Министерство науки и высшего образования Российской Федерации

Томский государственный университет
систем управления и радиоэлектроники

И. М. Васильев

Применение микроконтроллеров архитектуры RISC для создания электронной аппаратуры робототехнических систем

Методические указания по выполнению самостоятельной работы студентов

Томск
2023

УДК 004.02
ББК 3стд2-02
В 19

Рецензент:

Антипин М. Е. доцент каф. управления инновациями ТУСУР,
канд. физ. мат. наук

Васильев, Иван Михайлович
В 19 Применение микроконтроллеров архитектуры RISC для создания электронной аппаратуры робототехнических систем: Васильев, И. М. Применение микроконтроллеров архитектуры RISC для создания электронной аппаратуры робототехнических систем: Методические указания к лабораторным работам / И. М. Васильев. — Томск: ТУСУР, 2023. — 47 с. »

Методические указания содержат рекомендации и материалы, необходимые для самостоятельной работы студентов по дисциплине «Программирование микроконтроллеров для робототехнических систем» .

Для студентов высших учебных заведений.

Одобрено на заседании кафедры УИ, протокол 5 от 28.12.22.

УДК 004.02
ББК 3стд2-02

© Васильев И.М., 2023
© Томск. гос. ун-т систем упр. и радиоэлектронники, 2023

Оглавление

Общие положения 4
Лабораторная работа № 1. Работа с портами ввода/вывода. 5
Лабораторная работа № 2. Внешние прерывания микроконтроллеров. 14
Лабораторная работа № 3. Работа с таймерами-счётчиками AVR. 25
Лабораторная работа № 4. Работа с встроенным АЦП микроконтроллера AVR. 36
Список рекомендуемой литературы 44

Общие положения

Данные методические указания разработаны для студентов, обучающихся в Томском государственном университете систем управления и радиоэлектроники (далее - Университет).

Структура дисциплины «Программирование микроконтроллеров для робототехнических систем» предполагает выполнение студентами самостоятельной работы как по освоению теоретического материала, так и в рамках выполнения практических заданий. Рекомендации по выполнению самостоятельной работе студентов приведены в соответствующих методических указаниях.

В ходе выполнения самостоятельной работы студентам прививаются навыки работы с зарубежной технической документацией, навыки формирования и оптимизации алгоритмов обработки информации, проектирования электрических принципиальных схем простейших устройств на базе микроконтроллеров, структурирование программного кода, самостоятельного изучения архитектуры микроконтроллеров отвечать на вопросы, оформлять и представлять результаты работы.

Рекомендации подготовлены с целью помочь студентам в успешном освоении дисциплины и подготовке и прохождении промежуточных этапов аттестации.

Лабораторная работа № 1.
Работа с портами ввода/вывода.

Цель: изучить принцип работы портов ввода/вывода микроконтроллеров.

Краткие теоретические сведения

Каждый микроконтроллер (далее – «МК») имеет определенное количество ножек, позволяющих взаимодействовать с внешними устройствами путем подачи или считывания логического сигнала. Такие ножки имеют обозначение GPIO (General-purpose input/output). Внутри микроконтроллера они логически объединены в порт и обозначаются как I/O Ports (input/output ports). В рамках данной лабораторной работы будет рассматриваться микроконтроллер Atmega328P, входящий в состав платы Arduino UNO. Структурная схема внутреннего устройства GPIO приведена на рисунке 1.

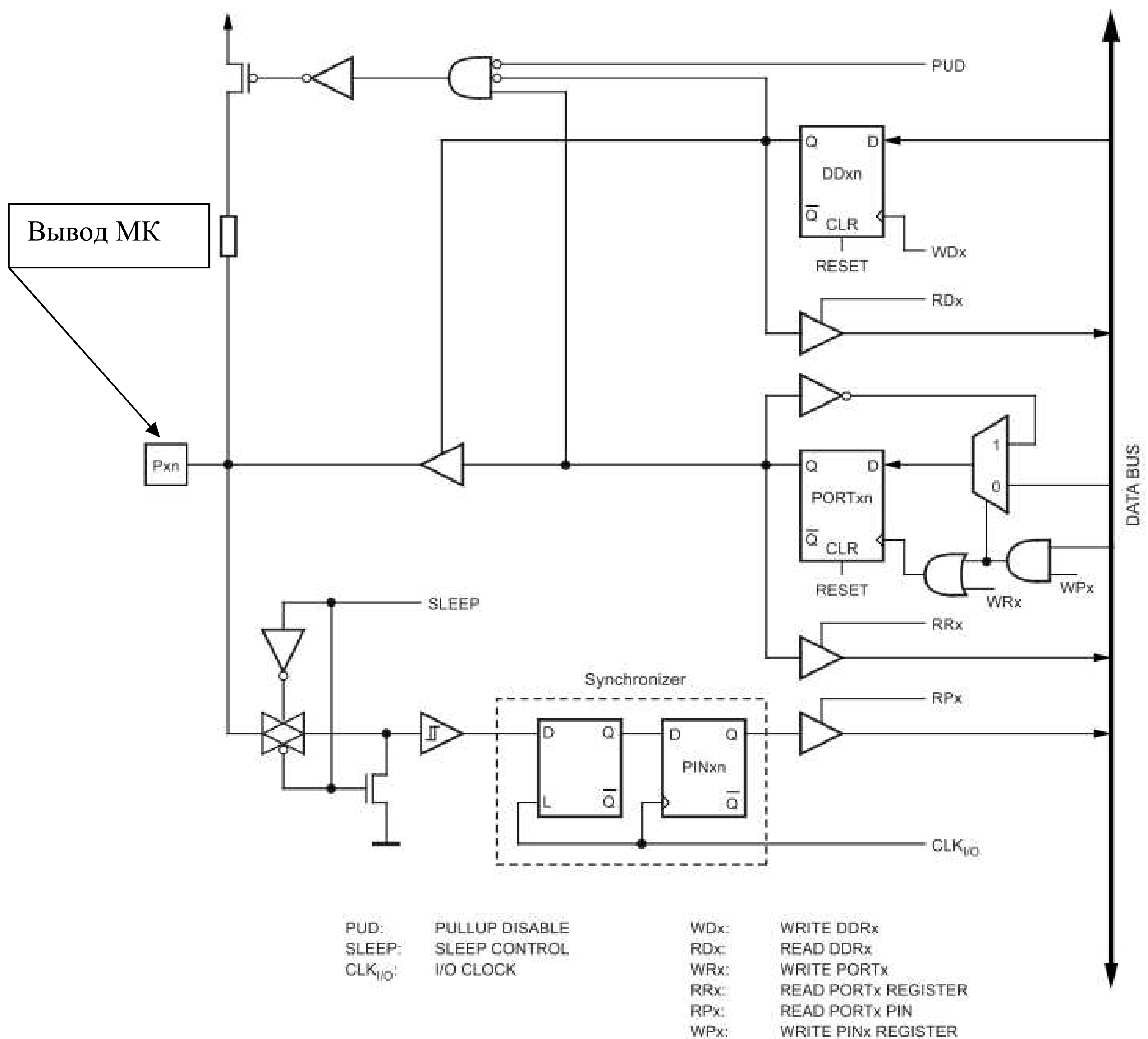

Рисунок 1 Структурная схема внутреннего устройства портов ввода/вывода Atmega328P

Микроконтроллер может, как считывать, так и выдавать сигнал на свой вывод. Причем считывать данные можно в любой момент, т.к. ножка Pxn подключена через согласующие блоки к биту регистра PINxn всё время, пока МК не находится в спящем режиме (синяя линия на рисунке 2). Здесь и далее x – буквенное обозначение порта, n – его разряд.

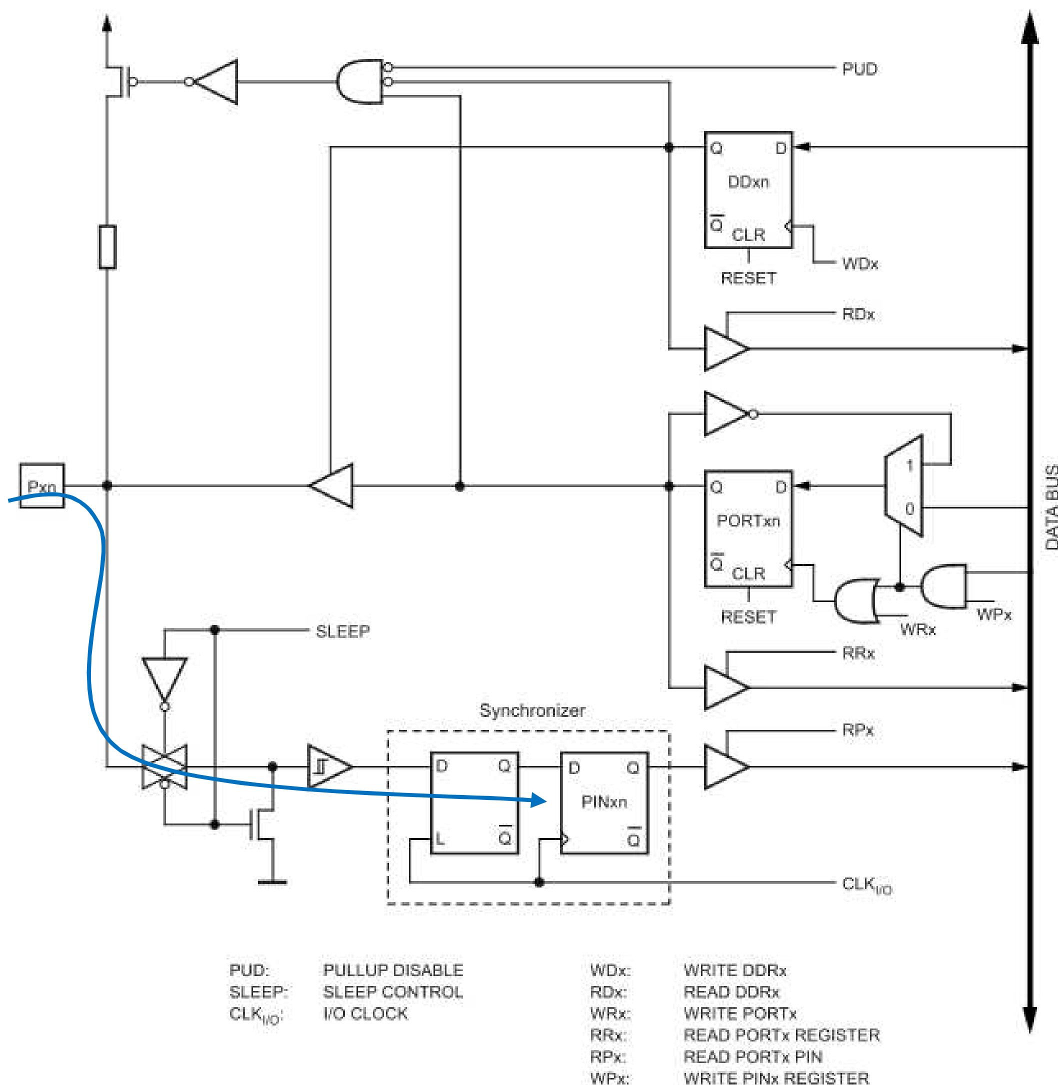

Рисунок 2. Направленность входных данных на устройство приема

Здесь стоит отметить, что регистр – простейшее устройство для хранения двоичных чисел. Согласно **разделу 13 документации** на Atmega328P PINn – восьмиразрядный регистр, т.е. содержит 8 бит данных. МК может обращаться как к отдельно взятым битам регистра, так и к его полному значению. Например если на ножке микроконтроллера PB0 (Port B, нулевой разряд) установлена «1», а на остальных ножках «0», то регистр PINB (регистр входных данных порта «B») примет значение PINB=0b00000001.

7	6	5	4	3	2	1	0
PINB7	**PINB6**	**PINB5**	**PINB4**	**PINB3**	**PINB2**	**PINB1**	**PINB0**
0	0	0	0	0	0	0	**1**

Для того что бы выдавать сигнал, необходимо разрешение выдачи данных на внешний вывод МК. Данные передаются через регистр **PORTx**, а разрешение данных выдает регистр **DDRx** (Data Direction Register) (красная линия на рисунке 3).

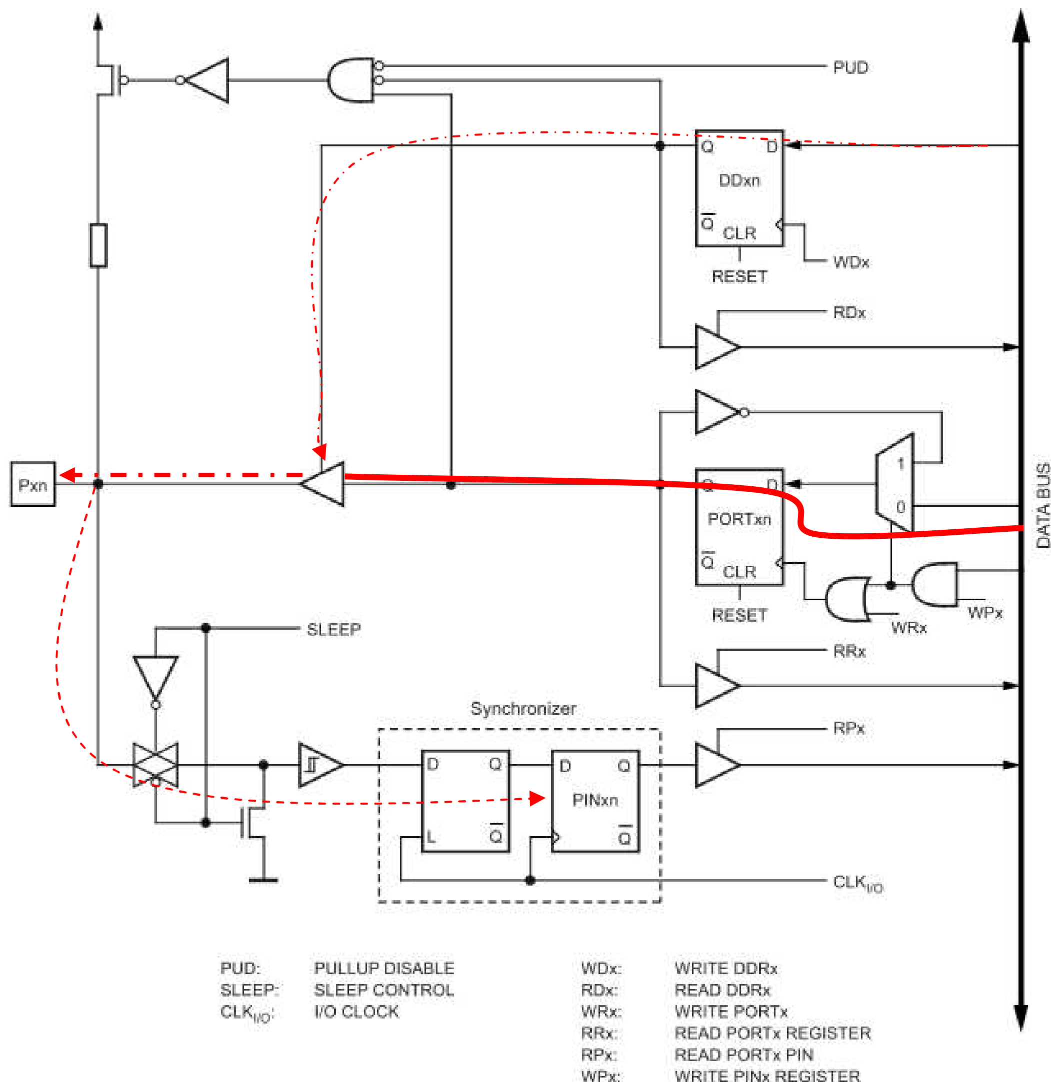

Рисунок 3. Направленность выходных данных на вывод МК

Буквенный постфикс n обозначает имя порта. Регистры управления имеют такой же постфикс, т.е., порт PORT A обслуживается регистрами PORTA и DDRA, порт PORT B — регистрами PORTB и DDRB и т.д.

Если нужно подать на ножку PB1 логический сигнал, нужно сначала разрешить выдачу данных записав «1» в бит 1 регистра DDRB, а потом записать нужный сигнал – «0» или «1» в бит 1 регистра PORTB=0b00000010.

DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	**DDRB1**	DDRB0
0	0	0	0	0	0	**1**	0
PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	**PORTB1**	PORTB0
1/0	1/0	1/0	1/0	1/0	1/0	**1/0**	1/0
PB7	PB6	PB5	PB4	PB3	PB2	**PB1**	PB0
0	0	0	0	0	0	**1/0**	0

Нагляднее всего рассматривать работу портов ввода вывода на примере работы светодиода и тактовой кнопки. Схема приведена на рисунке 4. Принципиальная электрическая схема приведена на рисунке 5

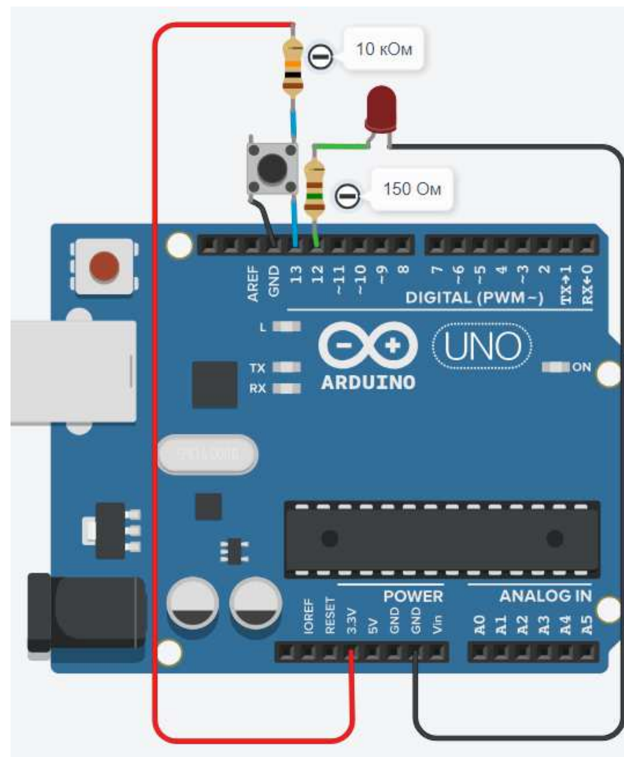

Рисунок 4. Схема подключения светодиода и тактовой кнопки

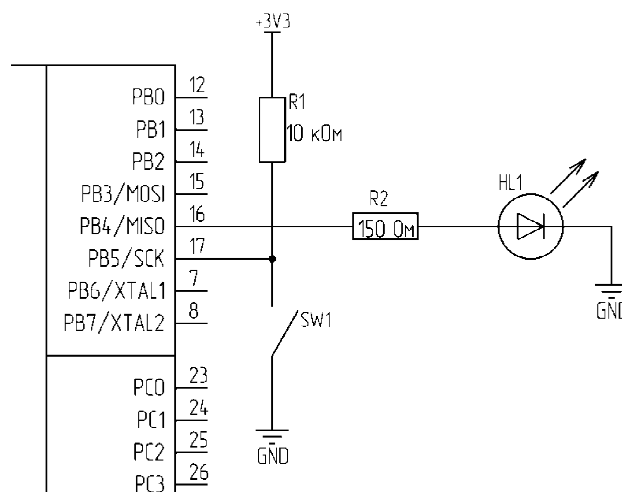

Рисунок 5 Принципиальная электрическая схема подключения светодиода и тактовой кнопки

Согласно принципиальной схеме платы (рисунок 6), 13ая ножка разъема Arduino UNO это **PB5**, а 12ая - **PB4**.

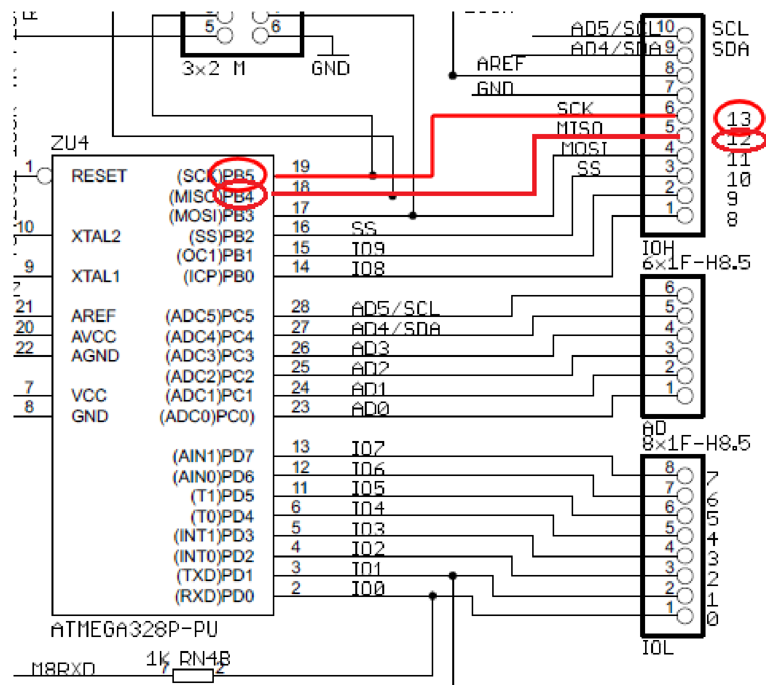

Рисунок 6. Принципиальная схема разъемов платы Arduino UNO

Код программы:

```
#define LED PB4     //присваваем выводу PB4 микроконтроллера имя
"LED"
#define BUTTON PB5 //присваваем выводу PB5 микроконтроллера имя
"BUTTON"
int main(void)
{
Serial.begin(115200);// начало работы последовательного порта
//полезная штука при отладке, здесь не используется

DDRB = 1<<DDB4);//
//ножку PB4 настраиваем на выход
while (1) //бесконечный цикл
{
  if (PINB & (1 << BUTTON))   // Если кнопка отпущена
  {
      PORTB &= ~(1 << LED); // то Выключаем светодиод на PB4
  }
  else // В другом случае(если кнопка нажата)
  {
      PORTB |= (1 << LED); // включаем светодиод на PB4
  }
}
}
```

Описание работы программы:

В блоке инициализации (задания условий работы МК) вывод МК PB4 настраивается на выдачу сигнала, путем записи логической «1» в бит **DDRB4** регистра направленности данных порта B **DDRB.**

В основном блоке программы зациклена следующая функция: если на вывод МК **PB5** путем нажатия кнопки поступает логический «0», то вывод **PB4** инвертирует сигнал, зажигая при этом светодиод. В любом другом случае (пока кнопка не нажата) на выводе **PB4** устанавливается логический «0» и светодиод не светится.

Код программы написан для использования компилятора со встроенным заголовочным файлом **avr/io.h,** т.е. для среды обработки кода известны названия всех регистров МК, а так же названия всех битов каждого регистра.

Ниже описаны основные операции, примененные в коде.

Операция присваивания

В строке

```
DDRB = (1<<DDB4);
```

используется знак «=», т.е. выполняется операция присваивания. При таком действии, все остальные биты регистра по умолчанию принимают значение «0».

	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0	
	1/0	1/0	**1/0**	**1/0**	1/0	1/0	1/0	1/0	
=	0	0	**1**	**1**	0	0	0	0	Присваиваемое значение
DDRB	0	0	**1**	**1**	0	0	0	0	Результат

Данная операция полезна при проведении инициализации МК, т.е. устанавливая нужные биты регистра мы автоматически присваиваем остальным битам значение «0», даже если в них было записано «1».

Логическое «ИЛИ»

Символ « | » обозначает логическое действие «ИЛИ». Так, например в строке

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

используется составной оператор « |= ». В данном случае мы тоже обращаемся к отдельному биту регистра данных **PORTB**, но производим операцию логического «ИЛИ». Результат операции логического «ИЛИ» принимает значение «1», если хотя бы один из операндов равен 1. Таблица истинности описывается следующим образом:

0 | 0 == 0

0 | 1 == 1

1 | 0 == 1

1 | 1 == 1

В нашем случае данный оператор применяется для того, что бы **не изменить значение остальных** битов регистра PORTB при изменении значения нужных нам битов. **ВАЖНО!** Маска принимает **не** значение бита, а его разряд. Что бы изменить значение бита, нужно провести логическую операцию нужного регистра с созданной битовой маской.

	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	
	1/0	1/0	**1/0**	**1/0**	1/0	1/0	1/0	1/0	
|=	0	0	**1**	**1**	0	0	0	0	Созданная маска
PORTB	1/0	1/0	**1**	**1**	1/0	1/0	1/0	1/0	Результат

Операция сдвига

В языке С, для микроконтроллеров или любых других платформ, битовая операция присвоения логической "1" в основном используется с помощью операции сдвига.

Это связано с тем, что битовые операции позволяют более гибко работать с битами в памяти, а сдвиг бит влево (с помощью `<<`) это метод установки определенного бита (установка бита в "1"). Кроме того, операции сдвига бит влево и вправо имеют низкую стоимость в терминах аппаратного обеспечения для архитектур без аппаратного умножителя, что позволяет выполнять эти операции быстрее и эффективнее в сравнении с присваиванием числа. Битовый сдвиг делает не что иное, как умножает или делит байт на 2 в степени. Данная операция удобна из-за особенностей архитектуры МК, на её выполнение уходит крайне мало времени, а именно – один такт процессора. Ниже описана работа оператора сдвига.

```
1 << 0 == 0b000000001        // сдвигаем число 1 влево 0 раз
1 << 1 == 0b000000010        // сдвигаем число 1 влево 1 раз
1 << 2 == 0b000000100        // и т.д.
1 << 3 == 0b000001000
...
1 << 8 == 0b100000000
```

В конечном итоге, использование операции сдвига для установки значений битов более эффективно и является хорошей практикой при программировании для микроконтроллеров в языке С.

Рассмотрим другой пример сдвига влево:

```
0b00000101 << 0 == 0b00000101   //начальное значение
0b00000101 << 1 == 0b00001010   //сдвигаем число влево 1 раз
0b00000101 << 2 == 0b00010100   //сдвигаем число влево 2 раза
0b00000101 << 3 == 0b00101000   // и т.д.
0b00000101 << 4 == 0b01010000
0b00000101 << 5 == 0b10100000
0b00000101 << 6 == 0b01000000
0b00000101 << 7 == 0b10000000
0b00000101 << 8 == 0b00000000
```

ВАЖНО! Если при сдвиге биты уходят за пределы регистра, их значение теряется.

В примере выше, в строке

```
DDRB = (1<<DDB4);
```

бит DDB4 (четвертый бит регистра DDRB) сдвигается на один бит влево, а т.к. DDB4 по умолчанию равен «0», а бит – одноразрядное двоичное число, то выполняется выражение

DDRB =(1<< DDB4)== 0b00010000 ➔ DDRB =(1 << 4) == 0b00010000

Таким образом бит DDB4 принимает значение «1».

Логическое «И»

В строке условия

```
if (PINB & (1 << BUTTON))
```

применяется оператор «&». Символ «&» обозначает логическую операцию «И». Результат операции логического «И» принимает значение «1», если **все** операнды равны «1». Таблица истинности описывается следующим образом:

0 & 0 == 0

0 & 1 == 0

1 & 0 == 0

1 & 1 == 1

В нашем случае данный оператор применяется для проверки условия нажатия кнопки. По умолчанию на вывод МК PB5, к которому подключена кнопка, подано напряжение питания через резистор 10 кОм, т.е. все время установлена логическая «1». При нажатии кнопки потенциал на выводе PB5 будет равен «0».

Условие `if (PINB & (1 << BUTTON))` выполнится (будет истинным), когда бит, соответствующий биту BUTTON (PB5), в регистре PINB (PINB5) будет установлен в «1».

Давайте рассмотрим эту конструкцию по шагам:

1. Операция (1 << BUTTON) создает маску, перемещая число 1 влево на позицию, соответствующую разряду биту PINB5 т.е. **5**. В данном примере BUTTON==5. Особенностью побитового обращения в языке C является то, что при считывании данных из бита результатом будет **не «0» или «1», а разряд бита.** В нашем случае BUTTON==5, а маска будет иметь вид 0b00**1**00000.

2. Операция PINB & (1 << BUTTON) выполняет побитовое И (&) между значением регистра PINB и созданной маской. Это позволяет проверить только нужный бит, соответствующий переменной BUTTON.

	7	6	5	4	3	2	1	0	
	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	
	1/0	1/0	**1/0**	1/0	1/0	1/0	1/0	1/0	
&	0	0	**1**	0	0	0	0	0	Созданная маска
Результат условия	0	0	**1/0**	0	0	0	0	0	Результат

3. Если результат логического «И» (`PINB & (1 << BUTTON)`) не равен нулю, то условие if будет выполнено, поскольку нужный бит в регистре PINB установлен в «1».

Логическое НЕ

В строке PORTB &= ~(**1** << LED) для подачи логического «0» на вывод PB4 используется составной оператор «&= ~ ». Символ «~» обозначает логическое действие «НЕ» и она инвертирует исходное значение. Таблица истинности приведена ниже

~0 == 1

~1 == 0

Давайте рассмотрим подробнее, как работает эта строка кода

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

1. Операция (1 << LED) создает маску, перемещая число 1 влево на позицию, соответствующую разряду биту PORTB4, т.е. 4 или в двоичной система 0b000**1**0000.

2. Операция ~(1 << LED) инвертирует эту маску, устанавливая все биты, кроме бита LED, в "1", в двоичной системе полученная маска будет равна 0b111**0**1111.

3. Далее, оператор побитового И (`&`) применяется к регистру PORTB и инвертированной маске. Это позволяет сохранить в PORTB только те биты, которые необходимо изменить, а остальные биты оставить без изменений.

	7	6	5	4	3	2	1	0	
	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	
	1/0	1/0	1/0	**1**	1/0	1/0	1/0	1/0	
&	1	1	1	**0**	1	1	1	1	Маска
PORTB	1/0	1/0	1/0	**0**	1/0	1/0	1/0	1/0	Результат

Таким образом, выполнение этой строки приведет к тому, что бит, соответствующий пину LED, будет сброшен в "0", **сохраняя остальные биты без изменений.**

Варианты заданий:

№	Задание
1.	Светодиод подключен к PC5 и по умолчанию светится. Пока нажата кнопка, подключенная к PD7, светодиод должен быть погашен.
2.	Светодиод подключен к PB0 и по умолчанию погашен. Пока нажата кнопка, подключенная к PB1, светодиод должен светиться.
3.	Первый светодиод подключен к PD6 и по умолчанию погашен. Второй светодиод подключен к PD7 и по умолчанию светится. Пока нажата кнопка, подключенная к PC2, оба светодиода меняют свое состояние.
4.	Два светодиода подключены к PD2 и PD3 и по умолчанию погашены. Пока нажата одна из кнопок, подключенных к PB0 и PB1 соответственно, светодиоды на PD2 и PD3 должны светиться. Каждая кнопка отвечает за свой светодиод.
5.	Светодиод подключен к PB4. Кнопки подключены к PC1 и PC0. Если нажаты обе кнопки одновременно – светодиод должен светиться.
6. *	Повторить результат программы, приведенной в примере, исключив из схемы внешнюю подтяжку (резистор R1)
7. *	Семисегментный индикатор подключен к порту D. Три переключателя подключены к PB3…PB5 и представляют собой трехразрядное двоичное число. Выводить это число на семисегментный индикатор в десятичном виде.
8. *	Организовать бегущие огни на семисегментном индикаторе, в которых каждый шаг происходит по нажатию кнопки.
9. *	Выводить на семисегментный индикатор цифру с клавиатуры 4x4. Допускается кнопку и светодиодный индикатор подключать к произвольным выводам МК.
* - задачи повышенной сложности	

Лабораторная работа № 2.
Внешние прерывания микроконтроллеров.

Цель: изучить принцип работы внешних прерываний микроконтроллеров.

Краткие теоретические сведения

Прерывание - инициируемый определенным образом процесс, временно переключающий микроконтроллер на выполнение другой программы **с последующим возобновлением** прерванной программы. Т.е. МК «всё бросает», и начинает выполнять функцию, написанную в обработчике прерываний. Однако после этого он возвращается именно в тот блок кода, на котором закончил работу в момент вызова прерываний.

Источников прерываний у МК **несколько** (подробнее в разделе 11), в рамках данной лабораторной работы будут рассмотрены прерывания от **внешних источников**, таких как кнопка, переключатель, логический сигнал другого устройства , т.е. есть то, что может подключаться к внешним выводам МК.

Суть в том, что системное ядро микроконтроллера не занимается опросом состояния внешнего вывода и не тратит на это время. Но как только на внешнем выводе изменяется логический уровень, МК немедленно обрабатывает прерывание и возвращается к работе.

Пример: нужно датчиком освещенности измерить длительность вспышки молнии, параллельно с этим непрерывно выдавая на LCD дисплей степень освещенности окружающей среды, температуру воздуха и атмосферное давление в реальном времени. Процедура выдачи данных о состоянии окружающей среды занимает достаточно долгое время, т.к. данные передаются последовательно. При опросе датчика освещенности внутри основного кода можно просто пропустить короткий импульс света молнии. Однако если организовать опрос через прерывание, длительность импульса молнии будет определяться **параллельно** основному коду программы аппаратными средствами МК, не затрагивая при этом основной код программы выдачи данных на LCD. При настроенном источнике прерываний МК сам определит момент начала и конца длительности импульса.

Прерывание необходимо использовать в тех случаях, когда неизвестен момент, изменения сигнала на ножке МК, а его нужно отслеживать постоянно и реагировать на него незамедлительно.

Так же, внешние прерывания используются для вывода МК из режима пониженного энергопотребления (режим сна). В этом режиме практически вся внешняя периферия отключена, и вывести его из этого режима можно лишь одним из прерываний, в том числе и **внешним**.

У Atmega328P есть 2 типа внешних прерываний: прерывания по изменению уровня конкретного вывода и прерывание по изменению состояния любого из группы выводов.

Внешнее прерывание INTn

МК Atmega328P имеет два входа внешнего прерывания (в других моделях может быть другое количество). **Входы INT0 и INT1 это выводы PD2 и PD3** соответственно (таблица 13.3.3 даташита). Обработчик прерываний может начинать работу как по **фронту** импульса (сразу как нажали кнопку), так и по **спаду** (только когда отпустили кнопку). Так же программа обработчика прерываний может выполняться до тех пор, пока на входах прерывания **определенное значение**. Режим работы обработчика прерываний **INTn** определяется следующими регистрами (данные из даташита, раздел 12):

SREG – AVR Status Register

Регистр состояний МК. Более подробно в 6.3.1. даташита.

В нашем случае – нас интересует только 1 бит данного регистра – **«I».** Установка этого бита разрешает **глобальные прерывания**, т.е. прерывания всех источников **можно будет разрешить**. Если его не установить в «1», то даже разрешив прерывания любых источников, они **не будут срабатывать.**

Глобальные прерывания так же разрешаются командой «sei()»;

EICRA – External Interrupt Control Register A

Bit	7	6	5	4	3	2	1	0
	-	-	-	-	**ISC11**	**ISC10**	**ISC01**	**ISC10**
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W

Биты 7…4 - не используются, всегда равны «0»

Биты 3, 2 – ISC11, ISC10 (Interrupt Sense Control)

В зависимости от установки этих битов – определяется режим работы обработчика внешнего прерывания.

ISC11	ISC10	Режим работы
0	0	**Логический «0»** на входе INT1 вызывает прерывание
0	1	**Любое изменение** логического уровня на входе INT1 вызывает прерывание
1	0	Прерывание вызывается по **спаду** логической «1» на входе INT1 (при переходе из «1» в «0»)
1	1	Прерывание вызывается по **фронту** логической «1» на входе INT1 (при переходе из «0» в «1»)

Такой же принцип соблюдается и для ввода **INT0** и битов **ISC01, ISC00.**

EIMSK – External Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0
	-	-	-	-	-	-	**INT1**	**INT0**
Read/Write	R	R	R	R	R	R	R/W	R/W

Биты 7…2 - не используются, всегда равны «0»

Биты INT0, INT1 разрешают прерывания (если разрешены глобальные прерывания), по источнику заданному в регистре **EICRA.** Если бит равен «0», то прерывания запрещены, если «1», то разрешены.

EIFR – External Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0
	-	-	-	-	-	-	**INTF1**	**INTF0**
Read/Write	R	R	R	R	R	R	R/W	R/W

Биты 7…2 - не используются, всегда равны «0»

INTF1, INTF0 – флаги состояний **прерывания.** Устанавливаются в «1», **если прерывание было вызвано**, и устанавливаются обратно в «0», при выходе из прерывания. Данные флаги можно устанавливать вручную, например, для отладки области кода. Для

данного МК эти биты будут всегда в «0», если выбран режим прерывания при определенном «0».

Рассматривать работу прерываний удобно на примере работы светодиода и переключателей. Схема приведена на рисунке 7.

Принципиальная электрическая схема приведена на рисунке 8.

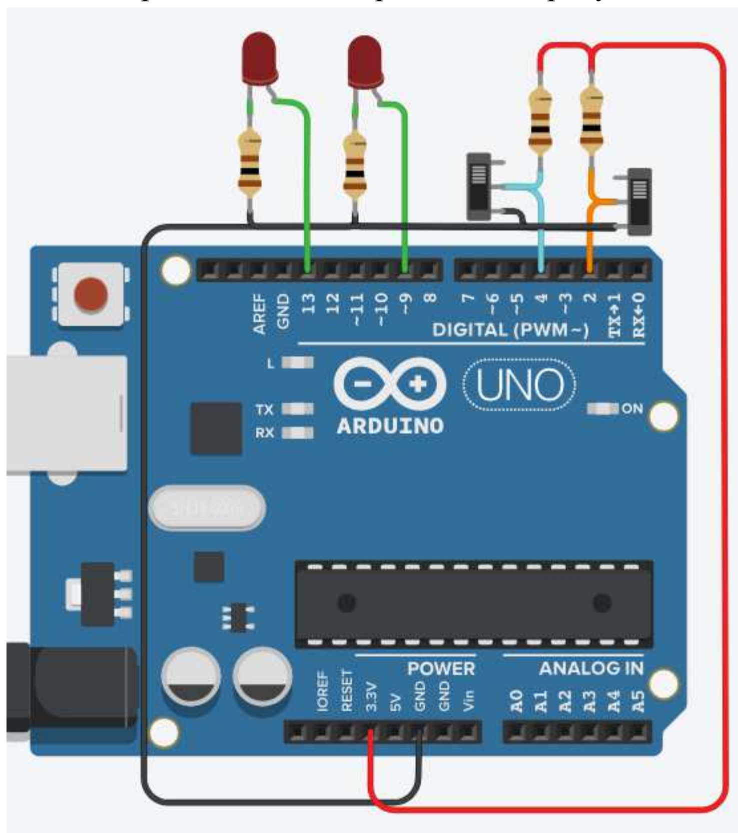

Рисунок 7. Схема подключения светодиодов и переключателей к выводам GPIO и INT0

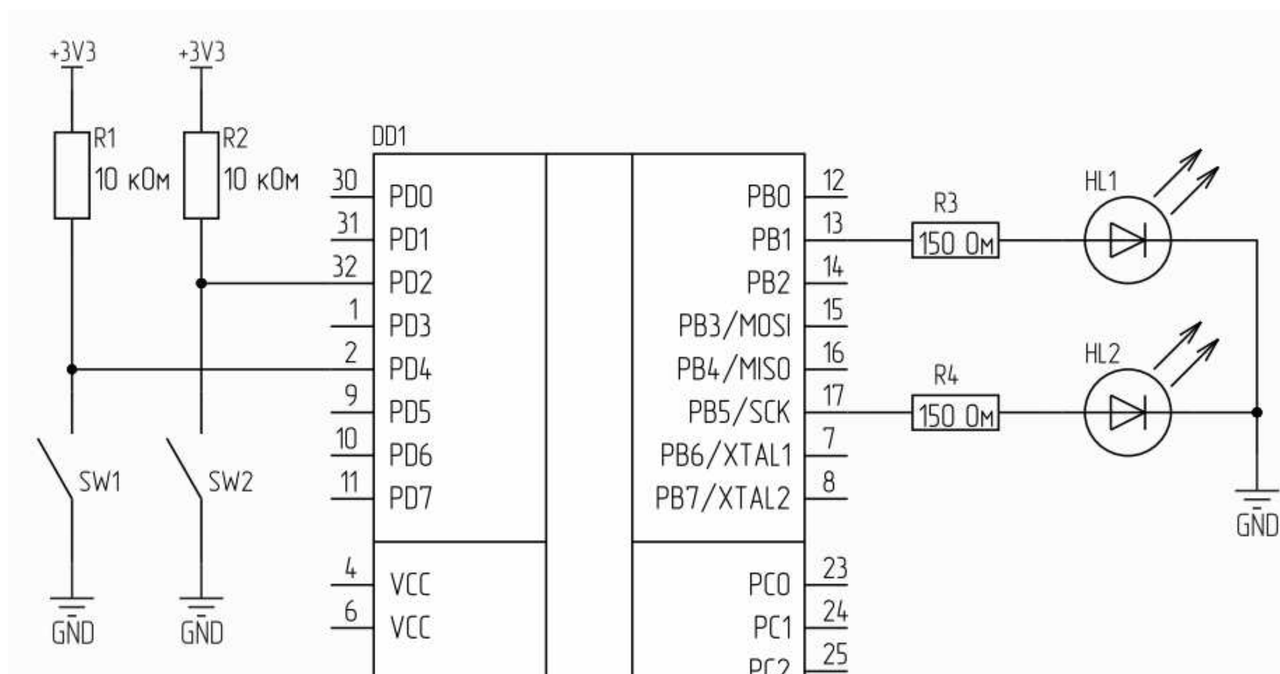

Рисунок 8 Принципиальная электрическая схема подключения переключателей и светодиодов к выводам GPIO и выводу INT0

Код программы:

```
#define LED_PIN PB1
#define BUTTON_PIN PD2
#define BUTTON PD4
#define LED PB5
void init_interrupt()
{// инициализация прерывания
    // Настройка внешнего прерывания INT0
    EICRA |= (1 << ISC00);       // Прерывание по смене уровня
    EIMSK |= (1 << INT0); // Разрешение прерывания INT0
}
void init_gpio()
{//инициализация GPIO
    DDRB |= (1 << LED_PIN)|(1 << LED);// Установка выводов LED MK
на выход
Serial.begin(9200);                //старт монитора для отладки
}

int main(void)
{
    init_interrupt();  // Инициализация прерывания
    init_gpio();       // Инициализация GPIO

    sei();  // Разрешение глобальных прерываний

    while (1)
    {// основной код программы
    Serial.println("start if");//пишем в монитор пользователя о
начале _delay_ms(1000000);       //искусственная задержка,
имитирующая длительное выполнение кода
    if (PIND & (1 << BUTTON))  // Если переключатель выключен
  {
      PORTB &= ~(1 << LED);     // то ВЫключаем светодиод на PB4
HL2
  }
  else // В другом случае(если включен)
  {
      PORTB |= (1 << LED);      // включаем светодиод на PB4 HL2
  }
      Serial.println("end if"); пишем в монитор пользователя о
завершении
}
}
// Обработчик прерывания по нажатию кнопки
ISR(INT0_vect)
{
```

```
    PORTB ^= (1 << LED_PIN); } // Инверсия состояния светодиода
HL1
// этот код выполняется немедленно при срабатывании прерывания, в
нашем случае – когда происходит смена уровня сигнала на ножке
INT0.
```

Описание работы программы:

Переключатели подключены на выводы PORTD. В блоке инициализации выводы МК PB1, PB5 настраивается на выдачу сигнала, путем записи логической «1» в соответствующие биты регистра направленности данных порта B **DDRB.** Вывод PD2 сконфигурирован как источник внешнего прерывания INT0. Так же **разрешаются** прерывание при смене уровня сигнала от «0» до «1» и от «1» до «0» на выводе **INT0 (PD2)**.

В основном блоке программы зациклена следующая функция: если на вывод МК **PD4** путем включения переключателя **SW1** поступает логический «0», то светодиод **HL2**, подключенный к **PB5** начинает светиться. В любом другом случае на выводе **PB5** устанавливается логический «0» и светодиод **HL2** не светится.

Т.е. Если включить **SW1**, то светодиод HL2 засветиться с существенной задержкой, что соответствует нормальной работе кода с искусственной задержкой

Так же, в монитор последовательных данных до начала опроса кнопки передаются строки ("start if") и ("end if").

В обработчике прерываний ISR(INT0_vect) описаны действия, выполняемые при прерывании. Прерывание сработает **немедленно** при изменении уровня на выводе **INT0 (PD2),** не смотря на искусственную задержку в основном коде программ, т.к. изменение состояния светодиода описано **не в основном коде программы, а в блоке обработки прерывания**.

При прерывании инвертируется состояние **HL1** изменением сигнала на выводе **PB1.** Обратите внимание, что постоянный опрос кнопки не требуется, т.к. в блоке инициализации мы уже описали алгоритм действий МК при переключении **SW2**, путем записи нужных бит в регистры управления.

Внешнее прерывание PCINTn (Pin Change INTerrupt)

Принцип работы данного прерывания схож с прерыванием **INTn,** но в отличии от **INTn,** данное прерывание срабатывает когда один из группы выводов МК **PCINTn** меняет свое состояние. Выводом **PCINTn** можно настроить практически любой вывод МК, подробнее в разделах (13.3.1… 13.3.3 даташита).

Обратите внимание, что оба источника внешнего прерывания могут работать как от внешнего сигнала, так и от **внутреннего**. Например, если регистр направленности данных DDRD настроен на выдачу данных, то прерывание **INT0** сработает, если МК выдаст сигнал на ножку **PD2 (INT0).** Это может быть полезным при создании собственных программных прерываний.

Режим работы обработчика прерываний **PCINTn** определяется следующими регистрами (данные из даташита, раздел 12):

PCICR – Pin Change Interrupt Control Register

Bit	7	6	5	4	3	2	1	0
	-	-	-	-	**-**	**PCIE2**	**PCIE1**	**PCIE0**
Read/Write	R	R	R	R	R	R/W	R/W	R/W

Биты 7…3 - не используются, всегда равны «0».

Бит 2 - PCIE2: Pin Change Interrupt Enable 2 разрешает прерывание **PCINT2**, если разрешены глобальные прерывания. Источником прерывание является изменение состояния хотя бы одной на одном из выводов МК **PCINT23..16.** Если бит равен «0», то прерывания запрещены, если «1», то разрешены.

Бит 1 – PCIE1: Pin Change Interrupt Enable 2 разрешает прерывание **PCINT1**, если разрешены глобальные прерывания. Источником прерывание является изменение состояния хотя бы одной на одном из выводов МК **PCINT14..8.** Если бит равен «0», то прерывания запрещены, если «1», то разрешены.

Бит 0 – PCIE0: Pin Change Interrupt Enable 0 разрешает прерывание **PCINT0**, если разрешены глобальные прерывания. Источником прерывание является изменение состояния хотя бы одной на одном из выводов МК **PCINT7..0.** Если бит равен «0», то прерывания запрещены, если «1», то разрешены.

PCIFR – Pin Change Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0
	-	-	-	-	**-**	**PCIF2**	**PCIF1**	**PCIF0**
Read/Write	R	R	R	R	R	R/W	R/W	R/W

Биты 7…3 - не используются, всегда равны «0».

PCIF2…PCIF0– флаги состояний **прерывания.** Устанавливаются в «1», **если соответствующее прерывание было вызвано**, и устанавливаются обратно в «0», при выходе из прерывания. Данные флаги можно устанавливать вручную, например, для отладки области кода.

PCMSK2 – Pin Change Mask Register

Bit	7	6	5	4	3	2	1	0
	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Каждый бит регистра соответствует своему выводу МК. Если бит установлен в «1», прерывания **PCINT2** по соответствующему выводу разрешены. Если бит установлен в «0», прерывания запрещены

PCMSK1 – Pin Change Mask Register 1

Bit	**7**	6	5	4	3	2	1	0
	-	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8
Read/Write	**R**	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Каждый бит регистра соответствует своему выводу МК. Если бит установлен в «1», прерывания **PCINT1** по соответствующему выводу разрешены. Если бит установлен в «0», прерывании я запрещены. Обратите внимание, что бит 7 – пустой, т.к. у данного МК ножка **PCINT15** отсутствует.

PCMSK0 – Pin Change Mask Register 0

Bit	7	6	5	4	3	2	1	0
	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Каждый бит регистра соответствует своему выводу МК. Если бит установлен в «1», прерывания **PCINT0** по соответствующему выводу разрешены.

Рассмотрим работу прерываний PCINT на примере работы светодиода, звукового излучателя и тактильных переключателей. Схема приведена на рисунке 9.

Принципиальная электрическая схема приведена на рисунке. 10.

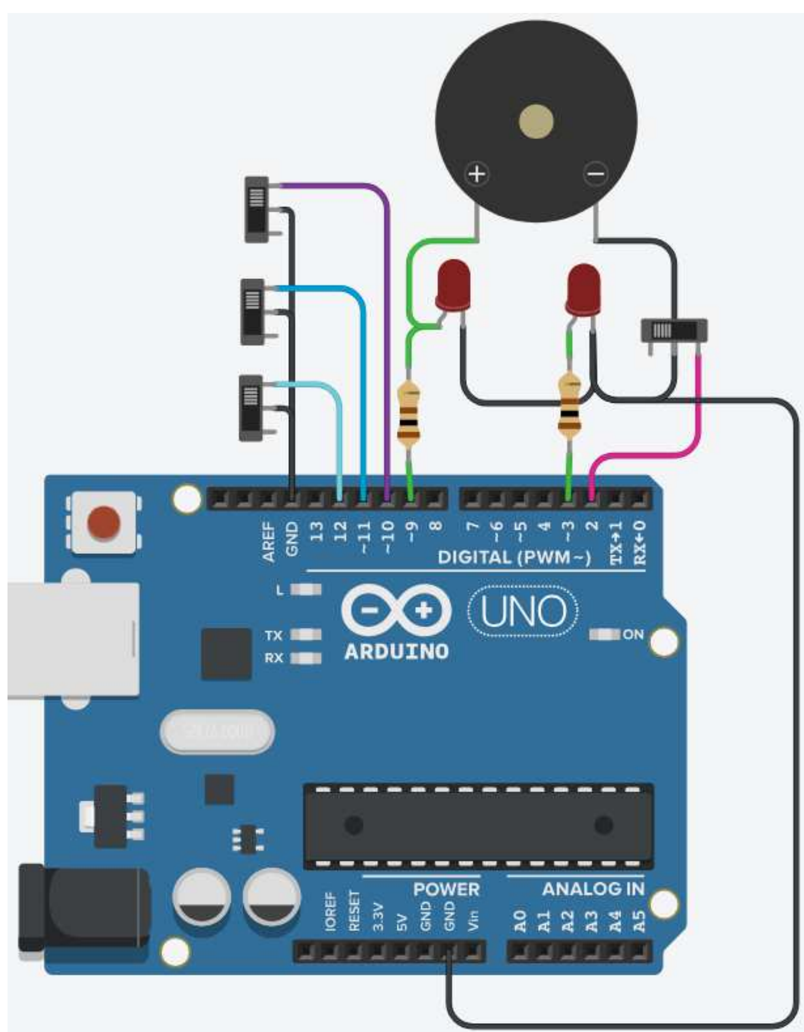

Рисунок 9 Схема подключения светодиодов, звукоизлучателя и переключателей к выводам GPIO, INT0 и PCINT.

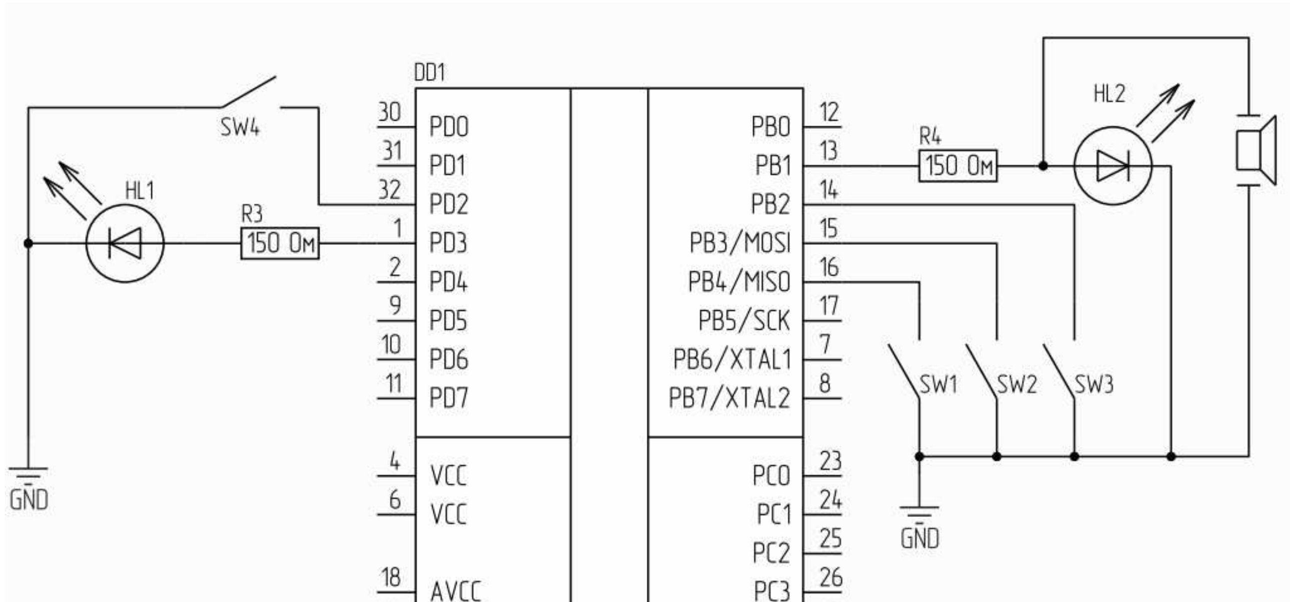

Рисунок. 10. Принципиальная схема подключения светодиодов, звукоизлучателя и переключателей к выводам GPIO, INT0 и PCINT.

Код программы:

```
#define LED_PIN PD3
#define LED1_PIN_PCINT0 PB1
#define BUTTON_PIN PD2

void init_interrupt()     //инициализация прерываний
{
    // Настройка внешнего прерывания INT0
    EICRA |= (1 << ISC00);      // Прерывание по смене уровня INT0
    EIMSK |= (1 << INT0);    // Разрешение прерывания INT0
      PCICR = 0b00000001;       // /Разрешаем прерывания PCINT0
      PCMSK0 =(1 << PB2)|(1 << PB3)|(1 << PB4);//указываем ножки, 
от которых
//будет срабатывать прерывание
}

void init_gpio()    //инициализация GPIO
{
    DDRD |= (1 << LED_PIN);      // Установка выводов светодиода на 
выход
    PORTB &= (1 << LED1_PIN_PCINT0);   // Установка светодиода в 
выключенное состояние
    PORTB |=(1 << PB2)|(1 << PB3)|(1 << PB4);
    DDRB|=(1 << LED1_PIN_PCINT0);//Установка выводов PCINT на 
выход
    DDRD &= ~(1 << BUTTON_PIN); // Установка вывода кнопки на вход
    PORTD |= (1 << BUTTON_PIN);

Serial.begin(115200);                //начало работы 
последовательного интерфейса
```

```
}

int main(void)
{
    init_interrupt();  // Начало инициализация прерывания
    init_gpio();       // Начало инициализация GPIO

    sei();  // Разрешение глобальных прерываний

    while (1)
    {
        if (PINB & (1 << PB2)) {  // Проверяем статус
переключателя 1
              Serial.println("ALARM1 ON");//выводим строку
        }
        if (PINB & (1 << PB3)) {  // Проверяем статус
переключателя 2
               Serial.println("ALARM2 ON");//выводим строку
        }
        if (PINB & (1 << PB4)) {  // Проверяем статус
переключателя 3
               Serial.println("ALARM3 ON");//выводим строку
        }

        _delay_ms(1000);  // Ждем 1 секунду, что бы не спамить
    }
}

ISR(INT0_vect)     // Обработчик прерывания INT0
{

    PORTD ^= (1 << LED_PIN);  // Инверсия состояния светодиода HL1

}

ISR(PCINT0_vect)   // Обработчик прерывания PCINT0
{

   PORTB |= (1 << LED1_PIN_PCINT0);  // Инверсия состояния
светодиода HL2
        if (((~PINB) & (1 << PB2)) && ((~PINB) & (1 << PB3))  &&
((~PINB) & (1 << PB4)))
//если все переключатели выключены
   PORTB &= ~ (1 << LED1_PIN_PCINT0); //то выключаем светодиод HL2
принудительно
}
```

Описание работы программы:

Данная программа имитирует работу сигнализации при срабатывании трех датчиков. Если один из переключателей **SW1…SW3** включится, имитируя срабатывание датчика сигнализации, то сработает звуковая и световая сигнализация (звуковой излучатель и светодиод HL2). Включение светодиода и сигнализации реализовано в обработчике прерываний **PCINT0**, в то время как в основном коде программы выполняется вывод через интерфейс последовательной передачи данных информации о включенных датчиках сигнализации.

Параллельно с этими действиями, **без дополнительного опроса** состояния выводов светодиод HL1 зажигается при срабатывании **SW4.**

В строке

```
if (((~PINB) & (1 << PB2)) && ((~PINB) & (1 << PB3))  && ((~PINB)
& (1 << PB4)))
```

используется оператор **«&&»** - логическое И. Отличие от битового И «&» в том, что битовое И выполняет операцию с битами и выдает конечный результат в численном виде, а логическое И устанавливает значение «true», если оба операнда равны «1», а в противном случае возвращает значение «false».

В рамках данной лабораторной работы рассматривались источники **внешнего** прерывания. Источники **внутренних** прерываний МК ATmega328P описаны в таблице **11.1 Interrupt Vectors in ATmega328P** даташита.

Варианты заданий:

№	Задание
1.	Светодиод подключен к PC5 и по умолчанию светится. Пока нажата кнопка, подключенная к PCINT7, светодиод должен быть погашен.
2.	Светодиод подключен к PB1 и по умолчанию погашен. Пока нажата кнопка, подключенная к INT1, светодиод должен светиться.
3.	Первый светодиод подключен к PD6 и по умолчанию погашен. Второй светодиод подключен к PD7 и по умолчанию светится. Пока включен переключатель, подключенный к PCINT14, оба светодиода меняют свое состояние.
4.	Два светодиода подключены к PD3 и PD4 и по умолчанию погашены. Пока включен один из переключателей, подключенных к PCINT7 и PCINT8 соответственно, светодиоды на PD3 и PD4 должны светиться. Каждый переключатель отвечает за свой светодиод.
5.	Светодиод подключен к PD4 и по умолчанию погашен. Переключатели подключены к PCINT12 и INT0. Если нажаты оба переключателя одновременно – светодиод должен светиться.
6. *	Организовать переключение состояния светодиода по нажатию кнопки. Допускается кнопку и светодиод подключать к произвольным выводам МК.
7. *	Организовать бегущие огни на 4ёх светодиодах. При нажатии одной из трех кнопок, светящиеся светодиоды должны смещаться влево. Допускается кнопки и светодиоды подключать к произвольным выводам МК.
8. *	Организовать бегущие огни на семисегментном индикаторе, в которых каждый шаг происходит по нажатию кнопки. Допускается кнопки и индикатор подключать к произвольным выводам МК.
* - задачи повышенной сложности	

ПРИМЕЧАНИЕ – в основном коде программы **ОБЯЗАТЕЛЬНО** ввести следующие строки

```
pinMode(LED_BUILTIN, OUTPUT);          //данную строку можно перенести в блок инициализации
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
  _delay_ms(1000);                       // wait for a second
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
  _delay_ms(1000);                       // wait for a second
```

Данные строки вводятся для наглядности применения прерывания при выполнении кода программы.

Лабораторная работа № 3.
Работа с таймерами-счётчиками AVR.

Цель: изучить принцип работы таймерами-счётчиками микроконтроллеров.

Краткие теоретические сведения

Таймер счётчик в микропроцессорах – устройство, которое считает количество импульсов опорной частоты и выполняет определённые действия при достижении конкретного значения счёта.

Таймер-счетчик является одним из самых ходовых ресурсов микроконтроллера. Таймеры-счетчики могут выполнять ряд дополнительных функций, например формирование ШИМ сигналов, подсчет длительности и количества входящих импульсов. Для этого существуют специальные режимы работы таймера-счетчика.

Источником опорной частоты могут служить как внешние источники (генераторы тактовой частоты и резонаторы), так и внутренний RC контур. Широкое применение нашли здесь кварцевые резонаторы – при низкой стоимости погрешность частоты находится в пределах 50 ppm. Внутренний RC контур как источник опорной частоты используется реже, т.к. согласно разделу 28.5 при нормальных климатических условиях его погрешность составляет 2%.

Входную частоту на готовых отладочных платах можно определить по маркировке кварцевого резонатора. При работе с отдельно взятым МК данная частота указывается при программировании битов конфигурации через программатор.

Вне зависимости от режима работы, таймер **считает количество импульсов входной частоты.** Т.к. длительность импульса входной частоты МК заранее известна (коэффициент заполнения для импульсов частоты равен 0,5), посчитав количество импульсов, можно определить, сколько прошло времени в секундах.

Пример приведен на рисунке 11. Таймер счетчик в обычном случае считает количество «периодов», т.е. изменяет значение счетчика 1 раз в период $T_и$. Т.к.

$$f = \frac{1}{T}, \text{ равно как и } T = \frac{1}{f},$$

то для входной частоты 16 МГц получим время дискретизации

$$T_и = \frac{1}{f} = \frac{1}{16\text{МГц}} = 62{,}5 \text{ нс}$$

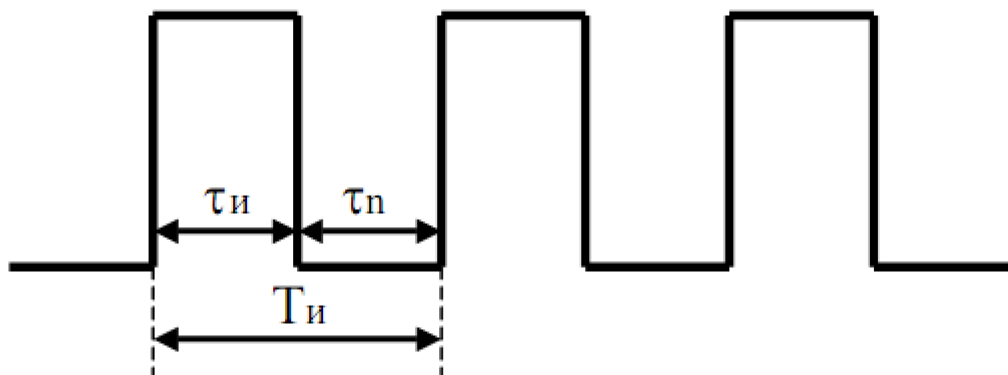

Рисунок 11. Форма и обозначение основных параметров входной частоты.

За время 62,5 нс свет пролетит всего лишь около 18 метров, а за это время данный МК может даже выполнить одну простейшую команду.

При прошествии одного периода таймер изменяет содержимое регистра счетчика на 1. Считывая значение регистра счетчика и умножив его на время дискретизации, мы получим **прошедшее время** с момента счёта.

МК Atmega328P имеет в составе 3 таймера-счётчика, которые отличаются друг от друга по конфигурации. Разные модели МК могут так же иметь разное количество таймеров и тем более разную конфигурацию, но основной принцип работы у них одинаковый, и описан выше.

Каждый таймер счетчик имеет несколько режимов работы

Рассмотрим подробнее **основные** режимы работ на примере восьмибитного таймера счетчика **Timer/Counter0.**

Normal mode

Простейший режим работы – счетный регистр (**TCNT0** – timer counter) увеличивает свое значение с течением времени. Регистр **TCNT0 -** восьмиразрядный, следовательно, считает он от 0 до 255. При достижении 255, счётчик переполняется и его значение снова становится 0. При этом флаг переполнения (бит **TOV0** регистра **TIFR0**) будет установлен в «1». Если разрешить прерывание, то действие, описанное в обработчике прерываний будет выполняться каждый раз **через одинаковый промежуток времени. Обратите внимание,** что содержимое счетчика можно изменить вручную, он продолжит считать с заданного вами значения, а при желании его можно обнулить.

Т.к. таймер считает по кругу от 0 до 255 и в последний момент может выполнить какое-либо действие. Это действие будет повторяться 1 раз в $256 * 62{,}5 \text{ нс} = \mathbf{15.938}$ мкс. Если мы хотим высокой точности, и выполнять действие например 1 раз в 10 мкс, то таймер должен посчитать

$$\frac{10\text{мкс}}{62{,}5\text{ нс}} = 160 \text{ раз}$$

Отслеживая значение регистра счёта **TCNT0=160 и обнуляя его в этот момент,** можно достичь равных промежутков времени в 10 мкс, но для более удобного достижения поставленной цели есть следующий режим работы.

Clear Timer on Compare Match (CTC) Mode

В данном режиме в регистр **OCR0A** (output compare register A) заносится значение, при котором счетчик **TCNT0** переполнится. Т.е. таймер будет считать не до 255, а до **OCR0A,** после чего вызовет прерывание и **обнулит TCNT0.** Это позволяет регулировать частоту до нужных значений на аппаратном уровне, не перегружая код программы лишним отслеживанием значения **TCNT0.** Т.е. по аналогии с предыдущим примером – если мы хотим повторить действие через 10 мкс, то выбрав режим **CTC** и записав **OCR0A=160** получится тот же результат.

Максимальная длительность отсчета в данном случае будет равняться $256 * 62{,}5 \text{ нс} = \mathbf{15.938}$ мкс. Для организации более длительных промежутков времени в структуре таймера предусмотрен **предделитель** частоты (**prescaler**). Работа предделителя – считывать каждый N-ый импульс входной частоты и только тогда увеличивать содержимое регистра счетчика. Счёт таймера для значения предделителя **8** приведен на рисунке 12 (данные из раздела 14).

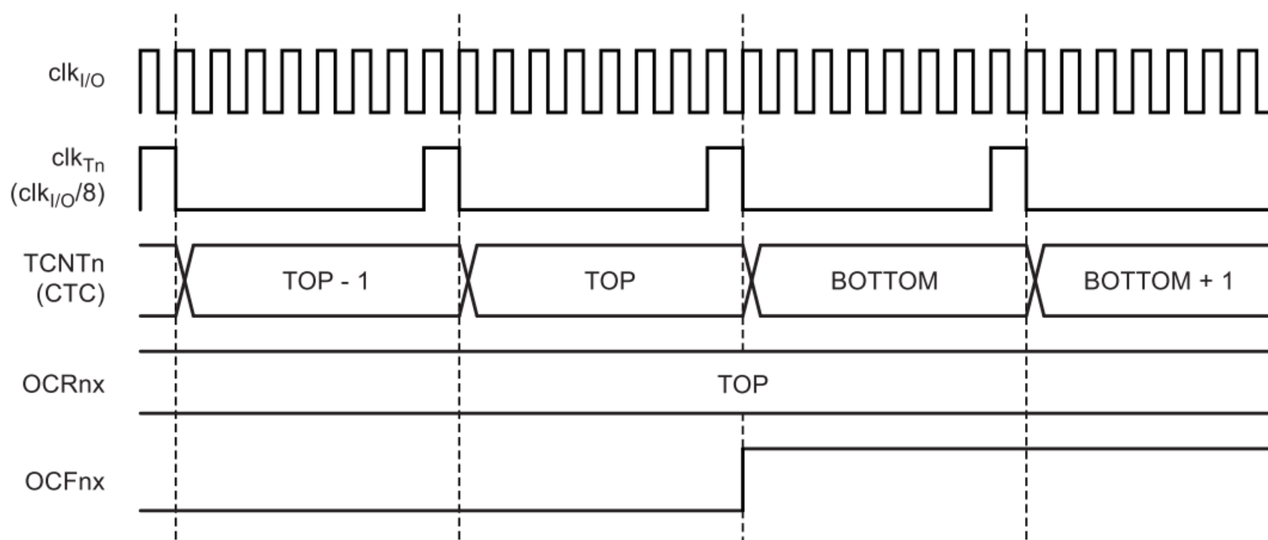

Рисунок 12. Работа таймера счетчика при значении предделителя 8.

Из данного рисунка видно, что предделитель считывает каждый восьмой импульс тактовой частоты. Как только таймер досчитал до значения **TOP**, записанного в регистре **OCRnx**, его счетный регистр **TCNTn** сбрасывается до нижнего значения **BOTTOM**=0. В этот момент также устанавливается флаг прерывания по **сравнению OCFA,** а не по **переполнению TOV0**, т.к. таймер не успел переполниться.

В данном режиме частота таймера будет рассчитываться по формуле

$$f = \frac{f_{clk_I/O}}{2 \times N \times (1 + OCR2A)}$$

где $f_{clk_I/O}$ – входная частота МК;

N – значение предделителя;

OCR2A – содержимое регистра сравнения.

Таким образом, если требуется выдать определенную частоту – достаточно выбрать нужный режим МК и задать правильное значение предделителя и регистра сравнения.

Fast PWM Mode (the fast pulse width modulation)

Данный режим отличается от предыдущего тем, что в нем можно регулировать не только частоту выдаваемых импульсов, но и коэффициент заполнения периода, т.е. длительность импульса вне зависимости от частоты.

ШИМ – широтно импульсная модуляция. Широко применяемый алгоритм задания аналогового сигнала **цифровым методом.** Поставив на выводе МК **нужный фильтр, можно** получить значение напряжения, отличное по амплитуде от значения логической «1». Наиболее часто, это применяется для регулировки яркости светодиодов и скорости вращения двигателей.

Подключив например двигатель постоянного тока к источнику батарейке напрямую мы получим фиксированное число оборотов в минуту, с помощью ШИМ это число можно понизить.

Так же в качестве повседневного примера можно привести конфорку электропечи – если включить её на максимум, то 220В из фазы бытовой электросети будет непрерывно греть, но если снизить регулятор, то можно невооруженным глазом увидеть, что конфорка

включается и отключается через промежутки времени, достигая в среднем более низкой температуры. Так же будет справедливо считать алгоритмом ШИМ случай, когда особо опытные пользователи пытаются определить температуру заведомо горячей поверхности – если прикоснуться на короткое время, то ожога может даже не произойти. Ещё один пример – управление высотой воздушного шара в полёте. Для поддержания высоты пилот включает подачу теплого воздуха только на короткое время, иначе бы он полетел осваивать верхние слои атмосферы.

Иллюстрация работы ШИМ приведена на рисунке 13.

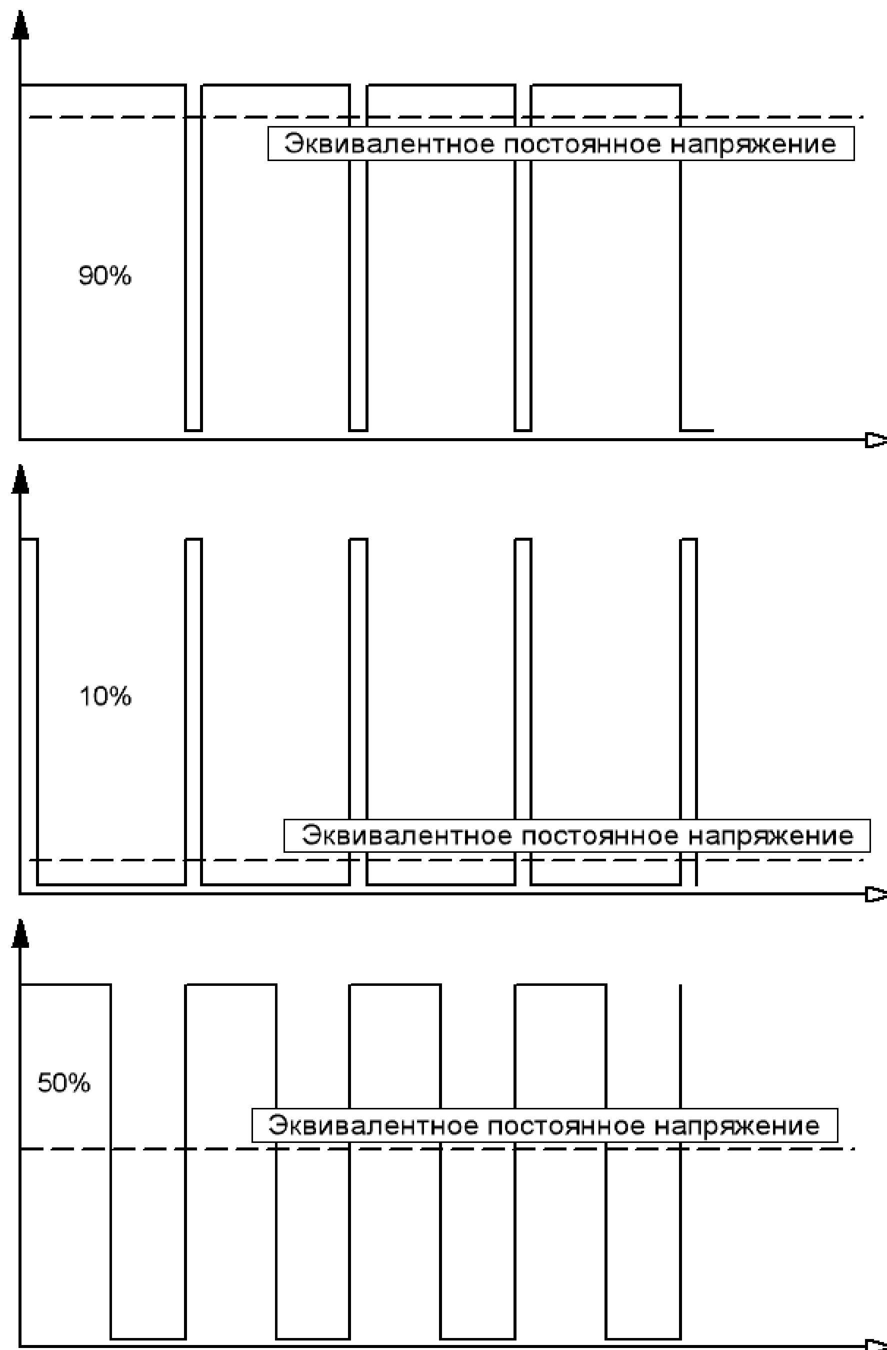

Рисунок 13 Иллюстрация работы ШИМ.

Работа ШИМ для таймера счётчика обеспечена его возможностью вызывать прерывание по совпадению с регистром сравнения не сбрасывая значение регистра счёта. Таймер может досчитать до конца и переполниться, либо, в зависимости от режима до любого другого значения.

Рассмотрим регистры управления таймера 0.

TCCR0A – Timer/Counter Control Register A

Bit	7	6	5	4	3	2	1	0
	COM0**A1**	COM0**A0**	COM0**B1**	COM0**B0**	**-**	**-**	**WGM01**	**WGM00**
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W

Биты **2, 3** - не используются, всегда равны «0».

Биты **0, 1 - WGM00… WGM01**

В сочетании с битом **WGM02**, находящимся в регистре **TCCR0B**, эти биты для выбора режима работы таймера. Ниже приведена таблица выбора режима из даташита.

Mode	WGM02	WGM01	WGM00	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on(1)(2)
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, phase correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, phase correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

Так же данная таблица поясняет режим работы, например в режиме **CTC** таймер будет считать до значения **OCR2A,** в момент достижения **OCR2A** изменится состояние вывода **OCRx,** а флаг переполнения сработает при достижении максимального значения регистра счетчика (т.е. никогда).

Биты **6, 7 - COM0A1, COM0A0 Compare Match Output A Mode**

Конфигурация данных бит определяет работу вывода сравнения **OC0A** таймера **0**. Т.е. таймер может управлять состоянием вывода МК. Если один или оба бита **COM0A[1:0]** установлены, выход **OC0A** отменяет нормальную функциональность порта ввода-вывода, к которому он подключен. Однако обратите внимание, что бит регистра направления данных **DDR**, соответствующий выводу **OC0A**, должен быть установлен, чтобы активировать выходной драйвер. Схема приведена на рисунке Рисунок 14. Данные о том, какие выводы МК подключены к таймерам приведены в **таблицах 13-3, 13-9** даташита .

В зависимости от режима работы функционал у данных битов разный, но они могут выполнять лишь 3 основных функции:

- **Clear** OC0A – установить «0» на выводе OC0A;
- **Set** OC0A - установить «1» на выводе OC0A;
- **Toggle** OC0A – инвертировать состояние вывода OC0A.

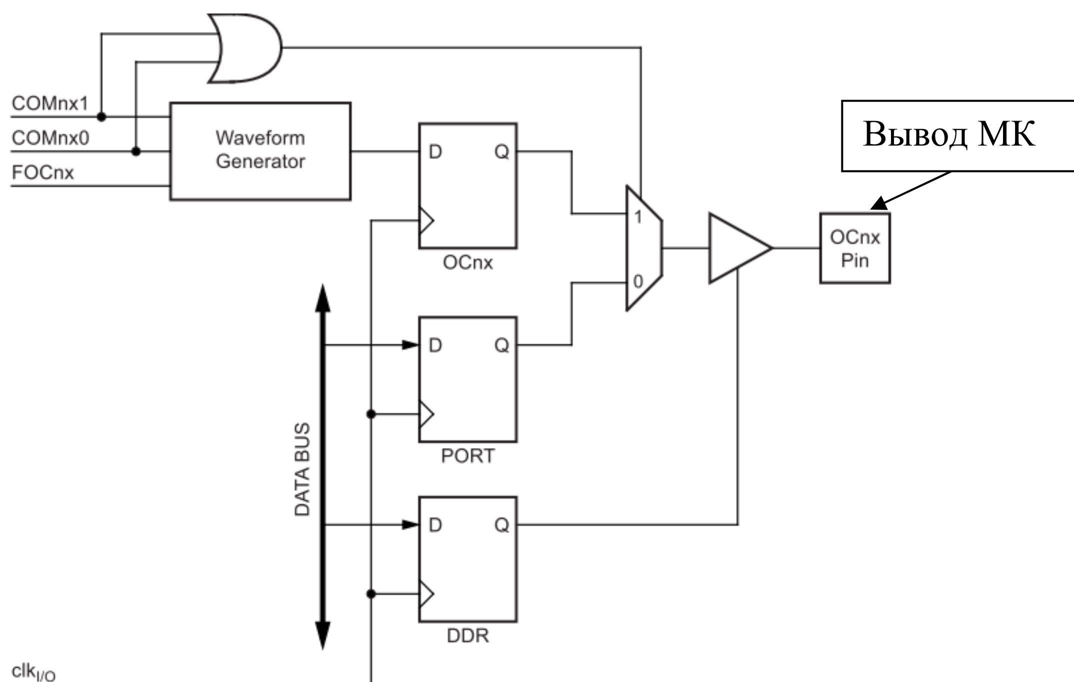

Рисунок 14. Электрическая схема таймера-счетчика.

Биты **5, 4 - COM0B1, COM0B0 Compare Match Output B Mode**

Конфигурация данных бит определяет работу вывода сравнения **OC0B** таймера **0**. Принцип работы тот же, что и у **COM0A1, COM0A0**

TCCR0B – Timer/Counter Control Register B

Bit	7	6	5	4	3	2	1	0
	FOC0A	FOC0B	-	-	**WGM02**	**CS02**	**CS01**	**CS00**
Read/Write	**W**	**W**	R	R	R/W	R/W	R/W	R/W

Биты 6, 7 - **FOC0B, FOC0A - Force Output Compare A, Force Output Compare B.**

Бит принудительного срабатывания сравнения. При записи в этот бит «1» происходит событие, соответствующее режиму работы. При этом не вызываются никакие прерывания и не очищает таймер в режиме CTC. Данные биты не требуют обнуления и всегда читаются как ноль.

Бит 3 - **WGM02.**

В сочетании с битами **WGM00** и **WGM01**, находящимся в регистре **TCCR0B**, этот бит предназначен для выбора режима работы таймера.

Биты 0…3 – **CS00…CS03 - Clock Select**

В соответствии с таблицей 14-9 даташита, устанавливают значение предделителя, либо назначает источником входной частоты внешние выводы МК.

TCNT0 - Timer/Counter Register

Регистр, увеличивающий свое значение с каждым периодом входной тактовой частоты, хранит в себе данные о количестве прошедшего времени.

OCR0A - Output Compare Register A

Регистр сравнения A содержит 8-битное значение, которое постоянно сравнивается со значением счетчика **TCNT0**. При равенстве значений можно вызвать прерывания сравнения выходов или изменение логического уровня на выводе **OC0A**.

OCR0B - Output Compare Register B

Регистр сравнения B содержит 8-битное значение, которое постоянно сравнивается со значением счетчика **TCNT0**. При равенстве значений можно вызвать прерывания сравнения выходов или изменение логического уровня на выводе **OC0B**.

TIMSK0 – Timer/Counter Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0
	-	-	-	-	**-**	**OCIE0B**	**OCIE0A**	**TOIE0**
Read/Write	R	R	R	R	R	R/W	R/W	R/W

Бит **0 - TOIE0: Timer/Counter0 Overflow Interrupt Enable**

Установка этого бита в «1» разрешает прерывание по переполнению таймера 0, если разрешены глобальные прерывания.

Бит **1** - **OCIE0A: Timer/Counter0 Output Compare Match A Interrupt Enable.**

Установка этого бита в «1» разрешает прерывания по сравнению **TCNT0** с **OCR0A.**

Бит **2** - **OCIE0B: Timer/Counter0 Output Compare Match B Interrupt Enable.**

Установка этого бита в «1» разрешает прерывания по сравнению **TCNT0** с **OCR0B.**

Все векторы прерываний приведены в таблице 11-1 даташита.

Рассмотрим работу таймера счетчика 2 на примере работы звукового излучателя и тактильных переключателей. Схема подключения приведена на рисунке 15

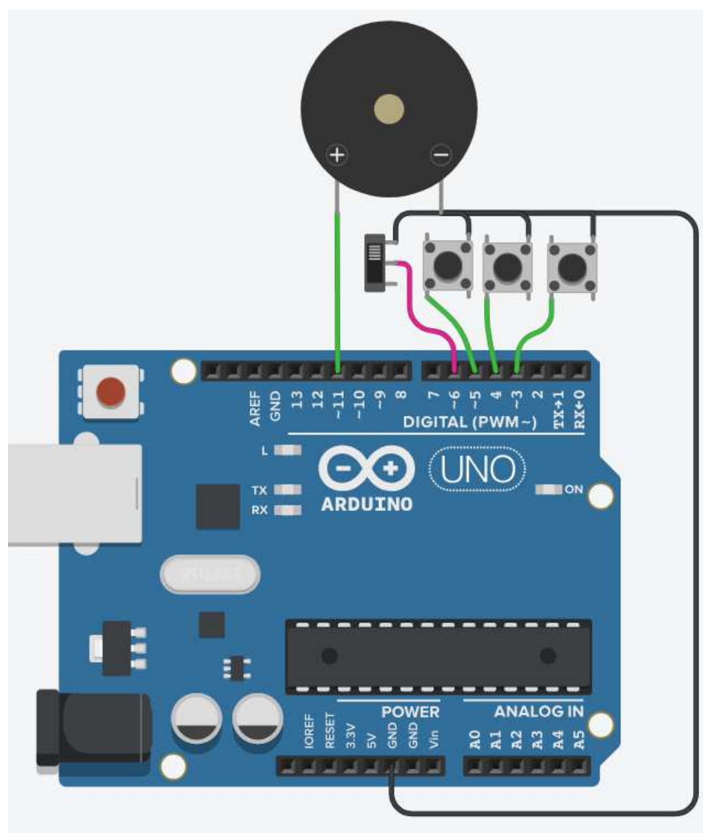

Рисунок 15. Схема подключения светодиодов, звукоизлучателя и переключателей к выводам GPIO, PCINT, OC2A.

Принципиальная электрическая схема приведена на рисунке 16.

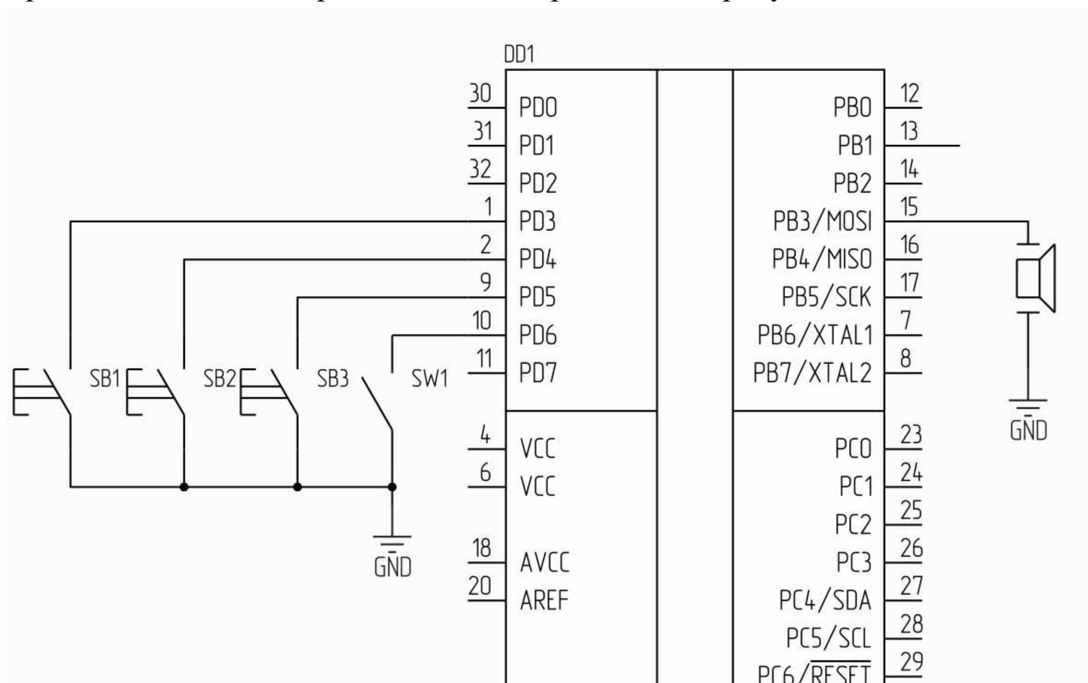

Рисунок 16. Электрическая хема подключения светодиодов, звукоизлучателя и переключателей

Код программы:

```
#define OC2A PB3//аппаратный вывод таймера 2 объявляем подобающим названием

ISR(TIMER2_COMPA_vect)
{
//обработчик прерывания по сравнению TCNT2 с OCR2A
//Здесь можно было написать кусок кода про переключению состояния вывода таймера
//но это действие уже прописано по умолчанию в блоке инициализации (toogle OC2A)
}

ISR(TIMER2_OVF_vect)
{
//обработчик  прерывания по переполнения таймера 2
}

ISR(PCINT2_vect)//обработчик прерывания по ИЗМЕНЕНИЮ уровня выводов PCINT16...PCINT23
{
  if (((PIND & (1 << PD6)))||(PIND&((1 << PD5) |(1<< PD4) | (1<< PD3)))==((1 << PD5) |(1<< PD4) | (1<< PD3)))
    //если тумблер отключен    и все кнопки отпущены
  {
  TCCR2B=0; //то отключаем счёт таймера 2, звуковой сигнал не поступает на динамик
  }
  else
  {
  TCCR2B |= (1 << CS22) | (1 << CS21) | (0 << CS20);// иначе включаем счёт таймера 2
  switch (PIND&((1 << PD5) | (1<< PD4) | (1<< PD3)))
    {
    case 24:
      OCR2A=59; //задаем частоту таймера 2 1046,50 Гц (До третьей октавы)
      break;
    case 40:
      OCR2A=52; //задаем частоту таймера 2 1174,60 (Ре третьей октавы)
      break;
    case 48:
      OCR2A=49;//задаем частоту таймера 2 1318,50 (Ми третьей октавы)
      break;
    }
  }
}

int main(void)
{
```

```
  Serial.begin(115200);           //инициализация монитора последовательного интерфейса
  pinMode(LED_BUILTIN, OUTPUT);        //инициализация встроенного светодиода
DDRB |=(1 << OC2A);               // вывод OC2A настраиваем на выход
TCCR2A = (1<<WGM21)|(1<<COM2A0); //режим CTC, toogle OC2A
TCCR2B |= (1 << CS22) |(1 << CS21) | (0 << CS20); // предделитель на 256
TIMSK2 = (1 << OCIE2A)|(1 << TOIE2); // Разрешить прерывание по сравнению с OCR2A и по переполнению
PCICR =(1 << PCIE2); // Разрешить прерывание по изменения уровня выводов PCINT16...PCINT23
PCMSK2 =(1<<PD6)|(1<<PD5)|(1<<PD4)|(1<<PD3);//Разрешить прерывание по изменения уровня выводов PD3...PD6
PORTD|=(1<<PD6)|(1<<PD5)|(1<<PD4)|(1<<PD3);//включаем внутреннюю подтяжку PD3...PD6

sei();//разрещаем глобальные прерывания

while(1)//основной код программы
  {
  Serial.println("Everything is awesome!");
  digitalWrite(LED_BUILTIN, HIGH);   // мигаем встроенным светодиодом всё время
  _delay_ms(500);
  digitalWrite(LED_BUILTIN, LOW);
  _delay_ms(500);
 }
}
```

Описание работы программы:

Выводы PORTD настроены на вход, и для них разрешено прерывание PCINT.

Таймер 2 по умолчанию отключен, но сконфигурирован так, что бы он работал в режиме CTC и его **вывод OC2A переключался** при переполнении. Т.е. таймер готов выдавать импульсы с разной частотой и скважностью 0,5.

Весь код обрабатывается в обработчике прерываний по изменению уровня выводов PCINT16...PCINT23.

По изменению уровня проверяется условие: если ползунковый переключатель отключен (выполняется условие (PIND & (1 << PD6)), то счёт таймера отключается.

Далее с помощью конструкции **switch case** программа определяет, какая кнопка нажата, и в зависимости от этого изменяет значение предделителя на актуальное, изменяя тем самым частоту импульсов таймера. Данные импульсы поступают на динамик, где электрические импульсы преобразуются в механические и вызывают звучание той же частоты.

Когда пользователь отпускает кнопку, опять срабатывает прерывание по изменению уровня и проверяется условие: если ни одна кнопка не нажата, то работа таймера запрещается и динамик перестает звучать.

Варианты заданий:

№	Задание
1.	Организовать бегущие огни на светодиодах.
2.	Организовать бегущие огни, меняющие направление по нажатию кнопки.
3.	Организовать бегущие огни, меняющие режим по нажатию кнопки. Кол-во режимов не менее трех.
4.	Организовать плавное свечение светодиода, регулируемое кнопками.
5.	По нажатию кнопки менять ноту звучания динамика в пределах одной октавы
6.	Организовать устранение дребезга тактовой кнопки.
7.	Двумя кнопками регулировать яркость свечения светодиода.
8.	Организовать плавное изменение свечения RGB светодиода во всем видимом спектре
9. *	Организовать воспроизведение простейшей мелодии.
10. *	С помощью ШИМ создать сигнал синусоидальной формы.
11. *	Организовать вращение сервоприводом на определенный угол, в зависимости от нажатой кнопки
12. *	Организовать вращение сервоприводом двумя тактильными кнопками – одна крутит в одну сторону, другая в другую
13. *	Организовать вращение шаговым двигателем с выводом скорости на экран. Скорость вращения регулировать кнопками
14. *	Организовать вращение шаговым двигателем с выводом скорости на экран. Направление вращения регулировать кнопками
15. *	Организовать вывод расстояния ультразвукового дальномера на семисегментный индикатор
16. *	Организовать тактильный приемник сигналов азбуки Морзе на вибромоторе. Данные отправлять через последовательный монитор.
* - задачи повышенной сложности	

ПРИМЕЧАНИЕ – бегущие огни должны быть с количеством светодиодов не меньше 6и. Запрещается использовать программируемые светодиодные ленты

Лабораторная работа № 4.
Работа с встроенным АЦП микроконтроллера AVR.

Цель: изучить принцип работы АЦП микроконтроллеров.

Краткие теоретические сведения

АЦП (Аналого-Цифровой Преобразователь) – устройство для преобразования аналогового сигнала в цифровой. В качестве аналогового сигнала выступает уровень напряжения, находящийся в пределах измеряемого значения. Преобразовав аналоговый сигнал в цифровой его становится возможно обработать цифровыми устройствами. Т.е. иными словами – микроконтроллер может знать какое напряжение подано на его вход в вольтах. В зарубежной литературе АЦП обозначается как **ADC** - **A**nalog-to-**D**igital **C**onverter.

Упрощенная схема параллельного АЦП приведена на рисунке 17.

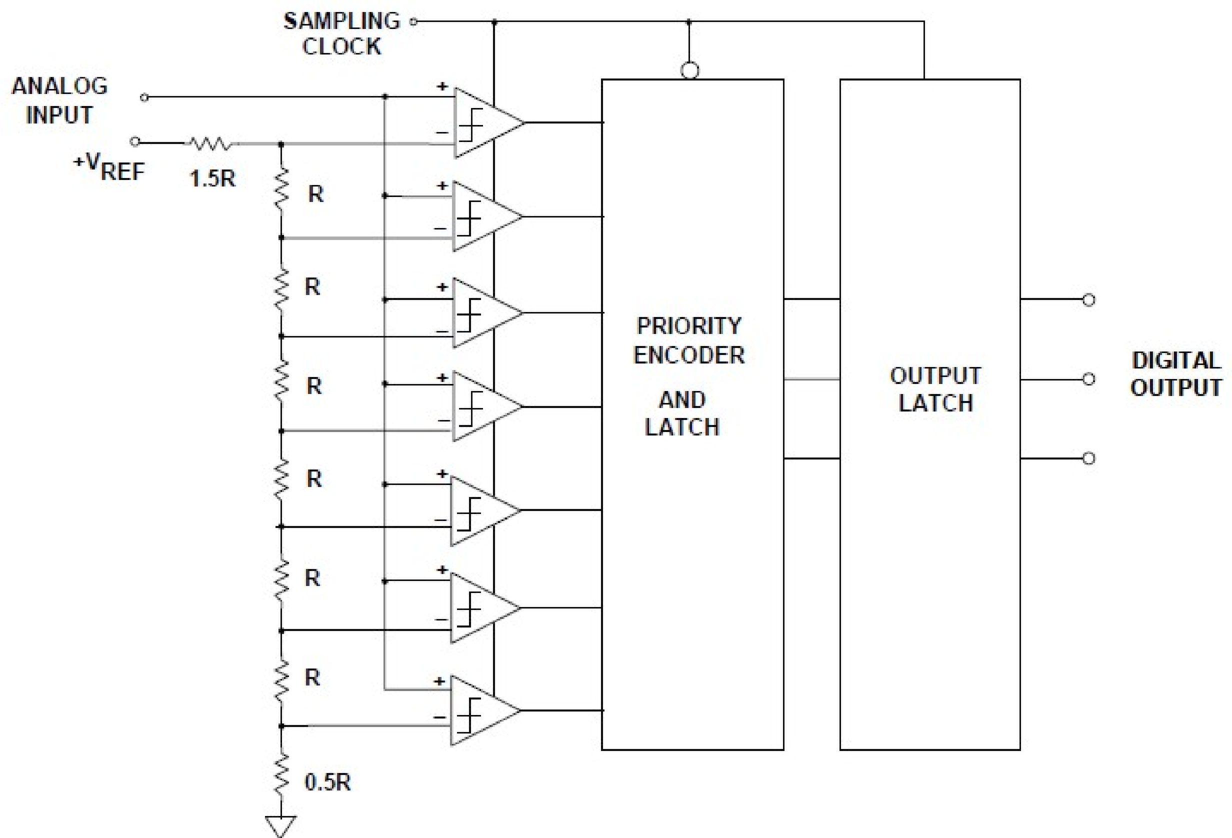

Рисунок 17. Упрощенная схема параллельного АЦП

Аналоговый сигнал со входа поступает на цепь резистивных делителей

Принцип действия АЦП предельно прост: входной сигнал поступает одновременно на все «плюсовые» входы компараторов, а на «минусовые» подается ряд напряжений, получаемых из опорного путем деления резисторами R. Для схемы на рисунке 17 этот ряд будет таким: (1/16, 3/16, 5/16, 7/16, 9/16, 11/16, 13/16) Vref, где Vref – опорное напряжение АЦП.

Пусть на вход АЦП подается напряжение, равное 1/2 Vref. Тогда сработают первые 4 компаратора (если считать снизу), и на их выходах появятся логические единицы.

Приоритетный шифратор (priority encoder) сформирует из «столбца» единиц двоичный код, который фиксируется выходным регистром.

В рассматриваемом микроконтроллере внутри корпуса есть один 10-битный АЦП. Количество бит определяет шаг измерений, чем больше разрешение АЦП, тем точнее будет измерение.

АЦП позволяет измерять напряжение не больше чем напряжение питания микроконтроллера, однако можно задать смещение напряжения по верхнему уровню, отсекая неинтересные пользователю значения, тем самым сужая диапазон измерения и увеличивая шаг измерения.

Схема АЦП внутри МК Atmega328P приведена на рисунке 18.

Все входы АЦП подключены к **PORTA.** Преобразовывать сигнал можно только с одного вывода МК, однако можно разрешить поочередное преобразование с помощью мультиплексора **Input MUX.** Т.е. если на несколько входов подключены разные напряжения, мы можем посмотреть напряжение на одном, потом только на втором и т.д. АЦП имеет свое время преобразования, и пока данные не считались, бесполезно пытаться считать новые, т.к. не сохраняется целостность данных. Скорость преобразования задается в блоке инициализации, и подстраивается при отладке устройства, т.к. чем выше скорость, тем больше вносимая ошибка.

Результатом преобразования будет 10-битное число, которое определяется по формуле:

$$ADC = \frac{U_{IN} \times 1024}{U_{REF}}$$

где U_{IN} – напряжение на входе АЦП;

U_{REF} – опорное напряжение, выбираемое в блоке инициализации.

Другими словами – опорное напряжение (ножка **AREF**) определяет в каких диапазонах нам измерять напряжение. Если подать на **AREF** 2,0 В, то измерять АЦП будет в пределах от 0 до 2,0 В. Так же можно использовать внутренний источник опорного напряжения 1,1 В, но ниже этого значения АЦП измерение достоверно не приведет.

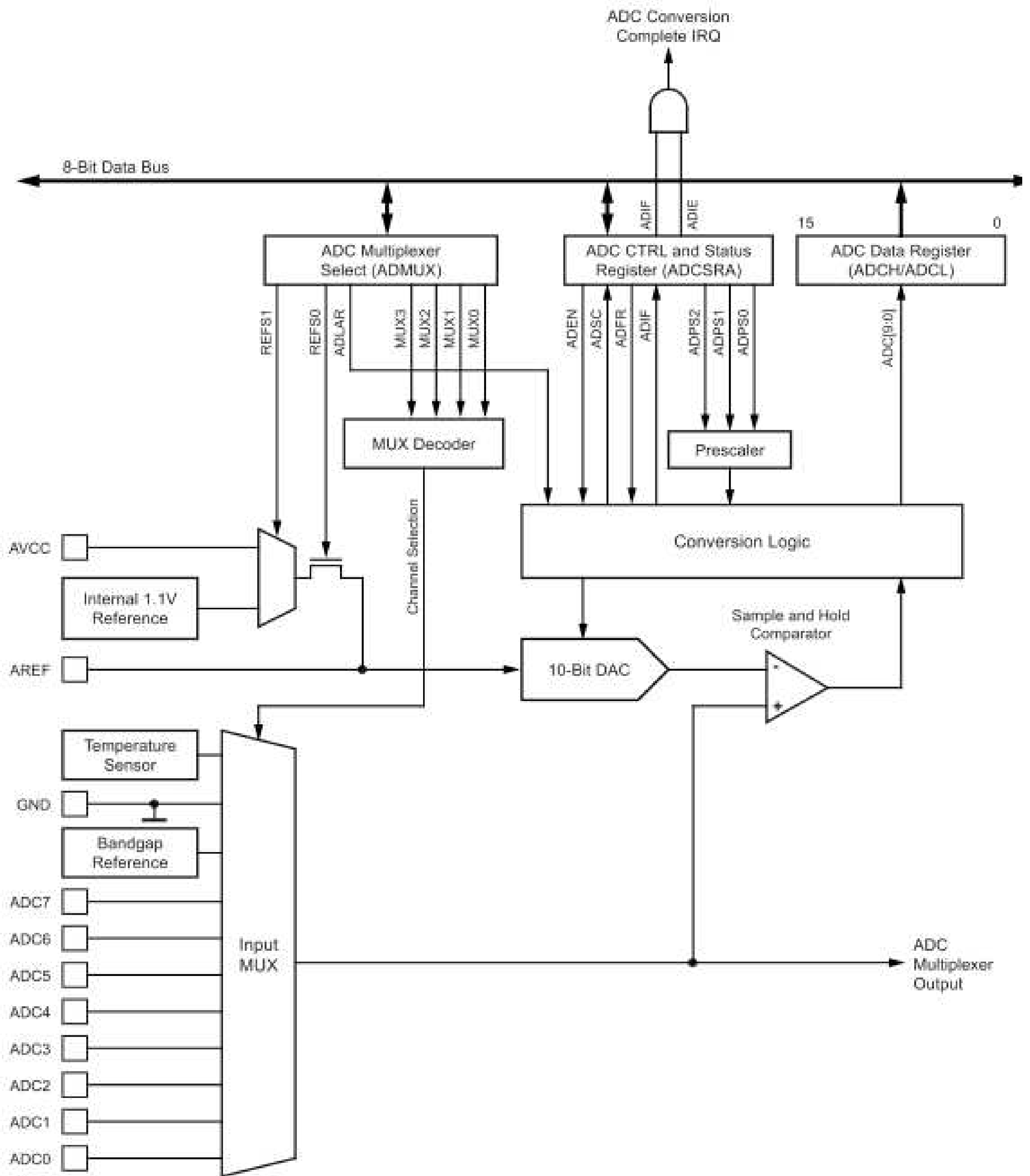

Рисунок 18. Схема АЦП Atmega328P

Рассмотрим регистры управления АЦП.

ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0
	REFS1	REFS0	ADLAR	**-**	**MUX3**	**MUX2**	**MUX1**	**MUX0**
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W

Биты **7, 6** - **Reference Selection Bits.** Биты выбора источника опорного напряжения. Ниже приведена таблица выбора режима из даташита.

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, internal V_{REF} turned off
0	1	AV_{CC} with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 1.1V voltage reference with external capacitor at AREF pin

Бит **5 – ADLAR.** Бит представления данных в регистре данных АЦП. Ниже приведено описание работы данного бита из даташита.

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	

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	

Бит **4 -** не используется, всегда равен «0».

Биты **3…0 – MUX3:0**. Биты выбора вывода для преобразования сигнала. Подробнее в таблице 23-4 даташита.

ADCSRA - ADC Control and Status Register A.

Bit	7	6	5	4	3	2	1	0
	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

Бит **7 – ADEN - ADC Enable.** Разрешение работы АЦП. Если установлен в «0», то АЦП выключен.

Бит **6 – ADSC - ADC Start Conversion.** Бит начала преобразования – если установлен в «1» и при этом разрешена работа АЦП, то начнется преобразование сигнала со входа в течении 13-25 тактов входной частоты. Когда преобразование завершится, этот бит вернет значение «0».

Бит **5 – ADATE - ADC Auto Trigger Enable.** Бит автоопределения импульса начала. Если установлен в «1», то преобразователь начнет работу по выбранному источнику сигнала начала.

Бит **4 – ADIF - ADC Interrupt Flag.** Флаг срабатывания прерывания АЦП. Когда АЦП вызывает прерывание начала преобразования, этот бит устанавливается в «1». После выхода из прерывания он снова примет значение «0» автоматически. Можно вызвать прерывание в ручную, установив этот бит в «1» программно, но возвращать его в «0» тоже придется самостоятельно.

Бит **3 – ADIE - ADC Interrupt Enable.** Разрешение прерывания АЦП.

Биты **2…0 - ADPS2:0: ADC Prescaler Select Bits.** Биты предделителя частоты преобразования АЦП. Подробнее в таблице 23-5 даташита.

ADCL and ADCH – The ADC Data Register. Регистр хранения выходных данных преобразователя. Это два раздельных восьмибитных регистра, если требуется значение всех 10-ти бит, нужно сделать специальное преобразование с помощью сдвига. Бит **ADLAR** определяет в какой из регистров положить старшие либо младшие разряды. Подробнее в п.23.9.3 даташита.

ADCSRB – ADC Control and Status Register B

Регистр выбора источнику сигнала начала преобразования

Bit	7	6	5	4	3	2	1	0
	-	**-**	**-**	**-**	**-**	**ADTS2**	**ADTS1**	**ADTS0**
Read/Write	R	R	R	R	R	R/W	R/W	R/W

Биты **7, 5…3 –** резервные, всегда «0»

Биты **2:0 – ADTS2…0 - ADC Auto Trigger Source.** Определяют по какому источнику сигнала начала преобразователь начнет работу. Подробнее в таблице 23-6 даташита.

DIDR0 – Digital Input Disable Register 0

Регистр запрещения цифровой обработки сигнала входов **PORTA**

Bit	7	6	5	4	3	2	1	0
	-	**-**	**ADC5D**	**ADC4D**	**ADC3D**	**ADC2D**	**ADC1D**	**ADC0D**
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W

Биты **7, 6 –** резервные, всегда «0»;

Биты **5…0 – ADC5D..ADC0D: ADC5..0 Digital Input Disable.** По умолчанию входной сигнал всегда приходит на регистр **PINA** внутри **PORTA.**
Т.е. можно считывать и цифровой сигнал из регистра PINA, и аналоговый из АЦП. Если цифровой сигнал не нужен – эти биты устанавливаются в «1» для снижения потребления. **ADC7** и **ADC6** не имеют цифровых буферов.

Рассмотрим работу АЦП на примере работы светодиода и переменного резистора. Схема приведена на рисунке 19.

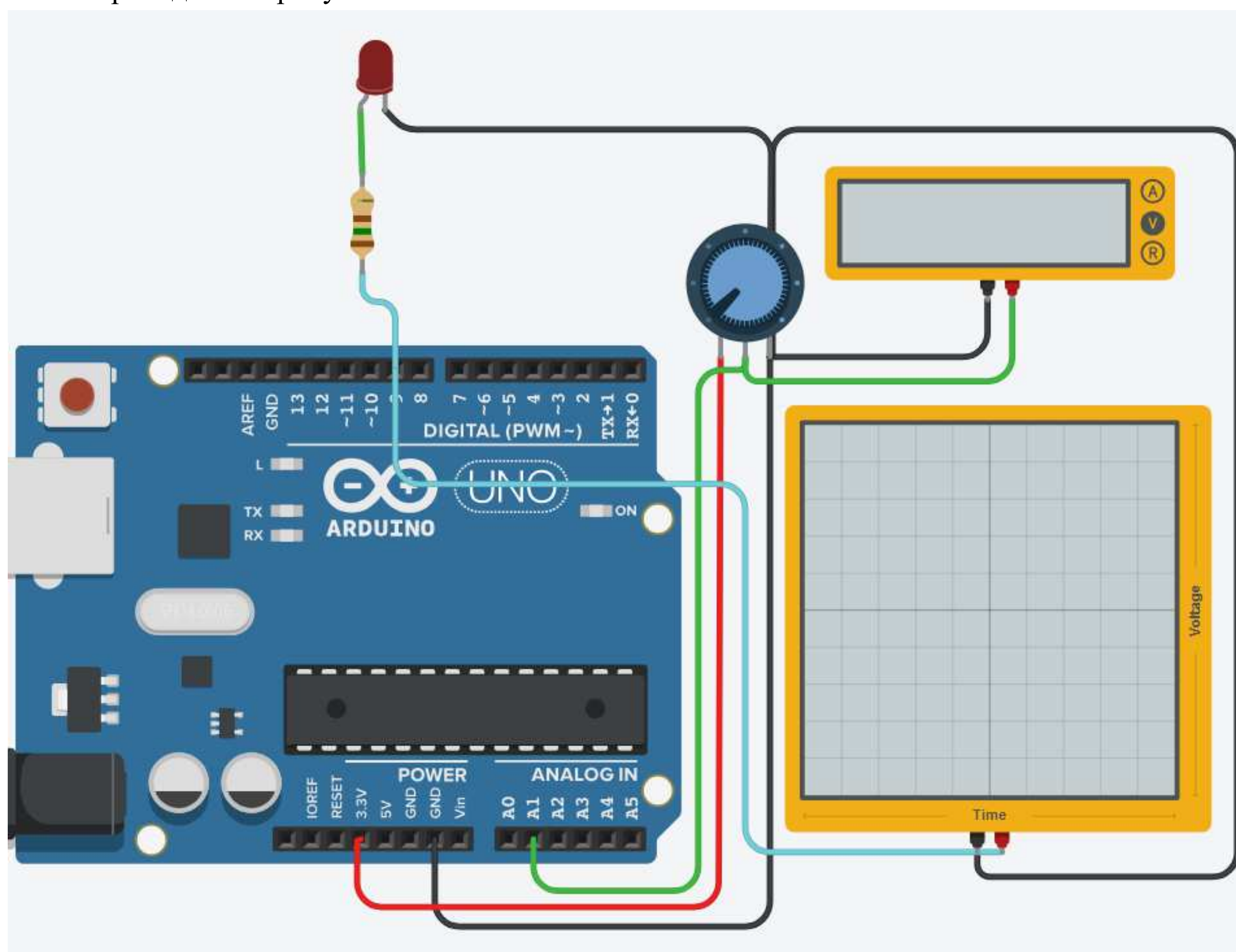

Рисунок 19. Схема подключения светодиодов и переменного резистора к МК

Код программы:

```
#define LED2 PB1

ISR(TIMER2_COMPA_vect) прерывание по COMPA
{
PORTB &= ~(1 << LED2);//гасим светодиод
}
ISR(TIMER2_COMPB_vect)// прерывание по COMPB
{//если разрешали прерывание, его обязательно надо прописать в
коде
}

ISR(TIMER2_OVF_vect) прерывание по переполнению
{
PORTB |= (1 << LED2); //включаем светодиод
}

ISR (ADC_vect)//прерывание преобразования АЦП
{
ADCSRA |= (1<<ADSC);//начинаем преобразование
OCR2A=(0.37)*ADC+2;//нехитрый расчет по формуле выше
}

int main (void)
{
DDRB |= (1 << LED2); // Set LED pins as outputs
  //блок иницализации таймера
  TCCR2A = (1<<WGM21)|(1<<WGM20);//fast WPM MODE
  TCCR2B = (1 << CS02) |(1 << CS01) | (1 << CS00); // Делитель на
1024
  OCR2A = 100; // Для периода 500 мс при частоте 16 МГц и делителе
на 1024
  OCR2B = 100;
  TIMSK2 = (1 << OCIE2B)| (1<<TOIE2)|(1 <<OCIE2A);// Разрешить
прерывание по сравнению с OCR2A и OCR2B
  sei(); // Enable global interrupts

    //блок иницализации АЦП
   ADMUX = (1<<REFS0)| //опорное напряжение = VCC(5В)
  (1<<MUX0); //Входное напряжение к AD1
  ADCSRA = (1<<ADEN)|(1<<ADSC)// //разрешаем работу АЦП и
инициируем первое преобразование
   |(0 << ADPS2) |(0 << ADPS1)| (0 <<ADPS0)  //предделитель на 128
   |(1 << ACIE);//разрешаем прерывания

  while(1)
  {
  //ничего не делаем, МК сам всё сделает
  }
}
```

Описание работы программы:

Напряжение на A1, приходящее через переменный резистор и вольтметр преобразуется в блоке прерываний **АЦП** и записывается в значение регистра сравнения таймера 2 **OCR2A.** Это задает длительность импульса ШИМ при той же частоте. Чем больше напряжение, тем больше длительность импульса. Среднее значение ШИМ будет выражаться яркостью свечения светодиода.

Преобразование АЦП происходит в блоке прерываний и завершается автоматически.

В основном блоке программы ничего не происходит, все действия выполняются в блоке прерываний. При вызове прерываний в обработчике не должно быть долгих операций и тем более задержек, иначе ход программы может пойти не корректно.

Варианты заданий:

№	Задание
1.	Переменным резистором регулировать скорость бегущих огней.
2.	Переменным резистором организовать шкалу на 6и светодиодах
3.	Организовать вывод данных с датчика влажности в последовательный монитор
4.	Организовать вывод данных с датчика температуры в последовательный монитор
5.	Переменным резистором дискретно регулировать частоту звучания ноты в пределах одной октавы
6. *	Организовать вывод данных с датчика температуры и датчика влажности в последовательный монитор
7. *	Организовать вращение сервопривода на угол, соответствующий углу поворота переменного резистора
8. *	Переменным резистором регулировать скорость вращения шагового двигателя
9. *	Переменным резистором регулировать частоту выдаваемой с помощью ШИМ синусоиды
10. *	Регулировать переменным резистором свечение RGB светодиода во всем видимом спектре
* - задачи повышенной сложности	

Список рекомендуемой литературы

1. Евстифеев, А.В. Микроконтроллеры AVR семейств Mega. Руководство пользователя / А.В. Евстифеев. - М.: ДМК, 2015. - 588 с.

2. Мортон, Д. Микроконтроллеры AVR. Вводный курс / Д. Мортон. - М.: ДМК, 2015. - 272 с. Моделирование и анализ бизнес-процессов [Текст] : учебное пособие / В. А. Силич, М. П. Силич ; Министерство образования и науки Российской Федерации, Томский государственный университет систем управления и радиоэлектроники. - Томск : ТУСУР, 2011. - 213 с.

3. Прокопенко, В.С. Программирование микроконтроллеров ATMELна языке С / В.С. Прокопенко. - СПб.: Корона-Век, 2015. - 320 с.

4. https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf (дата обращения 27.07.2022)