько таймеров, то для каждого из них существует свой отдельный регистр совпадения. Причем для восьмиразрядных таймеров регистр совпадения – это один восьмиразрядный регистр. Для шестнадцатиразрядных таймеров регистр совпадения – это два восьмиразрядных регистра.

Регистры сравнения также имеют свои имена. Например, регистр совпадения таймера T1 состоит из двух регистров: OCR1L и OCR1H. В ряде микроконтроллеров существуют два регистра совпадения. Так, во всех микроконтроллерах семейства «Tiny» существует два регистра совпадения для таймера T1. Это регистры OCR1A и OCR1B. Два регистра совпадения для таймера T1 имеет и микроконтроллер ATmega8x. Во втором случае как таймер, так и его регистры совпадения имеют шестнадцать разрядов.

Если регистр совпадения шестнадцатиразрядный, то физически он состоит из двух регистров ввода-вывода. Например, два регистра совпадения таймера T1 микросхемы ATmega328 представляют собой четыре регистра ввода-вывода с именами OCR1A, OCR1B. Эти регистры включаются в работу только тогда, когда выбран режим CTC. В этом режиме, как и в предыдущем, таймер производит подсчет входных импульсов. Текущее значение таймера из его счетного регистра постоянно сравнивается с содержимым регистров совпадения.

Если таймер имеет два регистра совпадения, то для каждого из этих регистров производится отдельное сравнение. Когда содержимое счетного регистра совпадет с содержимым одного из регистров совпадения, произойдет вызов соответствующего прерывания. Кроме вызова прерывания, в момент совпадения может происходить одно из следующих событий:

– сброс таймера (верно только для регистров совпадения OCR1A и OCR1B);

– изменение состояния одного из выводов микроконтроллера (верно для всех регистров).

Произойдет или не произойдет одно или оба события из вышеперечисленных, определяется при настройке таймера.

2. Нормальный режим (Normal)

В этом режиме таймер производит подсчет приходящих на его вход импульсов (от тактового генератора или внешнего устройства)

и вызывает прерывание по переполнению. Этот режим является единственным режимом работы для восьмиразрядных таймеров большинства микроконтроллеров семейства «Tiny» и для части микроконтроллеров семейства «Mega». Для всех остальных восьмиразрядных и всех шестнадцатиразрядных таймеров это всего лишь один из возможных режимов.

3. Режим «Захват» (Capture)

Суть этого режима заключается в сохранении содержимого счетного регистра таймера в определенный момент времени. Запоминание происходит либо по сигналу, поступающему через специальный вход микроконтроллера, либо от сигнала с выхода встроенного компаратора.

Этот режим удобен в том случае, когда нужно измерить длительность какого-либо внешнего процесса. Например, время, за которое напряжение на конденсаторе достигнет определенного значения. В таком случае напряжение с конденсатора подается на один из входов компаратора, а на второй его вход подается опорное напряжение.

Микроконтроллер должен одновременно запустить два этих процесса:

- подать напряжение на конденсатор;
- запустить таймер в режиме Capture.

Конденсатор начнет заряжаться, напряжение на нем при этом будет плавно расти. Одновременно счетчик таймера будет отсчитывать тактовые импульсы заданной частоты. В тот момент, когда напряжение на конденсаторе сравняется с опорным напряжением, логический уровень на выходе компаратора изменится на противоположный. По этому сигналу текущее значение счетного регистра запоминается в специальном регистре захвата. Имя этого регистра ICRx (для таймера T0 это будет ICR0, для T1 – ICR1 и т. д.). Одновременно вырабатывается запрос на прерывание.

Используя принцип измерения времени зарядки, удобно создавать простые схемы, работающие с различными аналоговыми датчиками (температуры, давления и т. д.). Если принцип работы датчика состоит в изменении его внутреннего сопротивления, то такой дат-

чик можно включить в цепь зарядки конденсатора. Емкостные датчики можно подключать напрямую.

4. Режим «Быстродействующий ШИМ» (Fast PWM)

ШИМ расшифровывается как «широтно-импульсная модуляция». На английском это звучит как «Pulse Width Modulation» (PWM). Сигнал с ШИМ часто используется в устройствах управления.

Сигнал с ШИМ можно использовать, например, для регулировки скорости вращения электродвигателя постоянного тока. Для этого вместо постоянного напряжения на двигатель подается прямоугольное импульсное напряжение. Благодаря инерции двигателя импульсы сглаживаются, и двигатель вращается равномерно. Меняя скважность импульсов (то есть отношение периода импульсов к их длительности), можно изменять среднее напряжение, приложенное к двигателю, и тем самым менять скорость его вращения.

Точно таким же образом можно управлять и другими устройствами. Например, нагревательными элементами, осветительными приборами и т. п. Преимущество импульсного управления в высоком КПД. Импульсные управляющие элементы рассеивают гораздо меньше паразитной мощности, чем управляющие элементы, работающие в аналоговом режиме.

Для формирования сигнала ШИМ используются те же самые регистры совпадения, которые работают и в режиме CTC. Формирование сигнала ШИМ может осуществляться несколькими разными способами. Работа таймера в режиме Fast PWM проиллюстрирована на рисунке 12.

Сигнал с ШИМ формируется на специальном выходе микроконтроллера. На вход таймера подаются импульсы от системного генератора. Таймер находится в состоянии непрерывного счета. При переполнении таймера его содержимое сбрасывается в ноль, и счет начинается сначала. В режиме ШИМ переполнение таймера не вызывает прерываний. На рисунке 12 это показано в виде пилообразной кривой, обозначенной как TCNTn. Кривая представляет собой зависимость содержимого счетного регистра от времени.

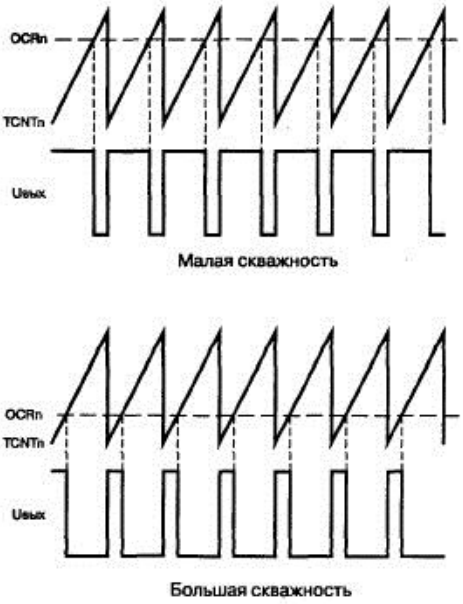


Рисунок 12 – Работа таймера в режиме Fast PWM

Содержимое счетного регистра непрерывно сравнивается с содержимым регистра совпадения. Пока число в регистре OCRn больше, чем число в счетном регистре таймера (TCNTn), напряжение на выходе ШИМ равно логической единице. Когда же в процессе счета содержимое счетного регистра TCNTn станет больше содержимого OCRn, на выходе ШИМ установится нулевой потенциал.

В результате на выходе получаются прямоугольные импульсы. Скважность этих импульсов будет зависеть от содержимого регистра OCRn. Чем меньше число в OCRn, тем выше скважность выходных импульсов. На рисунке 12 показана скважность импульсов для двух разных значений регистра OCRn.

Если содержимое OCRn достигнет своего максимального значения, то импульсы на выходе ШИМ исчезнут, и там постоянно будет присутствовать логическая единица. При уменьшении числа в OCRn появятся импульсы малой скважности (длительность почти равна периоду). Если плавно уменьшать число в OCRn, то скважность бу-

дет плавно уменьшаться. Когда содержимое OCRn достигнет нуля, импульсы на выходе ШИМ также исчезнут и там установится логический ноль.

Такой режим работы обеспечивает получение ШИМ высокой частоты, позволяющий использовать его для управления источниками питания, стабилизаторами, цифро-аналогового преобразования.

5. Режим «ШИМ с точной фазой» (Phase Correct PWM)

Описанный в предыдущем разделе режим ШИМ имеет один недостаток. При изменении длительности импульсов меняется и их фаза. Центр каждого импульса как бы сдвигается во времени. При управлении электродвигателем такое поведение фазы нежелательно. Поэтому в микроконтроллерах AVR предусмотрен еще один режим ШИМ. Это ШИМ с точной фазой. Принцип работы таймера в этом режиме изображен на рисунке 13.

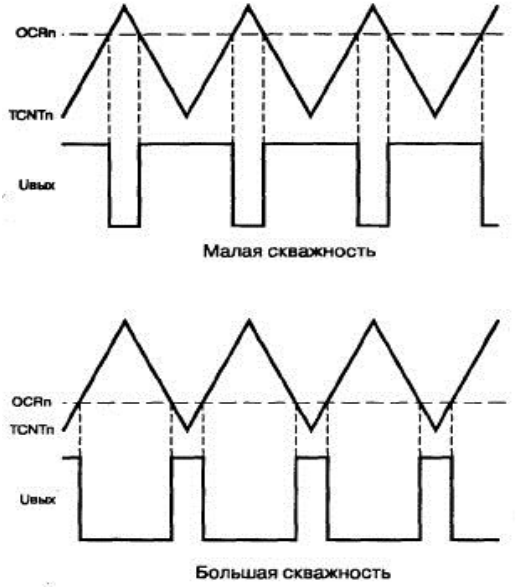


Рисунок 13 – Работа таймера в режиме Phase Correct PWM

Отличие режима «Phase Correct PWM» от режима «Fast PWM» заключается в режиме работы счетчика. Сначала счетчик считает так

же, как и в предыдущем режиме (от каждого входного импульса его значение увеличивается на единицу). Достигнув своего максимального значения, счетчик не сбрасывается в ноль, а переключается в режим реверсивного счета.

Теперь уже от каждого входного импульса его содержимое уменьшается на единицу. В результате пилообразная кривая, отображающая содержимое счетного регистра TCNTn, становится симметричной, как показано на рисунке 13. Система совпадения работает так же, как и в предыдущем случае.

Благодаря симметричности сигнала на таймере, фаза выходных импульсов в процессе регулировки скважности не изменяется. Середина каждого импульса строго привязана к точке смены направления счета таймера.

Недостатком режима «Phase Correct PWM» можно считать в два раза меньшую частоту выходного сигнала. Это существенно уменьшает динамичность регулирования. Кроме того, при использовании внешних фильтров для преобразования импульсного сигнала ШИМ в аналоговый, схема с более низкой частотой потребует применения комплектующих с большими габаритами и массой.

6. Асинхронный режим

В некоторых моделях микроконтроллеров таймер может работать в асинхронном режиме. В этом режиме на вход таймера подается сигнал либо от внутреннего кварцевого генератора, либо от внешнего генератора. Счетчик не вырабатывает никаких прерываний и дополнительных сигналов. В этом режиме он работает в качестве часов реального времени. Микроконтроллер может предустанавливать содержимое счетного регистра, а затем в любой момент считать это содержимое, получив, таким образом, текущее значение реального времени.

В состав контроллера ATmega328 входят один 16-битный и два 8-битных таймера. 16-битный таймер обладает несколько более широкой функциональностью.

В рассматриваемом устройстве таймер может быть использован для того, чтобы через определенный отрезок времени обновлять информацию на индикаторах.

Практическая часть

Программа:

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

uint8_t segments[]={
    0b00111111,   // 0 - A, B, C, D, E, F
    0b00000110,   // 1 - B, C
    0b01011011,   // 2 - A, B, D, E, G
    0b01001111,   // 3 - A, B, C, D, G
    0b01100110,   // 4 - B, C, F, G
    0b01101101,   // 5 - A, C, D, F, G
    0b01111101,   // 6 - A, C, D, E, F, G
    0b00000111,   // 7 - A, B, C
    0b01111111,   // 8 - A, B, C, D, E, F, G
    0b01101111,   // 9 - A, B, C, D, F, G
};

void InitPorts(void);
void send_data(uint8_t data, uint8_t ind);
void InitTimer0(void);
void Bin2Dec(uint16_t data);

volatile uint16_t cnt = 0;
volatile uint8_t switch_state = 0;
volatile uint8_t bcd_buffer[] = {0,0,0,0};

int main(void)
{
    InitPorts();
    InitTimer0();
    EIMSK|=(1<<INT0); //ВключитьINT0
```

```
        EICRA|=(1<<ISC01);    //Настройка INT0 на
прерывание по спаду

        sei();          //Глобальное разрешение пре-
рываний
        while(1)
        {
             if(switch_state == 0){
                  Bin2Dec(cnt);
                  if(cnt<9999){
                       cnt++;
                  }else{
                       cnt=0;
                  }
                  _delay_ms(100);
             }
        }
   }
   //-------------------------------------------

   ISR(TIMER0_COMPA_vect){
        send_data(bcd_buffer[3],0);
        send_data(bcd_buffer[2],1);
        send_data(bcd_buffer[1],2);
        send_data(bcd_buffer[0],3);
   }

   ISR(INT0_vect){
        if(switch_state == 0){
             switch_state = 1;
        } else {
             switch_state = 0;
             cnt = 0;
        }
   }
```

```
void InitPorts(void){
    DDRB = 0xFF;
    DDRC = (1<<PC0|1<<PC1|1<<PC2|1<<PC3);
    PORTC = 0x0F;
    DDRD = (0<<PD2);
    PORTD |= (1<<PD2);
}

void send_data(uint8_t data, uint8_t ind)
{
    PORTC = 0x0F &~ (1<<ind);
    PORTB = segments[data];
    _delay_ms(5);
    PORTB = 0;
    PORTC = 0x0F;
}

void InitTimer0(void)
{
    TCCR0A = (1<<WGM01);        //режим CTC -
Clear Timer on
                                        //Compare
    TCCR0B = (1<<CS02|1<<CS00);//prescaler =
sys_clk/1024
    TCNT0 = 0x00;               //начальное
значение счетчика
    OCR0A = 16;                         //порог
срабатывания
    TIMSK0 |= (1<<OCIE0A);
//включение прерывания при

//достижении порога A
}

void Bin2Dec(uint16_t data)
{
```

```
    bcd_buffer[3]=(uint8_t)(data/1000);
    data = data - bcd_buffer[3]*1000;
    bcd_buffer[2] = (uint8_t)(data/100);
    data = data - bcd_buffer[2]*100;
    bcd_buffer[1] = (uint8_t)(data/10);
    data = data - bcd_buffer[1]*10;
    bcd_buffer[0] = (uint8_t)(data);
}
```

Данная программа разбита на функции, что упрощает дальнейшее её сопровождение. Функции InitPorts и InitTimer0 выполняют первичную инициализацию контроллера. В функции InitPorts настраиваются необходимые для работы выводы контроллера. В функции InitTimer0, как следует из ее названия, производится настройка таймера 0. Таймер 0 – 8-битный таймер, и для его работы необходимо записать в его конфигурационные регистры соответствующие данные.

Регистр TCCR0A задает режим работы таймера – при приведенных настройках таймер сравнивает значение своего счетчика с заданным порогом и при его достижении вызывает прерывание и начинает счет с начала.

С помощью регистра TCCR0B настраивается работа предделителя. При приведенных настройках тактовая частота контроллера делится на 1024.

В регистре TCNT0 содержится стартовое значение счетчика таймера.

Таймер 0 имеет возможность срабатывания по двум пороговым значениям, но в данном случае в этом нет необходимости, и записанное в регистр TIMSK0 значение настраивает таймер на срабатывание по порогу A. В регистр OCR0A записывается значение порога A.

Таким образом, при работе данной программы содержимое переменной-счетчика cnt с помощью функции Bin2Dec переводится в вид десятичного числа, разряды которого хранятся в массиве bcd_buffer. Таймер 0 срабатывает с частотой около 60 Гц и выводит содержимое массива на индикаторы.

Несмотря на предпринятые усилия, такой секундомер все еще малопригоден для реального использования. Хотя для вывода информации на индикаторы и применяется таймер, увеличение значения самой переменной-счетчика производится с помощью все той же функции задержки, а следовательно, точность хода такого таймера явно оставляет желать лучшего. Логичным решением является использование таймера по его прямому назначению – для формирования точных отрезков времени. Использование двух таймеров одновременно нежелательно, так как возможен «перехлест» прерываний, и произойдет сбой. Поэтому процедуру обновления индикаторов необходимо перенести в основной цикл. Индикаторы должны обновляться с достаточно высокой частотой, что накладывает ограничения на время выполнения основного цикла. Чтобы уйти от этого ограничения, можно применить еще один способ подключения индикаторов – при помощи регистров.

Регистр – электронное устройство, предназначенное для хранения цифровой информации. Он состоит из набора триггеров и некоторого количества логических вентилей для управления им. Пример простого регистра на 4 бита представлен на рисунке 14.

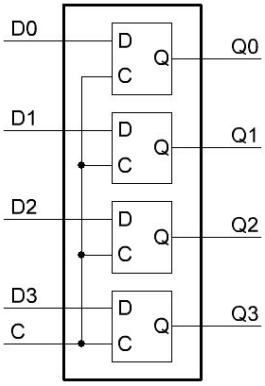


Рисунок 14 – Регистр с параллельной загрузкой

Приведенный на рисунке регистр состоит из 4 D-триггеров, и его функция заключается в том, что информация, поступающая на его

входы, при поступлении тактового импульса на вход С окажется на его выходах и там и останется даже при изменившихся состояниях входов либо до поступления следующего тактового импульса, либо до его выключения. Таким образом, регистр выполняет функцию ячейки памяти.

Если подключить такой регистр к индикатору, то индикатор будет отображать записанную в регистр информацию, пока в него не будет записана новая.

Изображенный на рисунке 14 регистр называется регистром с параллельной загрузкой, потому что триггеры внутри регистра соединены параллельно, и информация записывается одновременно во все. Но возможен и вариант с последовательной загрузкой (рис. 15).

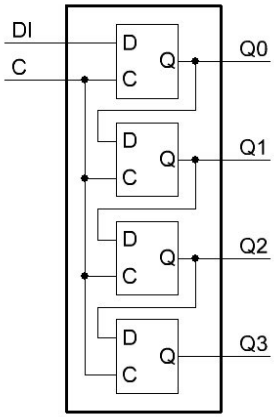


Рисунок 15 – Регистр с последовательной загрузкой

Работа такого регистра выглядит следующим образом (рис. 16):

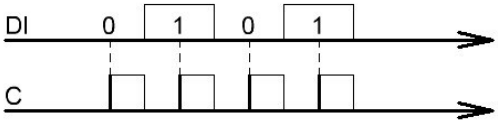


Рисунок 16 – Загрузка данных в регистр

Пусть на выходах Q[0..3] необходимо установить значения 0101. Сначала на вход DI подается значение старшего бита – 0, а на вход C подается тактовый импульс. По фронту сигнала C каждый триггер запишет в себя состояние выхода предыдущего триггера, а самый первый триггер – состояние входа DI. Так, после первого импульса на выходах останется значение 0000. Затем на DI подается следующий разряд – 1 и фиксируется вторым тактовым импульсом – значение станет равно 0001. Так же проходят 3-й (состояние выходов – 0010) и 4-й биты (0101).

Если продолжить подавать тактовые импульсы, то число, записанное в регистр, будет продолжать сдвигаться, поэтому регистры с подобным объединением триггеров также называются сдвиговыми. Развивая схему дальше, можно получить регистры с комбинированной загрузкой, с последовательным выходом, с реверсивным сдвигом и тому подобные. Такие регистры входят в состав контроллера и позволяют производить математические и логические операции над числами.

Приведенные на рисунках 14 и 15 регистры выпускаются в виде микросхем, и их модели есть в симуляторе Proteus. Однако и они не лишены недостатков. Для загрузки информации в регистр с параллельной загрузкой понадобится вести от контроллера 8 проводников: 7 линий данных и одна для тактового сигнала. Если регистров 4, то для раздельной загрузки каждого понадобится уже 4 тактовых линии. Не слишком экономное расходование выводов контроллера.

Если же использовать регистры с последовательной загрузкой, то во время загрузки информации в регистр на его выходах появляется не только то состояние, которое нужно, но и промежуточные, возникающие в процессе загрузки. То есть, пока регистр не загрузится полностью, на индикаторы будет выводиться мусор.

Решением проблемы является использование двухступенчатого регистра. Это регистр 74HC595, и его модель будет использоваться далее. Упрощенно его можно изобразить следующим образом (рис. 17).

Он состоит из уже знакомых регистров последовательной и параллельной загрузки. Первоначально информация записывается в

регистр последовательной загрузки, а после окончания этого процесса тактовый импульс на линии O переносит информацию в регистр параллельной загрузки, а значит и на выходы Q[0…3].

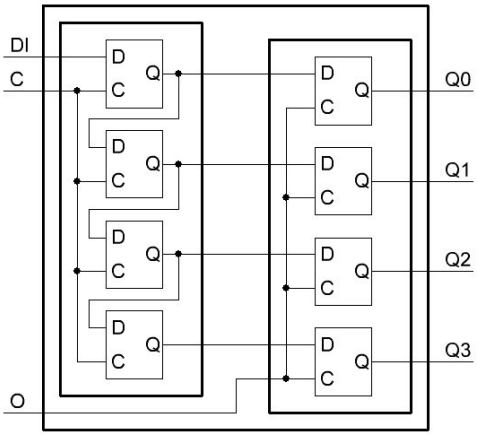


Рисунок 17 – Двухступенчатый регистр

На рисунке 18 приведен пример сдвигового регистра из базы данных Proteus.

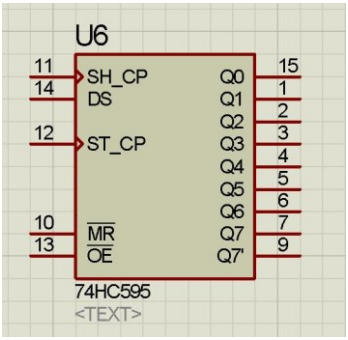


Рисунок 18 – Компонент 74HC595 в Proteus

Здесь вход DS соответствует входу DI на рисунке 8, SH_CP – тактовый сигнал последовательного регистра, ST_CP – параллельного. Дополнительно появляются выводы MR (Mass Reset – полное

стирание – применяется для начальной инициализации регистра, если важно его начальное состояние. Во время работы на этом выводе должен быть уровень логической 1) и OE (Output Enable – включение выхода – если неактивен, выходы Q переводятся в неактивное состояние. Во время работы на этом выводе должен быть уровень логической 1). Вывод Q7’ – это выход последнего триггера регистра последовательной загрузки. Если взять два таких регистра и подключить этот вывод первого регистра к входу данных второго, то появляется возможность по одной линии данных записывать информацию в два регистра – то есть информация, постоянно сдвигаясь, проходит сначала через первый регистр, выходит через вывод Q7’ и входит уже во второй регистр. Так можно подключить сколько угодно таких регистров, а затем разом обновить информацию на всех выходах всех регистров. Это и есть их основное достоинство.

Схема устройства с подключенными таким образом индикаторами приведена на рисунке 19.

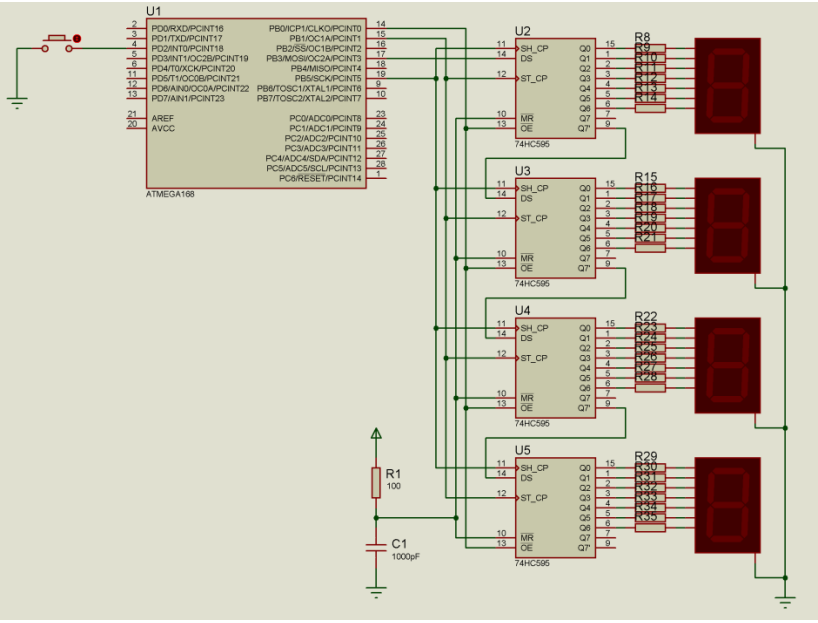


Рисунок 19 – Подключение индикаторов с помощью регистров

Регистры здесь подключены так, что выход Q7’ предыдущего регистра подключается к входу данных DS следующего. Выводы SH_CP, ST_SP и OE разных регистров объединены и подключаются к выводам контроллера. Выводы MR также объединены и подключены к RC-цепи, благодаря которой при включении содержимое регистров обнулится.

Рабочая программа микроконтроллера, реализующего функцию многоразрядного таймера, приведена ниже.

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

uint8_t segments[]={
    0b00111111,   // 0 - A, B, C, D, E, F
    0b00000110,   // 1 - B, C
    0b01011011,   // 2 - A, B, D, E, G
    0b01001111,   // 3 - A, B, C, D, G
    0b01100110,   // 4 - B, C, F, G
    0b01101101,   // 5 - A, C, D, F, G
    0b01111101,   // 6 - A, C, D, E, F, G
    0b00000111,   // 7 - A, B, C
    0b01111111,   // 8 - A, B, C, D, E, F, G
    0b01101111,   // 9 - A, B, C, D, F, G
};

void InitPorts(void);
void send_data(uint8_t data, uint8_t ind);
void InitTimer0(void);
void Bin2Dec(uint16_t data);
void InitTimer1(void);
void StartTimer1(void);
void StopTimer1(void);
void SendData(uint8_t data);
void DisplayData(uint16_t data);
```

```
volatile uint16_t cnt = 0;
volatile uint8_t switch_state = 0;
volatile uint8_t bcd_buffer[] = {0,0,0,0};

int main(void)
{
    InitPorts();
    InitTimer1();
    EIMSK |= (1<<INT0);   //разрешить прерыва-
ние INT0
    EICRA |= (1<<ISC01);//Запуск по заднему
фронту INT0

    sei();              //Разрешение прерываний
    PORTB &= ~(1<<PB0);   //OE = low (active)
    DisplayData(0);
    while(1)
    {    }
}

//-------------------------------------------

ISR(TIMER1_COMPA_vect){
    DisplayData(cnt);
    if(cnt<9999){
        cnt++;
    }else{
        cnt=0;
    }
}

ISR(INT0_vect){
    if(switch_state == 0){
        switch_state = 1;
        StartTimer1();
```

```
    }else{
        StopTimer1();
        DisplayData(cnt);
        switch_state = 0;
        cnt = 0;
    }
}

//-------------------------------------------

void InitPorts(void){
    DDRB = (1<<PB0|1<<PB1|1<<PB3|1<<PB5);
    DDRD &= ~(1<<PD2);
    PORTD |= (1<<PD2);
}

void InitTimer1(void){
    TCCR1A = 0;
    TCCR1B = (1<<CS11|1<<CS10|1<<WGM12);
    TCNT1 = 0;
    OCR1A = 15624;
}

void StartTimer1(void){
    TCNT1 = 0;
    TIMSK1 |= (1<<OCIE1A);
}

void StopTimer1(void){
    TIMSK1 &= ~(1<<OCIE1A);
}

void Bin2Dec(uint16_t data){
    bcd_buffer[3] = (uint8_t)(data/1000);
    data = data % 1000;
    bcd_buffer[2] = (uint8_t)(data/100);
```

```
    data = data % 100;
    bcd_buffer[1] = (uint8_t)(data/10);
    data = data % 10;
    bcd_buffer[0] = (uint8_t)(data);
}

void SendData(uint8_t data){
    for(uint8_t i=0; i<8; i++){
        PORTB &= ~(1<<PB5);          //CLK low
        if(0x80 & (data<<i)){
            PORTB |= 1<<PB3;  //DAT high
        } else {
            PORTB &= ~(1<<PB3);      //DAT low
        }
        PORTB |= (1<<PB5);           //CLK high
    }
}

void DisplayData(uint16_t data){
    Bin2Dec(data);
    PORTB &= ~(1<<PB1);         //clk_out = 0
    SendData(segments[bcd_buffer[0]]);
    SendData(segments[bcd_buffer[1]]);
    SendData(segments[bcd_buffer[2]]);
    SendData(segments[bcd_buffer[3]]);
    PORTB |= (1<<PB1);          //clk_out = 1
}
```

Как видно из текста программы, в основном цикле программы не выполняется ничего, вся обработка производится в прерываниях таймера и кнопки. Для отсчета равных промежутков времени используется таймер 1.

Для того чтобы таймер выполнял свою задачу в рассматриваемом устройстве, он должен срабатывать раз в секунду. При заданной тактовой частоте 1 МГц необходимо делить ее в миллион раз. Частота срабатывания таймера задается настройками предделителя тайме-

ра и порога срабатывания. Как для 16-битного, так и для 8-битных таймеров предделитель можно настроить лишь на фиксированные значения деления частоты, причем максимальный коэффициент деления – 1024, что означает, что для получения периода срабатывания 1 секунда необходимо установить порог срабатывания таймера 976,5625. Так как максимальное значение беззнакового целого, которое можно записать в 1 байт равно 511, то видно, что использование 8-битных таймеров для данной цели не представляется возможным (по крайней мере, без организации программного счетчика). Таким образом, для организации прерываний 1 раз в секунду необходимо использовать 16-битный таймер.

В ATmega328 16-битным таймером является Timer/Counter 1. Так как установить для таймера дробное значение порога срабатывания нельзя, необходимо подобрать коэффициент предделителя. Коэффициент 1000000 можно представить как 64×15625, где 64 – коэффициент предделителя, 15625 – пороговый уровень.

Для передачи информации в индикаторы используется функция DisplayData, в состав которой входят также функции перевода из двоичной системы в десятичную и функции отправки данных. Из текста видно, что сначала производится последовательная отправка 4-х байт, после чего посылается сигнал на линию загрузки параллельных регистров.

Функция отправки байта SendData выполняет побитовую отправку переданного в функцию байта. По сути, с ее помощью организуется передача данных по последовательному синхронному интерфейсу. Такой интерфейс носит название SPI.

Интерфейс SPI (*Serial Peripheral Interface*) разработан фирмой Motorola и используется для двунаправленной передачи данных по последовательному каналу. В системе со связью по SPI всегда присутствуют ведущее устройство (оно отправляет тактовые сигналы, всегда одно) и ведомые (может быть больше одного). Для выбора конкретного устройства из ведомых от ведущего к ведомому идет линия активизации ведомого – CS (или SS – Slave Select).

При помощи SPI осуществляется работа со многими внешними устройствами, такими как микросхемы памяти, АЦП, ЦАП, SD-карты и др.

Схемотехнически интерфейс SPI можно представить следующим образом (рис. 20):

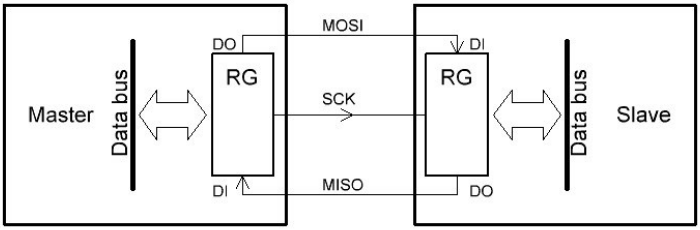


Рисунок 20 – Интерфейс SPI

Основным элементом интерфейса у обоих устройств является сдвиговый регистр с возможностью параллельной и последовательной загрузки и выдачи данных. Ведущий загружает в свой регистр данные по линии MOSI (*Master-Output-Slave-Input*) и начинает подавать тактовые сигналы по линии SCK. Таким образом, данные из регистра ведущего перемещаются в регистр ведомого, однако одновременно происходит и обратный процесс: ведущий получает информацию от ведомого по линии MISO (*Master-Input-Slave-Output*), хочет он того или нет. Часто эта информация не несет пользы и просто игнорируется. С другой стороны, наличие обеих линий не обязательно, и если предполагается односторонняя передача информации, то лишний проводник просто не делается.

Для настройки интерфейса необходимо сконфигурировать три регистра – SPI status register (SPSR), SPI control register (SPCR) и SPI data register (SPDR).

SPI control register (SPCR).

7	6	5	4	3	2	1	0	
SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
0	0	0	0	0	0	0	0	

Значения четырех старших разрядов задают основные параметры работы модуля SPI.

Бит 7 – SPIE: SPI Interrupt Enable.

Этот бит разрешает прерывания от SPI. Если будут установлены бит SPIF в регистре SPSR и бит I в регистре SREG (общее разрешение прерываний), то будет вызвана продпрограмма обработки прерывания SPI_STC.

– 0 - прерывания запрещены,

– 1 - разрешение прерывания.

Бит 6 – SPE: SPI Enable.

Этот бит разрешает работу SPI.

– 0 в этом разряде означает запрет любых операций в SPI,

– 1 в этом разряде – разрешение работы.

Бит 5 – DORD: Data Order.

Этот бит определяет порядок передачи данных.

– 0 в этом разряде – сначала будет передаваться младший разряд,

– 1 в этом разряде – сначала будет передаваться старший разряд.

Бит 4 – MSTR: Master/Slave Select.

Этот бит определяет режим работы модуля.

– 0 в этом разряде – режим ведомого,

– 1 в этом разряде – режим ведущего.

Четыре младших разряда определяют параметры тактового сигнала.

Бит 3 – CPOL: Clock Polarity.

Полярность сигнала синхронизации.

– 0 в этом разряде – тактовый сигнал в состоянии ожидания - 0,

– 1 в этом разряде – тактовый сигнал в состоянии ожидания - 1.

Таблица 2 – **Задание полярности сигнала синхронизации**

CPOL	Передний фронт	Задний фронт
0	Нарастающий	Спадающий
1	Спадающий	Нарастающий

Бит 2 – CPHA: Clock Phase.

Фаза сигнала синхронизации определяет, по какому фронту (переднему или заднему) будет производиться установка (выборка).

– 0 в этом разряде – установка (выборка) по заднему фронту,

– 1 в этом разряде – установка (выборка) по переднему фронту.

Таблица 3 – **Установка активного фронта сигнала синхронизации**

CPHA	Передний фронт	Задний фронт
0	Выборка	Установка
1	Установка	Выборка

Биты 1, 0 – SPR1, SPR0: SPI Clock Rate Select.

Совместно с битом SPI2x в регистре SPSR задают скорость передачи данных. Если устройство используется в режиме подчиненного, биты SPR1 и SPR0 не влияют на его работу. Отношение тактовой частоты SPI SCK к основной тактовой частоте микроконтроллера fosc приведено в таблице 4.

Таблица 4 – **Установка скорости обмена данными**

SPI2X	SPR1	SPR0	Частота SCK
0	0	0	fosc / 4
0	0	1	fosc / 16
0	1	0	fosc / 64
0	1	1	fosc / 128
1	0	0	fosc / 2
1	0	1	fosc / 8
1	1	0	fosc / 32
1	1	1	fosc / 64

Комбинация битов CPHA и CPOL задает четыре возможных режима последовательной передачи данных. Биты данных выводятся сдвигом и фиксируются на входе противоположными фронтами синхросигнала SCK, тем самым гарантируя достаточное время на установ-

ление уровней сигналов данных и окончание переходных процессов. Выбор режима работы SPI зависит от периферийного устройства, с которым осуществляется обмен данными. Возможна ситуация, когда к одному интерфейсу SPI подключены несколько ведомых периферийных устройств с разными режимами работы SPI. В таком случае необходима перенастройка ведущего устройства в соответствии с требованиями ведомого устройства при каждой передаче.

Форматы передачи данных для SPI представлены в таблице 5, а их временные диаграммы показаны на рисунках 21 и 22.

Таблица 5 – **Режимы работы SPI**

Значения битов	Передний фронт	Задний фронт	Режим SPI
CPOL = 0 CPHA = 0	Выборка нарастающим фронтом	Установка данных спадающим фронтом	0
CPOL = 0 CPHA = 1	Установка данных нарастающим фронтом	Выборка спадающим фронтом	1
CPOL = 1 CPHA = 0	Выборка спадающим фронтом	Установка данных нарастающим фронтом	2
CPOL = 1 CPHA = 1	Установка данных спадающим фронтом	Выборка нарастающим фронтом	3

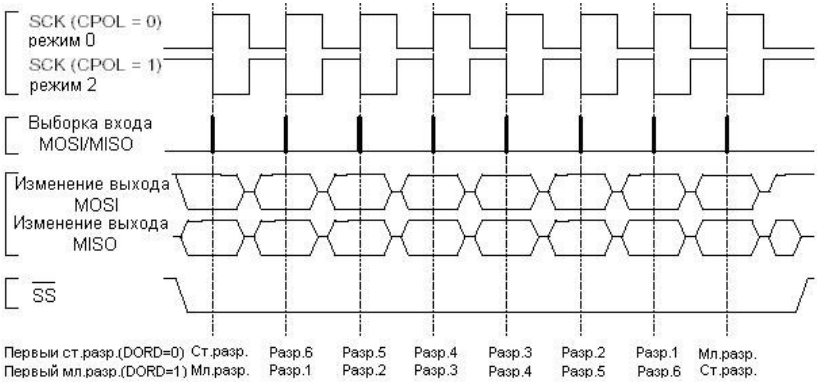


Рисунок 21 – Режимы 0 и 2 SPI

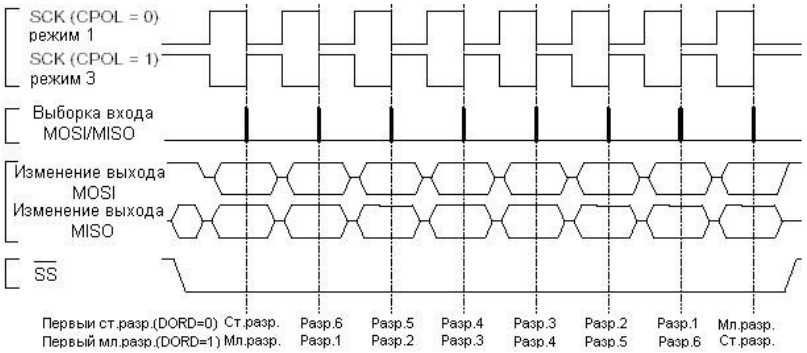


Рисунок 22 – Режимы 1 и 3 SPI

SPI status register (SPSR).

7	6	5	4	3	2	1	0	
SPIF	WCOL	–	–	–	–	–	SPI2X	SPSR
R	R	R	R	R	R	R	R/W	
0	0	0	0	0	0	0	0	

Бит 7 – SPIF: SPI Interrupt Flag.

Флаг завершения передачи устанавливается после завершения передачи (приема) и вызывает прерывание (как у ведущего, так и у ведомого). В вызванном прерывании можно считывать и записывать новые данные для передачи. Сбрасывается автоматически при входе в подпрограмму обработки прерывания.

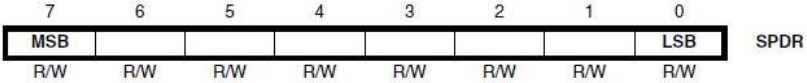
Бит 6 – WCOL: Write Collision flag.

Флаг наложения записи устанавливается при чтении либо записи данных при незавершенной передаче. Автоматически сбрасывается чтением регистра SPSR, а также при чтении (записи) в регистр данных (SPDR).

Бит 0 – SPI2X: Double SPI Speed Bit.

Совместно с битами SPR1 и SPR0 в регистре SPCR задает скорость передачи данных, при его установке скорость работы SPI (частота SCK) удвоится, если SPI находится в режиме ведущего. Это означает, что минимальный период SCK будет равен двум периодам fosc. Если SPI работает как ведомый, то работа SPI гарантирована только на частоте fosc / 4 или менее.

SPI Data Register (SPDR).



Регистр данных SPI, запись в данный регистр инициирует передачу данных. При чтении данного регистра фактически считывается содержимое приемного буфера сдвигового регистра.

Интерфейс SPI широко распространен и является стандартным для абсолютного большинства контроллеров. В ATmega168 также имеется встроенный SPI интерфейс, и его можно использовать для передачи информации на индикаторы. Передача данных идет только в одном направлении: от контроллера на индикаторы – нет необходимости в линии MISO; приемное устройство только одно – нет необходимости использовать линию SS.

На рисунке 19 индикаторы уже подключены нужным образом, но до сих пор выводы PB3 и PB5 использовались как обычные пины общего назначения и управлялись программно. Для использования интерфейса SPI необходимо включить их альтернативные функции. В код программы добавится функция инициализации SPI:

```
    void InitSPI(void){
        DDRB |= (1<<PB3 | 1<<PB5);//настроить MOSI
и CLK как выходы
        SPSR |= (1<<SPI2X);          //Fclk = Fosc/2
        SPCR = (1<<SPE | 1<<MSTR); //SPI включен,
мастер,
                        //MSB первый, CPOL=0, CPHA=0
        PORTB &= ~(1<<PB3 | 1<<PB5);
    //инициализация: DAT=0, CLK=0
    }
```

Данная функция настраивает режим работы SPI, частоту передачи, полярность и фазу сигналов.

Собственно передача информации по SPI будет производиться в той же функции SendData, которая заменит ее предыдущий вариант и будет выглядеть следующим образом:

```
void SPI_send (uint8_t data){
    SPDR = data;
    while(!(SPSR & (1<<SPIF)));
}
```

В этой короткой функции байт данных помещается в сдвиговый регистр интерфейса и программа ждет, пока не завершится передача.

Таким образом, в данной лабораторной работе были рассмотрены различные принципы вывода информации на индикаторы, примеры использования встроенных таймеров и основы последовательной передачи данных.

Задания и пояснения к выполнению работы

Напишите программу в соответствии с вариантом, предложенным преподавателем.

1 вариант.

Изучить документацию на MBI5039 (для реализации на отладочной плате) или TLC5925. С её помощью реализовать управление семисегментным индикатором со счётом от 0 до 99 (*использовать программный SPI*).

2 вариант.

Изучить документацию на WS2801 (для реализации на отладочной плате) или 74HC595. Реализовать визуальные эффекты на нескольких светодиодах (*использовать аппаратный SPI*).

Дополнительное задание на оценку «отлично» выдает преподаватель.

Контрольные вопросы

1. Почему нельзя просто подключить несколько индикаторов к портам контроллера?
2. Что такое динамическая индикация? Объясните её принципы и укажите область применения.

3. В состав ATmega168 входят несколько таймеров. Что это и для чего они нужны?

4. Назовите преимущества аппаратного таймера по сравнению с функцией задержки.

5. Что такое ШИМ? Какова область его применения?

6. Что такое регистр? Есть ли отличия между рассматриваемыми регистрами и регистрами контроллера?

7. Назовите виды и область применения регистров. В чем заключаются преимущества двухступенчатых регистров?

8. Для чего используется интерфейс SPI?

9. Как происходит обмен данными по SPI? Рассмотрите случай с несколькими подчиненными устройствами.

10. Как производится инициализация обмена по SPI в микроконтроллере?

3

Последовательный интерфейс UART и АЦП

Теоретические сведения

Контроллер ATmega328 имеет в своем составе *модуль АЦП*. Этот модуль позволяет контроллеру обрабатывать аналоговые величины, что существенно расширяет сферу его применения. Согласно техническому описанию контроллера, АЦП имеет разрядность 10 бит с абсолютной погрешностью ±2 LSB (Least Significant Bit – младший значащий бит), то есть теоретически напряжение может быть измерено с точностью около 0,2 %. Но на практике точность производимых измерений обычно значительно ниже, так как на неё влияет множество факторов, таких как стабильность питания и опорного напряжения, нагрев контроллера, помехи, проникновение цифровых помех в аналоговую часть и так далее.

Отличительные особенности данного АЦП:

- 10-разрядное разрешение;
- интегральная нелинейность 0,5 мл. разр.;
- абсолютная погрешность ±2 мл. разр.;
- время преобразования 13 - 260 мкс;
- 8 (6) мультиплексированных однополярных входов (количество зависит от корпуса микросхемы);
- представление результата с левосторонним или правосторонним выравниванием в 16-разрядном слове;
- диапазон входного напряжения АЦП 0…VCC;
- выборочный внутренний ИОН на 1,1 В;
- режимы одиночного преобразования и автоматического перезапуска;

- прерывание по завершении преобразования АЦП.

АЦП ATmega328 подключен к порту С контроллера через мультиплексор, что позволяет ему опрашивать 6 аналоговых входов. Диапазон измеряемых значений напряжения лежит от нуля до напряжения питания контроллера. Время одного преобразования – от 13 до 260 мкс.

Встроенный АЦП контроллера не подходит для решения задач, связанных с оцифровкой быстро меняющихся сигналов, или задач, требующих высокой точности, – для этого обычно применяются дискретные АЦП, но для остальных задач он подходит.

Структурная схема модуля АЦП приведена на рисунке 23.

АЦП преобразует входное аналоговое напряжение в 10-разрядный код методом последовательных приближений. Минимальное значение соответствует уровню GND, а максимальное – уровню AREF минус 1 младшего разряда. К выводу AREF опционально может быть подключено опорное напряжение от внутреннего ИОН на 1.1В путем записи соответствующих значений в биты REFSn в регистр ADMUX. Несмотря на то, что ИОН находится внутри микроконтроллера, к его выходу может быть подключен блокировочный конденсатор для снижения чувствительности к шумам, т.к. он связан с выводом AREF.

Канал аналогового ввода выбирается путем записи бита MUX в регистр ADMUX. В качестве однополярного аналогового входа АЦП может быть выбран один из входов ADC0…ADC7. АЦП включается установкой бита ADEN в регистре ADCSRA. АЦП не потребляет электроэнергии, когда выключен, поэтому рекомендуется выключать его перед переводом контроллера в один из режимов ожидания.

АЦП генерирует 10-разрядный результат, который помещается в пару регистров данных ADCH и ADCL. По умолчанию результат преобразования размещается в младших 10-ти разрядах 16-разрядного слова (выравнивание справа), но может быть опционально размещен в старших 10-ти разрядах (выравнивание слева) путем установки бита ADLAR в регистре ADMUX.

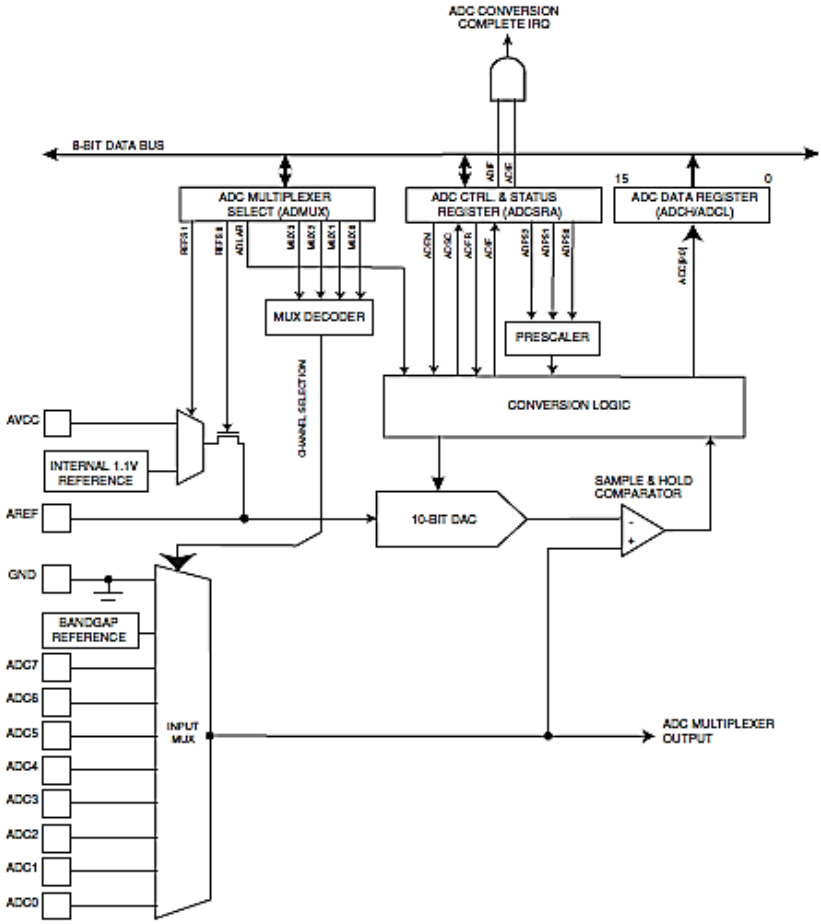


Рисунок 23 – Структурная схема модуля АЦП

Практическая полезность представления результата с выравниванием слева существует, когда достаточно 8-разрядное разрешение, т.к. в этом случае необходимо считать только регистр ADCH. В другом же случае следует первым считать содержимое регистра ADCL, а затем ADCH, чем гарантируется, что оба байта являются результатом одного и того же преобразования. Как только выполнено чтение ADCL, блокируется доступ к регистрам данных со стороны АЦП.

Это означает, что если считан ADCL и преобразование завершается перед чтением регистра ADCH, то ни один из регистров не может модифицироваться и результат преобразования теряется. После чтения ADCH доступ к регистрам ADCH и ADCL со стороны АЦП снова разрешается.

АЦП генерирует собственный запрос на прерывание по завершении преобразования. Если между чтением регистров ADCH и ADCL запрещен доступ к данным для АЦП, то прерывание возникнет, даже если результат преобразования будет потерян.

Одиночное преобразование запускается путем записи лог. 1 в бит запуска преобразования АЦП ADSC. Данный бит остается в высоком состоянии в процессе преобразования и сбрасывается по завершении преобразования. Если в процессе преобразования переключается канал аналогового ввода, то АЦП автоматически завершит текущее преобразование прежде, чем переключит канал.

В режиме автоматического перезапуска АЦП непрерывно оцифровывает аналоговый сигнал и обновляет регистр данных АЦП. Данный режим задается путем записи лог. 1 в бит ADFR регистра ADCSRA. Первое преобразование инициируется путем записи лог. 1 в бит ADSC регистра ADCSRA. В данном режиме АЦП выполняет последовательные преобразования, независимо от того, сбрасывается флаг прерывания АЦП ADIF или нет.

Модуль АЦП содержит предделитель, который формирует производные частоты свыше 100 кГц по отношению к частоте синхронизации ЦПУ. Коэффициент деления устанавливается с помощью бит ADPS в регистре ADCSRA. Предделитель начинает счет с момента включения АЦП установкой бита ADEN в регистре ADCSRA. Предделитель работает, пока бит ADEN = 1, и сброшен, когда ADEN = 0.

Предделитель АЦП устанавливается в соответствии с требуемой точностью преобразования. Если требуется максимальная разрешающая способность, то частота на входе схемы последовательного приближения должна быть в диапазоне 50…200 кГц. Если достаточно разрешение менее 10 разрядов, но требуется более высокая частота преобразования, то частота на входе АЦП может быть установлена свыше 200 кГц.

Если инициируется одиночное преобразование установкой бита ADSC в регистре ADCSRA, то преобразование начинается со следующего нарастающего фронта тактового сигнала АЦП.

Нормальное преобразование требует 13 тактов синхронизации АЦП. Первое преобразование после включения АЦП (установка ADEN в ADCSRA) требует 25 тактов синхронизации АЦП за счет необходимости инициализации аналоговой схемы.

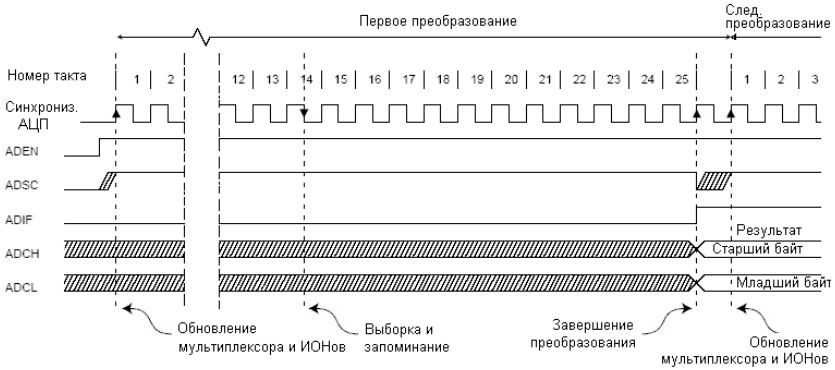


Рисунок 24 – Временная диаграмма работы АЦП при первом преобразовании в режиме одиночного преобразования

Рисунок 25 – Временная диаграмма работы АЦП в режиме одиночного преобразования

После начала нормального преобразования на выборку-хранение затрачивается 1,5 такта синхронизации АЦП, а после начала первого

преобразования – 13,5 тактов. По завершении преобразования результат помещается в регистры данных АЦП и устанавливается флаг ADIF. В режиме одиночного преобразования одновременно сбрасывается бит ADSC. Программно бит ADSC может быть снова установлен, и новое преобразование будет инициировано первым нарастающим фронтом тактового сигнала АЦП.

В режиме автоматического перезапуска новое преобразование начинается сразу по завершении предыдущего, при этом ADSC остается в высоком состоянии. Времена преобразования для различных режимов преобразования представлены в таблице 6.

Таблица 6 – **Время преобразования АЦП**

Тип преобразования	Длительность выборки-хранения (в тактах)	Время преобразования (в тактах)
Первое преобразование	13,5	25
Однократное преобразование	1,5	13
Автоматическое преобразование	2	13,5

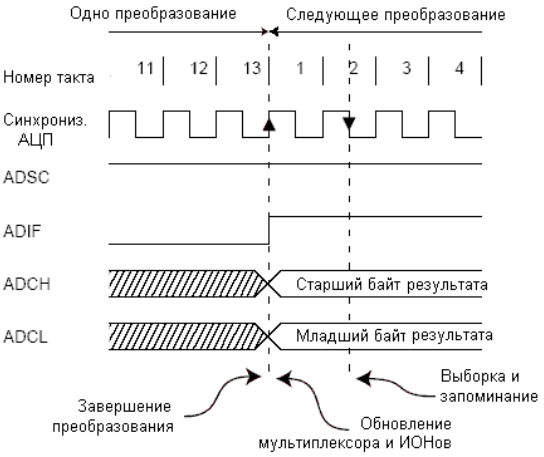


Рисунок 26 – Временная диаграмма работы АЦП в режиме автоматического перезапуска

При переключении входного канала необходимо учесть некоторые рекомендации, которые исключат некорректность переключения.

В режиме одиночного преобразования переключение канала необходимо выполнять перед началом преобразования. Переключение канала может произойти только в течение одного такта синхронизации АЦП после записи лог. 1 в ADSC. Однако самым простым методом является ожидание завершения преобразования перед выбором нового канала.

В режиме автоматического перезапуска канал необходимо выбирать перед началом первого преобразования. Переключение канала происходит аналогично: в течение одного такта синхронизации АЦП после записи лог. 1 в ADSC. Но самым простым методом является ожидание завершения первого преобразования, а затем переключение канала. Поскольку следующее преобразование уже запущено автоматически, то следующий результат будет соответствовать предыдущему каналу. Последующие преобразования отражают результат для нового канала.

Источник опорного напряжения (ИОН) для АЦП определяет диапазон преобразования АЦП. Если уровень однополярного сигнала выше $V_{ИОН}$, то результатом преобразования будет 0x3FF. В качестве $V_{ИОН}$ могут выступать AVCC, внутренний ИОН или внешний ИОН, подключенный к выводу AREF. В любом случае внешний вывод AREF связан непосредственно с АЦП, и поэтому можно снизить влияние шумов на опорный источник за счет подключения конденсатора между выводом AREF и общим. Напряжение $V_{ИОН}$ также может быть измерено на выводе AREF высокоомным вольтметром. Обратите внимание, что $V_{ИОН}$ является высокоомным источником, и поэтому внешне к нему может быть подключена только емкостная нагрузка.

Если пользователь использует внешний опорный источник, подключенный к выводу AREF, то не допускается использование другой опции опорного источника, т.к. это приведет к шунтированию внешнего опорного напряжения. Если к выводу AREF не приложено напряжение, то пользователь может выбрать AVCC и 1,1В качестве опорного источника. Результат первого преобразования после пере-

ключения опорного источника может характеризоваться плохой точностью, и пользователю рекомендуется его игнорировать.

Схема аналогового входа для однополярных каналов представлена на рисунке 27. Независимо от того, какой канал подключен к АЦП, аналоговый сигнал, подключенный к выводу ADCn, нагружается емкостью вывода и входным сопротивлением утечки. После подключения канала к АЦП аналоговый сигнал будет связан с конденсатором выборки-хранения через последовательный резистор, сопротивление которого эквивалентно всей входной цепи.

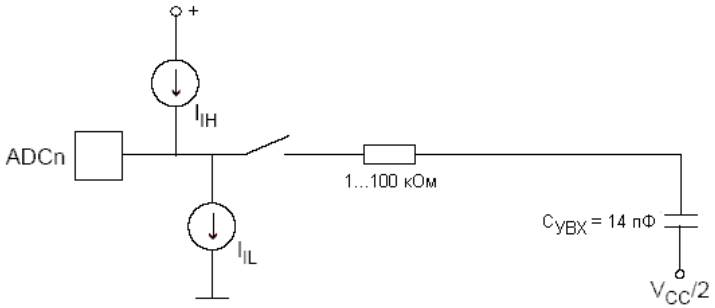


Рисунок 27 – Схема аналогового входа

АЦП оптимизирован под источники аналоговых сигналов с выходным сопротивлением не более 10 кОм. Если используется такой источник, то время выборки незначительно. Если же используется источник с более высоким входным сопротивлением, то время выборки будет определяться временем, которое требуется для зарядки конденсатора выборки-хранения источником аналогового сигнала. Рекомендуется использовать источники только с малым выходным сопротивлением и медленно изменяющимися сигналами, т.к. в этом случае будет достаточно быстрым заряд конденсатора выборки-хранения.

При обработке сигналов необходимо обеспечить выполнение требований теоремы Котельникова, в том числе путем установки фильтра низких частот.

Работа цифровых узлов внутри и снаружи микроконтроллера связана с генерацией электромагнитных излучений, которые могут негативно сказаться на точности измерения аналогового сигнала.

Если точность преобразования является критическим параметром, то уровень шумов можно снизить, придерживаясь следующих рекомендаций:

1. Пути прохождения аналоговых сигналов должны быть как можно более короткими. Аналоговые сигналы должны проходить над плоскостью (слоем) с аналоговой землей (экраном) и далеко от проводников, передающих высокочастотные цифровые сигналы.

2. Вывод AVCC необходимо связать с цифровым питанием VCC через LC-цепь в соответствии с рисунком 28.

3. Рекомендуется использовать функцию подавления шумов АЦП, внесенных работой ядра ЦПУ.

4. Если какой-либо из выводов АЦП используется как цифровой выход, то чрезвычайно важно не допускать переключения состояния этого выхода в процессе преобразования.

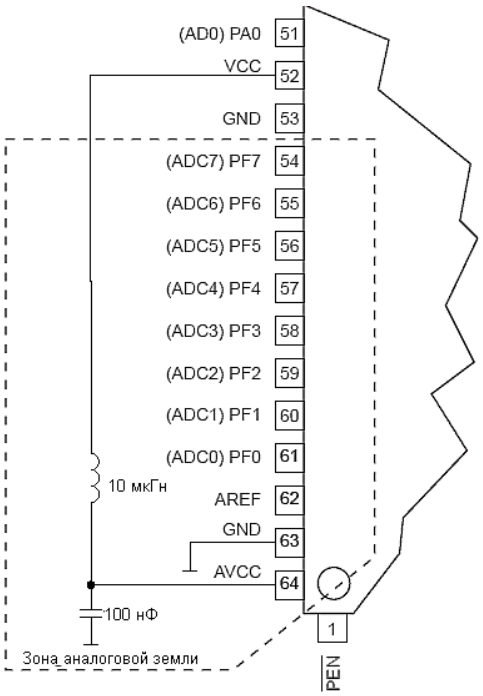


Рисунок 28 – Подключение питания АЦП

Определение погрешностей аналогово-цифрового преобразования

N-разрядный однополярный АЦП преобразует напряжение линейно между GND и $V_{ИОН}$ с количеством шагов 2^n (младших разрядов). Минимальный код = 0, максимальный = 2^n-1. Основные погрешности преобразования являются отклонением реальной функции преобразования от идеальной. К ним относятся:

- погрешность квантования;
- абсолютная погрешность;
- смещение;
- погрешность усиления;
- интегральная нелинейность;
- дифференциальная нелинейность.

Погрешность квантования. Возникает из-за преобразования входного напряжения в конечное число кодов. Погрешность квантования – интервал входного напряжения протяженностью 1 младший разряд (шаг квантования по напряжению), который характеризуется одним и тем же кодом. Всегда равен ±0,5 младшего разряда.

Абсолютная погрешность. Максимальное отклонение реальной (без подстройки) функции преобразования от реальной при любом коде. Является результатом действия нескольких эффектов: смещение, погрешность усиления, дифференциальная погрешность, нелинейность и погрешность квантования. Идеальное значение: ±0,5 младшего разряда.

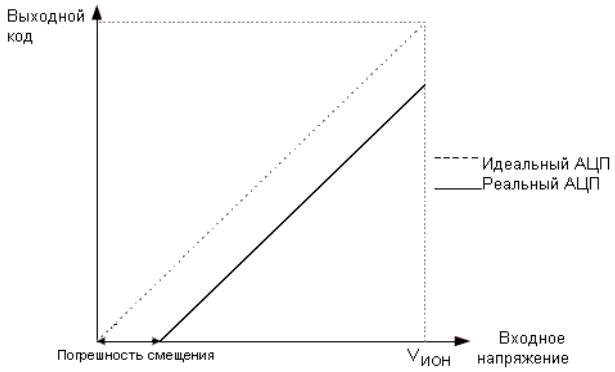


Рисунок 29 – Погрешность смещения

Смещение – отклонение первого перехода (с 0x000 на 0x001) по сравнению с идеальным переходом (т.е. при 0,5 младшего разряда). Идеальное значение: 0 младшего разряда.

Погрешность усиления. После корректировки смещения погрешность усиления представляет собой отклонение последнего перехода (с 0x3FE на 0x3FF) от идеального перехода (т.е. отклонение при максимальном значении минус 1,5 младшего разряда). Идеальное значение: 0 младшего разряда.

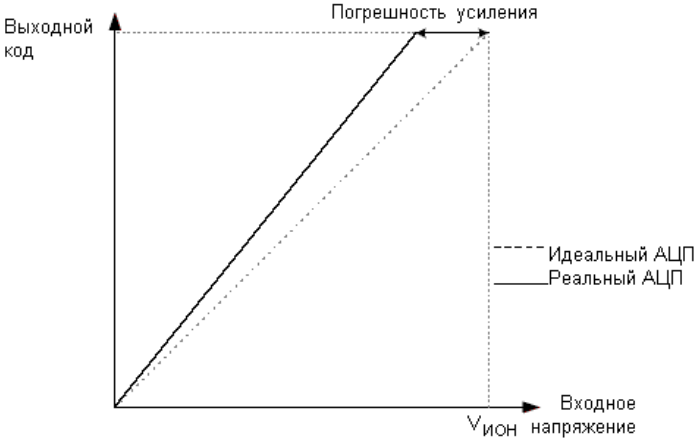


Рисунок 30 – Погрешность усиления

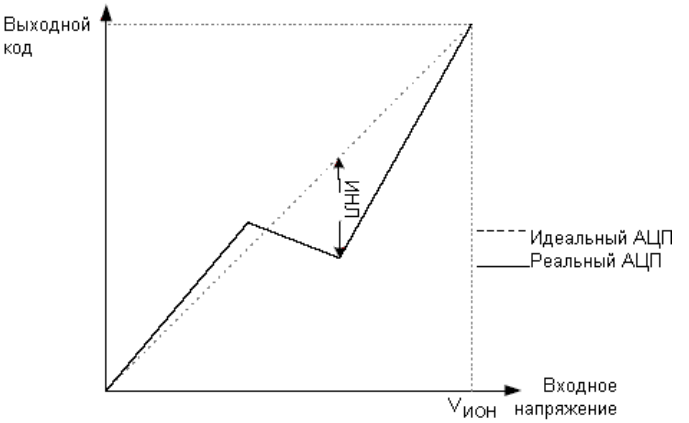


Рисунок 31 – Интегральная нелинейность (ИНЛ)

Интегральная нелинейность (ИНЛ). После корректировки смещения и погрешности усиления, ИНЛ представляет собой максимальное отклонение реальной функции преобразования от идеальной для любого кода. Идеальное значение ИНЛ – 0 младшего разряда.

Дифференциальная нелинейность (ДНЛ). Максимальное отклонение между шириной фактического кода (интервал между двумя смежными переходами) от ширины идеального кода (1 младшего разряда). Идеальное значение: 0 младшего разряда.

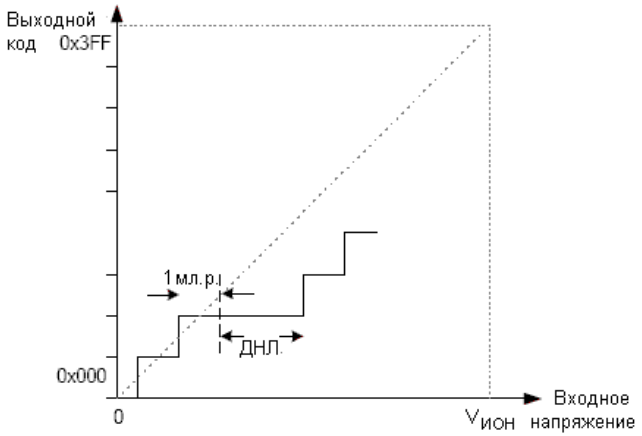


Рисунок 32 – Дифференциальная нелинейность (ДНЛ)

Результат преобразования АЦП

По завершении преобразования (ADIF = 1) результат может быть считан из пары регистров результата преобразования АЦП (ADCL, ADCH).

Для однополярного преобразования

$$ADC = \frac{V_{IN} \cdot 1024}{V_{REF}},$$

где V_{IN} – уровень напряжения на подключенном к АЦП входу; V_{REF} – напряжение выбранного источника опорного напряжения (см. табл. 7 и табл. 8). Код 0x000 соответствует уровню аналоговой земли, а 0x3FF – уровню напряжения ИОН минус 1 шаг квантования по напряжению.

Регистры АЦП

Регистр управления мультиплексором АЦП (ADMUX).

Bit	7	6	5	4	3	2	1	0
(0x7C)	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0
Read/write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W
Initial value	0	0	0	0	0	0	0	0

Разряды 7 и 6 – REFS1:0: Биты выбора источника опорного напряжения.

Данные биты определяют, какое напряжение будет использоваться в качестве опорного для АЦП (см. табл. 7). Если изменить значения данных бит в процессе преобразования, то новые установки вступят в силу только по завершении текущего преобразования (т.е. когда установится бит ADIF в регистре ADCSRA). Внутренний ИОН можно не использовать, если к выводу AREF подключен внешний опорный источник.

Таблица 7 – **Выбор опорного источника АЦП**

REFS1	REFS0	Опорный источник
0	0	AREF, внутренний ИОН отключен
0	1	AVCC с внешним конденсатором на выводе AREF
1	0	Зарезервировано
1	1	Внутренний источник опорного напряжения 1,1 В с внешним конденсатором на выводе AREF

Разряд 5 – ADLAR: Бит управления представлением результата преобразования.

Бит ADLAR влияет на представление результата преобразования в паре регистров результата преобразования АЦП. Если ADLAR = 1, то результат преобразования будет иметь левосторонний формат, в противном случае – правосторонний. Действие бита ADLAR вступает в силу сразу после изменения, независимо от выполняющегося параллельно преобразования.

Разряд 4 – зарезервировано.

Разряды 3..0 – MUX3..0: Биты выбора аналогового канала и коэффициента усиления.

Данные биты определяют, какие из имеющихся аналоговых входов подключаются к АЦП. Кроме того, с их помощью можно выбрать коэффициент усиления для дифференциальных каналов (см. табл. 8). Если значения бит изменить в процессе преобразования, то механизм их действия вступит в силу только после завершения текущего преобразования (после установки бита ADIF в регистре ADCSRA).

Таблица 8 – **Выбор входного канала**

MUX3..0	Входной канал
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5
0110	ADC6
0111	ADC7
1000	Зарезервировано
1001	Зарезервировано
1010	Зарезервировано
1011	Зарезервировано
1100	Зарезервировано
1101	Зарезервировано
1110	1,1 В (V_{BG})
1111	0 В (GND)

Регистр А управления и статуса АЦП (ADCSRA).

Bit	7	6	5	4	3	2	1	0
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
Read/write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial value	0	0	0	0	0	0	0	0

Бит 7 – ADEN: Разрешение работы АЦП.

Запись в данный бит лог. 1 разрешает работу АЦП. Если в данный бит записать лог. 0, то АЦП отключается, даже если он находился в процессе преобразования.

Бит 6 – ADSC: Запуск преобразования АЦП.

В режиме одиночного преобразования установка данного бита инициирует старт каждого преобразования. В режиме автоматического перезапуска установкой этого бита инициируется только первое преобразование, а все остальные выполняются автоматически. Первое преобразование после разрешения работы АЦП, инициированное битом ADSC, выполняется по расширенному алгоритму и длится 25 тактов синхронизации АЦП, вместо обычных 13 тактов. Это связано с необходимостью инициализации АЦП.

В процессе преобразования при опросе бита ADSC возвращается лог. 1, а по завершении преобразования – лог. 0. Запись лог. 0 в данный бит не предусмотрена и не оказывает никакого действия.

Бит 5 – ADATE: Выбор режима автоматического перезапуска АЦП.

Если в данный бит записать лог. 1, то АЦП перейдет в режим автоматического перезапуска. В этом режиме АЦП автоматически выполняет преобразования и модифицирует регистры результата преобразования через фиксированные промежутки времени. Запись лог. 0 в этот бит прекращает работу в данном режиме. Выбор источника импульсов запуска производится установкой битов ADTS в регистре ADCSRB.

Бит 4 – ADIF: Флаг прерывания АЦП.

Данный флаг устанавливается после завершения преобразования АЦП и обновления регистров данных. Если установлены биты ADIE и I (регистр SREG), то происходит прерывание по завершении преобразования. Флаг ADIF сбрасывается аппаратно при переходе на соответствующий вектор прерывания. Альтернативно флаг ADIF сбрасывается путем записи лог. 1 в него.

!Обратите внимание, что при выполнении команды «чтение-модификация-запись» с регистром ADCSRA ожидаемое прерывание может быть отключено.

Бит 3 – ADIE: Разрешение прерывания АЦП.

После записи лог. 1 в этот бит, при условии глобального разрешения прерываний, разрешается прерывание по завершении преобразования АЦП.

Биты 2..0 – ADPS2:0: Биты управления предделителем АЦП.

Данные биты определяют, на какое значение тактовая частота ЦПУ будет отличаться от частоты входной синхронизации АЦП.

Таблица 9 – **Управление предделителем АЦП**

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

Регистры данных АЦП (ADCL и ADCH)

ADLAR = 0

Bit	15	14	13	12	11	10	9	8	
(0x79)	–	–	–	–	–	–	ADC9	ADC8	ADCH
(0x78)	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 1

Bit	15	14	13	12	11	10	9	8	
(0x79)	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
(0x78)	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

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

Если выполнено чтение ADCL, то доступ к этим регистрам для АЦП будет заблокирован (т.е. АЦП не сможет в дальнейшем модифицировать результат преобразования), пока не будет считан регистр ADCH.

В зависимости от значения бита ADLAR, результат представлен в левостороннем или правостороннем формате, которые отличаются выравниванием результата в регистрах. Левосторонний формат представления результата удобно использовать, если достаточно 8 разрядов. В этом случае 8-разрядный результат хранится в регистре ADCH и, следовательно, чтение регистра ADCL можно не выполнять. При правостороннем формате необходимо сначала считать ADCL, а затем ADCH.

Результат преобразования обозначен как ADC9:0.

Регистр B управления и статуса АЦП (ADCSRB)

Bit	7	6	5	4	3	2	1	0
(0x7B)	–	ACME	–	–	–	ADTS2	ADTS1	ADTS0
Read/write	R	R/W	R	R	R	R/W	R/W	R/W
Initial value	0	0	0	0	0	0	0	0

Биты 7, 5..3 – зарезервированы.

Биты 2..0: ADTS2..0 – выбор источника импульсов автоматического запуска преобразования. Если установлен бит ADATE в регистре ADCSRA, значения данных бит выбирают источник импульсов запуска АЦП. Запуск преобразования будет производиться при установке соответствующего флага прерывания.

Таблица 10 – **Управление источником запуска преобразования АЦП**

ADTS2	ADTS1	ADTS0	Источник запуска
0	0	0	Свободный режим
0	0	1	Аналоговый компаратор
0	1	0	Внешнее прерывание EXT0
0	1	1	Срабатывание компаратора A таймера 0
1	0	0	Переполнение таймера 0
1	0	1	Срабатывание компаратора B таймера 1
1	1	0	Переполнение таймера 1
1	1	1	Событие захвата таймера 1

В данной работе модуль АЦП будет использован для измерения напряжения на переменном резисторе. Далее значение измеренного напряжения будет выводиться на индикаторы.

Для работы с модулем АЦП необходимо ознакомиться с его регистрами.

Так как АЦП имеет разрядность 10 бит, для представления результата преобразования используются два 8-битных регистра – ADCH и ADCL, содержащих соответственно старшую и младшую части результата.

ADMUX – в данном регистре содержится набор битов, отвечающих за выбор опрашиваемого канала, настройку опорного напряжения и представление результата. Интересен бит ADLAR, при включении которого результат преобразования начинает помещаться в выходные регистры так, что 8 старших бит оказываются в регистре ADCH, а два младших – в ADCL. Таким образом, можно искусственно ограничить точность преобразования АЦП, обрабатывая только данные регистра ADCH, получив взамен возможность простой работы с результатом.

ADCSRA – регистр содержит биты, позволяющие включить АЦП, запустить преобразование, установить режим работы АЦП, разрешить прерывание по окончании преобразования и установить значение делителя частоты.

ADCSRB – регистр содержит биты, отвечающие за выбор события, автоматически вызывающего запуск преобразовании АЦП. При программном запуске преобразования в данном регистре ничего менять не требуется.

Фрагмент схемы, на основе которой демонстрируется работа АЦП, приведен на рисунке 33.

В схему добавлен потенциометр RV1, в библиотеках Proteus он может быть найден по имени **POT-HG**. Один из входов потенциометра подключен к источнику питания, второй – заземлен, а выход движка соединен со входом АЦП микроконтроллера. Это позволяет подавать на вход АЦП любое напряжение в пределах от нуля до величины напряжения источника питания.

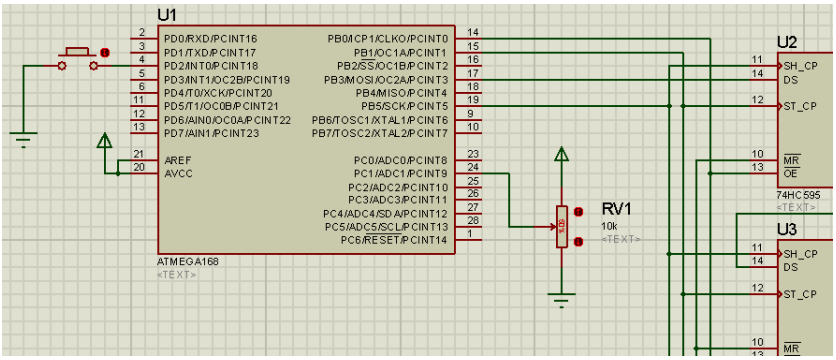


Рисунок 33 – Работа с АЦП

Блок UART

Еще одним важным блоком контроллера ATmega328 является интерфейс UART. Это интерфейс асинхронной последовательной передачи данных. В отличие от синхронного интерфейса SPI, в интерфейсе UART не используется линия передачи тактовых сигналов. Вместо этого приемник и передатчик должны быть синхронизированы по частоте так, чтобы прием и передача велись с одной, заранее оговоренной скоростью. Скорость UART для конкретного соединения указывается в бодах (что в данном случае соответствует битам в секунду). Существует общепринятый ряд стандартных скоростей: 300; 600; 1200; 2400; 4800; 9600; 19200; 38400; 57600; 115200; 230400; 460800; 921600 бод. Также UART позволяет связать между собой только два устройства.

Кроме данных, UART отправляет служебные биты, служащие для синхронизации, – так называемые *стартовый* и *стоповый* биты. При приёме эти биты не попадают в приемный буфер. Минимальная посылка по UART обычно состоит из одного байта, обрамленного стартовым и стоповым битами. Однако встречаются реализации UART, которые позволяют передавать по 5, 6, 7, 8 или 9 бит. Некоторые реализации UART дают возможность вставлять два стоповых бита при передаче для уменьшения вероятности рассинхронизации приёмника и передатчика при плотном трафике. Приёмник игнорирует второй стоповый бит, воспринимая его как короткую паузу на линии.

Принято соглашение, что пассивным (в отсутствие потока данных) состоянием входа и выхода UART является логическая 1. Стартовый бит всегда логический 0, поэтому приёмник UART ждёт перепада из 1 в 0 и отсчитывает от него временной промежуток в половину длительности бита (середина передачи стартового бита). Если в этот момент на входе всё ещё 0, то запускается процесс приёма минимальной посылки. Для этого приёмник отсчитывает 9 битовых длительностей подряд (для 8-бит данных) и в каждый момент фиксирует состояние входа. Первые 8 значений являются принятыми данными, последнее значение проверочное (стоп-бит). Значение стоп-бита всегда 1; если реально принятое значение иное, UART фиксирует ошибку.

Для формирования временных интервалов передающий и приёмный UART имеют точные источники тактирования. Точность этих источников должна быть такой, чтобы сумма погрешностей (приёмника и передатчика) установки временного интервала от начала стартового импульса до середины стопового импульса не превышала половины (а лучше хотя бы четверти) битового интервала. Рекомендуемый допуск на точность тактирования UART составляет не более 1,5 %.

Самым известным примером использования UART служит COM-порт персонального компьютера. Его логика работы в точности совпадает с UART, однако отличаются уровни рабочих напряжений – COM-порт, он же RS-232, работает с уровнями -5…-15 В в качестве логического нуля и +5…+15 В в качестве логической единицы. Для согласования уровней напряжений промышленность выпускает интегральные микросхемы согласования, например, MAX232.

Таким образом, интерфейс RS-232 может быть использован для подключения контроллера к персональному компьютеру. Однако в современных условиях интерфейс RS-232 повсеместно вытесняется интерфейсом USB, в то время как реализация USB присутствует лишь на более мощных контроллерах, а работа с ним достаточно трудоемка. Для разрешения данной проблемы выпускаются микро-

схемы-переходники с интерфейса USB на UART. Примером такой микросхемы может служить FT232RL. Данная микросхема позволяет подключить необходимое устройство к ПК через интерфейс UART путем создания на ПК виртуального COM-порта, работа с которым не отличается от работы с физическим COM-портом. Таким образом, с точки зрения целевого устройства, работа ведется через интерфейс UART.

У ATmega есть регистр UDR (UART Data Register). На самом деле это два разных регистра, но имеют один адрес. При записи данные попадает в один регистр (регистр передатчика), а при чтении данные вычитываются из другого регистра (регистр приемника).

Когда байт принят в регистр UDR, может сработать прерывание по завершению приема (если оно разрешено), которое вызывается сразу после получения приемником всех битов данных (обычно 8, но может быть и 9, в зависимости от настройки UART).

Поскольку передача во времени происходит не мгновенно, то необходимо дождаться окончания передачи предыдущего байта. О том, что UDR пуст и готов к приему нового байта, сигнализирует бит UDRE, он же вызовет аппаратное прерывание по опустошению буфера.

Настройка UART

Все настройки приемопередатчика хранятся в регистрах конфигурации: UCSRA, UCSRB и UCSRC. Скорость задается в паре регистров UBBRH:UBBRL.

Регистр А управления и статуса UART (UCSRA).

Bit	7	6	5	4	3	2	1	0
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn
Read/write	R	R/W	R	R	R	R	R/W	R/W
Initial value	0	0	1	0	0	0	0	0

Бит 7 – RXCn – флаг окончания приема.

Устанавливается в случае, когда в приемном буфере есть непрочитанная информация. Снимается автоматически при опустошении буфера.

Бит 6 – TXCn – флаг окончания передачи.

Устанавливается, когда вся необходимая информация передана и выходной буфер пуст (информация уже передана, новая еще не поступила).

Бит 5 – UDREn – флаг пустого буфера.

Показывает, что передающий буфер пуст и готов к приему новой информации.

Бит 4 – FEn – ошибка кадрирования.

Данный флаг выставляется, если ожидался стоп бит (лог. 1), а принят 0. При записи в регистр UCSRA данный бит всегда должен быть равен 0.

Бит 3 – ORn – переполнение буфера.

Ошибка возникает в случае, если данные поступают быстрее, чем программа успела их читать из UDR.

Бит 2 – PEn – ошибка контроля четности.

Бит устанавливается в 1 при нарушении заданного правила контроля четности. При записи в UCSRA бит должен быть равен 0.

Бит 1 – U2Xn – бит удвоения скорости при работе в асинхронном режиме. Его надо учитывать при расчете значения в UBRRH:UBRRL

Бит 0 – MPCMn – режим мультипроцессорной связи.

Когда он установлен, все входящие фреймы данных игнорируются, если не содержат адресной информации.

Регистр B управления и статуса UART (UCSRB).

Bit	7	6	5	4	3	2	1	0
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n
Read/write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
Initial value	0	0	0	0	0	0	0	0

Бит 7 – RXCIEn – включение прерывания по окончании приема.

Бит 6 – TXCIEn – включение прерывания по окончании передачи.

Бит 5 – UDRIEn – включение прерывания по опустошении буфера.

Бит 4 – RXENn – включение приемника.

Бит 3 – TXENn – включение передатчика.

Бит 2 – UCSZn2 – совместно с битом UCSZn1:0 в регистре UCSRnC устанавливает количество бит данных в посылке.

Бит 1 – RXB8n – девятый принятый бит данных при работе с 9-битными посылками.

Бит 0 – TXB8n – девятый передаваемый бит данных при работе с 9-битными посылками.

Регистр C управления и статуса UART (UCSRC)

Bit	7	6	5	4	3	2	1	0
	UMSELn1	UMSELn0	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn
Read/write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial value	0	0	0	0	0	1	1	0

Биты 7..6 – UMSELn1:0 – установка режима работы UART. Варианты настройки приведены в таблице 11.

Таблица 11 – **Режимы работы UART**

UMSELn1	UMSELn0	Режим
0	0	Асинхронный USART
0	1	Синхронный USART
1	0	Зарезервировано
1	1	Мастер SPI

Биты 5..4 – UPMn1:0 – Режим проверки четности. Если проверка четности включена, передатчик UART автоматически вычисляет и добавляет к каждой посылке данных бит четности. Приёмник проверяет входные данные в соответствии с настройками, указанными в таблице 12, и в случае обнаружения несоответствия поднимает флаг UPEn в регистре UCSRnA.

Таблица 12 – **Режимы контроля четности**

UPMn1	UPMn0	Режим проверки четности
0	0	Выключение
0	1	Зарезервировано
1	0	Включен, проверка на четность
1	1	Включен, проверка на нечетность

Бит 3 – USBSn – Настройка стоп-бита.

– 0 – 1 стоп-бит;

– 1 – 2 стоп-бита.

Биты 2..1 – UCSZn1:0 – Размер посылки.

Задает количество бит в одной посылке данных. Количество бит может меняться от 5 до 9 бит на посылку. Комбинации флагов для установки необходимого значения приведены в таблице 13.

Таблица 13 – **Установка количества бит в посылке**

UCSZn2	UCSZn1	UCSZn0	Размер посылки данных
0	0	0	5 бит
0	0	1	6 бит
0	1	0	7 бит
0	1	1	8 бит
1	0	0	Зарезервировано
1	0	1	Зарезервировано
1	1	0	Зарезервировано
1	1	1	9 бит

Бит 0 – UCPOLn – Полярность тактового сигнала.

Данный бит имеет значение только при использовании режима синхронного обмена.

Регистры установки скорости обмена (UBRRnL и UBRRnH).

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	UBRRn[11:8]				UBRRnH
	UBRRn[7:0]								UBRRnL
	7	6	5	4	3	2	1	0	
Read/write	R	R	R	R	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Биты с 11 по 0 содержат значение скорости обмена, при этом младший байт соответствует регистру UBRRnL, а 4 старших бита – регистру UBRRnH.

Для стандартных частот обмена расхождение в скорости обмена не должно превышать 0,5% и рассчитывается по формуле

$$\text{Ошибка}[\%] = \left(\frac{\text{Установленная скорость}}{\text{Стандартная скорость}} - 1\right) \cdot 100\%.$$

Для работы с UART необходимо использовать два вывода контролера – RXD и TXD. По линии RXD производится прием данных, по TXD – передача. Таким образом, для соединения двух устройств по UART необходимо соединить выводы RXD и TXD крест-накрест, чтобы линия передачи одного устройства была подключена к линии приема второго, и наоборот.

Практическая часть

Программа для работы с АЦП выглядит следующим образом:

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

uint8_t segments[]={
    // GFEDCBA
    0b00111111,   // 0 - A, B, C, D, E, F
    0b00000110,   // 1 - B, C
    0b01011011,   // 2 - A, B, D, E, G
    0b01001111,   // 3 - A, B, C, D, G
    0b01100110,   // 4 - B, C, F, G
    0b01101101,   // 5 - A, C, D, F, G
    0b01111101,   // 6 - A, C, D, E, F, G
    0b00000111,   // 7 - A, B, C
    0b01111111,   // 8 - A, B, C, D, E, F, G
    0b01101111,   // 9 - A, B, C, D, F, G
};

void InitPorts(void);
void InitTimer1(void);
void Bin2Dec(uint16_t data);
```

```
    void SendData(uint8_t data);
    void DisplayData(uint16_t data);
    void InitSPI(void);
    void InitADC(void);

    volatile uint8_t bcd_buffer[] = {0,0,0,0};
    volatile uint16_t display_val = 0;

    int main(void){
        InitPorts();
        InitSPI();
        InitTimer1();
        EIMSK |= (1<<INT0);         //Enable INT0
        EICRA |= (1<<ISC01); //Trigger on falling
edge of INT0
        InitADC();

        sei();        //global interrupt enable
        PORTB &= ~(1<<PB0);   //OE = low (active)
        DisplayData(0);
        while(1){
            DisplayData(display_val);
        }
    }
    //------------------------------------------------

    ISR(TIMER1_COMPB_vect){

    }

    ISR(INT0_vect){

    }

    ISR(ADC_vect){
        display_val=ADC;
```

```
}
//-------------------------------------------

void InitPorts(void){
    DDRB = (1<<PB0|1<<PB1|1<<PB3|1<<PB5);
    DDRD = (0<<PD2);
    PORTD |= (1<<PD2);
}

void InitTimer1(void){
    TCCR1A = 0;
    TCCR1B = (1<<CS11|1<<CS10|1<<WGM12);
    TCNT1 = 0;
    TIMSK1 |= (1<<OCIE1B);
    OCR1A = 1562;
    OCR1B = 1562;
}

void Bin2Dec(uint16_t data){
    bcd_buffer[3] = (uint8_t)(data/1000);
    data = data % 1000;
    bcd_buffer[2] = (uint8_t)(data/100);
    data = data % 100;
    bcd_buffer[1] = (uint8_t)(data/10);
    data = data % 10;
    bcd_buffer[0] = (uint8_t)(data);
}

void SendData(uint8_t data){
    SPDR = data;
    while(!(SPSR & (1<<SPIF)));
}

void DisplayData(uint16_t data){
    Bin2Dec(data);
    PORTB &= ~(1<<PB1);         //clk_out = 0
```

```
        SendData(segments[bcd_buffer[0]]);
        SendData(segments[bcd_buffer[1]]);
        SendData(segments[bcd_buffer[2]]);
        SendData(segments[bcd_buffer[3]]);
        PORTB |= (1<<PB1);          //clk_out = 1
    }

    void InitSPI(void){
        DDRB |= (1<<PB3|1<<PB5); //configure MOSI
and CLK as out
        SPSR |= (1<<SPI2X);      //Fclk = Fosc/2
        SPCR = (1<<SPE|1<<MSTR);   //SPI enable,
master mode,
        PORTB &= ~(1<<PB3|1<<PB5); //init values -
DAT low, CLK low
    }

    void InitADC(void){
        ADMUX = (1<<MUX0);    //Align left, ADC1
        ADCSRB = (1<<ADTS2|1<<ADTS0);    //Start on
Timer1 COMPB
        //Enable, auto update, IRQ enable
        ADCSRA = (1<<ADEN|1<<ADATE|1<<ADIE);
    }
```

Из программы удалены части кода, которые отвечали за работу секундомера и реакцию на нажатие кнопки. Теперь Таймер 1 настроен на срабатывание раз в 100 мс. Срабатывание таймера запускает преобразование АЦП (это настроено установкой бит ADTS2 и ADTS0 в регистре ADCSRB). После окончания преобразования вызывается соответствующее прерывание АЦП, в котором выделенной переменной `display_val` присваивается значение, полученное при измерении. Стоит отметить, что АЦП используется в 10-битном формате, потому выделенная переменная 16-битная и в нее сохраняется содержимое обоих регистров данных АЦП – ADCH

и ADCL. Чтение данных двух регистров имеет свою особенность: при поочередном их чтении сначала должен быть прочитан ADCL, и только потом – ADCH. При записи вида `display_val=ADC;` за этим следит компилятор.

В основном цикле программы происходит периодический вывод переменной на индикаторы.

Таким образом, блок АЦП обладает широкими возможностями по настройке и позволяет получить необходимую логику работы.

UART

Для демонстрации работы с UART к соответствующим выводам контроллера в Proteus подключается виртуальный прибор – **Virtual Terminal** (рис. 34).

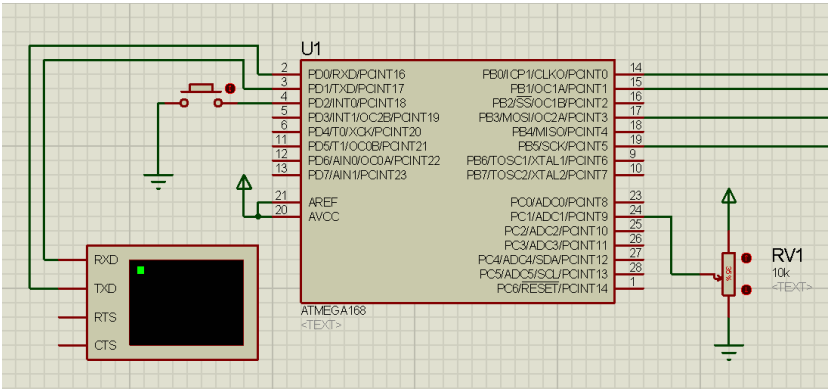


Рисунок 34 – Подключение виртуального прибора

В данном терминале будет отображаться вся информация, отправленная контроллером по UART. Настройки UART контроллера и терминала должны совпадать.

Для настройки UART служит функция `InitUSART`:

```
void InitUSART(){
    UCSR0B = (1<<RXEN0|1<<TXEN0);
    UCSR0C = (1<<UCSZ01|1<<UCSZ00);
    UBRR0H = 0;
    UBRR0L = 0x0C;
}
```

В данной функции осуществляется настройка скорости обмена данными, включаются прием и передача информации, настраивается количество бит данных и служебных бит. Установлены следующие параметры: скорость 4800 бод, 8 бит данных, 1 старт-бит, 1 стоп-бит, контроль четности отключен. Такие же настойки должны быть установлены для виртуального терминала.

Передача одного байта данных осуществляется с помощью функции `SendChar`:

```
void SendChar(char symbol){
    while(!(UCSR0A & (1<<UDRE0))); //wait un-
til flag cleared
    UDR0 = symbol;
}
```

В данной функции сначала осуществляется проверка состояния флага – бита UDRE0 в регистре UCSR0A. Этот бит указывает на состояние регистра данных: когда он установлен, регистр данных пуст. Таким образом, функция ждет, пока освободится регистр данных, а затем помещает в него байт, который должен быть отправлен.

Для отправки текстовой строки предназначена функция `SendString`:

```
void SendString(char* buffer){
    while(*buffer != 0){
        SendChar(*buffer++);
    }
}
```

Данная функция принимает на входе указатель на массив элементов типа char и отправляет их по очереди с помощью вышеописанной функции `SendChar`.

Используя указанные функции, легко можно отправлять информацию по UART. Так, для отправки данных с АЦП при нажатии на кнопку, в обработчик нажатия помещается следующий код:

```
ISR(INT0_vect){
    SendString("Value = ");
    SendChar(0x30 + bcd_buffer[3]);
```

```
        SendChar(0x30 + bcd_buffer[2]);
        SendChar(0x30 + bcd_buffer[1]);
        SendChar(0x30 + bcd_buffer[0]);
        SendString("\r\n");
    }
```

К данным `bcd_buffer` прибавляется значение 0x30, потому что символы, выводимые в терминале, соответствуют кодировке ASCII, а в таблице ASCII символы цифр начинаются с позиции 0x30.

Результат отображается в терминале (рис. 35).

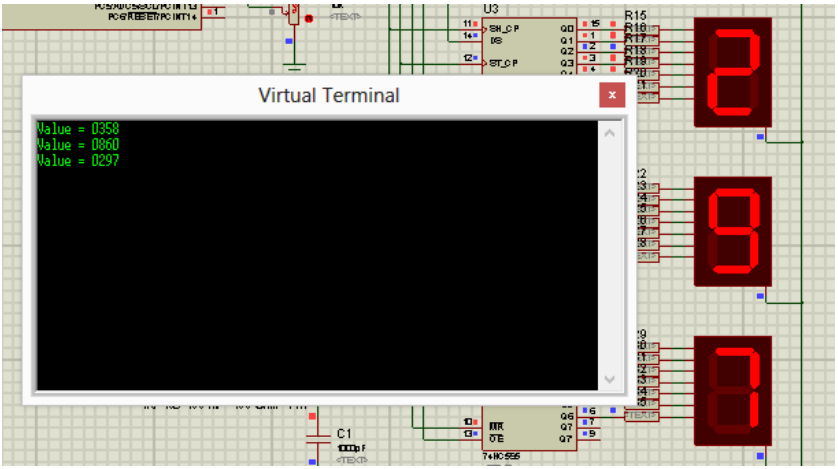


Рисунок 35 – Отправка результата измерений по UART

Интерфейсу UART безразлично, какая информация через него передается – в любом случае она представляется в двоичной форме. Так, чтобы отобразить на экране терминала слово «Hello» можно поступить двумя способами:

1) `SendString("Hello");`

2)
```
buf[] = {0x48, 0x65, 0x6C, 0x6C, 0x6F};
SendString(buf);
```

В обоих случаях по UART будет отправлена одинаковая информация, только в первом случае задачу формирования ASCII кодов из текстовой строки берет на себя компилятор.

Виртуальный терминал Proteus может не только принимать данные, но и передавать – он передает коды символов, соответствующие нажатым клавишам. Так, при нажатии на пробел в UART будет отправлен код 0x20 – код пробела в таблице ASCII.

Таким образом, при помощи прерывания, настроенного на прием, можно осуществлять прием информации. Прерывание по приему включается в регистре UCSR0B:

```
UCSR0B=(1<<RXEN0|1<<TXEN0|1<<RXCIE0);
```

Вектор прерывания называется `USART_RX_vect`, и после каждого приема байта необходимо читать регистр данных UDR0, чтобы сбросить флаг прерывания, иначе оно будет вызываться постоянно.

В данном обработчике прерывания, если принятый символ соответствует нажатию на пробел, в UART отправляется ответное сообщение.

```
ISR(USART_RX_vect){
    if(UDR0 == 0x20){
        SendString("It was SPACE button\r\n");
    }
}
```

Используя это прерывание, можно создать функцию приема строки, но следует иметь в виду, что необходимо задать какой-либо признак конца передачи строки, так как для UART передаваемая информация не является связанной и каждый байт передается независимо от остальных. Так что при приеме строки необходимо либо заранее оговаривать длину передаваемой строки, либо считать признаком конца строки какой-либо символ.

Полный текст программы:

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

uint8_t segments[] = {
    // GFEDCBA
```

```
        0b00111111,   // 0 - A, B, C, D, E, F
        0b00000110,   // 1 - B, C
        0b01011011,   // 2 - A, B, D, E, G
        0b01001111,   // 3 - A, B, C, D, G
        0b01100110,   // 4 - B, C, F, G
        0b01101101,   // 5 - A, C, D, F, G
        0b01111101,   // 6 - A, C, D, E, F, G
        0b00000111,   // 7 - A, B, C
        0b01111111,   // 8 - A, B, C, D, E, F, G
        0b01101111,   // 9 - A, B, C, D, F, G
    };

    void InitPorts(void);
    void InitTimer1(void);
    void Bin2Dec(uint16_t data);
    void SendData(uint8_t data);
    void DisplayData(uint16_t data);
    void InitSPI(void);
    void InitADC(void);

    void InitUSART(void);
    void SendChar(char symbol);
    void SendString(char * buffer);

    volatile uint8_t bcd_buffer[] = {0,0,0,0};
    volatile uint16_t display_val = 0;

    int main(void)
    {
        InitPorts();
        InitSPI();
        InitTimer1();
        EIMSK |= (1<<INT0);  //Enable INT0
        EICRA |= (1<<ISC01); //Trigger on falling
edge of INT0
        InitADC();
```

```
    InitUSART();

    sei();        //global interrupt enable
    PORTB &= ~(1<<PB0);   //OE = low (active)
    DisplayData(0);
    SendString("Hello\r\n");
    while(1)
    {
        DisplayData(display_val);
    }
}
//-----------------------------------------------

ISR(TIMER1_COMPB_vect){}
ISR(INT0_vect){
    SendString("Value = ");
    SendChar(0x30 + bcd_buffer[3]);
    SendChar(0x30 + bcd_buffer[2]);
    SendChar(0x30 + bcd_buffer[1]);
    SendChar(0x30 + bcd_buffer[0]);
    SendString("\r\n");
}

ISR(ADC_vect){
    display_val = ADC;
}

ISR(USART_RX_vect){
    if(UDR0 == 0x20){
        SendString("Roger that\r\n");
    }
}
//-----------------------------------------------

void InitPorts(void){
```

```
        DDRB = (1<<PB0 | 1<<PB1 | 1<<PB3 |
1<<PB5);
        DDRD = (0<<PD2);
        PORTD |= (1<<PD2);
    }

    void InitTimer1( void){
        TCCR1A = 0;                 //CTC mode -
Clear Timer on Compare
        //prescaler = sys_clk/64
        TCCR1B = (1<<CS11 | 1<<CS10 | 1<<WGM12);

        TCNT1 = 0;          //start value of counter
        TIMSK1 |= (1<<OCIE1B);
        OCR1A = 1562;
        OCR1B = 1562;
    }

    void Bin2Dec(uint16_t data){
        bcd_buffer[3] = (uint8_t)(data/1000);
        data = data % 1000;
        bcd_buffer[2] = (uint8_t)(data/100);
        data = data % 100;
        bcd_buffer[1] = (uint8_t)(data/10);
        data = data % 10;
        bcd_buffer[0] = (uint8_t)(data);
    }

    void SendData (uint8_t data){
        SPDR = data;
        while(!(SPSR & (1<<SPIF)));
    }

    void DisplayData (uint16_t data){
        Bin2Dec(data);
        PORTB &= ~(1<<PB1);         //clk_out = 0
```

```
        SendData(segments[bcd_buffer[0]]);
        SendData(segments[bcd_buffer[1]]);
        SendData(segments[bcd_buffer[2]]);
        SendData(segments[bcd_buffer[3]]);
        PORTB |= (1<<PB1);          //clk_out = 1
    }

    void InitSPI( void){
        DDRB |= (1<<PB3 | 1<<PB5);//configure MOSI
and CLK as out
        SPSR |= (1<<SPI2X);   //Fclk = Fosc/2
        //SPI enable, master mode, MSB first,
CPOL=0, CPHA=0
        SPCR = (1<<SPE | 1<<MSTR);
        //init values - DAT low, CLK low
        PORTB &= ~(1<<PB3 | 1<<PB5);     }

    void InitADC( void){
        ADMUX = (1<<MUX0);    //Align left, ADC1
        ADCSRB = (1<<ADTS2 | 1<<ADTS0);  //Start on
Timer1 COMPB
        //Enable, auto update, IRQ enable
        ADCSRA = (1<<ADEN | 1<<ADATE | 1<<ADIE);

    }

    void InitUSART(){
        UCSR0B = (1<<RXEN0 | 1<<TXEN0 |
1<<RXCIE0);
        UCSR0C = (1<<UCSZ01 | 1<<UCSZ00);
        UBRR0H = 0;
        UBRR0L = 0x0C;
    }

    void SendChar(char symbol){
        while (!(UCSR0A & (1<<UDRE0)));
```

```
    UDR0 = symbol;
}

void SendString(char * buffer){
    while(*buffer != 0){
        SendChar(*buffer++);
    }
}
```

Возможны и другие варианты использования UART. Этот интерфейс благодаря своей простоте и широким возможностям очень популярен. Так, часто в сложных устройствах оставляется так называемый «отладочный UART». В него отправляются сообщения, сигнализирующие о режимах работы устройства и происходящих внутри него процессах. Этот интерфейс может быть даже не выведен на разъем и при обычной работе устройства никак не использоваться. Однако при настройке и диагностике к нему можно подключиться и узнать, в чем проблема.

Интерфейс также может использоваться и в синхронном режиме. В этом случае для передачи данных необходим дополнительный проводник для передачи тактовых импульсов. Источником тактирования может быть как сам контроллер, так и внешнее устройство, с которым производится обмен. Введение дополнительной линии тактирования позволяет повысить надежность и скорость связи.

Таким образом, рассмотрена работа блоков АЦП и UART, приведены примеры работы с ними и исследованы возможные варианты их применения.

Задания и пояснения к выполнению работы

Разработать устройство и программу, позволяющие выполнять команды, отправленные по UART (команда состоит из нескольких символов). Предусмотреть команды установки настроек АЦП, запроса настроек АЦП, команду вывода числового значения на индикаторы (значение также передается по UART). Настроить АЦП на работу с двумя аналоговыми входами.

Контрольные вопросы

1. Что такое АЦП? Для чего он используется?
2. Какие факторы влияют на точность работы АЦП?
3. Назовите виды погрешностей, возникающих при работе АЦП.
4. У АЦП ATmega168 есть функция, позволяющая выравнивать результат преобразования АЦП по левой границе. Для чего?
5. Назовите возможные режимы работы АЦП.
6. Что такое UART? Чем он отличается от SPI?
7. Опишите процесс передачи байта через UART.
8. Почему необходимо точно установить частоту работы UART?
9. Каким образом можно использовать UART для подключения микроконтроллера к компьютеру?
10. Для чего при передаче данных с АЦП в данной работе к каждому полученному числу прибавлялось число 0x30?

4

Протокол работы с ЖКИ WH1602

Теоретические сведения

Ранее уже рассматривались методы отображения информации на светодиодных семисегментных индикаторах. Такие индикаторы позволяют наглядно отображать цифровую информацию, однако отображение какой-либо другой информации попросту невозможно. Для преодоления данной трудности сначала было предложено увеличить количество сегментов – так появились 9-, 14- и 16-сегментные цифро-буквенные индикаторы. Дальнейшим шагом стала замена сегментов на матрицу точек – такие светодиодные матрицы повсеместно используются для создания так называемых «бегущих строк», а версии матриц с большим количеством полноцветных светодиодов используются для создания светодиодных экранов.

Однако у таких индикаторов есть общие недостатки – большие габариты и высокое энергопотребление. Если стоит задача создания носимого портативного устройства, работающего от аккумуляторов, в котором необходимо отображение текстовой информации, светодиодная индикация совершенно неприемлема. Решение данной задачи заключается в использовании жидкокристаллических индикаторов (ЖКИ). ЖКИ компактны, экономичны и дешевы, но из их недостатков следует отметить хрупкость и подверженность влиянию низких температур.

Управление непосредственно ЖКИ – достаточно сложная задача, поэтому для ее упрощения промышленностью выпускаются ЖКИ с встроенным контроллером. Такой контроллер берет на себя функции хранения и отображения символьной информации, поэтому

управление подобным индикатором заключается в отправке данному контроллеру управляющих команд.

Рассмотрим ЖКИ WH1602 (рис. 36). Данный индикатор позволяет одновременно отображать две строки по 16 символов, при этом каждый символ отображается с помощью матрицы точек 5x8. Существует множество видов подобных индикаторов, при этом возможны вариации в размере матрицы символа, количестве строк и/или столбцов, но видов управляющих контроллеров гораздо меньше.

Рассматриваемый индикатор имеет в своем составе контроллер HD44780. Данный контроллер широко распространен и применяется во многих индикаторах. Он фактически является промышленным стандартом и широко используется при производстве алфавитно-цифровых модулей.

Рисунок 36 – Внешний вид индикатора WH1602

Алфавитно-цифровые ЖКИ модули представляют собой удобное решение, позволяющее сэкономить время и ресурсы при разработке новых изделий, и при этом обеспечивают отображение большого объема информации при хорошей различимости и низком энергопотреблении.

Контроллер HD44780 потенциально может управлять двумя строками по 40 символов в каждой (для модулей с 4 строками по 40 символов одновременно используются сразу два контроллера) при матрице символа 5x7 точек.

Существует несколько стандартных конфигураций ЖКИ модулей (символов x строк): 8x2, 16x1, 16x2, 16x4, 20x1, 20x2, 24x2, 40x2, 40x4. Однако принципиальных ограничений на комбинации и коли-

чество отображаемых символов контроллер не накладывает – модуль может иметь любое количество символов от 1 до 80, хотя адресация в некоторых вариантах может оказаться весьма неудобной.

Рассмотрим имеющиеся у индикатора выводы (табл. 14).

Таблица 14 – **Выводы индикатора WH1602**

№ вывода	Символ	Лог. уровень	Описание
1	V_{DD}	5,0 В	Вывод питания
2	V_{SS}	0 В	Вывод питания
3	VO	Переменный	Установка контраста
4	RS	H/L	Сигнал выбора регистра
5	R/W	H/L	Лог. 1 – чтение модуля, лог. 0 – запись в модуль
6	E	H, H→L	Сигнал выбора модуля
7	DB0	H/L	Линия информационной шины
8	DB1	H/L	Линия информационной шины
9	DB2	H/L	Линия информационной шины
10	DB3	H/L	Линия информационной шины
11	DB4	H/L	Линия информационной шины
12	DB5	H/L	Линия информационной шины
13	DB6	H/L	Линия информационной шины
14	DB7	H/L	Линия информационной шины

Выводы 1 и 2 необходимы для подачи питания на индикатор, вывод 3 служит для настройки контрастности индикатора. Обычно к нему подключается скользящий контакт потенциометра на 10-20 кОм, включенного между линиями питания и земли. Однако при известных номиналах можно заменить его обычным резисторным делителем.

Передача информации на контроллер ЖКИ осуществляется по параллельной шине данных DB0…DB7. По ней передаются как команды контроллеру, так и коды выводимых символов. Вид передаваемой информации (команды или данные) задается с помощью вывода 4 (RS): при низком уровне на данном выводе контроллер воспринимает принятую информацию как команду, при высоком – как данные. Для синхронизации передачи служит вывод 6 (E) – инфор-

мация с шины данных записывается во входной регистр контроллера по спаду сигнала на данном выводе.

Контроллер ЖКИ не только принимает информацию, но и может ее передавать – режим чтение/запись устанавливается с помощью вывода 5 (R/W): при низком уровне на данном выводе происходит запись в контроллер, при высоком – чтение.

Таким образом, для подключения индикатора к управляющему контроллеру необходимо использовать 11 выводов. Для уменьшения данного количества возможна работа ЖКИ с 4-проводной шиной данных – к ЖКИ подключаются только выводы DB7…DB4, при этом передача байта на ЖКИ происходит в два этапа: сначала передаются биты DB7…DB4, затем DB3…DB0. Количество используемых выводов контроллера уменьшается до 7.

Количество выводов можно сократить еще на 1, если отказаться от использования линии R/W, при этом контроллер ЖКИ сможет только принимать информацию. В таком случае возникает следующее затруднение: в режиме чтения контроллер ЖКИ отдает управляющему контроллеру состояние флага занятости и адрес курсора ЖКИ. Если линия R/W не используется, управляющая программа лишается доступа к данной информации; поэтому если в обычном режиме программа после отправки какой-либо команды ждала очистки флага занятости контроллера ЖКИ, то в данном случае необходимо ждать определенное для каждой команды время, чтобы команда гарантированно успела выполниться ЖКИ до отправки следующей.

Последовательности действий, которые необходимо выполнить управляющей системе при совершении операций записи и чтения для 8- и 4-разрядной шин приведены ниже.

Операции записи для 8-разрядной шины

1. Установить значение линии RS.
2. Вывести значение байта данных на линии шины DB0...DB7.
3. Установить линию E = 1.
4. Установить линию E = 0.
5. Установить линии шины DB0...DB7 = HiZ.

Операции чтения для 8-разрядной шины

1. Установить значение линии RS.
2. Установить линию R/W = 1.
3. Установить линию E = 1.
4. Считать значение байта данных с линий шины DB0...DB7.
5. Установить линию E = 0.
6. Установить линию R/W = 0.

Операции записи для 4-разрядной шины

1. Установить значение линии RS.
2. Вывести значение старшей тетрады байта данных на линии шины DB4...DB7.
3. Установить линию E = 1.
4. Установить линию E = 0.
5. Вывести значение младшей тетрады байта данных на линии шины DB4...DB7.
6. Установить линию E = 1.
7. Установить линию E = 0.
8. Установить линии шины DB0...DB7 = HiZ.

Операции чтения для 4-разрядной шины

1. Установить значение линии RS.
2. Установить линию R/W = 1.
3. Установить линию E = 1.
4. Считать значение старшей тетрады байта данных с линий шины DB4...DB7.
5. Установить линию E = 0.
6. Установить линию E = 1.
7. Считать значение младшей тетрады байта данных с линий шины DB4...DB7.
8. Установить линию E = 0.
9. Установить линию R/W = 0.

Приведенные операции подразумевают, что время выполнения каждого шага составляет не менее 250 нс. При использовании со-

временных контроллеров это условие легко может быть нарушено, поэтому необходимо тщательно контролировать минимальные значения временных интервалов, указанных в таблицах 15 и 16, и при необходимости вводить задержки. Временные диаграммы операций изображены на рисунках 37 и 38.

Таблица 15 – **Временные характеристики сигналов при записи (нс)**

Параметр	Обозн.	Мин.	Макс.
Период сигнала E	t_{CYC}	500	–
Положительный полупериод сигнала E	t_{PW}	230	–
Фронт/спад сигнала E	t_{ER}/ t_{EF}	–	20
Установление адреса	t_{AS}	40	–
Удержание адреса	t_{AH}	10	–
Установление данных	t_{DSW}	80	–
Удержание данных	t_H	10	–

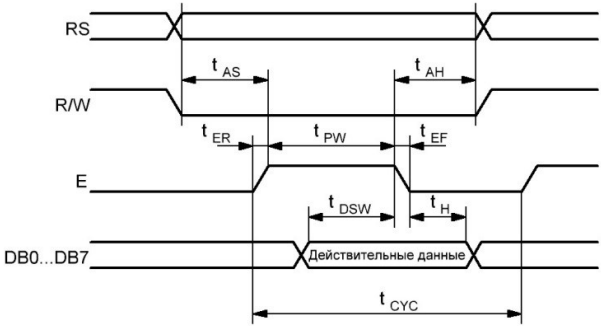


Рисунок 37 – Временная диаграмма операции записи

Таблица 16 – **Временные характеристики сигналов при чтении (нс)**

Параметр	Обозн.	Мин.	Макс.
Период сигнала E	t_{CYC}	500	–
Положительный полупериод сигнала E	t_{PW}	230	–
Фронт/спад сигнала E	t_{ER}/ t_{EF}	–	20
Установление адреса	t_{AS}	40	–
Удержание адреса	t_{AH}	10	–
Установление данных	t_{DSW}	–	160
Удержание данных	t_H	5	–

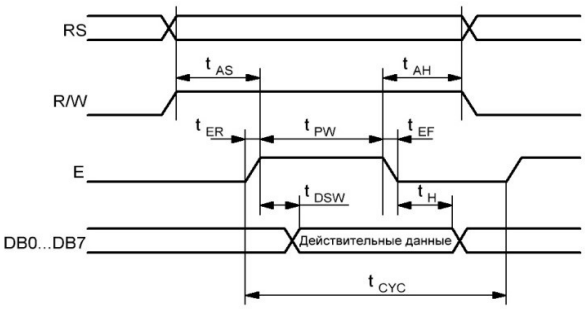


Рисунок 38 – Временная диаграмма операции чтения

Описанные операции записи/чтения байта являются базовыми для осуществления обмена данными с ЖКИ модулем. Реализация данных операций – единственное, что отличает процесс обмена с модулем для 8- и 4-разрядной шин данных. На их основе строятся все виды операций программирования и управления.

Рассмотрим структуру контроллера HD44780 (рис. 39).

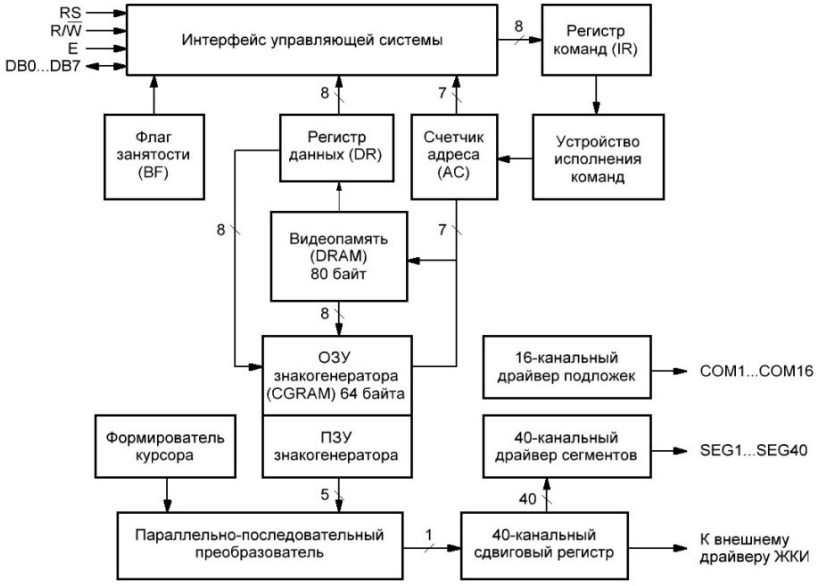


Рисунок 39 – Структурная схема контроллера HD44780

На данной схеме можно выделить основные элементы, с которыми приходится взаимодействовать при программном управлении: регистр данных (DR), регистр команд (IR), видеопамять (DDRAM), ОЗУ знакогенератора (CGRAM), счетчик адреса памяти (AC), флаг занятости контроллера.

Другие элементы участвуют в процессе регенерации изображения на ЖКИ и не взаимодействуют с управляющей программой напрямую.

Для связи с управляющим контроллером контроллер ЖКИ имеет в своем составе два регистра: регистр инструкций IR и регистр данных DR. Команды, отдаваемые управляющим контроллером, помещаются в регистр IR. Список команд, принимаемых контроллером, представлен в таблице 17.

Таблица 17 – **Управляющие комбинации регистра IR**

D7	D6	D5	D4	D3	D2	D1	D0	Назначение
0	0	0	0	0	0	0	1	Очистка экрана, AC = 0, адресация AC на DDRAM
0	0	0	0	0	0	1	–	AC = 0, сброшены сдвиги, начало строки адресуется в начале DDRAM
0	0	0	0	0	1	I/D	S	Выбирается направление сдвига курсора или экрана
0	0	0	0	1	D	C	B	Выбирается режим отображения
0	0	0	1	S/C	R/L	–	–	Команда сдвига курсора/экрана
0	0	1	DL	N	F	–	–	Определение параметров развертки и ширины шины данных
0	1	AG	AG	AG	AG	AG	AG	Присвоение счетчику AC адреса в области CGRAM
1	AD	AD	AD	AD	AD	AD	AD	Присвоение счетчику AC адреса в области DDRAM

Представленные команды служат для изменения состояния управляющих флагов контроллера. Назначение данных флагов приведено в таблице 18.

Таблица 18 – **Управляющие флаги контроллера HD44780**

Флаг	Назначение
I/D	Режим смещения счетчика адреса AC: 0 – уменьшение, 1 – увеличение
S	Флаг режима сдвига содержимого экрана: 0 – сдвиг не производится, 1 – после записи в DDRAM очередного кода экран сдвигается в направлении, определяемом флагом I/D: 0 – вправо, 1 – влево. При сдвиге не изменяется содержимое DDRAM, изменяются только внутренние указатели видимого положения начала строки.
S/C	Флаг-команда, производящая вместе с флагом R/L операцию сдвига содержимого экрана (также без изменений в DDRAM) или курсора. Определяет объект смещения: 0 – курсор, 1 – экран.
R/L	Флаг-команда, производящая вместе с флагом S/C сдвиг экрана или курсора. Задает направление смещения: 0 – влево, 1 – вправо.
D/L	Флаг, определяющий ширину шины данных: 0 – 4 бит, 1 – 8 бит.
N	Флаг режима развертки изображения: 0 – одна строка, 1 – две строки.
F	Размер матрицы символов: 0 – 5×8 точек, 1 – 5×10 точек.
D	Наличие изображения: 0 – выключено, 1 – включено.
C	Курсор в виде подчеркивания: 0 – выключен, 1 – включен.
B	Курсор в виде мигающего знакоместа: 0 – выключен, 1 – включен.

Адресация регистров DR и IR производится с помощью линии RS – если на ней лог. 1, то идет обращение к DR, иначе к IR.

Данные через регистр данных DR могут в зависимости от текущего режима помещаться (или прочитываться) в видеопамять DDRAM или ОЗУ знакогенератора CGRAM по текущему адресу, указываемому счетчиком AC. Информация, попавшая в регистр IR, интерпретируется устройством выполнения команд как управляющая последовательность. Чтение регистра IR возвращает значение счетчика AC, а в старшем разряде – флаг занятости BF. Этот флаг имеет значение 1, когда контроллер занят, и 0, когда он свободен.

Большинство операций, выполняемых контроллером, занимают значительное время, от 40 мкс до единиц миллисекунд, поэтому цикл ожидания снятия флага BF обязательно должен присутствовать в программах драйвера ЖКИ модуля и предшествовать выполнению любой операции.

Видеопамять объемом 80 байт предназначена для хранения кодов символов, отображаемых на ЖКИ. Видеопамять организована в две строки по 40 символов в каждой. Эта организация является жесткой и не зависит от количества строк в конкретном индикаторе.

Вывод символа на экран производится записью его кода в регистр DR. При этом символ размещается в DDRAM по текущему адресу, указываемому AC, а само значение AC при этом увеличивается или уменьшается на 1. Для перестановки курсора на новую позицию необходимо присвоить AC соответствующее значение.

В техническом описании на контроллер ЖКИ приведены алгоритмы инициализации индикатора для различных вариантов его подключения. Рассмотрим алгоритм инициализации для подключения по 4-проводной шине данных. Он состоит из следующих этапов:

1) При включении ожидание не менее 40 мс после того момента, как напряжение питания превысит 2,7 В.
2) Отправка команды 0x30.
3) Ожидание не менее 4,1 мс.
4) Отправка команды 0x30.
5) Ожидание не менее 100 мкс.
6) Отправка команды 0x30. После данной команды контроллер выходит на нормальный режим работы: может выполнять данные ему инструкции и сигнализировать об их выполнении флагом BF.
7) При включении ЖКИ по умолчанию настроен на работу с 8-разрядной шиной данных, поэтому в первую очередь необходимо установить 4-разрядный режим путем установки флага D/L в ноль командой 0x20. После данной команды все передачи по шине данных будут проходить в 2 этапа: сначала передача старшей тетрады, затем – младшей.
8) Отправка команды 0x28 – данная команда установит значения флагов N и F таким образом, что ЖКИ будет настроен на отображение двух строк с матрицами символов 5x8 точек.
9) Отправка команды 0x0C – установка значений флагов D, C, B – включение отображения на дисплей, символы курсоров не отображаются.

10) Отправка команды 0x06 – установка значений флагов I/D и S – настройка счетчика адреса AC на увеличение при записи символа, выключение сдвига экрана при записи.

После выполнения данной последовательности работа с дисплеем заключается в записи по определенным адресам кодов символов, которые необходимо вывести на экран. При приведенных выше настройках при отправке в ЖКИ кода символа он отображается в том месте экрана, на которое указывает счетчик AC, – после инициализации он указывает на первый символ первой строки. После записи символа значение счетчика автоматически увеличивается, и он начинает указывать на второй символ. При записи последнего символа первой строки AC автоматически начинает указывать на первый символ второй.

Для непосредственного указания, какой символ требуется переписать, управляющий контроллер отправляет команду, содержащую новый адрес AC. Так, адрес первого символа первой строки – 0x00, второго – 0x01 и т. д. Однако адреса второй строки, вне зависимости от количества знакомест в строке ЖКИ, начинаются с адреса 0x40. Далее – 0x41, 0x42 и так по порядку.

Практическая часть

В третьем разделе разработанное устройство было способно работать с АЦП и UART, выводя данные АЦП на набор семисегментных индикаторов, и отправлять их по нажатию кнопки в UART.

Заменим семисегментные индикаторы на ЖКИ модуль. В Proteus имеется модель ЖКИ 16x2 с контроллером HD44780. Данная модель находится в библиотеке **Optoelectronics** под названием **LM016L** – **16x2 Alphanumeric LCD**. Общая схема устройства с подключенным по четырехпроводной шине данных ЖКИ показана на рисунке 40.

К сожалению, Proteus производит исключительно логическое моделирование данного ЖКИ, нет возможности управлять яркостью или контрастностью дисплея. Так, выводы с 1 по 3, отвечающие за подключение питания и делителя напряжения для управления контрастностью, сделаны исключительно для соответствия реальному

индикатору, в моделировании они не участвуют и вообще могут не подключаться. Однако приведенная схема подключения работоспособна для реального ЖКИ.

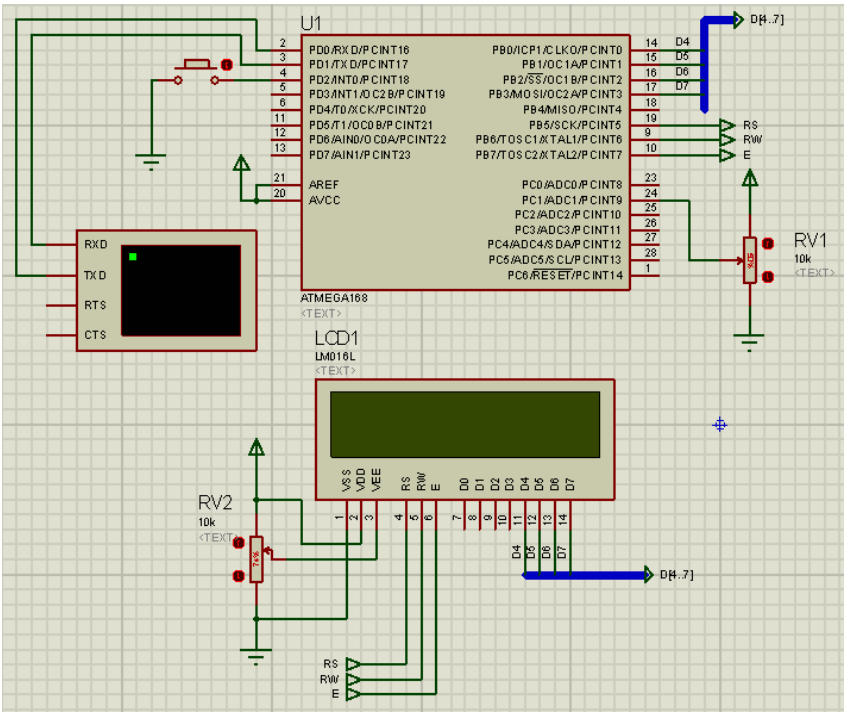


Рисунок 40 – Общая схема устройства

Изменения в схеме устройства неизбежно требуют изменений в тексте управляющей программы. Так как ЖКИ подключен к порту B, к которому ранее подключались 7-сегментные индикаторы, из текста программы необходимо убрать все, что относится к работе модуля SPI, расположенного на порту B, и иначе настроить режим работы порта: настроить все выводы порта на выход и задать их начальное состояние.

```
        DDRB=0xFF;     //настройка выводов управле-
ния на выход
        PORTB=0;
```

Для упрощения понимания программы назначим выводам порта B имена, соответствующие их функциональному назначению:

```
#define D4   PB0
#define D5   PB1
#define D6   PB2
#define D7   PB3
#define RS   PB5
#define RW   PB6
#define E    PB7
```

Также создадим две константы, которые понадобятся при создании функции отправки байта в ЖКИ:

```
#define CMD      0
#define DATA 1
```

Инициализация ЖКИ производится по приведенному выше алгоритму и располагается в функции `InitLCD`, которая вызывается до начала основного цикла программы.

```
void InitLCD(void){
    uint8_t BF = 0x80;          //объявление
переменной-флага BF
    _delay_ms(40);              //ожидание со-
гласно алгоритму
        PORTB &= ~(1<<RS);
        PORTB = (0x30>>4);      //выставление
на шину данных
                                //старшей тет-
рады команды 0x30
        PORTB |= (1<<E); // выставление сигна-
ла E
        asm("nop");         // ожидание в течение
3-х тактов
        asm("nop");         // на частоте 8 МГц
1 такт=125нс
        asm("nop");         //требуется удержания
E не менее 250нс
        PORTB &= ~(1<<E);       // сброс E
```

```
            PORTB = 0;
        _delay_ms(5);          // ожидание согласно
алгоритму
            PORTB &= ~(1<<RS);
            PORTB = (0x30>>4);     // вторая от-
правка команды 0x30
            PORTB |= (1<<E);
            asm("nop");
            asm("nop");
            asm("nop");
            PORTB &= ~(1<<E);
            PORTB = 0;
        _delay_us(150);        // ожидание согласно
алгоритму
            PORTB &= ~(1<<RS);
            PORTB = (0x30>>4);     // третья от-
правка команды 0x30
            PORTB |= (1<<E);
            asm("nop");
            asm("nop");
            asm("nop");
            PORTB &= ~(1<<E);
            PORTB = 0;
        _delay_ms(5);
        do{                    // ожидание до тех пор,
пока флаг BF
                               // не освободится
            BF = (0x80 & LCD_Read());
        }while(BF == 0x80);
            PORTB &= ~(1<<RS);
            PORTB = (0x20 >> 4);   //установка ши-
рины шины данных
                                                //в 4 бит
            PORTB |= (1<<E);
            asm("nop");
            asm("nop");
```

```
            asm("nop");
            PORTB &= ~(1<<E);
            PORTB = 0;
        do{
            BF = (0x80 & LCD_Read());
        }while(BF == 0x80);
        // здесь обмен начинает работать по 4-
проводной шине
        LCD_Write(CMD,0x28); // установка режима 2
строки,
                             // знакоместо 5*8
        LCD_Write(CMD,0x0C); // включение отобра-
жения
        LCD_Write(CMD,0x06); // автоинкремент
счетчика
    }
```

Функция `LCD_Write` принимает на вход тип передаваемой информации (команда или данные) и собственно передаваемый байт. Содержимое функции следующее:

```
    void LCD_Write(uint8_t type,char data){
        uint8_t BF = 0x80;
        do{                 // ожидание завершения
предыдущей операции
            BF = 0x80 & LCD_Read();
        }while(BF == 0x80);
        PORTB |= (type<<RS); // установка RS в за-
висимости от
                                    // типа данных
        PORTB |= (1<<E);
        PORTB &= ~(0x0F);
        PORTB |= (0x0F & (data>>4));     // переда-
ча старшей тетрады
        PORTB &= ~(1<<E);
        asm("nop");
        asm("nop");
```

```
        asm("nop");
        PORTB |= (1<<E);
        PORTB &= ~(0x0F);
        PORTB |= (0x0F & data);          // переда-
ча младшей тетрады
        PORTB &= ~(1<<E);
        PORTB = 0;
    }
```

Для проверки статуса флага BF используется функция LCD_Read:

```
    char LCD_Read(void){
        char retval = 0;
        PORTB &= ~(1<<RS);
        PORTB |= (1<<RW);           // переход в
режим чтения
        DDRB &= ~(1<<D4|1<<D5|1<<D6|1<<D7);    //
настройка выводов

    // на вход
        PORTB |= (1<<E);
        asm("nop");
        asm("nop");
        retval = (PINB & 0x0F)<<4; //чтение стар-
шей тетрады байта
        PORTB&=~(1<<E);
        asm("nop");
        asm("nop");
        asm("nop");
        PORTB |= (1<<E);
        asm("nop");
        asm("nop");
        retval |= (PINB & 0x0F);    // чтение млад-
шей тетрады
        PORTB&=~(1<<E);
        DDRB |= (1<<D4|1<<D5|1<<D6|1<<D7);     //
настройка выводов
```

```
    // на выход
        PORTB = 0;
        return retval;     // возвращаемое значение
– прочитанный байт
    }
```

После прохождения инициализации ЖКИ готов к приему информации. В качестве примера отобразим следующую информацию (рис. 41):

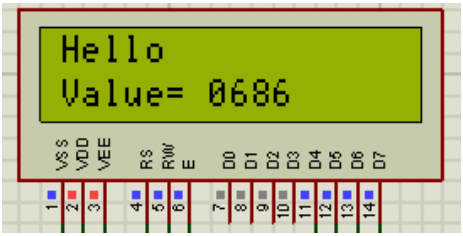


Рисунок 41 – Пример вывода на ЖКИ

Здесь 0686 – значение, получаемое с помощью АЦП. До начала основного цикла программы располагается следующий программный код:

```
        LCD_Write(DATA,'H');
        LCD_Write(DATA,'e');
        LCD_Write(DATA,'l');
        LCD_Write(DATA,'l');
        LCD_Write(DATA,'o');
        LCD_Write(CMD,0x40|0x80);  // переход на
вторую строку
        LCD_Write(DATA,'V');
        LCD_Write(DATA,'a');
        LCD_Write(DATA,'l');
        LCD_Write(DATA,'u');
        LCD_Write(DATA,'e');
        LCD_Write(DATA,'=');
        LCD_Write(DATA,0x20);  // вывод пробела
```

Символы, выведенные на ЖКИ, остаются на экране до тех пор, пока не будут переписаны другими символами. Таким образом, для получения изображения, показанного на рисунке 41, достаточно один раз вывести слова «Hello» и «Value = », а символы цифр обновлять в соответствии с данными АЦП. Для этого в основной цикл программы помещается следующий код:

```
        Bin2Dec(ADC_val);
        LCD_Write(CMD,0x47|0x80);  //установка
курсора на 7-е

   //знакоместо 2-й строки
        LCD_Write(DATA,0x30+bcd_buffer[3]);
        LCD_Write(DATA,0x30+bcd_buffer[2]);
        LCD_Write(DATA,0x30+bcd_buffer[1]);
        LCD_Write(DATA,0x30+bcd_buffer[0]);
```

Функция `Bin2Dec` преобразует двоичное значение, полученное с АЦП, в 4 числа, соответствующие разрядам десятичного числа, хранящиеся в массиве `bcd_buffer`. Двоичные данные, отправляемые на контроллер ЖКИ для вывода символов на экран, в основном соответствуют кодам символов в кодировке ASCII, поэтому для вывода цифры от 0 до 9 необходимо к ней прибавить значение 0x30, так как символы цифр в кодировке ASCII начинаются с кода 0x30. Однако соответствие кодов символов ASCII не полное.

В зависимости от производителя контроллера и основного рынка их продажи таблица символов контроллера может выглядеть поразному. Существует как минимум четыре модификации таблиц символов: японская, европейская, французская и русская, поэтому для конкретного ЖКИ всегда необходимо уточнять данную информацию в его техническом описании. Пример таблицы символов приведен на рисунке 42.

Таким образом, управление ЖКИ сводится к периодической отправке выходных данных и управляющих команд, задачи хранения информации и обновления изображения на ЖКИ берет на себя вспомогательный контроллер. Использование вспомогательного контроллера значительно упрощает работу со сложными устройствами, поэтому они широко распространены.

Рисунок 42 – Один из вариантов таблицы кодов символов контроллера HD44780

Таким образом, рассмотрена работа с ЖКИ. Более полная информация по командам и алгоритмам работы с данными устройствами располагается в технических описаниях на соответствующие компоненты.

Задания и пояснения к выполнению работы

Написать программу, позволяющую принимать значения по USART и передавать их на ЖКИ.

Написать программу-калькулятор с выводом результата на ЖКИ.

Контрольные вопросы

1. В чем заключаются недостатки светодиодных индикаторов?
2. Назовите преимущества и недостатки жидкокристаллических индикаторов.
3. Для чего в модули ЖКИ часто встраивают дополнительные контроллеры?
4. Назовите основные функциональные блоки контроллера HD44780.
5. Каким образом происходит вывод необходимых символов на ЖКИ?
6. Вам необходимо отобразить на середине первой строки индикатора символ «Ы». Опишите порядок действий.

5

Протокол работы с датчиком температуры DS18B20

Теоретические сведения

Зачастую связь с контроллером осуществляется по цифровому интерфейсу, который имеет серьезные преимущества перед аналоговой линией. Например, проектируемое устройство должно измерять температуру некоторого объекта, находящегося на расстоянии нескольких метров от него. Если в качестве температурного датчика использовать терморезистор или термопару и измерять их параметры с помощью АЦП контроллера, возникают следующие проблемы:

- длинные провода до термодатчика из-за наличия собственного сопротивления вносят ошибку в измеряемый параметр;
- эти же длинные провода работают как антенна, воспринимая аналоговые помехи;
- осуществление гальванической развязки аналогового сигнала – достаточно нетривиальная задача.

Поэтому в настоящее время выпускаются цифровые термодатчики, имеющие в своем составе встроенный контроллер, осуществляющий все необходимые преобразования данных от термочувствительного элемента, связь с которым осуществляется по цифровому каналу передачи данных. При цифровой передаче данных достигается большая помехоустойчивость, легко организуется гальваническая развязка сигнала.

Примером подобного датчика является DS18B20 фирмы Maxim. Этот широко известный датчик работает по цифровому интерфейсу 1-Wire, требующему для работы всего одну линию для передачи данных. Технически необходимы три провода: провод питания, про-

вод для линии данных и земля, однако есть возможность использовать всего два провода, избавившись от провода питания. При этом датчик получает энергию от линии передачи данных: в то время, пока не идет обмена данными, датчик заряжает от линии встроенный конденсатор, который играет роль источника питания в те моменты, когда происходит обмен данными. Такая технология получила название «паразитного питания».

Каждый датчик DS18B20 имеет собственный уникальный 64-битный идентификатор, записанный в его память. Управляющий контроллер, работающий с датчиком, обращается к нему с помощью данного идентификатора, поэтому количество однотипных устройств, подключенных по одной шине 1-Wire, теоретически не ограничено. Структура идентификатора представлена на рисунке 43.

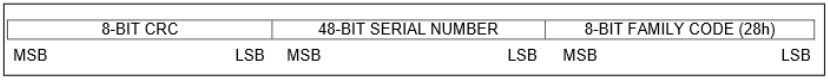


Рисунок 43 – Идентификатор датчика

Младший байт – код семейства датчиков (для DS18B20 равен 0x28), далее идет уникальный 48-битный серийный номер датчика, и в старшем байте содержится CRC код, вычисленный для младших 56 бит идентификатора.

Физически шина 1-Wire представляет собой проводник, подтягиваемый к напряжению питания через резистор, номинал которого обычно составляет 4,7 кОм. К шине могут быть подключены одно мастер-устройство и сколько угодно подчиненных устройств. Как мастер, так и подчиненные, могут лишь притягивать уровень линии к нулю, высокий уровень устанавливается с помощью подтягивающего резистора. Подключение нескольких устройств одновременно возможно благодаря тому факту, что низкий уровень имеет преимущество – то есть, если хотя бы одно устройство выставит низкий уровень на линии (просто замкнет его на землю), то на всей линии установится низкий уровень, вне зависимости от того, какой уровень пытаются выставить остальные устройства. Обмен по шине всегда инициируется мастером.

Протокол обмена 1-Wire устанавливает несколько типов сигналов со строгими временными характеристиками: импульс сброса, импульс присутствия, запись 0, запись 1, чтение 0, чтение 1. Все эти сигналы, кроме импульса присутствия, инициируются мастером.

Датчик обладает настраиваемой точностью преобразования – после включения его АЦП работает с разрядностью 12 бит, но можно установить также 11, 10 и 9 бит. При пониженной разрядности, естественно, снижается точность измерений, зато значительно уменьшается время, необходимое на одно преобразование: если для преобразования с точностью 12 бит требуется 750 мс, то со снижением разрядности на каждый бит это время уменьшается вдвое. Так, для 9 бит время преобразования составит 93,75 мс.

Результат преобразования температуры хранится в двух 8-разрядных регистрах 0 и 1, расположенных в памяти датчика, как показано на рисунке 44.

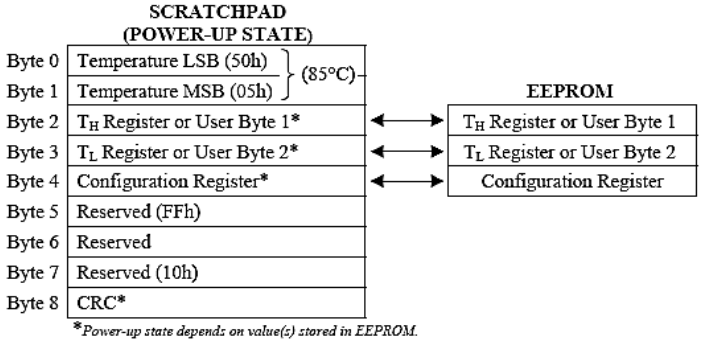


Рисунок 44 – Память датчика

Термометр DS18B20 имеет в своем составе 8 байт оперативной памяти (scratchpad, а далее ОЗУ) и 3 байта EEPROM.

Байты 0 и 1 – измеренная температура. Начальное значение температуры (при подаче питания) +85 градусов. Формат регистров:

	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
LS BYTE	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}

	BIT 15	BIT 14	BIT 13	BIT 12	BIT 11	BIT 10	BIT 9	BIT 8
MS BYTE	S	S	S	S	S	2^6	2^5	2^4

,

где S – знак температуры (0 – положительная температура, 1 – отрицательная), BIT10…BIT4 – целая часть значения температуры, BIT3…BIT0 – дробная часть.

Отрицательная температура представлена в дополнительном коде.

Байты 2 и 3 – верхний и нижний температурные пороги. Значения этих байтов задаются пользователем. Начальное значение этих байтов зависит от содержимого EEPROM памяти. Формат обоих регистров следующий:

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
S	2^6	2^5	2^4	2^3	2^2	2^1	2^0

,

где S – знак температуры (0 – положительная температура, 1 – отрицательная), BIT6…BIT0 – целая часть значения температурного порога.

Байт 4– конфигурационный регистр. В нем задается температурное разрешение DS18B20. Начальное значение регистра зависит от содержимого EEPROM памяти. В датчике DS18B20 температурное разрешение по умолчанию 12 бит.

Байты с 5 по 7 зарезервированы и не используются.

Байт 8 – это CRC (циклический избыточный код). Значение CRC используется для проверки целостности принятых данных.

Любой обмен по шине 1-Wire происходит в определенной последовательности:

1) процедура инициализации;

2) передача ROM-команды (с необходимой последующей информацией);

3) передача функциональной команды (с необходимой последующей информацией).

Рассмотрим каждый из этапов.

Этап 1. Инициализация

Перед началом любого обмена мастер производит отправку импульса сброса, на который подчиненные отвечают импульсами присутствия. Микроконтроллер на 480 мкс устанавливает на 1-Wire шине ноль, а затем освобождает ее. Если к шине подключен термо-

метр DS18B20, то он обнаруживает положительный перепад и после паузы в 15-60 мкс отвечает микроконтроллеру импульсом присутствия – устанавливает на шине ноль на время от 60 до 240 мкс. Временная диаграмма данного процесса приведена на рисунке 45.

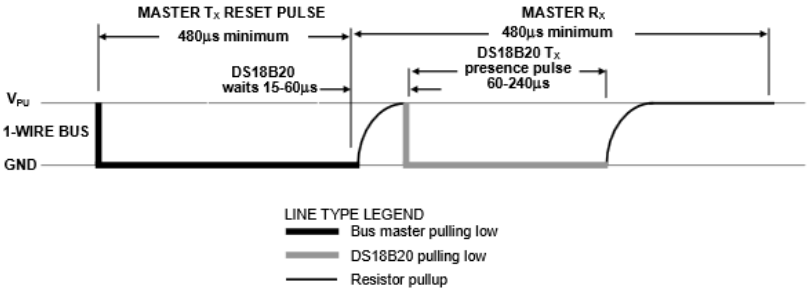


Рисунок 45 – Процедура инициализации

Этап 2. ROM-команды

После инициализации мастер передает код ROM-команды. ROM-команды направлены на работу с идентификаторами датчиков и служат для выбора конкретного устройства из множества подключенных к шине. Эти команды позволяют мастеру определить, сколько и какие устройства подключены к шине. Всего есть 5 ROM-команд, каждая из которых состоит из 1 байта:

1) Search ROM [0xF0] – данная команда используется при первичной инициализации системы для получения идентификаторов устройств.

Любое 1-Wire устройство имеет 64-разрядный код. Этот код хранится в ПЗУ и используется для адресации устройства на шине. Младший байт кода определяет семейство, к которому принадлежит 1-Wire устройство (например, код датчиков DS18B20 – 0x28). Следующие 6 байт – это серийный номер устройства. И старший байт – CRC (циклический избыточный код), вычисленный из первых 7 байтов.

Процедура получения адресов выглядит следующим образом.

Выполнив инициализацию, микроконтроллер посылает команду Search ROM, и начинается цикл чтения 64-разрядного кода. Микроконтроллер формирует на 1-Wire шине два тайм-слота чтения.

В первый тайм-слот все 1-Wire устройства, подключенные к шине, выдают первый бит своего 64-разрядного кода. Во второй тайм-слот – инвертированное значение первого бита. Если у всех 1-Wire устройств первый бит адреса – единица, то микроконтроллер примет сначала единицу, а затем ноль. Если хотя бы у одного 1-Wire устройства первый бит адреса нулевой, то микроконтроллер в обоих случаях примет ноль. Если активных устройств на шине нет, микроконтроллер в обоих случаях примет единицу. В ситуации, когда микроконтроллер принимает два нуля, возникает неоднозначность: как узнать, у каких устройств преданный бит адреса ноль, а у каких – единица? Протокол решает эту проблему просто: после двух тайм-слотов чтения микроконтроллер должен ответить 1-Wire устройствам, с какими из них он продолжит работу. Для этого он выставляет на шине соответствующий бит: ноль или единицу. Устройства, у которых переданный бит соответствует выставленному микроконтроллером, продолжат работу, остальные замолчат (станут неактивными) до следующего сигнала сброса.

Далее процедура повторяется еще 63 раза: формирование первого тайм-слота чтения, чтение состояния шины, формирование второго тайм-слота чтения, чтение состояния шины, ответ подчиненным устройствам.

После завершения цикла чтения 64-разрядного кода, микроконтроллер будет знать адрес одного 1-Wire устройства. Для получения следующего адреса нужно снова запустить цикл чтения, но на этот раз в случае неоднозначности выставить другой бит. Сколько 1-Wire устройств подключено к шине, столько раз и следует провести описанную процедуру.

2) Read ROM [0x33] – команда может использоваться только в случае, когда на шине всего одно подчиненное устройство. Позволяет напрямую получить его идентификатор.

3) Match ROM [0x55] – дает возможность мастеру выбрать конкретное устройство на шине: после отправки данной команды мастер отправляет идентификатор адресата, который и будет выполнять команду, которую затем отправит мастер.

4) Skip ROM [0xCC] – команда используется для адресации всех устройств на шине одновременно. Следующая за этой команда будет выполнена всеми подчиненными одновременно. Также может использоваться, если на шине всего одно подчиненное устройство.

5) Alarm Search [0xEC] – команда аналогична Search ROM, но отвечать будут лишь те устройства, у которых установлен Alarm-флаг. Подробнее см. в техническом описании DS18B20.

Этап 3. Функциональные команды

После выбора необходимых подчиненных устройств с помощью ROM-команды мастер передает функциональную команду, которая определяет действия подчиненного. Всего имеется 6 функциональных команд.

1) Convert T [0x44] – запуск преобразования температуры. Если DS18B20 работает в режиме паразитного питания, то после выдачи команды Convert T микроконтроллер должен подключить 1-Wire шину к источнику питания на время преобразования. При работе от внешнего питания во время хода преобразования датчик удерживает на линии 0, по окончании выставляет 1.

2) Read Scratchpad [0xBE] – эта команда позволяет микроконтроллеру считывать содержимое ОЗУ датчика DS18B20. Данные передаются младшим битом вперед с нулевого по восьмой байт. Микроконтроллер может в любой момент выдать на 1-Wire шину сигнал сброса и прекратить прием данных, если, например, требуются только первые два байта ОЗУ.

3) Write Scratchpad [0x4E] – запись данных в память датчика: передается 3 байта, первый пишется в регистр TH, второй – в TL, третий – в регистр конфигурации.

4) Copy Scratchpad [0x48] – переписывает содержимое регистров TH, TL и регистра конфигурации в энергонезависимую память датчика. Если DS18B20 работает в режиме паразитного питания, то после выдачи команды Copy Scratchpad микроконтроллер должен подключить 1-Wire шину к источнику питания не менее чем на 10 мс.

5) Recall E^2 [0xB8] – восстанавливает содержимое регистров TH, TL и регистра конфигурации из энергонезависимой памяти дат-

чика. Микроконтроллер может отслеживать процесс выполнения этой операции, формируя на 1-Wire шине тайм-слоты чтения. Пока операция выполняется, на шине будет логический ноль, когда операция завершится – логическая единица. Выполняется автоматически при включении датчика.

6) Read Power Supply [0xB4] – используется мастером для определения, используется ли каким-либо из датчиков на шине паразитное питание.

Практическая часть

Рассмотрим работу с датчиком DS18B20 в 10-битном режиме для случая 1 датчика на шине 1-Wire. Полученные данные температуры отправляются в UART. Схема устройства приведена на рисунке 46.

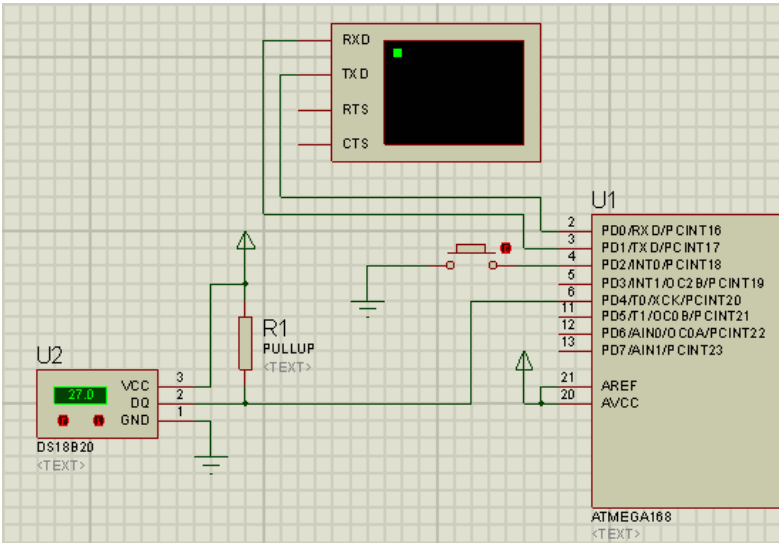


Рисунок 46 – Подключение датчика

Для упрощения работы определяются макросы:

```
#define WIRE            PD4
#define SET_1_WIRE          PORTD|=(1<<WIRE);
```

```
#define SET_0_WIRE        PORTD&=~(1<<WIRE);
```

Также необходимо настроить тактовую частоту контроллера на 8 МГц. Для этого следует в код программы добавить макрос

```
#define F_CPU    8000000UL ,
```

а кроме того, в настройках контроллера в Proteus установить фьюзбит CLKDIV8 в 1 (Unprogrammed). При этом также необходимо изменить настройки делителя частоты UART для сохранения заданной скорости обмена.

В функцию первоначальной настройки портов добавится настройка вывода PD4 (WIRE), к которому подключен датчик:

```
void InitPorts(void){
    DDRB = 0xFF;       //настройка выводов управления дисплеем
    PORTB = 0;
    DDRD = (0<<PD2|1<<PD4);     //настройка вывода кнопки и 1-Wire
    PORTD |= (1<<PD2|1<<PD4);  //начальное состояние - 1
}
```

Любой обмен данными с DS18B20 начинается с процедуры инициализации. Для инициализации мастер устанавливает линию в 0 не менее чем на 480 мкс, после чего освобождает линию. Резистор подтяжки возвращает высокий уровень на линии. Датчик отмечает этот нарастающий фронт и через 15…60 мкс устанавливает линию в 0 на 60…240 мкс. Мастер, обнаруживая этот низкий уровень, делает вывод о наличии на линии подчиненного устройства. Промежуток времени от освобождения мастером шины до начала передачи ROM-команды – не менее 480 мкс.

Функция, обеспечивающая такой порядок действий, выглядит следующим образом:

```
void OneWire_Init(void){
    DDRD |= (1<<WIRE);    // шину на выход
    SET_0_WIRE;           // установить 0
    _delay_us(500);    // ждать не менее 480 мкс
    SET_1_WIRE;                // установить 1
```

```
        DDRD &= ~(1<<WIRE);    // шину на вход
        _delay_us(60);         // ожидается импульс
сброса
        if((PIND & 1<<WIRE) == 0){ // если низкий
уровень
            while((PIND & 1<<WIRE) == 0);
            _delay_us(420);
        }else{
            SendString("No response\r\n");     //
нет подчиненных }
    }
```

После инициализации производится последовательная отправка ROM-команды и функциональной команды. Данные операции разбиваются на более простые этапы.

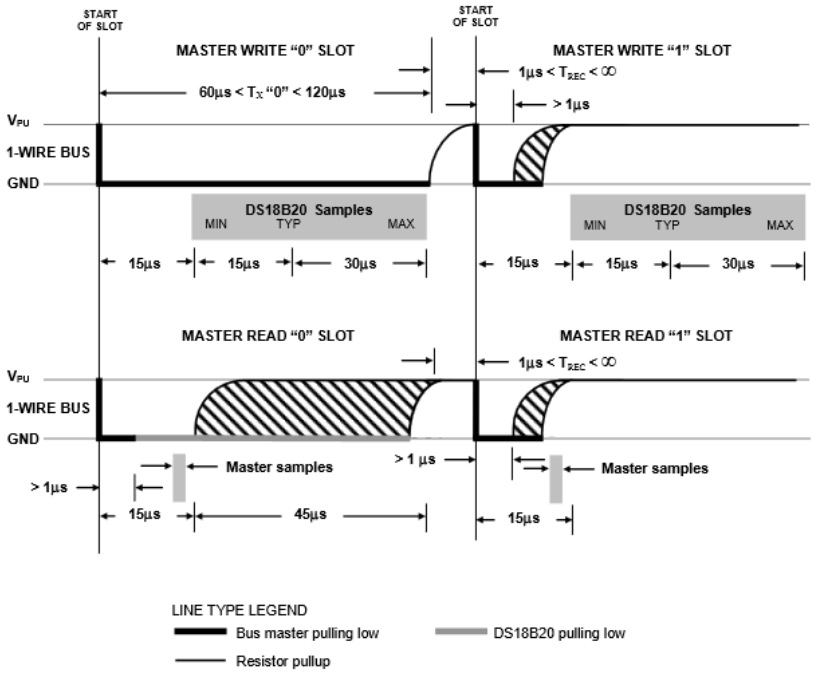


Рисунок 47 – Временные диаграммы записи и чтения битов

Базовыми функциями при работе с 1-Wire являются запись и чтение нуля и единицы. Временные диаграммы для этих действий приведены на рисунке 47.

1) Запись 0 – мастер устанавливает линию в 0 и удерживает от 60 до 120 мкс. При этом датчик читает состояние линии в промежутке от 15 до 60 мкс от момента установки линии в 0.

2) Запись 1 – мастер устанавливает линию в 0 и удерживает от 1 до 15 мкс, после чего освобождает линию. Резистор подтяжки при этом восстанавливает состояние линии в 1. Датчик читает состояние линии в промежутке от 15 до 60 мкс от момента установки линии в 0. Время между отдельными циклами записи – не менее 1 мкс.

3) Чтение бита – мастер устанавливает линию в 0 не менее чем на 1 мкс, после чего читает состояние линии. Датчик, уловив спад напряжения на линии, передает 0 или 1 в течение 15 мкс после спада, поэтому в данном промежутке мастер должен считать состояние линии.

Функция записи бита выглядит следующим образом:

```
    void OneWire_Send_1_0(uint8_t bit){
        _delay_us(2);      // промежуток между опе-
рациями записи
        DDRD |= (1<<WIRE);    // шину на выход
        SET_0_WIRE;           // установить 0
        _delay_us(10);        // ожидание 10 мкс
        if(bit != 0){
            SET_1_WIRE;       // если передается 1,
установить 1
            _delay_us(90);
        }else{
            SET_0_WIRE;       // если передается 0,
удерживать 0
            _delay_us(90);    // установить 1 после
передачи
            SET_1_WIRE
        }
    }
```

Функция чтения бита:

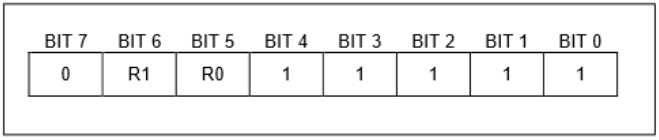
```
uint8_t OneWire_Read_1_0(void) {
    uint8_t bit=0;          // возвращаемая пере-
менная
    _delay_us(2);       // промежуток между опе-
рациями чтения
    DDRD |= (1<<WIRE);     // шину на выход
    SET_0_WIRE;            // установить 0
    _delay_us(2);       // ожидание 2 мкс
    SET_1_WIRE;            // установить 1
    DDRD &= ~(1<<WIRE);    // шину на вход
    _delay_us(2);
    if((PIND & 1<<WIRE) == 0){ // чтение со-
стояния линии
        bit=0;
    }else{
        bit=1;
    }
    _delay_us(90);          // ожидание промежут-
ка времени чтения
    return bit;             // возврат прочитан-
ного бита
}
```

С помощью приведенных функций записи и чтения бита можно сконструировать более сложные функции записи и чтения байта:

```
void OneWire_SendByte(uint8_t data){
    for(uint8_t i=0; i<8; i++){      // 8 бит
        OneWire_Send_1_0(0x01 & data);// от-
правка младшего бита
        data = data>>1;  // сдвиг вправо на 1
    }
}
uint8_t OneWire_ReadByte(void){
    uint8_t retval=0;     // возвращаемая пере-
менная
    for(uint8_t i=0; i<8; i++){
```

```
            // чтение в переменную начиная с млад-
шего бита
            retval |= OneWire_Read_1_0() << i;
        }
        return retval;
    }
```

С помощью приведенных функций можно полноценно работать с датчиком. В первую очередь до начала основного цикла следует произвести настройку разрешения датчика. Для этого необходимо записать в его регистр конфигурации соответствующее значение. Структура регистра представлена на рисунке 48.



BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0	R1	R0	1	1	1	1	1

R1	R0	RESOLUTION (BITS)	MAX CONVERSION TIME	
0	0	9	93.75ms	(t_{CONV}/8)
0	1	10	187.5ms	(t_{CONV}/4)
1	0	11	375ms	(t_{CONV}/2)
1	1	12	750ms	(t_{CONV})

Рисунок 48 – Конфигурационный регистр

Запись данного регистра производится с помощью функциональной команды Write Scratchpad (0x4E). Выбор датчика можно произвести ROM-командой Skip ROM (0xCC), так как на шине всего один датчик.

После выполнения команды Write Scratchpad необходимо передать 3 байта – TH, TL и конфигурационный. Регистры TH и TL в данном случае не используются, поэтому передать можно любые данные. Процедура установки разрешения датчика выглядит следующим образом:

```
OneWire_Init();
OneWire_SendByte(0xCC);   // SkipROM
```

```
   OneWire_SendByte(0x4E);   // Write scratchpad
   OneWire_SendByte(0x00);   // TH
   OneWire_SendByte(0x00);   // TL
   OneWire_SendByte(0x3F);   // config
```

После данной последовательности датчик начинает работать с разрешением 10 бит, однако при моделировании единственное отличие при работе с различными разрешениями заключается во времени преобразования, информация с датчика поступает та же самая. В то же время для реального датчика после установки пониженного разрешения соответствующие младшие биты перестают учитывать, так как не гарантируется, что их значения соответствуют реальности.

Для периодического опроса датчика температуры и вывода данных в необработанном виде на ЖКИ в основной цикл программы помещается следующий код:

```
   OneWire_Init();
   OneWire_SendByte(0xCC);         // Skip ROM
   OneWire_SendByte(0x44);         // Запуск пре-
образования
   while(OneWire_Read_1_0() == 0);      // ожида-
ние окончания
                                   // преобразования
   OneWire_Init();
   OneWire_SendByte(0xCC);         // Skip ROM
   OneWire_SendByte(0xBE);         // Чтение памя-
ти датчика,
                                   // в т.ч. температуры
   temperature = 0x00FF & OneWire_ReadByte();
   temperature |= OneWire_ReadByte() << 8;

   // Вывод необработанных данных на ЖКИ
   LCD_Write(CMD,0x03|0x80);
   LCD_Write(DATA,char_tab[(temperature>>12) &
0x0F]);
   LCD_Write(DATA,char_tab[(temperature>>8) &
0x0F]);
```

```
   LCD_Write(DATA,char_tab[(temperature>>4)  &
0x0F]);
   LCD_Write(DATA,char_tab[(temperature>>0)  &
0x0F]);
```

Здесь `temperature` – 16-битная переменная, предназначенная для хранения результата измерения температуры.

После выполнения данной программы вывод на ЖКИ для температуры 25 градусов выглядит следующим образом:

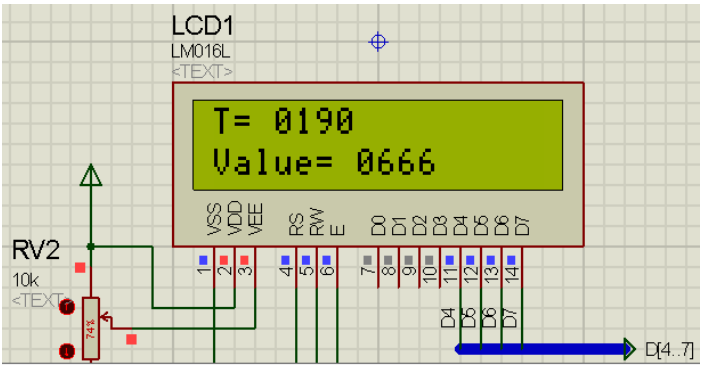


Рисунок 49 – Результат выполнения программы

Полный код программы приведен ниже.

```
#define F_CPU 8000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

#define D4    PB0
#define D5    PB1
#define D6    PB2
#define D7    PB3
#define RS    PB5
#define RW    PB6
#define E     PB7
#define CMD       0
```

```
   #define DATA 1

   #define WIRE PD4
   #define SET_1_WIRE       PORTD |= (1<<WIRE);
   #define SET_0_WIRE       PORTD &= ~(1<<WIRE);

   void InitPorts(void);
   void InitTimer1(void);
   void Bin2Dec(uint16_t data);
   void DisplayData (uint16_t data);
   void InitADC(void);

   void InitUSART(void);
   void SendChar(char symbol);
   void SendString(char * buffer);

   void InitLCD(void);
   void LCD_Write(uint8_t type, char data);
   char LCD_Read(void);

   void OneWire_Init(void);
   uint8_t OneWire_Read_1_0(void);
   uint8_t OneWire_ReadByte(void);
   void OneWire_Send_1_0(uint8_t);
   void OneWire_SendByte(uint8_t);

   volatile uint8_t bcd_buffer[] = {0,0,0,0};
   volatile uint16_t ADC_val, temperature = 0;
   const char char_tab[] =
{'0','1','2','3','4','5','6','7','8',

'9','A','B','C','D','E','F'};

   int main(void)
   {
       InitPorts();
```

```
        InitTimer1();
        EIMSK |= (1<<INT0);
        EICRA |= (1<<ISC01);
        InitADC();
        InitUSART();
        InitLCD();

        sei();
        SendString("Hello\r\n");

        LCD_Write(DATA, 'T');
        LCD_Write(DATA, '=');
        LCD_Write(DATA, 0x20);
        LCD_Write(CMD, 0x40 | 0x80);
        LCD_Write(DATA, 'V');
        LCD_Write(DATA, 'a');
        LCD_Write(DATA, 'l');
        LCD_Write(DATA, 'u');
        LCD_Write(DATA, 'e');
        LCD_Write(DATA, '=');
        LCD_Write(DATA, 0x20);

        OneWire_Init();
        OneWire_SendByte(0xCC);    // Skip ROM
        OneWire_SendByte(0x4E);    // Write
scratchpad
        OneWire_SendByte(0x00);    // TH
        OneWire_SendByte(0x00);    // TL
        OneWire_SendByte(0x3F);    // config

        while(1)
        {
            Bin2Dec(ADC_val);
            LCD_Write(CMD, 0x47 | 0x80);
            LCD_Write(DATA, 0x30+bcd_buffer[3]);
            LCD_Write(DATA, 0x30+bcd_buffer[2]);
```

```
            LCD_Write(DATA, 0x30+bcd_buffer[1]);
            LCD_Write(DATA, 0x30+bcd_buffer[0]);

            OneWire_Init();
            OneWire_SendByte(0xCC);  // Skip ROM
            OneWire_SendByte(0x44);  // Convert T
            while (OneWire_Read_1_0() == 0);

            OneWire_Init();
            OneWire_SendByte(0xCC);  // Skip ROM
            OneWire_SendByte(0xBE);  // Read
Scratchpad
            temperature = 0x00FF & One-
Wire_ReadByte();
            temperature |= OneWire_ReadByte() << 8;

            LCD_Write(CMD, 0x03 | 0x80);
            LCD_Write(DATA, char_tab[(temperature
>> 12) & 0x0F]);
            LCD_Write(DATA, char_tab[(temperature
>> 8) & 0x0F]);
            LCD_Write(DATA, char_tab[(temperature
>> 4) & 0x0F]);
            LCD_Write(DATA, char_tab[(temperature
>> 0) & 0x0F]);
        }
    }
    //-----------------------------------------

    ISR(TIMER1_COMPB_vect){
        //DisplayData(0x1E61);
    }

    ISR(INT0_vect){
        Bin2Dec(ADC_val);
        SendString("Value = ");
```

```
        SendChar(0x30 + bcd_buffer[3]);
        SendChar(0x30 + bcd_buffer[2]);
        SendChar(0x30 + bcd_buffer[1]);
        SendChar(0x30 + bcd_buffer[0]);
        SendString("\r\n");
    }
    ISR(ADC_vect){
        ADC_val = ADC;
    }

    ISR(USART_RX_vect){
        if(UDR0 == 0x20){
            SendString("Roger that\r\n");
        }
    }
    //-------------------------------------------
    void InitPorts(void){
        DDRB = 0xFF;  //настройка выводов управле-
ния дисплеем
        PORTB = 0;
        DDRD = (0<<PD2 | 1<<PD4);  //настройка вы-
вода кнопки
        PORTD |= (1<<PD2 | 1<<PD4);
    }

    void InitTimer1( void){
        TCCR1A = 0;
        TCCR1B = (1<<CS11 | 1<<CS10 | 1<<WGM12);
        TCNT1 = 0;
        TIMSK1 |= (1<<OCIE1B);
        OCR1A = 12500;
        OCR1B = 12500;
    }

    void Bin2Dec(uint16_t data){
        bcd_buffer[3] = (uint8_t)(data/1000);
```

```
        data = data % 1000;
        bcd_buffer[2] = (uint8_t)(data/100);
        data = data % 100;
        bcd_buffer[1] = (uint8_t)(data/10);
        data = data % 10;
        bcd_buffer[0] = (uint8_t)(data);
    }

    void DisplayData (uint16_t data){
        Bin2Dec(data);
    }

    void InitADC( void){
        ADMUX = (1<<MUX0);
    //Align left, ADC1
        ADCSRB = (1<<ADTS2 | 1<<ADTS0);  //Start on
Timer1 COMPB
        //Enable, auto update, IRQ enable
        ADCSRA = (1<<ADEN | 1<<ADATE | 1<<ADIE);

    }

    void InitUSART(){
        UCSR0B = (1<<RXEN0 | 1<<TXEN0 |
1<<RXCIE0);
        UCSR0C = (1<<UCSZ01 | 1<<UCSZ00);
        UBRR0H = 0;
        UBRR0L = 0x67;
    }
    void SendChar(char symbol){
        while (!(UCSR0A & (1<<UDRE0)));
        UDR0 = symbol;
    }

    void SendString(char * buffer){
        while(*buffer != 0){
```

```
            SendChar(*buffer++);
        }
    }

    void InitLCD( void){
        uint8_t BF = 0x80;
        _delay_ms(40);
            PORTB &= ~(1<<RS);
            PORTB = (0x30 >> 4);
            PORTB |= (1<<E);
            asm("nop");
            asm("nop");
            asm("nop");
            PORTB &= ~(1<<E);
            PORTB = 0;
        _delay_ms(5);
            PORTB &= ~(1<<RS);
            PORTB = (0x30 >> 4);
            PORTB |= (1<<E);
            asm("nop");
            asm("nop");
            asm("nop");
            PORTB &= ~(1<<E);
            PORTB = 0;
        _delay_us(150);
            PORTB &= ~(1<<RS);
            PORTB = (0x30 >> 4);
            PORTB |= (1<<E);
            asm("nop");
            asm("nop");
            asm("nop");
            PORTB &= ~(1<<E);
            PORTB = 0;
        _delay_ms(5);
        do{
            BF = (0x80 & LCD_Read());
```

```
        } while (BF == 0x80);
            PORTB &= ~(1<<RS);
            PORTB = (0x20 >> 4);
            PORTB |= (1<<E);
            asm("nop");
            asm("nop");
            asm("nop");
            PORTB &= ~(1<<E);
            PORTB = 0;
        do{
            BF = (0x80 & LCD_Read());
        } while (BF == 0x80);
        // заработал 4-проводной интерфейс
        LCD_Write(CMD, 0x28); //2 строки, 5*8
        LCD_Write(CMD, 0x0C); //display on, cursor
on
        LCD_Write(CMD, 0x06); //cnt++, shift ena-
bled
    }

    void LCD_Write(uint8_t type, char data){
        uint8_t BF=0x80;
        do{
            BF = 0x80 & LCD_Read();
        } while (BF == 0x80);
        PORTB |= (type<<RS);
        PORTB |= (1<<E);
        PORTB &= ~(0x0F);
        PORTB |= (0x0F & (data>>4));   //старшая
тетрада
        PORTB &= ~(1<<E);
        asm("nop");
        asm("nop");
        asm("nop");
        PORTB |= (1<<E);
        PORTB &= ~(0x0F);
```

```
    PORTB |= (0x0F & data);  //младшая тетрада
    PORTB &= ~(1<<E);
    PORTB = 0;
}

char LCD_Read( void){
    char retval=0;
    PORTB &= ~(1<<RS);
    PORTB |= (1<<RW);
    DDRB &= ~(1<<D4 | 1<<D5 | 1<<D6 | 1<<D7);
    PORTB |= (1<<E);
        asm("nop");
        asm("nop");
    retval = (PINB & 0x0F)<<4;
    PORTB &= ~(1<<E);
    asm("nop");
    asm("nop");
    asm("nop");
    PORTB |= (1<<E);
        asm("nop");
        asm("nop");
    retval |= (PINB & 0x0F);
    PORTB &= ~(1<<E);
    DDRB |= (1<<D4 | 1<<D5 | 1<<D6 | 1<<D7);
    PORTB = 0;
    return retval;
}

void OneWire_Init( void){
    DDRD |= (1<<WIRE);    // шину на выход
    SET_0_WIRE;           // установить 0
    _delay_us(500);       // ждать не менее
480 мкс
    SET_1_WIRE;
    DDRD&= ~(1<<WIRE);    // шину на вход
    _delay_us(60);
```

```
        if ((PIND& 1<<WIRE) == 0){ // ожидается
импульс сброса
            while ((PIND & 1<<WIRE) == 0);
            _delay_us(420);
        } else {
            SendString("No response\r\n");
        }
    }

    void OneWire_Send_1_0(uint8_t bit){
        _delay_us(2);
        DDRD |= (1<<WIRE);
        SET_0_WIRE;
        _delay_us(10);
        if (bit != 0){
            SET_1_WIRE;
            _delay_us(90);
        } else {
            SET_0_WIRE;
            _delay_us(90);
            SET_1_WIRE
        }
    }

    uint8_t OneWire_Read_1_0(void){
        uint8_t bit = 0;
        _delay_us(2);
        DDRD |= (1<<WIRE);
        SET_0_WIRE;
        _delay_us(2);
        SET_1_WIRE;
        DDRD &= ~(1<<WIRE);
        _delay_us(2);
        if((PIND & 1<<WIRE) == 0){
            bit = 0;
        } else {
```

```
            bit = 1;
        }
        _delay_us(90);
        return bit;
    }

    void OneWire_SendByte(uint8_t data){
        for (uint8_t i=0; i<8; i++){
            OneWire_Send_1_0(0x01 & data);
            data = data >> 1;
        }
    }

    uint8_t OneWire_ReadByte( void){
        uint8_t retval = 0;
        for (uint8_t i=0; i<8; i++){
            retval |= OneWire_Read_1_0() << i;
        }
        return retval;
    }
```

Таким образом, рассмотрена работа с датчиком DS18B20 по протоколу 1-Wire. Более полная информация по командам и алгоритмам работы с данными устройствами располагается в технических описаниях на соответствующие компоненты.

Задания и пояснения к выполнению практического задания

Изучить техническое описание на микросхему DS18B20.

Организовать опрос двух датчиков DS18B20, подключенных по шине 1-Wire с внешним питанием.

Идентификаторы датчиков считать заранее известными. Один из датчиков должен работать с разрешением 12 бит, второй – 10 бит.

Информацию от датчиков перевести из двоичных кодов в десятичные значения температуры и отобразить на ЖКИ.

При изменении значения какого-либо из параметров, оно передается в USART.

Контрольные вопросы

1. Назовите преимущества цифровых датчиков температуры по сравнению с аналоговыми.
2. Что такое «паразитное питание» для датчика DS18B20?
3. Интерфейс 1-Wire позволяет подключать к одной шине множество подчиненных устройств. Как происходит обращение к конкретному устройству?
4. Опишите общую последовательность обмена данными по шине 1-Wire.

6

Использование бесконтактных считывателей RFID

Теоретические сведения

RFID (англ. Radio Frequency IDentification, радиочастотная идентификация) – способ автоматической идентификации объектов, в котором посредством радиосигналов считываются или записываются данные, хранящиеся в так называемых транспондерах, или RFID-метках.

Любая RFID-система состоит из считывающего устройства (считыватель, ридер или интеррогатор) и транспондера (он же RFID-метка, иногда также применяется термин RFID-тег).

Большинство RFID-меток состоит из двух частей. Первая – интегральная схема (ИС) для хранения и обработки информации, модулирования и демодулирования радиочастотного (RF) сигнала и некоторых других функций. Вторая – антенна для приёма и передачи сигнала.

Wiegand26 протокол

Протокол Wiegand26 появился в 80-х годах как промышленный стандарт для систем контроля доступа – трехпроводная линия соединяет считыватель магнитных карт с платой контроллера, который обеспечивает хранение и контроль номеров карт.

В настоящее время протокол Wiegand26 был расширен и дополнен, но до сих пор активно используется как стандарт в большинстве охранных систем.

Рисунок 50 – Считыватель RFID карт

Физический уровень

Для передачи данных используются два провода – D0 / D1 и один – для объединения земли ридера и контроллера. С учетом линии питания, минимально для подключения считывателя необходимо 4 жилы. Остальные 3-4 используются для управления дополнительными функциями считывателя.

В состоянии покоя линии D0 и D1 имеют высокий уровень (+5V) или неопределенный в расчете на подтяжку на стороне контроллера. Для передачи бита данных считыватель притягивает одну из линий к земле на 50 мкс, отпускает на 2 мс и передает следующий бит, притягивая соответствующую линию на 50-200 мкс – и снова пауза в 2 мс. Если пауза между передачами больше 2 мс, значит передача данных завершена. Время кадра 52-58 мс.

Состояние бита 0 или 1 определяется по тому, какая линия (D0 или D1) была притянута к земле в момент передачи бита. Линия D0 отвечает за нули, линия D1 – за единицы.

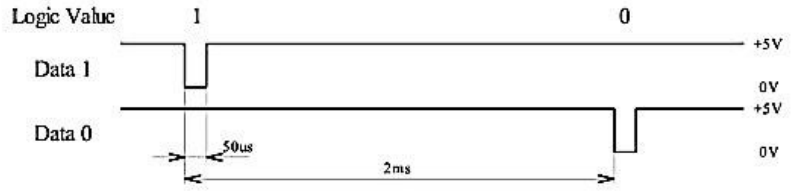


Рисунок 51 – Диаграммы сигналов Wiegand протокола

Однако следует учитывать, что у разных производителей ширина импульсов и пауз между ними может отличаться. Импульсы от 20 до 200 мкс и паузы от 200 до 3000 мкс.

Если быть точнее, то стоит говорить не про промежутки (паузы), а про длину посылки одного бита, впрочем, это видно на графике, т.к. время считается от начала посылки одного бита до начала посылки второго и включает в себя сам бит.

Логический уровень

На логическом уровне протокол Wiegand26 состоит из 26 бит данных, внутри которых хранятся 2 номера – facility code и card number.

Facility code – номер объекта или зоны.

Card number – номер карточки доступа.

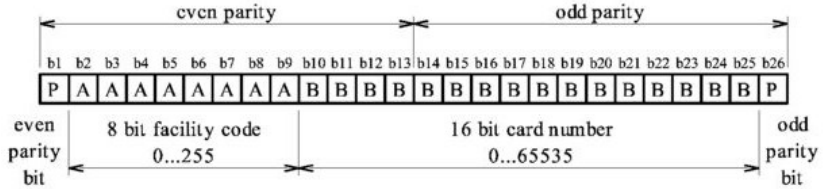


Рисунок 52 – Формат кадра Wiegand протокола

Первый бит используется для контроля первой половины данных (b1-b13): если количество единиц в первой половине посылки нечетное – бит равен 1. Следующие 8 бит (b2-b9) – facility code (старший бит впереди). Затем идут 16 бит (b14-b25) с номером карты (старший бит впереди). Последний бит в посылке отвечает за контроль второй половины собранных данных (b14-b26): если количество единиц во второй половине посылки четное, то бит равен 0.

Пример кадра:

D0 1 0101 0100 0011 0111 1010 1001 0

D1 0 1010 1011 1100 1000 0101 0110 1

После инверсии каналов D0 и D1 получаем данные кадра

Frame = 1 0101 0100 0011 0111 1010 1001 0.

Очевидно, что биты четности соответствуют принятым данным. В шестнадцатеричном представлении кадр выглядит следующим образом:

54h 37h A9h

Считыватель с интерфейсом UART (TTL)

Рисунок 53 – RDM6300 – бесконтактный считыватель RFID карт (125 кГц EM4100) с интерфейсом UART (TTL)

RDM6300 – бесконтактный считыватель предназначен для дистанционного считывания номера (ID) RFID-брелка или карты стандарта EM4100, EM4305, работающих на частоте 125 кГц, и передачи этого номера через последовательный интерфейс UART (TTL) на контроллер, который управляет электрозамком в системах контроля доступа в домах, квартирах, гаражах и других объектах.

Технические характеристики RDM6300:

– рабочая частота 125 кГц;
– битрейт 9600 (TTL уровень RS232);
– интерфейс Wiegand26 или TTL уровень RS232;
– дальность считывания 20…50 мм (в зависимости от карты/брелка, производителя);
– напряжение питания постоянного тока 5 В.

RDM6300 имеет 9 контактов, где Vcc и GND повторены дважды. Контакт Vcc должен быть подключен к 5V DC. ANT1 и ANT2 используются для подключения антенны к плате. TX используется для передачи данных, а RX используется для извлечения данных. Вывод LED можно использовать для отображения того, была ли RFID-

метка успешно прочитана. Если метка RFID отсутствует, вывод LED находится на 5V (HIGH).

Когда RFID-тег был обнаружен, RDM6300 отправляет кадр в 14 байтами: Head [1 байт], Data [10 байт], CRC [2 байт] и Tail [1 байт]. Head (или преамбула) всегда равна 0x02. Точно так же Tail всегда равен 0x03. Поле данных содержит значения HEX, закодированные в ASCII. Иногда производители стремятся разделить поле данных метки RFID на две части: версию и тег. Поэтому в зависимости от типа метки RFID, первые два байта данных могут быть версией, а остальные 8 байтов – фактическим тегом RFID. Чтобы вычислить контрольную сумму (CRC) из поля данных, необходимо выполнить XOR-операцию по всем записям данных.

Практическая часть

В практической части рассмотрим чтение данных RFID-карты по интерфейсу UART. Пример посылки, принятой с RDM6300 (ASCII):

02h 30h 37h 30h 30h 35h 34h 33h 37h 41h 39h 43h 44h 03h.

Переведем фрагмент данных и контрольной суммы к HEX виду. Отметим, что значения ASCII от 30h до 39h соответствуют значениям от 0 до 9, значения от 41h до 46h – значениям от A до F. Результат выглядит так:

DATA = 07h 00h 54h 37h A9h CRC = CDh

Проверка:

CRC = 07h *XOR* 00h *XOR* 54h *XOR* 37h *XOR* A9h = CDh

Здесь:

Version = 07h

Facility code = 0054h

Card number = 37A9h,

что соответствует надписи на карте RFID 0005519273 084,14249,

где 0005519273_{10} является десятичным представлением 005437A9h.

084_{10} = 0054h 14249_{10} = 37A9h

Для моделирования реальной системы воспользуемся схемой из двух микроконтроллеров, один из которых является генератором тестовых сигналов.

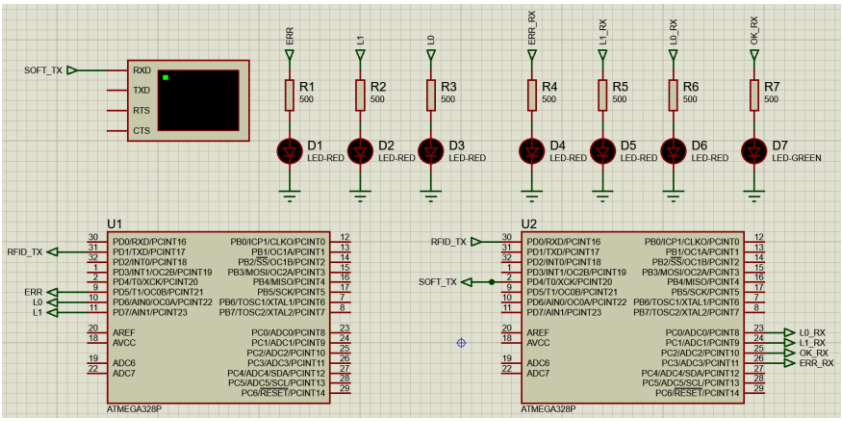


Рисунок 54 – Тестовая схема для моделирования RFID

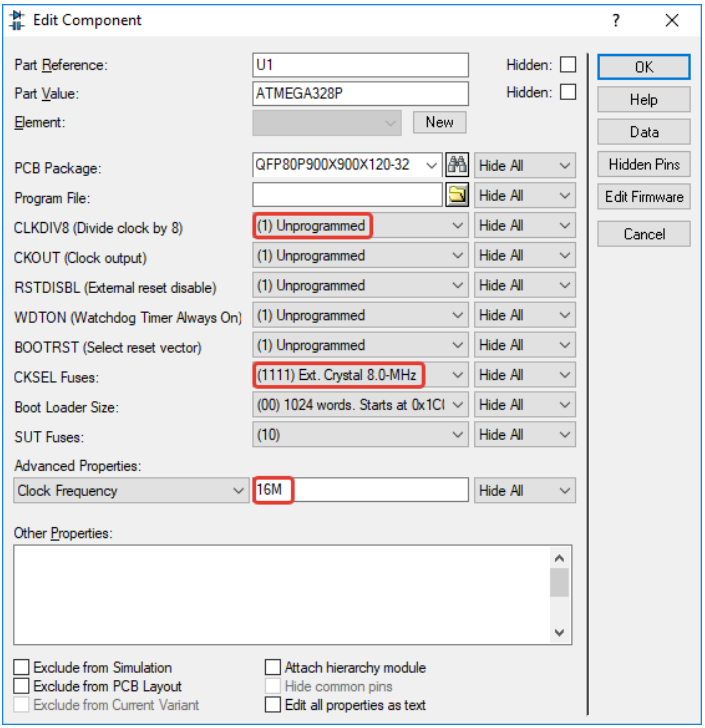


Рисунок 55 – Настройка фьюз-бит микроконтроллеров

Исходный код передатчика

```
#define F_CPU 16000000UL
#define BAUDRATE 9600
#include <avr/io.h>
#include <util/delay.h>
#define CARD_DATA_SIZE 14
#define BASE_DELAY 100

#define LED_ERR 5
#define LED0 6
#define LED1 7
#define LEDS_OFF() PORTD &= ~((1 << LED1) | (1 << LED0) | (1
<< LED_ERR))

//Массив данных для 4 карт
volatile char cardData[56] = {
    0x02, 0x30, 0x37, 0x30, 0x30, 0x35, 0x34, 0x33, 0x37, 0x41,
0x39, 0x43, 0x44, 0x03,
    0x02, 0x30, 0x37, 0x30, 0x30, 0x35, 0x34, 0x33, 0x37, 0x41,
0x37, 0x43, 0x33, 0x03,
    0x02, 0x30, 0x37, 0x30, 0x30, 0x35, 0x34, 0x33, 0x37, 0x41,
0x35, 0x43, 0x31, 0x03,
    0x02, 0x30, 0x37, 0x30, 0x30, 0x35, 0x34, 0x33, 0x37, 0x41,
0x34, 0x43, 0x37, 0x03};

void InitPorts(){
    DDRD = 0xFF;
    PORTD = 0x00;
}

void InitUart(){
    UCSR0B = (1 << TXEN0);
    UCSR0C = (1 << UCSZ01 | 1 << UCSZ00);
    UBRR0H = 0;
    UBRR0L = F_CPU/BAUDRATE/16 - 1;
```

```
}

void SendChar(char symbol){
    while(!(UCSR0A & (1<<UDRE0)));
    UDR0 = symbol;
}

void SendPacket(char* dat){
    int i = 0;
    while(i < CARD_DATA_SIZE)
    SendChar(dat[i++]);
}

void Leds_On(uint8_t value){
    PORTD |= (value >> 1) << LED1;
    PORTD |= (value & 0x01) << LED0;
}

void Delay_Func(uint8_t iteration){
    //Функция необходима для формирования изменяемой задержки
    //так как _delay_ms() не принимает изменяемые переменные в качестве параметра
    int i = 0;
    while(i++ < iteration)
        _delay_ms(BASE_DELAY);
}

int main(void)
{
    InitPorts(); //Инициализация портов
    InitUart(); //Инициализация UART
    uint8_t cardIdx = 0;
    int shift = 0;
    uint8_t iterations =0;
```

```
    uint8_t doError = 0;
  char sendData[CARD_DATA_SIZE];
    while (1)
    {
        LEDS_OFF(); //Вызов макроса выключения свето-
диодов
        cardIdx = rand() & 0x03; //Выбор случайной карты
        Leds_On(cardIdx); //Индикация номера кадра

        //Формирование кадра для отправки
        shift = cardIdx * CARD_DATA_SIZE;
        for (int i = 0; i < CARD_DATA_SIZE; i++)
            sendData[i] = cardData[i + shift];
        //Внесение ошибки в кадр
        doError = rand() & 0x01; //
        if(doError) sendData[1] = 0x35;
            PORTD |= doError << LED_ERR; //Индикация
наличия ошибки
        //Отправка данных
        SendPacket(sendData);
        //Случайная задержка отправки сигнала
        //от 100мс до 1.7 сек
        iterations = rand() & 0x0F;
        Delay_Func(++iterations);
    }
}
```

Исходный код приемника

```
#define F_CPU 16000000UL
#define BAUDRATE 9600

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
```

```
#define CARD_DATA_SIZE 5
#define BASE_DELAY 100

#define LED0 0
#define LED1 1
#define LED_OK 2
#define LED_ERR 3
#define LEDS_OFF PORTC = ~((1 << LED1) | (1 << LED0) | (1 << LED_ERR) | (1 << LED_OK))

#define READY 1
#define RECEIVING 2
#define RECEIVED 3

#define UART_RCV_ON UCSR0B = (1 << RXEN0) | (1 << RXCIE0)
#define UART_RCV_OFF UCSR0B = 0x00

#define TX_SOFT 4
#define SET_0_SOFT PORTD &= ~(1 << TX_SOFT)
#define SET_1_SOFT PORTD |= (1 << TX_SOFT)

//Массив данных для 4 карт
volatile char cardData[20] = {0x07, 0x00, 0x54, 0x37, 0xA9,
                              0x07, 0x00, 0x54, 0x37, 0xA7,
                              0x07, 0x00, 0x54, 0x37, 0xA5,
                              0x07, 0x00, 0x54, 0x37, 0xA3};

char arr[15] = {0x02, 0x35, 0x37, 0x30, 0x30, 0x35, 0x34, 0x33, 0x37, 0x41, 0x35, 0x43, 0x31, 0x03, 0x00}; //Буфер приема
volatile uint8_t state = READY;
volatile uint8_t uCnt = 0;
volatile char uValue = 0;
volatile uint8_t cardCount = 4;
```

```
ISR(USART_RX_vect){
  uValue = UDR0;
  if(uValue == 0x02)  {
    uCnt = 0;  state = RECEIVING;
  }
  if(state == RECEIVING)  {
    arr[uCnt++] = uValue;
    if(uCnt > 14) uCnt = 0; //Зацикливание записи в массив приема
  }
  //Условие окончания приема данных
  if(uValue == 0x03 && uCnt > 0)  {
    state = RECEIVED;
  }
}

void InitPorts(){
  DDRD = 0xFF;
  PORTD = 0x00;
  DDRC = 0xFF;
  PORTC = 0x00;
}

void InitUart(){
  UCSR0B = (1 << RXEN0) | (1 << RXCIE0);
  UCSR0C = (1 << UCSZ01 | 1 << UCSZ00);
  UBRR0H = 0;
  UBRR0L = F_CPU/BAUDRATE/16 - 1;
}

void SendCharSoftUART(char symbol)  {
  //BAUDRATE_SOFT = 57600 bps
  //t = 1 / BAUDRATE = 17.36 us
  uint8_t i = 0x01;
  SET_0_SOFT; // Старт бит
  _delay_us(17.36);
```

```
  while(i > 0){ // 8 бит данных
    if(symbol & i) SET_1_SOFT;
    else SET_0_SOFT;
    i <<= 1; // Младший бит первый (LSB)
    _delay_us(17.36);
  }
  SET_1_SOFT; // Стоп бит
  _delay_us(17.36);
}

void SendStringSoftUART(char * buffer)  {
  while(*buffer != 0){
    SendCharSoftUART(*buffer++);
  }
}

void Leds_On(uint8_t value){
  PORTC |= (value >> 1) << LED1;
  PORTC |= (value & 0x01) << LED0;
}

uint8_t ConvertCharToByte(char val)  {
  switch(val)  {
    case '0': return 0x00;
    case '1': return 0x01;
    case '2': return 0x02;
    case '3': return 0x03;
    case '4': return 0x04;
    case '5': return 0x05;
    case '6': return 0x06;
    case '7': return 0x07;
    case '8': return 0x08;
    case '9': return 0x09;
    case 'A': return 0x0A;
    case 'B': return 0x0B;
```

```
        case 'C': return 0x0C;
        case 'D': return 0x0D;
        case 'E': return 0x0E;
        case 'F': return 0x0F;
        default: return 0x00;
      }
    }

    int main(void)
    {
      //Инициализация
      InitPorts();
      InitUart();
      SET_1_SOFT;
      SendStringSoftUART("Hello\r\n");
      sei();
      int cardIdxEst = -1; // Вычисленный индекс карты
      int curMatch = -1;
      char tempArr[CARD_DATA_SIZE + 1];
      uint8_t CRC = 0; // Вычисленное CRC
      uint8_t rCRC = 0; // Принятое CRC
      int tmp = 0;
      int shift;

      while (1)
      {
        //Ожидание приема кадра
        if(state == RECEIVED)   {
          state = READY; // Сброс состояния в режим ожидания
          // Проверка CRC, извлечение информации из принятого
массива и запись во временный массив
          rCRC = (ConvertCharToByte(arr[11]) << 4);
          rCRC |= ConvertCharToByte(arr[12]);
          CRC = 0;
          for (int i = 0; i < CARD_DATA_SIZE; i++)   {
```

```
            tmp = (ConvertCharToByte(arr[2 * i + 1]) << 4) | Con-
vertCharToByte(arr[2 * i + 2]);
            tempArr[i] = tmp & 0xFF;
            CRC ^= tempArr[i];
          }
          LEDS_OFF;
          //Отключаем аппаратный UART, чтобы не влияло на тай-
минг программного UART
          UART_RCV_OFF;
          SendStringSoftUART(arr);
          //Проверка контрольной суммы
          if((rCRC == CRC) & (rCRC != 0))  {
            PORTC |=  1 << LED_OK;
            SendStringSoftUART(" CRC OK ");
          }
          else {
            PORTC |=  1 << LED_ERR;
            SendStringSoftUART(" CRC InCorrect ");
          }
          //Поиск совпадения по номеру карты
          cardIdxEst = -1;
          shift = 0;
          for(int i = 0; i < cardCount; i++)  {
            curMatch = tempArr[CARD_DATA_SIZE - 1] ^ cardDa-
ta[shift + CARD_DATA_SIZE - 1];
            for (int j = CARD_DATA_SIZE - 2; j >= 0; j--)  {
              curMatch += tempArr[j] ^ cardData[shift + j];
              if(curMatch > 0) break;
            }
            shift += CARD_DATA_SIZE;
            if(curMatch == 0) {
              cardIdxEst = i; break;
            }
          }
          //Индикация индекса карты
```

```
        Leds_On(cardIdxEst & 0x03);
        if(cardIdxEst >= 0)   {
          SendStringSoftUART("CARD INDEX = ");
          SendCharSoftUART(cardIdxEst + 0x30);
        }
        else SendStringSoftUART("CARD NOT FOUND ");
        SendStringSoftUART("\r\n");

        UART_RCV_ON; //Включаем аппаратный UART
      }
    }
  }
```

Задания и пояснения к выполнению практической работы

Напишите программу в соответствии с вариантом, предложенным преподавателем.

Для получения положительной оценки написать программу для работы по протоколу Wiegand26.

Контрольные вопросы

1. Какую информацию содержат идентификационные признаки?
2. Что является носителем идентификационных признаков или параметров?
3. Какие бывают идентификаторы?
4. Опишите устройство и функции систем радиочастотной идентификации (RFID).
5. Что представляют собой RFID считыватели, или ридеры?
6. Для чего предназначены стационарные считыватели?
7. Для чего предназначены настольные считыватели?
8. Для чего предназначены мобильные считыватели?

ЗАКЛЮЧЕНИЕ

Представленное учебное пособие призвано дать обучающимся базовые навыки по работе с микроконтроллерами. За рамками данного издания остались вопросы:

- цифровой обработки сигналов на микроконтроллерах;
- взаимодействия с внешними устройствами памяти;
- управления мощными нагрузками;
- реализации различных видов регуляторов в системах автоматического управления и т.д.

Однако, пользуясь полученными знаниями и навыками, студенты без труда смогут решать вопросы разработки и программирования перечисленных устройств, поскольку рассмотренные в практической части каждого раздела схемы являются их неотъемлемой частью.

Представленные здесь программы разработаны на языке С. С одной стороны, это облегчает и ускоряет процесс разработки и отладки программ, а с другой – нельзя не отметить некоторые негативные моменты: в отдельных случаях это может ограничить возможности разработчика в плане полноты и гибкости использования ресурсов микроконтроллера, эффективности использования его памяти, а также привести к снижению быстродействия программы. В подобных случаях необходим переход к программам, написанным на языке ассемблера.

Таким образом, настоящее учебное издание, давая необходимые базовые навыки разработки устройств на микроконтроллерах, обозначает и пути их дальнейшего развития и совершенствования.

СПИСОК ЛИТЕРАТУРЫ

1. Белов, А. В. Самоучитель разработчика устройств на микроконтроллерах AVR. Книга + диск / А. В. Белов. – Изд. 2-е, перераб. и доп. – СПб.: Наука и Техника, 2010. – 527 с. : ил. + 1 эл. опт. диск (CD-ROM).

2. Евстифеев, А. В. Микроконтроллеры AVR семейства Classic фирмы ATMEL / А. В. Евстифеев – М.: Додэка-XX1, 2006. – 288 с.

3. Предко, Майк. Руководство по микроконтроллерам: в 2 т. / М. Предко; пер. с англ. И. И. Шагурина, С. Б. Лужанского. – М.: Простмаркет, 2001. (Библиотека современной электроники).

4. Консультационно-технический центр по микроконтроллерам. Алфавитно-цифровые индицирующие ЖК-модули на основе контроллера HD44780. 2000 [Электронный ресурс]. – URL: http://www.cec-mc.ru/ techinfo /list /#item_694 (дата обращения 10.03.2015)/

5. Atmel Corporation. ATmega 48/88/168 datasheet. 2011 [Electronic resource]. – URL: http://www.atmel.com/images/doc2545.pdf (accessed: 10.03.2015).

6. Hitachi, Ltd. HD44780U. Dot Matrix Liquid Crystal Display Controller. Datasheet. 1998 [Electronic resource]. – URL: https://www.sparkfun.com/datasheets/LCD/ HD44780.pdf (accessed: 10.03.2015).

7. Maxim Integrated. DS18B20 Programmable Resolution 1-Wire Digital Thermometer. 2008 [Electronic resource]. – URL: http://datasheets.maximintegrated.com/en/ds/ DS18B20.pdf (accessed: 10.03.2015).

8. NXP Semiconductors. 74HC595, 74HCT595 8-bit serial-in, serial or parallel-out shift register with output latches. Datasheet. 2015 [Electronic resource]. – URL: http://www.nxp.com/documents/data_sheet/ 74HC_HCT595.pdf (accessed: 10.03.2015).

9. Wiegand26 протокол на AVR [Electronic resource]. – URL: http://zps-electronics.com/docs/wiegand_rfid_reader_avr/ (accessed: 25.02.2018).

Инструкция по прошивке микроконтроллера

Наиболее доступным программатором для AVR является USBAsp.

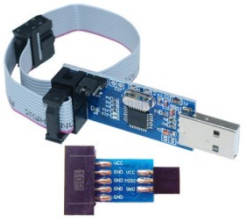


Рисунок П1 – Программатор USBAsp для микроконтроллеров AVR

Для проверки корректности работы прошивки напишем простейшую программу перемигивания светодиодов на отладочной плате Arduino Uno.

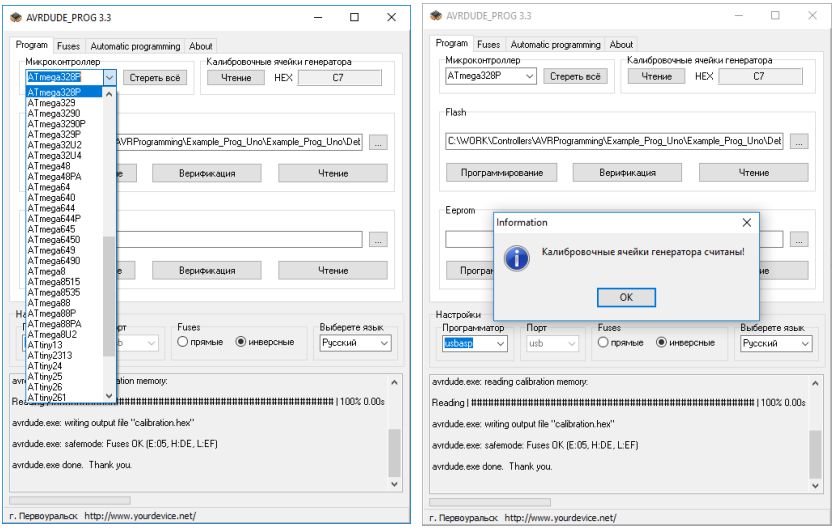
Тестовая программа

```
#define F_CPU 16000000UL //На плате Arduino Uno установлен кварцевый резонатор 16МГЦ
#include <avr/io.h>
#include <util/delay.h>
int main(void)
{
    DDRD = 0xFF;
    uint8_t dir = 0;
  while (1)
  {
            dir ^= 1;
            if(dir)
                    PORTD = 1 << PORTD0;
            else
                    PORTD = 1 << PORTD1;
            _delay_ms(300);
  }
}
```

Для программирования микроконтроллера программатором **USBAsp** используется программа AVRDUDE_PROG 3.3. Программа AVRDUDE_PROG работает не только с USBAsp, но и другими программаторами.

Прошивка микроконтроллера

Шаг 1. В программе AVRDUDE_PROG 3.3, переходим в блок «Микроконтроллер» и выбираем ATmega328P (рис. П2*а*).



а) б)

Рисунок П2 – Программа AVRDUDE, выбор микроконтроллера (а)**, чтение калиброванных ячеек** (б)

Для проверки правильности подключения программатора рекомендуется считать калибровочные ячейки генератора, нажав кнопку «Чтение». В случае правильного подключения программатора, правильного выбора микроконтроллера и его исправности – появится окно как на рисунке П2*б*.

Шаг 2. Далее необходимо выбрать прошивку, в блоке «Flash» нажимаем «. . .», переходим в папку тестового проекта и выбираем *.hex.

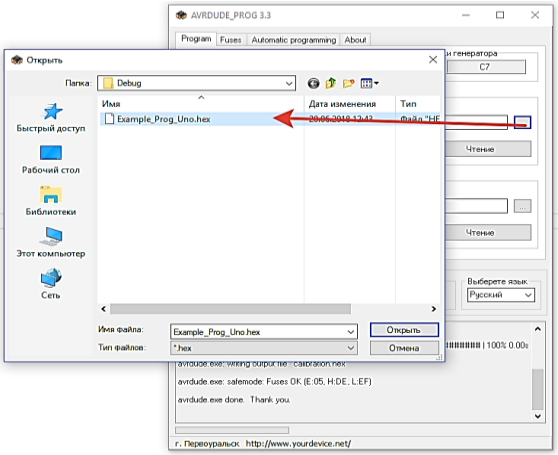


Рисунок П3 – Программа AVRDUDE. Выбор файла прошивки

Шаг 3. Подключаем программатор к плате «Arduino UNO R3» и нажимаем кнопку «Программирование».

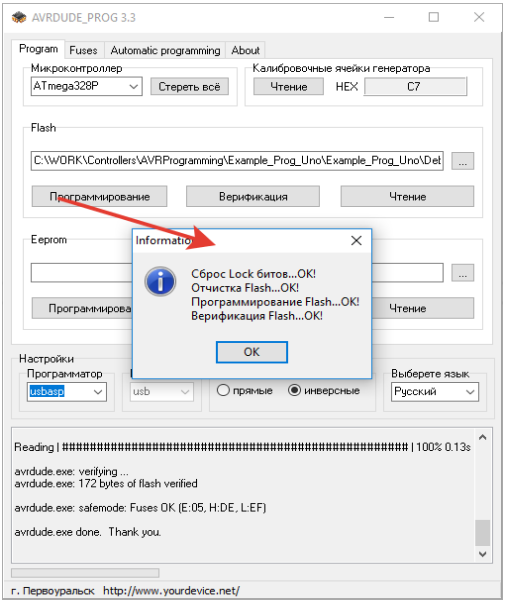


Рисунок П4 – Программа AVRDUDE. Программирование МК

По окончании программирования выйдет диалоговое окно об удачном окончании процесса.

Фьюз-биты

Внимание! Изменение фьюз-бит следует производить только с разрешения и под присмотром преподавателя, так как неправильная установка приведет к выводу отладочной платы из строя.

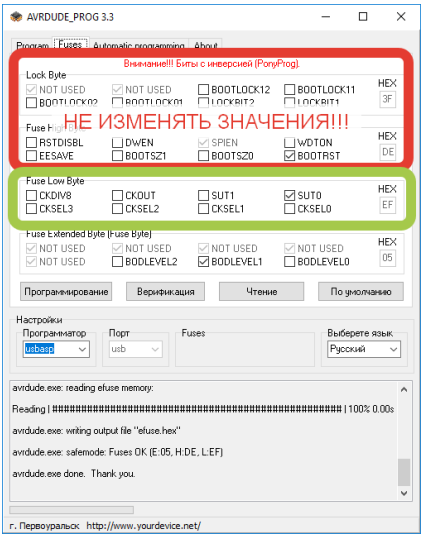


Рисунок П5 – Программа AVRDUDE. Настройка фьюз-бит

ПРИЛОЖЕНИЕ 2

Правила охраны труда и техники безопасности при выполнении лабораторных и практических работ

1. Общие требования безопасности

1.1. К работе с электроизмерительными приборами, электроустановками, ЭВМ (под руководством преподавателя или ответственного за лабораторию) допускаются лица, прошедшие инструктаж по охране труда, медицинский осмотр и не имеющие противопоказаний по состоянию здоровья.

1.2. При работе в лаборатории студенты должны соблюдать правила поведения, расписание учебных занятий, установленные режимы труда и отдыха.

1.3. При работе с электроизмерительными приборами возможно воздействие на работающих следующих опасных факторов:

– поражение электрическим током при прикосновении к оголенным проводам и при работе с приборами, находящимися под напряжением;

– травмирование рук при использовании неисправного инструмента.

ПОМНИТЕ! Электрический ток может вызвать ожоги, обморок, судороги, прекращение дыхания, даже смерть.

ЗНАЙТЕ! К индивидуальным средствам защиты относятся: для глаз – защитные очки, для лица – маски, для рук – диэлектрические перчатки, для дыхательных путей – респиратор.

1.4. При выполнении лабораторных работ должны использоваться указатели напряжений и изолированный инструмент.

1.5. В лаборатории для выполнения лабораторных работ должна быть медицинская аптечка с набором необходимых медикаментов и перевязочных средств.

1.6. Студенты обязаны соблюдать правила пожарной безопасности, знать места расположения первичных средств пожаротушения.

1.7. При несчастном случае пострадавший или очевидец несчастного случая обязан немедленно сообщить преподавателю или зав. лабораторией, который сообщает об этом администрации ПГТУ.

При неисправности электроизмерительных приборов, инструмента прекратить работу и поставить об этом в известность преподавателя или зав. лабораторией.

1.8. В процессе работы соблюдать правила ношения спецодежды, пользования индивидуальными и коллективными средствами защиты, соблюдать правила личной гигиены, содержать в чистоте рабочее место.

1.9. Студенты, допустившие невыполнение или нарушение инструкции по охране труда, привлекаются к дисциплинарной ответственности в соответствии с правилами внутреннего трудового распорядка ПГТУ и подвергаются внеочередной проверке знаний правил охраны труда.

2. Требования безопасности перед началом работы

2.1. Получив разрешение на проведение практических работ, ПРОВЕРЬТЕ состояние и исправность электроизмерительных приборов и инструмента, наличие и исправность защитного заземления.

2.2. Подготовьте необходимые для работы материалы, приспособления и разложите на свои места, уберите с рабочего стола все лишнее.

2.3. Подготовьте к работе средства индивидуальной защиты, убедитесь в их исправности.

3. Требования безопасности во время работы

3.1. ПОМНИТЕ! Электрический ток величиной 0,1 А и напряжением свыше 42 В опасен для жизни человека.

3.2. Пребывание студентов в лаборатории разрешается только в присутствии преподавателя или ответственного за лабораторию.

3.3. Практические работы студенты выполняют только в присутствии преподавателя или ответственного за лабораторию.

3.4. ЗАПРЕЩАЕТСЯ применять электрические приборы и устройства, не соответствующие требованиям безопасности труда. Не применять оборудование, приборы и кабели с открытыми токоведущими частями.

3.5. При сборке электрической схемы использовать провода с наконечниками без повреждений изоляции, избегать пересечений проводов, источник тока подключать в последнюю очередь.

3.6. Собранную электрическую схему включать под напряжение только после проверки ее преподавателем или ответственным за лабораторию, не производить переключений в цепях до отключения источника тока.

3.7. Наличие напряжения в электрической цепи проверять только приборами.

3.8. Не допускать предельных нагрузок измерительных приборов.

3.9. Не оставлять без надзора невыключенные электрические устройства и приборы.

4. Требования безопасности в аварийных ситуациях

4.1. При обнаружении неисправности в работе электроизмерительных приборов или лабораторной установки, находящихся под напряжением (повышенном их нагревании, появлении искрения и т.д.), немедленно отключить источник электропитания, вывесить табличку о неисправности оборудования и сообщить об этом преподавателю или зав. лабораторией.

4.2. При коротком замыкании в электроизмерительных приборах и лабораторных установках и их загорании немедленно отключить их от электросети, сообщить о пожаре в пожарную часть по телефону 01 и приступить к тушению очага возгорания углекислотным (порошковым) огнетушителем.

4.3. При получении травмы сообщить преподавателю или зав. лабораторией, оказать первую помощь пострадавшему, при необходимости отправить пострадавшего в ближайшее лечебное учреждение, сообщить об этом администрации ПГТУ.

5. Требования безопасности по окончании работы

5.1. Отключить электроизмерительные приборы и лабораторные установки от электросети.

5.2. Привести в порядок рабочее место.

5.3. Сообщить преподавателю или ответственному за лабораторию об окончании работы и получить разрешение на уход из лаборатории.

Учебное издание

РОЖЕНЦОВ Алексей Аркадьевич
БАЕВ Алексей Александрович
ГАРИПОВА Юлия Евгеньевна
ОХОТНИКОВ Сергей Аркадьевич

ПРИМЕНЕНИЕ МИКРОКОНТРОЛЛЕРОВ В РАДИОТЕХНИЧЕСКИХ И БИОМЕДИЦИНСКИХ СИСТЕМАХ

Учебное пособие

Редактор *Л. С. Емельянова*
Компьютерная верстка *С. Н. Эштыкова, А. А. Баев, Ю. Е. Гарипова*
Дизайн обложки *С. Н. Эштыкова*

Подписано в печать 20.04.2018. Формат 60×84 $^{1}/_{16}$
Бумага офсетная. Печать офсетная.
Усл. печ. л. 10,0. Тираж 100 экз. Заказ № 968.

Поволжский государственный технологический университет
424000 Йошкар-Ола, пл. Ленина, 3

Отпечатано с готового оригинал-макета в ООО «Принтекс»
Республика Марий Эл, г. Йошкар-Ола, б-р Победы, 14