Министерство образования и науки Российской Федерации
федеральное государственное бюджетное образование учреждение
высшего профессионального образования

«Российский государственный университет
нефти и газа имени И.М. Губкина»

Кафедра автоматизации технологических процессов

Учебное пособие для бакалавров направления
*220400 «*Управление в технических системах*»*

В.В. ЮЖАНИН

ТИПЫ ДАННЫХ И ПОРТЫ ВВОДА-ВЫВОДА МИКРОКОНТРОЛЛЕРОВ

Издательский центр
РГУ нефти и газа имени И.М. Губкина
Москва - 2015

ББК 74.58

И 65 - поправить

Рецензенты:

____

Южанин В.В.

Типы данных и порты ввода-вывода . Учебное пособие. – М.: Издательский центр РГУ нефти и газа имени И.М. Губкина, 2015. – 41 с.

В данном пособии рассматриваются вопросы низкоуровневого программирования микроконтроллеров.

Учебное пособие предназначено для бакалавров, обучающихся по направлению 220400 «Управление в технических системах».

Данное издание является собственностью РГУ нефти и газа имени И.М. Губкина и его репродуцирование (воспроизведение) любыми способами без согласия университета запрещается

© РГУ нефти и газа имени И.М. Губкина, 2015

© В.В. Южанин, 2015

© _______ – оформление серии, 2015

I. О чем этот курс?

Основной предмет изучения данного курса – аппаратные средства микропроцессорной техники. Говоря точнее – даже не сами аппаратные средства, а принципы их работы, благодаря которым «примитивная» электроника (регистры, триггеры, аппаратные сумматоры) превращается в нечто значительно более «умное» – процессор, исполняющий сложнейшие программы.

Современные микропроцессоры проделали длинный путь эволюции (с 1970-х годов) и превратились в чрезвычайно сложные устройства, работающие под управлением не менее сложного программного обеспечения. Поэтому в данном курсе рассматриваются их значительно более простые аналоги – *микроконтроллеры*.

Из микропроцессоров, как известно, получаются домашние (и не только) компьютеры. Что же можно сделать из микроконтроллеров? Лишь основные примеры – смартфон, промышленный контроллер (ПЛК) для управления технологическим процессом, а также платформа Arduino, представляющая собой своеобразный конструктор Lego для детей старше 20 лет.

В курсе можно выделить два крупных блока – изучение аппаратных средств и разработка программного обеспечения. Подобно своим старшим собратьям (микропроцессорам), микроконтроллеры не могут решать задачи и быть полезными без программного обеспечения. Чтобы научиться решать задачи с помощью микроконтроллеров, разобраться в их аппаратной части, в данном курсе предлагается **изучение аппаратных средств микроконтроллера через их программирование**.

Программирование на уровне аппаратных средств называется низкоуровневым программированием /не в смысле уровня квалификации программиста, а в смысле уровня организации вычислительной машины, с

которым программист взаимодействует!/. Низкоуровневое программирование требует знаний аппаратных средств /мы этого и хотим от курса!/, структур целых и вещественных типов данных на битовом уровне (уровне нулей и единиц).

Языком программирования микроконтроллеров обычно является язык *C* и наш курс не стал исключением. К слову, язык *C*, хотя и появился в 1970-е годы, значительно повлиял на современные языки программирования, некоторые из которых называются почти так же: *C++*, *Java, C#*.

Лабораторные стенды курса снабжены собственно микроконтроллером (серии Atmel Mega 16), к которому подключаются модули светодиодов, кнопок и ЖК-дисплея. К слову, иногда возникает вопрос, почему курс не сделан на базе упомянутой платформы Arduino. Дело в том, что эта платформа имеет слишком много готовых наработок, упрощающих жизнь любителям, но избавляющих от необходимости думать, что на практике приведет массовому скачиванию кода из интернета на защитах лабораторных работ.

В завершении курса рассматривается важная тема – чем программирование промышленных контроллеров (ПЛК) отличается от программирования микроконтроллеров, и как знание микроконтроллеров поможет в работе с ПЛК. Повторим, микроконтроллеры используется как элементная база ПЛК. Программирование промышленных контроллеров стремятся упрощать так, чтобы разработчик алгоритмов ПЛК как можно меньше думал об устройстве ПЛК, и сосредоточился на задачах управления технологическим процессом. В этом, кстати, и состоит идея высокоуровневых языков программирования в противоположность низкоуровневым. Среди разработчиков АСУТП встречаются специалисты-электрики, электронщики, а также программисты. Чтобы максимально задействовать опыт и интуицию каждой из этих групп, в стандарт МЭК-

61131-3 включено несколько языков программирования, каждый из которых будет более понятен соответствующей группе специалистов (нетрудно сообразить какой для кого). Однако, несмотря на общую тенденцию к упрощению, во многих ПЛК при программировании до сих пор приходится в существенной степени владеть низкоуровневым программированием.

II. Представление целых беззнаковых чисел в микропроцессорах. Позиционные системы счисления.

В микропроцессорной технике со времен фон Неймана (конец 1940-х гг.) в основном используется двоичная система счисления. При этом для человека привычнее десятичная система счисления по числу пальцев на руках. Поэтому представляют интерес правила перевода из двоичной системы счисления в десятичную и обратно. Помимо двоичной системы счисления в микропроцессорной технике часто применяется шестнадцатеричная, ее роль также будет рассмотрена в данной лекции.

При работе с микропроцессорами большое значение имеет разработка программного обеспечения на том или ином языке программирования. В данной лекции будут рассмотрено внутреннее устройство целых беззнаковых типов данных языка Си (unsigned int, unsigned char, unsigned long).

Позиционные системы счисления

В этой привычной нам системе счисления число есть строка цифр, каждой цифре приписан вес. Значение числа – взвешенная сумма его цифр (разрядов):

$$1734 = 1 \cdot 10^3 + 7 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$$

Для представления дробных чисел можно использовать отрицательные степени числа 10:

$$187.25 = 1 \cdot 10^2 + 8 \cdot 10^1 + 7 \cdot 10^0 + 2 \cdot 10^{-1} + 5 \cdot 10^{-2}$$

Величина $r = 10$ называется основанием (base, radix) системы счисления. В общем случае для произвольного основания системы счисления имеем:

$$D = \sum_{i=-n}^{p-1} d_i \cdot r^i = d_{p-1} d_{p-2} \ldots d_0 \ . \ d_{-1} d_{-2} \ldots d_{-n} \qquad (1)$$

где d_i – разряды числа D, $0 \le d_i < r$ или $d_i \in [0, r - 1]$,

d_{p-1} – старший разряд числа, most significant digit (MSD),

d_{-n} – младший разряд числа, least significant digit (LSD),

p – количество разрядов целой части числа,

n – количество разрядов дробной части числа.

Если $n = 0$, то число D – целое. Если $n > 0$, $p > 0$ и фиксированы, то такой формат представления называется числом с фиксированной точкой. Если n и p могут изменяться, то такой формат чисел называется числами с плавающей точкой. В архитектурах x86 и AVR для представления чисел с плавающей точкой используется стандарт IEEE-754, который рассматривается в одном из следующих разделов.

Роль двоичной системы счисления в микропроцессорной технике

Рассмотрим более подробно двоичную систему счисления, т.е. случай $r = 2$.

Пример:

$110110_{(2)} = \boldsymbol{1}\cdot 2^5 + \boldsymbol{1}\cdot 2^4 + \boldsymbol{0}\cdot 2^3 + \boldsymbol{1}\cdot 2^2 + \boldsymbol{1}\cdot 2^1 + \boldsymbol{0}\cdot 2^0 = 32 + 16 + 4 + 2 + 0 = 54_{(10)}$

Значение двоичной системы в цифровой технике настолько велико, что разряды двоичного числа получили свое собственное название – бит (bit – Binary digiT).

Двоичные разряды-биты естественно хранить в электронном регистре. Интересно, что хранимые в регистре двоичные числа могут быть не только целыми, но и вещественными. Для вещественных чисел часть битов регистра надо оставить целыми, присвоив им вес 2^i ($i \ge 0$), а остальные биты сделать вещественными, присвоив им вес 2^i ($i < 0$). Таким образом, с

помощью регистров можно хранить целые и вещественные числа (в виде чисел с фиксированной и плавающей точкой).

Обратим внимание, что не все числа можно представить описанным способом. Так, число $0.1_{(10)}$ в двоичном коде является периодической дробью: $0.0(0011)_{(2)}$. Почему получается именно так, рассмотрим позднее. Сейчас же заметим, что под хранение периодической дроби требуется бесконечное число разрядов ($n \to \infty$), в то же время разрядность любого электронного регистра (сумма $n + p$) конечна. Если записать в регистр конечное количество разрядов числа $0.0(0011)_{(2)}$, мы по существу округлим его, т.е. *не сможем представить точно*.

Исторически в микропроцессорной технике возникли понятия *байта* и *слова*. ***Байт*** – единица хранения информации, соответствующая 8 двоичным разрядам, т.е. 8 битам. Кроме того, существует понятие полубайт или тетрада – 4 бита. ***Слово*** – характерный размер данных, измеряемый в битах, для которого аппаратно реализована операция сложения (которое, кстати, автоматически дает вычитание при использовании *дополнительного кода*). Слово зависит от архитектуры процессора и является важной его характеристикой. Так, процессоры Intel Pentium I-IV 32-разрядные (размер их слова 32 бита), Intel Core 2 – 64-разрядные. Микроконтроллер AVR, изучаемый в данном курсе – 8-разрядный или, что то же самое, 8-битный.

Диапазон беззнаковых типов данных

Важный практический вопрос при программировании – как связана разрядность целых беззнаковых типов данных (unsigned char, unsigned int, unsigned long) и диапазон их возможных значений. Рассмотрим для примера восьмиразрядный тип unsigned char. Известно, что количество кодовых комбинаций *N* для *n*-разрядного целого числа равно:

$$N = 2^n$$

Для типа unsigned char количество кодовых комбинаций $N = 2^8 = 256$. Минимальное значение очевидно равно 0, если положить в (1) все биты равными нулю. Максимальное значение выглядит как «все единицы»:

$$max = 1111\ 1111_{(2)}$$

Как быстро вычислить это число? Если к нему прибавить 1, получим:

$$max + 1 = 1\ 0000\ 0000_{(2)}$$

Это число имеет единственный ненулевой бит с весом 2^8, значит оно и само равно 2^8. Тогда для unsigned char максимальное значение равно $2^8 - 1 = 255$. В общем случае для произвольной разрядности n:

$$\max(n) = 2^n - 1$$

Посчитайте самостоятельно диапазон значений для 16- и 32-битных целых беззнаковых чисел (для архитектуры AVR это будет соответствовать типам unsigned int и unsigned long соответственно).

Шестнадцатиразрядная система счисления

Разряд шестнадцатеричной системы счисления лежит в диапазоне от 0 до 15:

$$d \in [0, 16) = [0, 15] \qquad (2)$$

Записи разрядов для диапазона 0-9 соответствуют десятичной системе, а цифры 10-15 задаются латинскими буквами A-F (табл. 1).

Пример. Переведем число $AF3_{(16)}$ в десятичную систему счисления с помощью формулы (1):

$$AF3_{(16)} = A \cdot 16^2 + F \cdot 16^1 + 3 \cdot 16^0 = 10 \cdot 16^2 + 15 \cdot 16^1 + 3 \cdot 16^0 =$$

$$= 2560 + 240 + 3 = 2803.$$

табл. 1. Значения шестнадцатеричных чисел

Основание системы счисления	**Значение разряда**															
16	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
10	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Связь двоичной и шестнадцатеричной систем счисления

Существует простая связь двоичной и шестнадцатеричной систем счисления. Используя эту связь, программист может в голове осуществить перевод чисел между этими системами. Благодаря этому улучшается читаемость программистом кода. Например, вместо 32-разрядного двоичного числа (в коде займет примерно половину экрана) можно "отделаться" 8-разрядным шестнадцатеричным.

Рассмотрим связь этих двух систем счисления. Для примера запишем двоичное 10-разрядное целое число:

$$D=b_8\cdot 2^8 + b_7\cdot 2^7 + b_6\cdot 2^6 +b_5\cdot 2^5 +b_4\cdot 2^4 +b_3\cdot 2^3 +b_2\cdot 2^2 +b_1\cdot 2^1 +b_0\cdot 2^0$$

Сгруппируем слагаемые следующим образом:

$$D= (b_8\cdot 2^0) \cdot 2^8 + (b_7\cdot 2^3 + b_6\cdot 2^2 +b_5\cdot 2^1 +b_4\cdot 2^0) \cdot 2^4 + (b_3\cdot 2^3 +b_2\cdot 2^2 +b_1\cdot 2^1 +b_0\cdot 2^0) \cdot 2^0$$

Легко заметить, что полученное выражение похоже на запись шестнадцатеричного числа:

$$D = d_2 \cdot 16^2 + d_1 \cdot 16^1 + d_0 \cdot 16^0$$

Кроме того, можно показать, что выражения в скобках, например ($b_7\cdot 2^3 + b_6\cdot 2^2 +b_5\cdot 2^1 +b_4\cdot 2^0$) лежат в диапазоне [0, 15], т.е. удовлетворяют требованию к разрядам 16-разрядного числа см. формулу (1). Просто выписав шестнадцатеричные коды выражений в скобках, получим строковую запись числа в шестнадцатеричной системе счисления. Очевидно, легко сделать и обратное преобразование

Примеры:

$1101\ 0101_{(2)} = 13_{(10)}\ 5_{(10)} = D_{(16)}\ 5_{(16)} = D5_{(16)}$

$AF3_{(16)} = 1010\ 1111\ 0011_{(2)}$

Общий подход к преобразованию чисел между системами счисления

Мы рассмотрели системы счисления с основаниями 2, 10, 16. В работе надо уметь переводить из каждой системы в любую другую. Простую связь двоичной и шестнадцатеричной рассмотрели только что. Для

перевода из двоичной или шестнадцатеричной в десятичную нужно просто расписать число по формуле (1).

Остались переводы из десятичной в двоичную и из десятичной в шестнадцатеричную. Попробуем снова применить формулу (1):

$$135_{(10)} = 1 \cdot A^2 + 3 \cdot A^1 + 5 \cdot A^0$$

Получается сложно, потому что для перевода придется вести вычисления в шестнадцатеричной системе счисления. Другой способ – подгонка. Нам необходимо подобрать разряды d_2, d_1, d_0 в разложении:

$$135_{(10)} = d_2 \cdot 16^2 + d_1 \cdot 16^1 + d_0 \cdot 16^0$$

Сколько раз по $16^2 = 256$ содержится в 135? Ни одного, поэтому разряд $d_2 = 0$. Разряд d_1 указывает, сколько раз по 16^1 содержится в 135. Очевидно, 8. Дальше легко определить d_0.

Рассмотренное правило подбора, однако, оставляет ощущение необоснованности. Изучим более основательный способ. Пусть задано число в некоторой системе счисления. Требуется его перевести в систему счисления с основанием *r*. Рассмотрим отдельно перевод его целой и дробной части. В соответствии с (1) целая часть числа задана *p* разрядами, а дробная часть числа задана n разрядами. Известный способ перевода целого десятичного числа в двоичное состоит в «делении столбиком».

Пример. Переведем $13_{(10)}$ в двоичную систему счисления.

Делимое	Делитель	Частное	Остаток
13	2	6	1
6	2	3	0
3	2	1	1
1	2	0	1

Остатки от деления, записанные в обратном порядке, дадут разряды в двоичной системе счисления:

$$13_{(10)} = 1101_{(2)}$$

Но почему остатки от деления – это разряды числа в новой системе счисления, в частности первый остаток – это младший разряд d_0? Целая часть числа в нужной нам системе с основанием r (на интересуют $r = 2$ и $r = 16$) в соответствии с (1) имеет вид:

$$D = \sum_{i=0}^{p-1} d_i \cdot r^i = d_{p-1} \cdot r^{p-1} + d_{p-2} \cdot r^{p-2} + \ldots + d_1 \cdot r^1 + d_0 \qquad (3)$$

При «делении столбиком» D делилось на 2, в общем случае делим на основание r:

$$\frac{D}{r} = \underbrace{d_{p-1} \cdot r^{p-2} + d_{p-2} \cdot r^{p-3} + \ldots + d_1 \cdot r^0}_{Q} + \underbrace{\frac{d_0}{r}}_{F}$$

По определению частное равно целой части результата, а остаток можно найти, умножив дробную часть результата на делитель. Что в полученном выражении целая часть и что дробная? Каждое слагаемое Q является произведением целой степени целого числа r на целый разряд d_i, поэтому и само Q целое. Поскольку $d_0 < r$, то F – чисто дробное число ($F < 1$). Вся целая часть числа D/r сосредоточилась в Q, а вся вещественная часть – в F. Следовательно, умножая дробную часть F на делитель r, получим остаток от деления D/r. Этот остаток равен:

$$F \cdot r = (d_0 / r) \cdot r = d_0$$

Полученный вывод обосновывает корректность «деления столбиком», точнее его первого деления. Разряд d_1 будет остатком от деления Q на r. Повторяя процедуру, пока $Q \neq 0$, получим все разряды числа в системе счисления с основанием r в обратном порядке.

Рассмотрим перевод дробной части числа:

$$D = \sum_{i=-n}^{-1} d_i \cdot r^i = d_{-1} \cdot r^{-1} + d_{-2} \cdot r^{-2} + \ldots + d_{-n} \cdot r^{-n} \qquad (4)$$

Умножим D на r:

$$D \cdot r = \underbrace{d_{-1}}_{Q} + \underbrace{d_{-2} \cdot r^{-1} + \ldots + d_{-n} \cdot r^{-n+1}}_{F} \qquad (5)$$

Можно показать, что сумма *F* всех слагаемых выражения, кроме d_{-1}, строго меньше 1. Следовательно, целая часть *Q* состоит из единственного слагаемого d_{-1} (старший разряд дробной части числа в новой системе счисления). Обнулим целую часть $D \cdot r$, снова умножим на *r*, получим d_{-2} и так далее. Отметим, что и при переводе целой, и при переводе дробной части мы не сделали никаких предположений относительно основания системы *r*, поэтому описанный способ подходит для перевода десятичных чисел в двоичные или шестнадцатеричные.

Пример. Переведем $0.1_{(10)}$ в двоичную систему счисления.

D	***D·r***	**Целая часть *Q***	**Дробная часть *F***
0.1	0.2	$0 = d_{-1}$	0.2
0.2	0.4	$0 = d_{-2}$	0.4
0.4	0.8	$0 = d_{-3}$	0.8
0.8	1.6	$1 = d_{-4}$	0.6
0.6	1.2	$1 = d_{-5}$	0.2
0.2	0.4	$0 = d_{-6}$	0.4
0.4	0.8	$0 = d_{-7}$	0.8
0.8	1.6	$1 = d_{-8}$	0.6
0.6	1.2	$1 = d_{-9}$	0.2

… и так далее – получили периодическую дробь $0.1_{(10)} = 0.0(0011)_{(2)}$

Для удобства все рассмотренные способы перевода сведены в табл. 2.

табл. 2. Правила перевода между системами счисления

Из системы счисления	***В систему счисления***		
	Bin(ary), 2	**Dec(imal), 10**	**Hex(ademical), 16**
Bin(ary), 2	–	формула (1)	Связь 2 <–> 16

Dec(imal), 10	«Общее правило перевода»	–	«Общее правило перевода» или подбор
Hex(ademical), 16	Связь 2 <–> 16	формула (1)	–

III. Машинное представление знаковых чисел

В курсе «Устройства цифровой автоматики» рассматривался аппаратный двоичный сумматор (рис. 1). Этот сумматор способен складывать целые неотрицательные числа. Оказывается, что с помощью сумматора можно выполнять вычитание, если на его вход подавать число, предварительно преобразовав их в так называемый ДОПОЛНИТЕЛЬНЫЙ КОД.

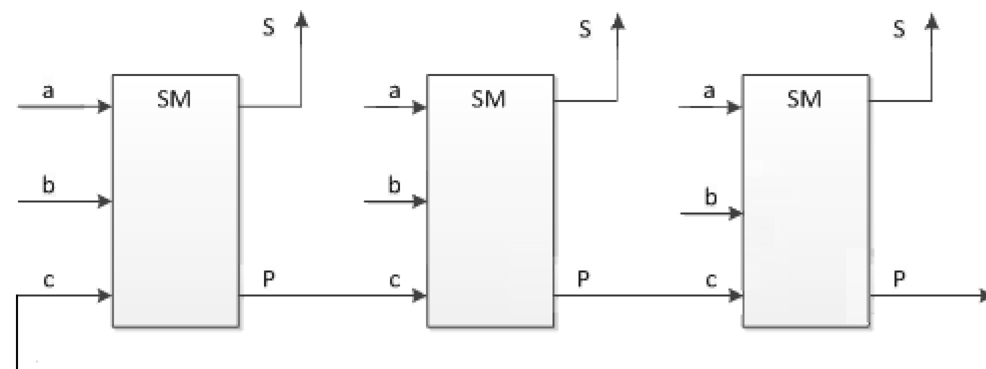


рис. 1. Аппаратный двоичный сумматор

Прежде чем перейти к дополнительному коду, рассмотрим некоторые важные особенности данного сумматора. Типичный сценарий его использования в вычитательной машине состоит в сложении двух чисел, хранящихся в n-разрядных регистрах с последующей записью результата также в n-разрядных регистрах. Этот сценарий происходит при работе следующего кода:

```
char a = 10, b = 10;
char c = a + b;
```

Проблема состоит в том, что при сложении двух n-разрядных чисел результат в общем случае имеет разрядность $n + 1$, поэтому при записи результата в n-разрядном регистре старший бит в общем случае потеряется.

Пример 1:

 1101 0111
 0111 0101
10100 1100

Появился 9 разряд.

Пример 2:

1010 0011
0100 1011
1110 1110

Количество разрядов сохранилось.

С точки зрения математики, такое сложение называют: *Сложение по модулю 2^n*.

$$C = (a + b)\ mod\ 2^n$$

Если $a + b$ девятиразрядное, то $a + b = \sum_{i=0}^{n} d_i \cdot r^i$, где $n = 8$.

Найдем остаток от деления $a + b$ на 2^n:

$$\frac{a+b}{2^n} = \frac{\sum_{i=0}^{n} d_i \cdot r^i}{2^n} = d_i + \frac{\overbrace{\sum_{i=0}^{n-1} d_i \cdot r^i}^{Q}}{2^n}$$

Число Q – есть искомый остаток от деления, при этом оно отличается от $a + b$ отброшенным страшим разрядом, что нам и требовалось проверить.

Оказывается, можно приспособить двоичный сумматор к вычитанию. Как известно, вычитание и сложение суть алгебраическое сложение, поэтому вычитание сводится к сложению числа со знаком «минус». Рассмотрим, как это можно сделать с помощью *дополнительного кода*. При этом автоматически станет ясно, как представляются знаковые числа в микропроцессорах.

Определение. *Дополнительный код целого n-разрядного числа x в r-ичной системе счисления имеет вид:*

$$\tilde{x} = r^n - x \quad (6)$$

r – основание системы счисления,

n – разрядность числа,

Продемонстрируем основное свойство дополнительного кода, выполнив сложение на аппаратном сумматоре:

$$y \oplus \tilde{x} = y \oplus (2^n - x) = \left(2^n + (y - x)\right) mod\ 2^n$$

В случае, если $y \geq x$, старший разряд выражения в скобках обусловлен числом 2^n:

1	*0*	*0*	*0*	*0*	*0*	2^n
0	b_{n-1}	b_{n-2}	…	…	b_0	$y - x$
1	b_{n-1}	b_{n-2}	…	…	b_0	$2^n + y - x$

Таким образом, при сложении у с дополнительным кодом x, результат равен разности $y - x$:

$$y \oplus \tilde{x} = \left(2^n + (y - x)\right) mod\ 2^n = y - x \quad (7)$$

Для разбора случая $y < x$, представим выражение следующим образом:

$$y \oplus \tilde{x} = \left(2^n - (x - y)\right) mod\ 2^n$$

Поскольку $2^n - (x - y) < 2^n$, то все выражение в скобках является остатком от деления на 2^n :

$$\left(2^n - (x - y)\right) mod\ 2^n = 2^n - (x - y)$$

Обратим внимание, что последнее выражение соответствует определению дополнительного кода **(6)**, при $r = 2$, то есть результатом является дополнительный код выражения:

$$y \oplus \tilde{x} = 2^n - (x - y) = (\widetilde{x - y}) \quad (8)$$

Если мы когда-нибудь потом сложим $(\widetilde{x - y})$ с еще каким-нибудь числом, то это сложение автоматически будет учитывать отрицательность числа $(\widetilde{x - y})$.

Таким образом, для того, чтобы осуществить вычитание двух чисел, достаточно рассчитать сумму уменьшаемого с дополнительным кодом вычитаемого. Для того, чтобы воспользоваться данными свойствами дополнительного кода, нужен быстрый способ его расчета. Вычисление же дополнительного кода по определению **(6)** трудоемко, т.к. само требует вычитания. Рассмотрим быстрый способ расчета двоичного дополнительного кода. Используя математический прием «прибавить и отнять единицу», запишем:

$$\tilde{x} = 2^n - x = 2^n - 1 + 1 - x$$

Очевидно, что число $(2^n - 1)$ представляет собой число из n единиц. Тогда:

$$\tilde{x} = \underbrace{11\ldots11}_{\text{число из } n \text{ единиц}} - x + 1$$

При вычитании $(1111\,1111 - x)$ получаем число $\bar{x}$, биты которого инверсны по отношению к битам x. Действительно выражение $1 - b_i$ равно 1, если $b_i = 0$ и равно 0, если $b_i = 1$. Кроме того, поскольку $1 \geq b_i$, то нигде не встречаются «заемы» от старшего бита. В итоге имеем:

$$\tilde{x} = \overline{x} + 1$$

Учитывая полученный результат ясно, что двоичный сумматор превращается в вычитатель, если проинвертировать одно из входных чисел и задать «1» на вход переноса (c = 1) на сумматоре младшего разряда.

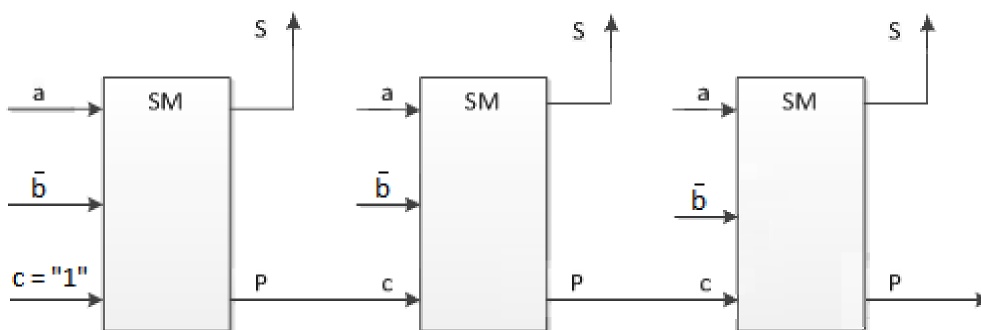


рис. 2. Аппаратный двоичный вычитатель

Приведем без обоснования формулу для определения значения знакового числа по его битовому набору:

$$val(x) = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i \quad (9)$$

В знаковых числах (каким и является число a) старший разряд отвечает за знак. И при записи битового набора числа b мы должны учитывать это.

Эта операция получила название «операция распространения знака». Рассмотрим простой и часто встречающийся код:

```
int a = - 43;
long b = a;
```

В коде 16-разрядная переменная в дополнительном коде a, записывается в 32-разрядном b. Ожидаемое поведение кода должно обеспечить равенство значений a и b.

$$val(a) = val(b)$$

Однако, если просто скопировать 16-разрядный битовый набор переменной a в 16 младших разрядов переменной b, а остальные (старшие) разряды b оставить положить равными нулю (что вроде бы «логично»), то нетрудно сообразить, что результат достигнут не будет, хотя бы потому, что число с нулевым старшим разрядом (а у b аж 16 старших разрядов равны нулю!) не может быть отрицательным.

Рассмотрим сначала более простой случай – пусть число b – не 32-разрядное, а 17-и разрядное. Снова потребуем:

$$val(a) = val(b)$$

Тогда:

$$-b_{15}2^{15} + \sum_{i=0}^{14} b_i 2^i = -b_{16}2^{16} + \sum_{i=0}^{15} b_i 2^i$$

Выделим старшее слагаемое из суммы в правой части тождества:

$$-b_{15}2^{15} + \sum_{i=0}^{14} b_i 2^i = -b_{16}2^{16} + b_{15}2^{15} + \sum_{i=0}^{14} b_i 2^i$$

$$-b_{15}2^{15} = -b_{16}2^{16} + b_{15}2^{15}$$

$$b_{16}2^{16} = 2 \cdot b_{15}2^{15} = b_{15}2^{16}$$

Отсюда,

$$b_{16} = b_{15}$$

т.е. новый разряд следует положить равным старшему разряду исходного битового набора.

Аналогичным образом получают значения последующих битов – 1, 7, 18 и так до получения 32-разрядного числа. Поскольку старший бит в исходном числе отвечает за знак, то описанный алгоритм приведения типов получила название алгоритм распределения знака.

Итак, при приведении целой знаковой переменной меньшей разрядности к знаковой переменной большей разрядности свободные старшие разряды не заполняются нулями (что «логично»), а заполняются значением старшего разряда исходного числа (что правильно).

IV. Машинное представление вещественных чисел. Формат IEEE-754

Прежде чем глубоко погрузится в формат чисел с плавающей точкой, бросим въедливый взгляд на табл. 4, а именно на типы long и float. Оба типа имеют одинаковую разрядность – 32 бита или 4 байта (для архитектуры AVR, для x86 размеры целых типов другие!), однако разные диапазоны значений. Многих это вводит в заблуждение, автор этих строк однажды даже видел следующее рассуждение на интернет-форуме: «поскольку оба типа 4-байтные, то их максимальное значение должно быть одинаковым, следовательно, значение $3.4 \cdot 10^{38}$ для float ошибочное». Однако, ошибки никакой нет. Фокус состоит в том, что отведенные 32 бита можно использовать по-разному. У <u>*целых беззнаковых*</u> чисел биты (или

битовый набор) интерпретируются по формуле (1), у знаковых – по формуле (9). Правила интерпретации битового набора для вещественных чисел совершенно другие, не похожие на (1) и (9), благодаря чему получается теми же 32 битами задать числа с большим диапазоном. Рассмотрим эти правила.

Представление вещественных чисел основано на нормализованной экспоненциальной записи числа:

$$x = \pm M \cdot r^e$$

где M – мантисса, e – порядок. Для десятичной системы счисления $M \in [1{,}10)$, для двоичной системы счисления $M \in [1{,}2)$.

Пример. Представим десятичное число в нормализованной экспоненциальной записи**:**

$$155.625 = \underbrace{1.55625}_{\text{мантисса } M} \cdot \underbrace{10^2}_{r^e}$$

Можно это представить себе так: множитель 10^2 «двигает» точку в мантиссе *M* вправо, в результате получается исходное число.

Пример. Переведем то же самое десятичное число в двоичный вид и затем в двоичную нормализованную форму.

$$155.625 = 9B_{(16)} + 0.101_{(2)} = 10011011.101$$

Учтем, что роль множителя $r^e = 2^e$ аналогична десятичной системе, тогда Нормализованная экспоненциальная запись этого двоичного числа:

$$10011011.101 = \underbrace{1.0011011101}_{\text{мантисса } M} \cdot \underbrace{2^7}_{r^e}$$

Если взять отрицательный порядок, например *e* = -3, то сдвиг запятой происходит влево:

$$1.0011011101 \cdot 2^{-3} = 0.0010011011101$$

Обратим внимание, что мантисса представляет собой вещественное число с фиксированной точкой:

$$M = \sum_{i=-n}^{0} b_i 2^i$$

Поскольку $M \in [1,2)$, то для любого значения мантиссы старший разряд b_0 равен единице, поэтому его можно отбросить, сэкономив один бит памяти.

Сделав предварительные замечания, рассмотрим правила интерпретации битового набора вещественных чисел по формуле, заданной стандартом IEEE-754:

$$x = (-1)^s \cdot \underbrace{1.F}_{\text{мантисса } M} \cdot 2^{\varepsilon - 127} \quad (10)$$

F – вещественная часть мантиссы;

s – бит знака (s = 1 для x< 0);

$e = \varepsilon - 127$ – порядок числа, величина $\varepsilon \in [0{,}255]$.

Как видно, формула учитывает равенство единице старшего бита мантиссы в экспоненциальной записи числа.

Порядок e может быть положительным и отрицательным. Например, если $\varepsilon = 120, e = -7$.

Формат IEEE-754 определяет способы машинного представления вещественных чисел с одинарной точностью (32 бита, тип float в языке C) и двойной точностью (64 бита, тип double в язык C). Одинарная точность от двойной отличается количеством разрядов мантиссы и порядка (рис. 3).

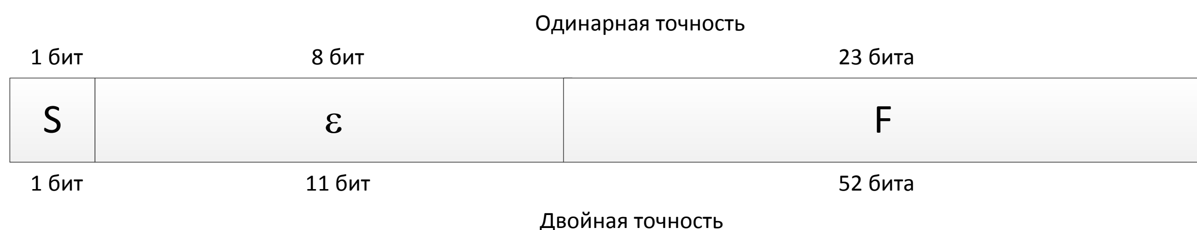
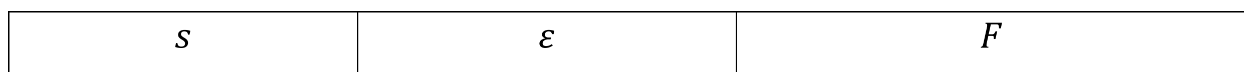


рис. 3. Формат IEEE-754 с двойной и одинарной точностью

Пример. Представим знакомое нам число $155.625 = 1.0011011101 \cdot 2^7$ в формате с одинарной точностью.

$$e = 7 \Rightarrow \varepsilon = e + 127 = 134 = 86_{(16)} = 10000110_{(2)}$$



s	ε	F

0	10000110	0011011101 $\underbrace{0000000000000}_{\text{дописываем нули с конца}}$

Вернемся к вопросу, с которого начали: получилось ли с помощью IEEE-754 хранить в 32-битной переменной значения, большие максимального 32-разрядного целого? Максимальное число типа unsigned long, если задать все его биты равными единице, равно:

$$N_{max} = 2^n - 1 = 2^{32} - 1 \approx 4.3 \cdot 10^9$$

Максимальное число в формате IEEE-754 с одинарной точностью получится, если задать максимальную мантиссу $M_{max} = 1.11 \dots 1 \approx 2$ и максимальный порядок $e_{max} = \varepsilon_{max} - 127 = 128$, равно:

$$x_{max} = M_{max} \cdot 2^{e_{max}} = 2 \cdot 2^{128} = 2^{129} \approx 6.8 \cdot 10^{38}$$

Как видно, формат IEEE-754 обеспечивает $e_{max} \gg n$, поэтому $2^{e_{max}} \gg 2^n$.

В отличие от целых чисел, для вещественных чисел имеется нетривиальное понятие минимального по модулю числа, также называемого **машинным нулем**. Зададим $M_{min} = 1.00 \dots 0$, $e_{min} = -127$, тогда

$$x_{min} = 2^{-127} = 5.8 \cdot 10^{-39} \neq 0.$$

Получается неожиданный вывод, что используя (4) нельзя задать число, строго равное нулю.

> *Обратим внимание, что величины x_{min} и x_{max} не совпадают с диапазоном вещественных типов в табл. 4, хотя и близки к ним. Это связано, с тем, что в данном изложении опущены некоторые детали формата IEEE-754 (например, т.н. денормализованные числа), слабо влияющие на общее понимание, однако сильно его затрудняющие. Кроме того, из той же таблицы видно, что вместо формата с двойной точностью double и формата long double в AVR используется float.*

Как же задать строгий ноль? Только отойдя от правила (4) в особых случаях. В табл. 3 приведены специальные комбинации, которые машиной

обрабатываются особо. Помимо понятных вариантов с нулем и бесконечностью, имеется специальное значение NaN (not a number, не число). Значение NaN нужно, например, для обозначения некорректного результата при попытке посчитать квадратный корень из -1.

табл. 3. Специальные битовые комбинации IEEE-754

Значение числа	**Битовая комбинация числа** (x – бит может быть любым)		
	s	ε	F
+0	0	00000000	00000000000000000000000
-0	1	00000000	00000000000000000000000
$+\infty$	0	11111111	00000000000000000000000
$-\infty$	1	11111111	00000000000000000000000
NaN	x	11111111	$xxxxxxxxxxxxxxxxxxxxxxx$ (за исключением случая «все нули», соответствующего $\pm\infty$)

Понятие точности представления чисел с плавающей точкой также значительно сложнее по сравнению с целыми числами. Если записать вещественное число в целочисленную переменную (int, long), то его дробная часть просто отбросится. Если то же число записать в вещественную переменную (float), то дробная часть целиком не отбросится, но и без потерь не обойдется – т.к. разрядность ограничена.

Зададимся вопросом, какова будет разница между истинным числом округления при представлении произвольного числа (например, числа π).Оценим эту разницу:

$$\Delta x = x - \tilde{x} = (M - \widetilde{M}) \cdot 2^e$$

$\tilde{x}$- число, которое необходимо представить, x – округленное до конечной разрядности $\tilde{x}$, величины $M, \widetilde{M}$ – мантиссы чисел $x, \tilde{x}$.Абсолютная

ошибкаΔ зависит от порядка e (который у чисел x и $\tilde{x}$ один и тот же).Перейдем к относительной ошибке, которая от порядкаeне зависит:

$$\gamma = \frac{\Delta x}{\tilde{x}} = \frac{\tilde{x} - x}{\tilde{x}} = \frac{(\widetilde{M} - M) \cdot 2^e}{\widetilde{M} \cdot 2^e} = \frac{\Delta M}{\widetilde{M}} \qquad (11)$$

Мантисса округленного числа x содержит конечное число разрядов и равна:

$$M = \sum_{i=-n}^{0} b_i 2^i$$

Мантисса истинного числа $\tilde{x}$ содержит бесконечное число разрядов и равна:

$$\widetilde{M} = \sum_{i=-\infty}^{0} b_i 2^i$$

Тогда

$$\Delta M = \sum_{i=-\infty}^{0} b_i 2^i - \sum_{i=-n}^{0} b_i 2^i = \sum_{i=-\infty}^{-n-1} b_i 2^i = 2^{-n} \underbrace{\sum_{i=-\infty}^{-1} b_{i-n} 2^i}_{\leq 0.11111\ldots\approx 1}$$

Отсюда

$$\Delta M \leq 2^{-n}$$

Как известно, за счет округления до ближайшего можно снизить погрешность в два раза, поэтому

$$\Delta M \leq \max \Delta M = 2^{-(n+1)}$$

Максимальная величина относительной погрешности γ будет достигнута при максимальном ΔM (равном $2^{-(n+1)}$) и минимальном $\widetilde{M}$ (по определению равном единице). Тогда

$$\gamma \leq \gamma_{max} = \frac{\max \Delta M}{\min \widetilde{M}} = 2^{-(n+1)}$$

Величина γ_{max} называется машинным эпсилоном. Машинный эпсилон – числовое значение, меньше которого невозможно задавать точность любого алгоритма, возвращающего вещественное число.

Пример. Рассмотрим, насколько может отличаться число π от своего представления в формате IEEE-754 с одинарной точностью:

$$\gamma = \frac{\Delta x}{\tilde{x}} = \frac{\pi - x}{\pi} \le \gamma_{max}$$

$$\Delta x = \pi - x \le \pi \cdot \gamma_{max} = 3.141593 \cdot 2^{-(n+1)} \approx 1.8 \cdot 10^{-7}$$

табл. 4. Типы данных в архитектуре AVR ATMega

<table>
<tr><th colspan="2">Тип</th><th>Знак</th><th>Разрядность</th><th>Диапазон значений</th></tr>
<tr><td rowspan="8">целый тип</td><td>char = unsigned char</td><td>беззнаковый</td><td rowspan="2">8</td><td>0 … +255</td></tr>
<tr><td>signed char</td><td>знаковый</td><td>-128 ... +127</td></tr>
<tr><td>int = signed int</td><td>знаковый</td><td rowspan="4">16</td><td>-32768 … 32767</td></tr>
<tr><td>unsignedint</td><td>беззнаковый</td><td>0 … 65535</td></tr>
<tr><td>short int =
= signed short int</td><td>знаковый</td><td>-32768 … 32767</td></tr>
<tr><td>unsigned short int</td><td>беззнаковый</td><td>0 … 65535</td></tr>
<tr><td>long = long int =
= signed long int</td><td>знаковый</td><td rowspan="2">32</td><td>-2147483648 … +2147483647</td></tr>
<tr><td>unsigned long int</td><td>беззнаковый</td><td>0 … 4294967295</td></tr>
<tr><td rowspan="3">вещественный тип</td><td>float</td><td>знаковый</td><td>32</td><td>$\pm 1.17 \cdot 10^{-38}$ - $\pm 3.4 \cdot 10^{38}$</td></tr>
<tr><td>double</td><td>знаковый</td><td>32</td><td>$\pm 1.17 \cdot 10^{-38}$ - $\pm 3.4 \cdot 10^{38}$</td></tr>
<tr><td>long double</td><td>знаковый</td><td>32</td><td>$\pm 1.17 \cdot 10^{-38}$ - $\pm 3.4 \cdot 10^{38}$</td></tr>
</table>

V. Порты ввода/вывода микроконтроллера Atmel Mega

С помощью портов ввода/вывода (ВВ) микроконтроллер взаимодействует с другими цифровыми устройствами. Поэтому большинство выводов типичного микроконтроллера относятся к его портам ВВ. Микроконтроллеры серии Atmel Mega имеют несколько портов ВВ, обозначающихся буквами (PORTA, PORTB и т.д.). Эти порты являются двунаправленными 8-разрядными, т.е. каждый такой порт имеет 8 выводов, что позволяет читать или записывать до 8 цифровых сигналов. При этом выводы порта можно конфигурировать независимо, т.е. часть выводов может работать на ввод, а другая часть на вывод.

Рассмотрим примеры подключения устройств к портам ВВ. При вводе типичными источниками сигнала являются микросхемы с ТТЛ-уровнями (например, выход микросхемы И-НЕ, цифровой код от АЦП в параллельной форме) и кнопки.

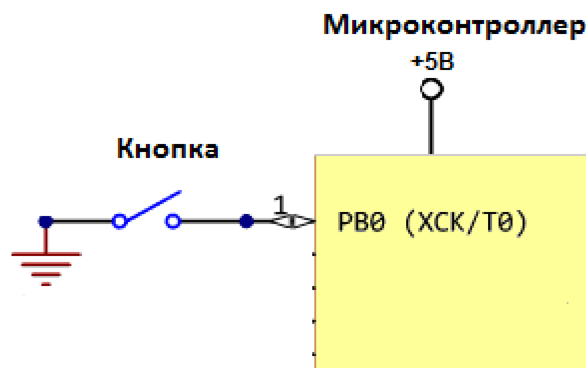


рис. 4. Пассивный источник сигнала – кнопка

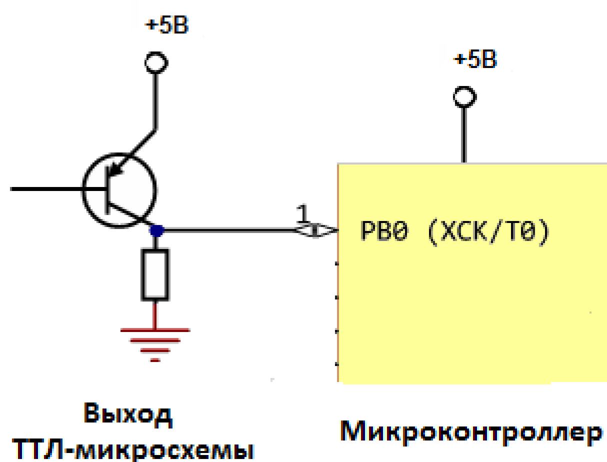


рис. 5. Активный источник сигнала: выход ТТЛ (Транзисторно-транзисторной логики)

Рассмотрим более подробно работу кнопки (рис. 4), которая подключена к нулевому выводу порта В (PB0). Если контакт кнопки замкнут, очевидно, на входе порта будет напряжение лог. «0». Если кнопка разомкнута, то потенциал на входе PB0 определяется зарядом, который скопился на соответствующе ножке микросхемы. Поскольку емкость ножки микросхемы мала, то любая маломощная помеха может зарядить или разрядить ее, т.е. считываемый сигнал может оказаться любым. Следовательно, в первоначальном варианте схема неработоспособна. Решением является подача напряжения во входную цепь микроконтроллера. Тогда при разомкнутой кнопке на выводе будет гарантированный высокий уровень, а при нажатой кнопке – низкий. Напряжение в цепь подает сам микроконтроллер, подключая к ней свое питание +5В через специальный резистор, называемый подтягивающим резистором. Резистор кроме того позволяет избежать короткого замыкания при нажатой кнопке.

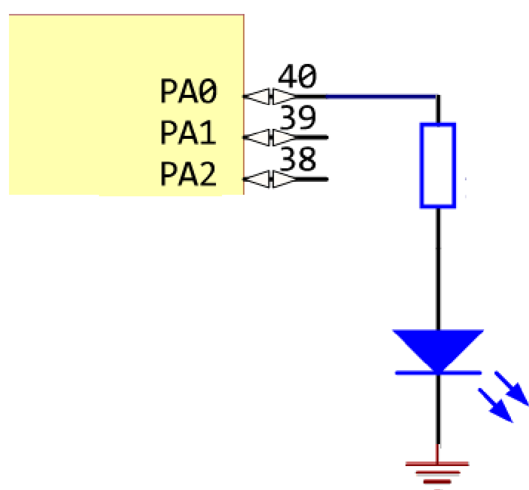


рис. 6. Потребитель логического сигнала: светодиод (схема подключения 1)

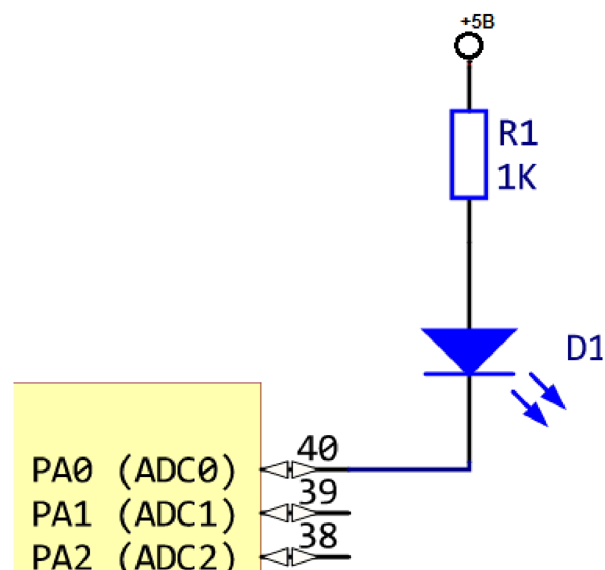


рис. 7. Потребитель логического сигнала: светодиод (схема подключения 2)

При работе с активным источником сигнала (выход ТТЛ-микросхемы – рис. 5) подтягивающий резистор не нужен, так как микросхема участвует в формировании напряжения сигнала.

Примером потребителя сигнала является светодиод (рис. 6, рис. 7). За счет управления выводом порта задается разность потенциалов на светодиоде и появляющийся при этом ток зажигает светодиод. Для схемы рис. 6 диод ожидаемо включается лог. «1» по PA1, одного при подключении нескольких диодов через цепи микросхем проходит большой ток, что нежелательно. Этот негативный эффект устранен в схеме рис. 7. Однако, управление диодом становится неожиданным: при лог. «1» на PA1 разность потенциалов на диоде равна нулю, тока нет, диод не горит: при лог. «0» на PA1 ток течет через диод от ИН в микросхему.

Из сказанного выше ясно, что при использовании портов ВВ сначала требуется каждый из выводов сконфигурировать на ввод или вывод. При этом в случае работы на ввод необходимо также сконфигурировать использование подтягивающего резистора. Управление встроенной периферией (к которой помимо портов ВВ относятся АЦП, таймер и др.) в

микроконтроллерах ATMega осуществляется через так называемые регистры ввода/вывода. Регистры ввода/вывода отображены в памяти данных микроконтроллера, т.е. к ним можно обратиться по адресу как к обычной ячейке памяти.

Для управления каждым из портов ВВ имеются три 8-разрядных регистра ВВ: DDRx, PORTx и PINx. Биты регистра DDRx задают режим ввода (лог. «0») или вывода (лог. «1»). Регистр PINx позволяет считать текущее значение порта, независимо от того, какой режим ввода/вывода установлен. Регистр PORTx в режиме вывода задает выходные сигналы на выводах порта, а в режиме ввода задает включение (лог. «1») или отключение (лог. «0») подтягивающих резисторов для выводов порта.

табл. 5. Возможные режимы работы портов ввода-вывода

<table>
<tr><th>Режим</th><th>DDRx</th><th>PORTx</th><th>PINx</th></tr>
<tr><td>Вывод</td><td>1</td><td>Задает состояние выводов</td><td>Не используется</td></tr>
<tr><td>Ввод активного источника сигнала</td><td>0</td><td>0
(Выключить подтягивающий резистор)</td><td rowspan="2">Содержит значение входного сигнала</td></tr>
<tr><td>Вывода пассивного источника сигнала</td><td>0</td><td>1
(Включить подтягивающий резистор)</td></tr>
</table>

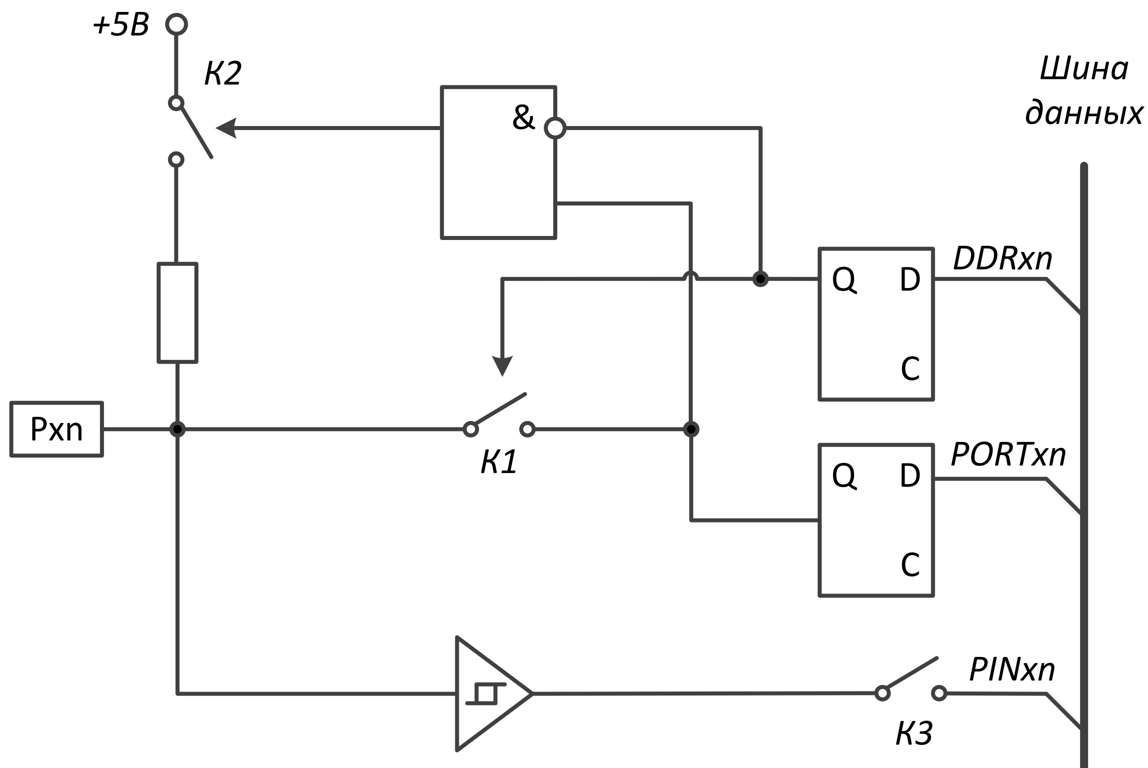


рис. 8. Принципиальная схема **одного** вывода порта.

DDRxn – n-й бит регистра DDRx;

PORTxn – n-й бит регистра PORTx;

PINxn – n-й бит регистра PINx

Принципиальная схема одного из выводов порта ВВ показана на рис. 8. Из рис. 8 видно, как обеспечена описанная выше функциональность регистров DDRx, PORTx и PINx. Компаратор с гистерезисом устраняет дребезг контактов, обеспечивает строгие фронты сигнала, подаваемого на Pxn. Блок синхронизации отодвигает фронты сигнала в допустимые периоды так, чтобы они оказались между фронтами системного тактового импульса.

Приведем пример рабочего кода. Предполагается, что на выводы порта А подключены кнопки по схеме рис. 4, а на выводы порта Б – светодиоды по схеме рис. 7.

```
#include <avr/io.h> // для определения DDRx, PORTx, PINx
int main()
{
    // конфигурирование портов
    DDRA = 0b00000000; // порт A на ввод
    PORTA = 0b11110000; // включить подтягивающие резисторы порта A
    DDRB = 0b11111111;  // порт B на вывод

    // операции ввода/вывода
    while (1)
    {
        char port_value = PINA; // чтение текущего значения порта A
        if (port_value == 0b11111111)
            PORTB = 0b00000000;     // запись данных порт B
        else
            PORTB = 0b10101010;     // запись данных порт B
    }
    return 0;
}
```

Внимательно прочитайте комментарии в коде. Конфигурирование портов должно быть понятно из сказанного выше. Бесконечный цикл while (цикл выполняется бесконечно, поскольку условие продолжения всегда истинно) необходим, поскольку опрос порта требуется производить в непрерывном режиме (Подумайте, что произойдет, если операторы в теле цикла исключить из цикла).

С точки зрения пользователя код будет работать следующим образом: пока нет нажатых кнопок (т.е. port_value == 0b11111111), светодиоды будут гореть. Как только будет нажата и удерживаться любая кнопка (т.е. port_value != 0b11111111), часть светодиодов будет потушена.

Техника программирования при работе с портами ввода-вывода.

Регистрация нажатия кнопки

При работе рассмотренного выше кода микросхема зажигает светодиоды, только пока кнопка удерживается нажатой. В реальных задачах обычно требуется «помнить» о факте нажатия после самого нажатия. В данном случае можно оставить зажженным нужный диод. Чтобы добиться такого поведения, потребуется программно фиксировать момент нажатия кнопки.

Рассмотрим код:

```
char prev_value;
while (1)
{
    char curr_value = PIND;
    /* тело цикла */
    prev_value = curr_value;
}
```

В этом коде в теле цикла имеется текущее и предыдущее значение порта D в переменных curr_value и prev_value соответственно. Эти переменные позволяют зафиксировать момент нажатия кнопки.

Допустим, осуществляется обработка нажатия кнопки, подключенной к выводу 4 порта D, при этом остальные кнопки не нажимаются. Четвертый бит регистра PIND изменяется, а остальные биты всегда равны единице, т.к. кнопки не нажимаются. Тогда временная диаграмма примет вид рис. 9.

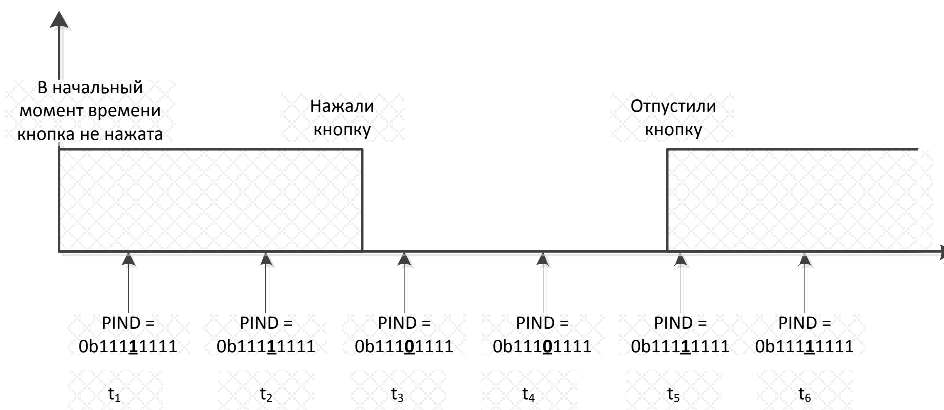


рис. 9. Временная диаграмма состояния порта Pxn при нажатии кнопки

В момент нажатия кнопки (момент времени t_3) переменная curr_value равна 0b11101111, а prev_value равна 0b11111111. Для того, чтобы зафиксировать момент нажатия кнопки 4, необходимо сравнить текущее и предыдущее значение соответствующего бита регистра PIND. Текущий бит равен 0 при значении предыдущего бита равном 1. Такое сравнение реализуется с логической фунцкией, соответствующей таблице истиности.

Бит	**Бит**	**Результат**	**Пояснение**

prev_value, отвечающий за кнопку	**curr_value, отвечающий за кнопку**		
1	0	1	Кнопка только что нажата
0	0	0	Кнопка нажата и удерживается
1	1	0	Кнопка не нажата
0	1	0	Кнопку только что отпустили

Выделим нужные нам значения битов из переменных curr_value и prev_value:

```
char curr_button_value = curr_value & 0b00010000;
char prev_button_value = prev_value & 0b00010000;
```

Для тех битов, где значение константы 0b00010000 равно нулю результат побитового И всегда будет равен нулю, независимо от значения этих битов в переменной curr_value. Для единственного, не равного нулю бита в константе 0b00010000, результат побитового И будет зависеть от величины этого бита в curr_value. Таким образом, значения переменных curr_button_value, prev_button_value могут быть равны либо 0b00000000 (кнопка нажата), либо 0b00010000 (кнопка не нажата). Сразу после нажатия кнопки (момент времени t_3), переменная curr_button_value равна 0b00000000, а переменная prev_button_value равна 0b00010000. Тогда код проверки нажатия кнопки будет выглядеть следующим образом:

```
if (curr_button_value == 0b00000000 && prev_button_value == 0b00010000)
    // мы здесь, если кнопка нажата
```

Убедитесь, что это условие действительно реализует требуемую таблицу истинности. Учитывая рассмотренный выше код, получим:

```
PORTD = 0b00000000; // включить светодиоды при запуске программы
char prev_value = 0xFF;
while (1)
{
    char curr_value = PIND;
    char curr_button_value = curr_value & 0b00010000;
    char prev_button_value = prev_value & 0b00010000;
    if (curr_button_value == 0b00000000 && prev_button_value == 0b00010000)
    {
        PORTD = 0b11111111; // выключить светодиоды
    }
    prev_value = curr_value;
}
```

Единственное, что в этом коде нового – инициализация переменной prev_value значением 0xFF. Как мы знаем, переменная prev_value является автоматической и память под нее выделяется на стеке, причем заранее неизвестно в какой ячейке стека, поэтому исходное значение переменной prev_value может оказаться любым.

Сделаем окончательную доводку кода. Если трактовать переменные curr_button_value, prev_button_value как логические, то для краткости можно записать этот код так:

```
if (!curr_button_value && prev_button_value)
    // мы здесь, если кнопка нажата
```

Константа 0b00010000 несколько раз используется в коде и при этом может быть изменена, если нужно будет обрабатывать другую кнопку. Поэтому целесообразно определить именованную константу

```
#define BUTTON_MASK 0b00010000
```

и заменить все константы 0b00010000 на BUTTON_MASK.

Кроме того, совсем необязательно заводить отдельные переменные curr_button_value и prev_button_value, можно записать:

```
if ((~curr_value & BUTTON_MASK) && (prev_button & BUTTON_MASK))
     // мы здесь, если кнопка нажата
```

Снова соберем все вместе и получим код хотя и короткий, но с большим числом тонкостей, описанных выше.

```
PORTD = 0b00000000; // включить светодиоды при запуске программы
char prev_value = 0xFF;
while (1)
{
    char curr_value = PIND;
    if (!(curr_value & BUTTON_MASK) && (prev_button & BUTTON_MASK))
    {
        PORTD = 0b11111111; // выключить светодиоды
    }
    prev_value = curr_value;
}
```

Регистрация нажатий произвольного набора кнопок

Часто требуется анализировать несколько кнопок. Тогда необходимо проверять переход из единицы в нуль для всех битов регистра PIND, т.е. реализовать таблицу истинности, приведенную выше, для всех битов порта. Убедимся, что очень простое выражение

```
char buttons = ~curr_value & prev_value;
```

реализует требуемую функцию для всех битов. Результат выражения (переменная buttons) будет содержать логические единицы для битов, соответствующих нажатиям кнопок. Например, необходимо различать 3 кнопки: пуск (нулевой вывод порта), стоп (вывод 1), пауза (вывод 2). Код для обработки нажатий этих кнопок будет выглядеть следующим образом:

```
#define BUTTON_PLAY  0b00000001
#define BUTTON_STOP  0b00000010
#define BUTTON_PAUSE 0b00000100
...
char prev_value = 0xFF;
while (1)
{
    char curr_value = PIND;
    char buttons = ~curr_value & prev_value;
    prev_value = curr_value;
    if (buttons & BUTTON_PLAY)
    {
        // нажали Play
    }
    if (buttons & BUTTON_STOP)
    {
        // нажали Stop
    }
    if (buttons & BUTTON_PAUSE)
    {
        // нажали Pause
    }
}
```

Обратите внимание, что обновление переменной prev_value можно осуществлять сразу после присваивания переменной buttons, т.к. дальше по коду она не используется. Если этот код не ясен, пора подумать и перечитать текст внимательнее.

Оформим код определения нажатия кнопок в отдельную функцию:

```
char detect_buttons()
{
    static char previous_port = 0xFF;
    char current_port = PIND;
    char buttons = ~current_port & previous_port;
    previous_port = current_port;
    return buttons;
}
```

Подчеркнем, что статической переменной previous_port значение 0xFF присваивается только один раз при запуске программы, при этом переменная сохраняет значение между вызовами функции detect_buttons, т.к. в отличие от автоматических переменных статическим переменным всегда соответствует одна и та же ячейка памяти.

Эту переменную можно было объявить глобально, однако переменная изменяется только в функции detect_buttons, поэтому с точки зрения читаемости кода лучше объявить ее внутри функции. Благодаря использованию функции код становится очень простым:

```
int main()
{
    while (1)
    {
        char buttons = detect_buttons();
        if (buttons & BUTTON_PLAY)
        {
            // нажали Play
        }
        if (buttons & BUTTON_STOP)
        {
            // нажали Stop
        }
        if (buttons & BUTTON_PAUSE)
        {
            // нажали Pause
        }
    }
}
```

Бегущий светодиод

Рассмотрим реализацию бегущего светодиода с самым простым законом его «бега»: перемещение из начального положения (первый светодиод) в последнее (8-й светодиод), из последнего обратно в начальное и так далее.

```
DDRA  = 0xFF;
PORTA = 0b11111110;
while (1)
{
    PORTA = PORTA << 1;
    _delay_ms(1000);
}
```

Библиотечная функция _delay_ms реализует паузу на заданное количество миллисекунд и определена в заголовочном файле util/delay.h. При выполнении программы значения порта А будут равны:

0b11111110
0b11111100
0b11111000
…
0b10000000
0b00000000

Т.е. вместо бегущего светодиода получится последовательное включение 1, 2, 3, …, 8 светодиодов. В действительности требуется такой закон:

0b11111110
0b11111101
…
0b10111111
0b01111111
0b11111110
0b11111101
..

Для этого достаточно в приведенном коде заменить операцию сдвига PORTA << 1 на операцию циклического сдвига (рис. 10). Операцию циклического сдвига реализует выражение:

PORTA << 1 | PORTA >> 7;

Цикличность обеспечивается за счет сдвига PORTA >> 7: младшие семь разрядов регистра PORTA отбрасываются, и в младший разряд результата записывается старший разряд регистра PORTA.

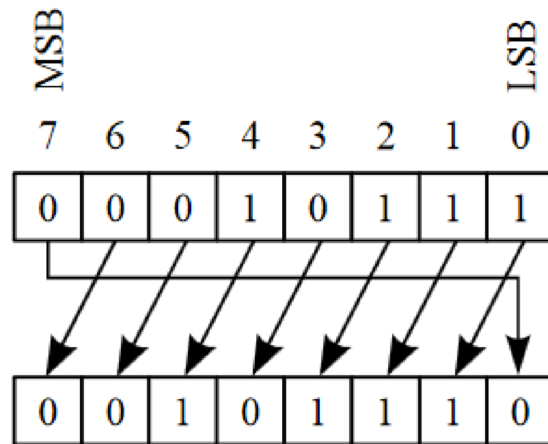


рис. 10. Операция циклического сдвига

Усложним задачу. Пусть, теперь есть два бегущих диода, которые сдвигаются на 2 разряда. Тогда сначала в регистр PORTA следует записать 0b11111100, и циклический сдвиг осуществлять на 2 разряда:

```
PORTA << 2 | PORTA >> 6;
```

Рассмотрим более сложные законы смены состояния светодиодов, которые трудно реализовать с помощью циклических сдвигов. В этом случае нужно записать последовательность из n значений порта в массиве, а затем выводить их из массива в циклической последовательности, т.е. индекс элемента массива, который в данный момент выводится в порт, изменяется от 0 до n-1, затем переход в 0, опять до n-1 и так далее. Для удобства определим именованную константу для размера массива.

```
#define PORT_VALUES_COUNT 4
char port_values[] = {0b11110101, 0b11111010, 0b01011111, 0b10101111};
int main()
{
    DDRA = 0xFF;
    PORTA = 0b00000001;
    char index = 0;
    while (1)
    {
        PORTA = port_values[index];
        index = (index + 1) % PORT_VALUES_COUNT;
        _delay_ms(1000);
    }
}
```

– Докажите, что переменная index действительно принимает значения от 0 до n-1.

- Можно ли исправить первые две строчки тела цикла while таким образом:

  PORTA = port_values[index++];

  index %= PORT_VALUES_COUNT;

Бегущий светодиод с обработкой нажатий кнопок

Попробуем добавить обработку нажатия кнопок. Представим код с бегущим светодиодом:

```
int main()
{
    char prev_value = 0xFF;
    while (1) {
        PORTA = PORTA << 1 | PORTA >> 7;
        _delay_ms(1000);

        char curr_value = PIND;
        char buttons = ~curr_value & prev_value;
        prev_value = curr_value;
        if (buttons & BUTTON_PLAY)
             ...

    }
}
```

Код довольно прост. Он скомпонован из предыдущих решений, но, к сожалению, не достаточно хорош, т.к. опрос кнопок осуществляется с периодом 1000 мс, т.е. 1 сек (на временной диаграмме t3 – t2 = 1000 мс).

Недостаток такого кода – во время паузы нельзя обрабатывать кнопки. Программа, имеющая такую особенность, в технологии программирования называется однопоточной.

Если снизить паузу с 1000 до 100, то время реакции уже будет неплохое, но диод будет бегать слишком быстро. Это наводит на мысль, что если добиться того, что на один шаг светодиода осуществлять несколько опросов кнопок, то можно добиться ожидания времени с параллельной обработкой нажатий кнопок.

```c
int main()
{
    char prev_value = 0xFF;
    while (1) {
        PORTA = PORTA << 1 | PORTA >> 7;
        for (char index = 0; index < 10; index++)  {
            char curr_value = PIND;
            _delay_ms(100);
            char buttons = ~curr_value & prev_value;
            prev_value = curr_value;
            if (buttons) // если нажата хотя бы одна кнопка
                break;
        }
        // обработка кнопок
        if (buttons & BUTTON_PLAY)
            ...
    }
}
```

Цикл for в приведенном коде работает следующим образом:

- ожидает 100 мс, если не было нажатий кнопки.
- если кнопки нажаты, цикл заканчивается досрочно

Заметим, что продолжительность временного кванта можно еще уменьшить, например до 10 мс, не забыв исправить пределы изменения переменной index в цикле for.

Выделим приведенный цикл в отдельную функцию:

```c
char detect_buttons_in_delay(int delay)
{
    static char previous_port = 0xFF;
    int quant_count = delay / 10;
    for (int quant_number = 0; quant_number < quant_count; ++quant_number)
    {
        char current_port = PIND;
        char buttons = ~current_port & previous_port;
        previous_port = current_port;
        if (buttons)
            return buttons;
        _delay_ms(10);
    }

    return 0x00;
}
```

При использовании функции основной код становится гораздо проще:

```
int main()
{
    char prev_value = 0xFF;
    while (1)
    {
        PORTA = PORTA << 1 | PORTA >> 7;
        char button = detect_buttons_in_delay(1000);
        if (buttons & BUTTON_PLAY)
            ...
    }
}
```

Общая структура программы с обработкой ввода/вывода

Посмотрим на изложенное выше с более общих позиций. Приведенный код может быть использован для мониторинга любых событий, появление которых можно зарегистрировать с помощью портов ВВ. Помимо события нажатия кнопки, в цикле можно, например, опрашивать флаг завершения аналого-цифрового преобразования микросхемой АЦП или флаг завершения счета встроенного таймера. Общая структура кода для всех подобных задач имеет вид:

```
int main()
{
    Конфигурирование портов
    while (1)
    {
        Опрос портов ввода/вывода
        Обработка данных
        Вывод данных
    }
    return 0;
}
```

VI. Побитовые и логические операции

Побитовые операции – мощный и изящный инструмент, полезный при низкоуровневом программировании аппаратных средств. Однако мощный инструмент требует правильного использования. Рассмотрим сущность

побитовых операции, их свойства и применение при работе с портами ввода/вывода.

Определим **побитовые** операции, сравнив их с родственными **логическими** операциями (*И, ИЛИ*). При вычислении логической операции на вход подаются два булевых операнда, на выходе получаем булево значение, например:

1 И 0 = 0

1 ИЛИ 0 = 1

Каждая **одиночная** побитовая операция представляет собой **серию** логических операции над парами операндов. Пример:

```
  A = 10101
&
  B = 00111
-------------
      00101
```

Здесь выполнено побитовое *И* – серия логических *И* для пар битов пятиразрядных двоичных чисел A и B. (Знак «&» - амперсанд обозначает побитовое *И* в языке Си). Аналогичным образом, побитовое *ИЛИ* (знак вертикальная черта «|» в Си) представляет собой серию логических *ИЛИ*:

```
  A = 110011
|
  B = 100111
--------------
      110111
```

Рассмотрим важные в дальнейшем свойства логических *И*, *ИЛИ*. Рассмотрим выражение x && 1 при различных значениях x:

$$x \ \&\& \ 1 = \begin{cases} 0, \text{если } x = 0 \\ 1, \text{если } x = 1 \end{cases}$$

Проще говоря, логическое *И* с единицей не меняет x:

1°) $x \ \&\& \ 1 = x$

Рассмотрим теперь выражение x && 0:

$$x \ \&\& \ 0 = \begin{cases} 0, \text{если } x = 0 \\ 0, \text{если } x = 1 \end{cases}$$

То есть логическое *И* с нулем всегда дает 0, независимо от x.

2°) $x \,\&\&\, 0 = 0$

Аналогично анализируя результат выражения при различных значениях x, получим правила для логического *ИЛИ*:

3°) $x \,||\, 1 = 1$

4°) $x \,||\, 0 = x$

(докажите это самостоятельно).

Основываясь на изложенном, научимся решать задачи:

- Задача 1. Проверка отдельного бита числа.
- Задача 2. Установить нужный бит числа в 0, остальные оставить неизменными.
- Задача 3. Установить заданный бит числа в 1, остальные не трогать.

Необходимость проверки отдельного бита (задача 1) возникает при обработке нажатий кнопки, подключенной к одной из ножек порта ввода/вывода микроконтроллера. Допустим, нас интересует кнопка, подключенная к ножке PD2 (порт D, вторая ножка). Чтобы программно определить, что кнопка нажата, надо определить состояние второго бита регистра PIND. Рассчитаем результат выражения PIND & 0b00000100 при всевозможных значениях PIND (учтем свойства 1° и 2°):

$$\begin{array}{rl} & \text{PIND} = b_7b_6b_5b_4b_3b_2b_1b_0 \\ \& & 0\;0\;0\;0\;0\;1\;0\;0 \\ \hline & 0\;0\;0\;0\;0\;b_2\;0\;0 \end{array}$$

Второй бит результата равен второму биту PIND, остальные биты PIND не влияют на результат. Таким образом:

при нажатой кнопке: PIND & 0b00000100 = 0b00000000

при отпущенной кнопке: PIND & 0b00000100 = 0b00000100 ≠ 0

Учитывая это, код проверки положения кнопки можно написать так:

```
int year;
if (PIND & 0b00000100)
    year = 1824;
else
    year = 1799;
```

Очевидно, в случае нажатой кнопки в переменной year год рождения гениального поэта, а при отпущенной кнопке – великого писателя.

Изменять значение одного бита, не трогая другие (задачи 2, 3) нужно, если мы хотим зажечь (или погасить) один из светодиодов, подключенный к порту ввода/вывода. Пусть нас интересует диод, подключенный к ножке PB1. Чтобы его зажечь, надо записать 0 в 1-й бит регистра PORTB, оставив неизменными. Рассмотрим фрагмент кода в языке Си:

```
PORTB = PORTB & 0b11111101;
```

Рассмотрим, что окажется в PORTB после выполнение операции:

$$
\begin{array}{rl}
\& & PORTB = b_7b_6b_5b_4b_3b_2b_1b_0 \\
 & 1\,1\,1\,1\,1\,1\,0\,1 \\
\hline
 & b_7b_6b_5b_4b_3b_2 0\, b_0
\end{array}
$$

Результат операции PORTB & 0b11111101 совпадает с исходным значением PORTB за исключением 1-го бита (снова вследствие свойств 1° и 2°).

Если мы теперь хотим погасить диод на PB1, то нам надо записать в бит b_1 единицу, как и раньше не испортив (т.е. оставив неизменными) остальные биты. Заметим, что это нельзя сделать с помощью побитового *И* (самостоятельно рассмотрите результат выражения PORTB & 0b00000010, которое так и тянет использовать). Рассмотрим, как это достигается побитовым *ИЛИ*:

```
PORTB = PORTB | 0b00000010;
```

Рассмотрим результат побитовой операции (учтем свойства 3° и 4°):

$$
\begin{array}{rl}
| & PORTB = b_7b_6b_5b_4b_3b_2b_1b_0 \\
 & 0\,0\,0\,0\,0\,0\,1\,0 \\
\hline
 & b_7b_6b_5b_4b_3b_2 1\, b_0
\end{array}
$$

Результат отличается от начального значения PORTB только первым битом.

Отметим важную особенность побитовых операций: если требуется установить бит в 0, то надо использовать побитовое *И*; если надо установить бит в 1, используется побитовое *ИЛИ*.

Литература

1. Уэйкерли Дж. Ф. Проектирование цифровых устройств, том 1 М.: Постмаркет, 2002. – 544с
2. Самарский А.А., Гулин А.В. Численные методы: Учеб. Пособие для вузов. – М.: Наука. Гл. ред. физ-мат. лит., 1989. – 432 с.
3. IEEE-754. Стандарт двоичной арифметики с плавающей точкой [Электронный ресурс] // Режим доступа: http://www.softelectro.ru/ieee754.html.– (Дата обращения: 30.08.2015).

УЧЕБНОЕ ПОСОБИЕ

ЮЖАНИН ВИКТОР ВЛАДИМИРОВИЧ

ТИПЫ ДАННЫХ И ПОРТЫ ВВОДА-ВЫВОДА МИКРОКОНТРОЛЛЕРОВ

Редактор ___________________

Художник-график ___________

Технический редактор _______

Корректор _________________

Компьютерная верстка_______

Подписано в печать __________. Формат 60х90/16. Усл. п.л. 1,25.

Гарнитура «Таймс». Печать офсетная. Тираж 150 экз. Заказ №231

Издательский центр

РГУ нефти и газа имени И.М. Губкина

119991, Москва, Ленинский проспект, 65

Тел./факс: __________