Лекция 5. 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 (или пинов, транскрипция англ. pin) 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.

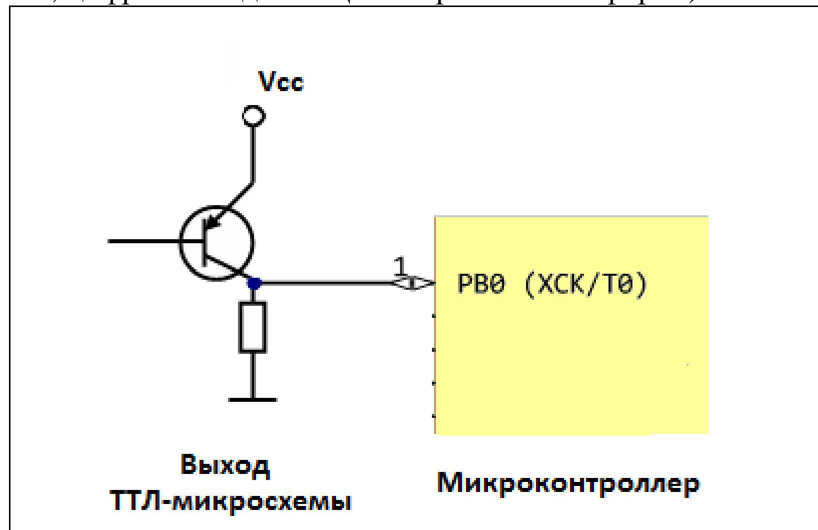


рис. 1 Активный источник сигнала: выход ТТЛ логики

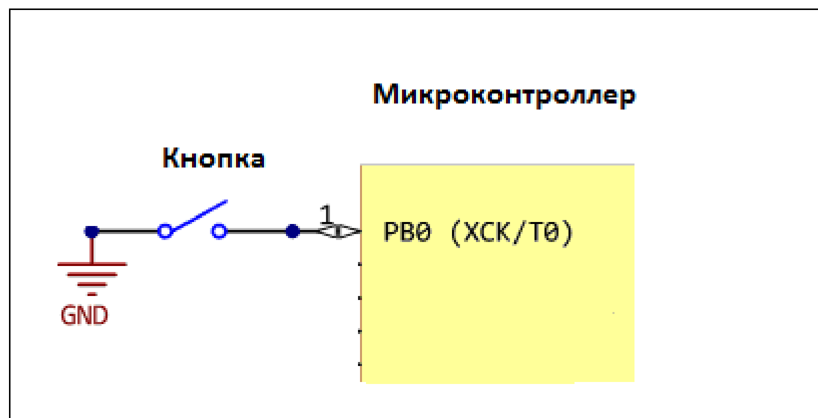


рис. 2 Пассивный источник сигнала: кнопка

Рассмотрим более подробно работу кнопки (рис. 2), 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 питания Vcc через специальный резистор,

называемый подтягивающим резистором. Резистор нужен для избегания короткого замыкания при нажатой кнопке.

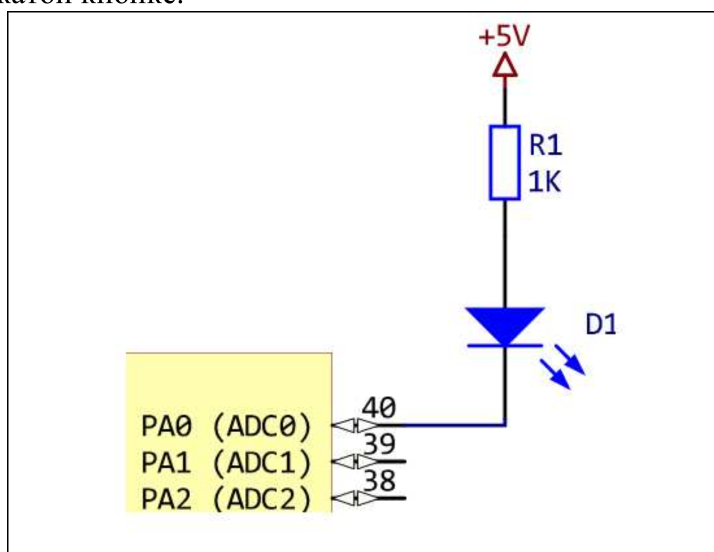


рис. 3 Потребитель логического сигнала:
светодиод (схема подключения 1)

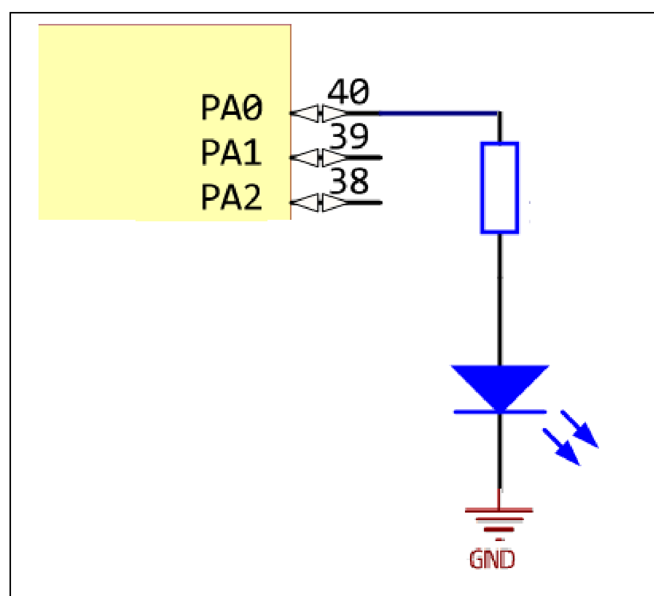


рис. 4 Потребитель логического сигнала:
светодиод (схема подключения 1)

Из сказанного 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 с поддержкой ШИМ и др.) в микроконтроллерах ATMega осуществляется через регистры ввода/вывода. Регистры ввода/вывода отображаются в память 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 DDR 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.

Принципиальная схема одного из выводов порта ВВ показана на рис. 5 и рис. 6. Из рис. 5 видно как обеспечена описанная выше функциональность регистров DDRx, PORTx и PINx. Компаратор с гистерезисом устраняет дребезг, обеспечивает строгие фронты сигнала, подаваемого на Pxn. Блок синхронизации отодвигает фронты сигнала в допустимые периоды так, чтобы они были между фронтами системного тактового импульса. На рис. 6 видно показаны защиты выводов портов микроконтроллера от слишком высоких и отрицательных напряжений (диоды VD1 и VD2).

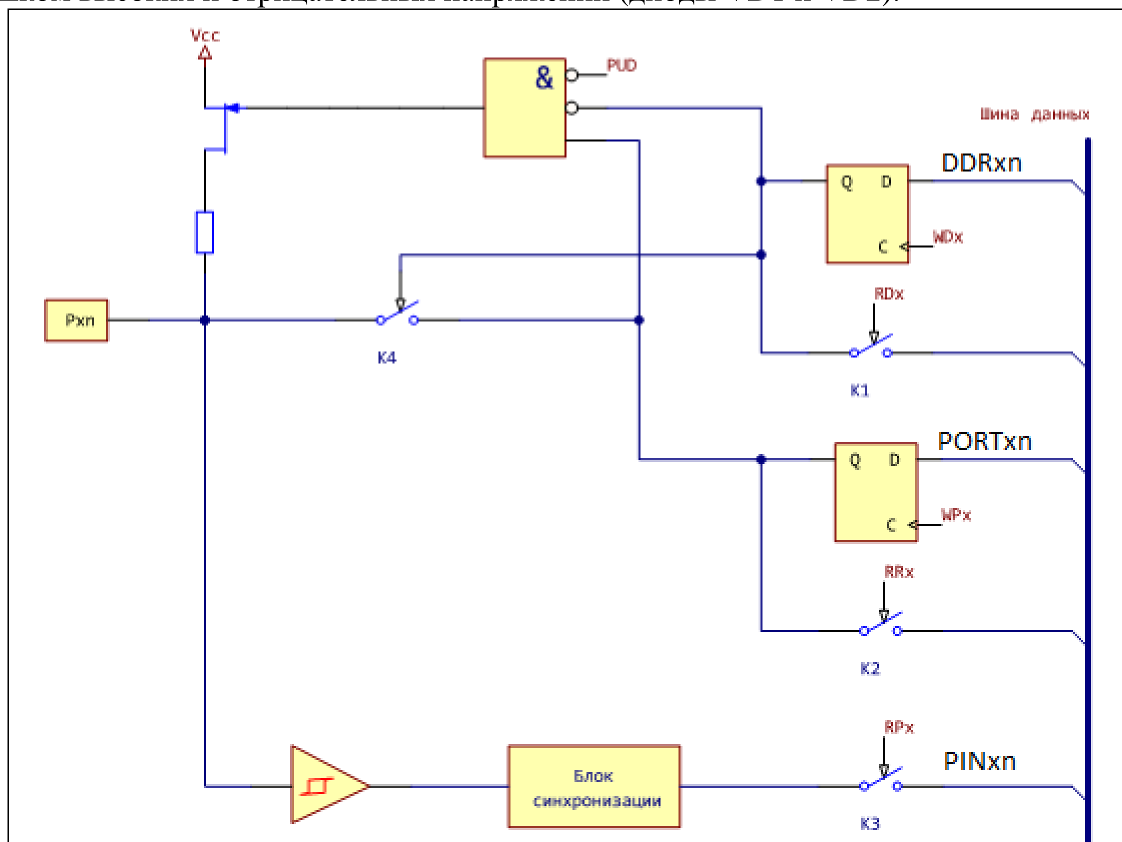


рис. 5 Принципиальная схема вывода порта

PUD – pull-up disable, отключение подтягивающих резисторов для всех портов при PUD = 1; **Pxn** – n-й вывод порта x (ножка микроконтроллера); **DDRxn** – n-й бит регистра DDRx; PORTxn – n-й бит регистра PORTx; **PINxn** – n-й бит регистра PINx

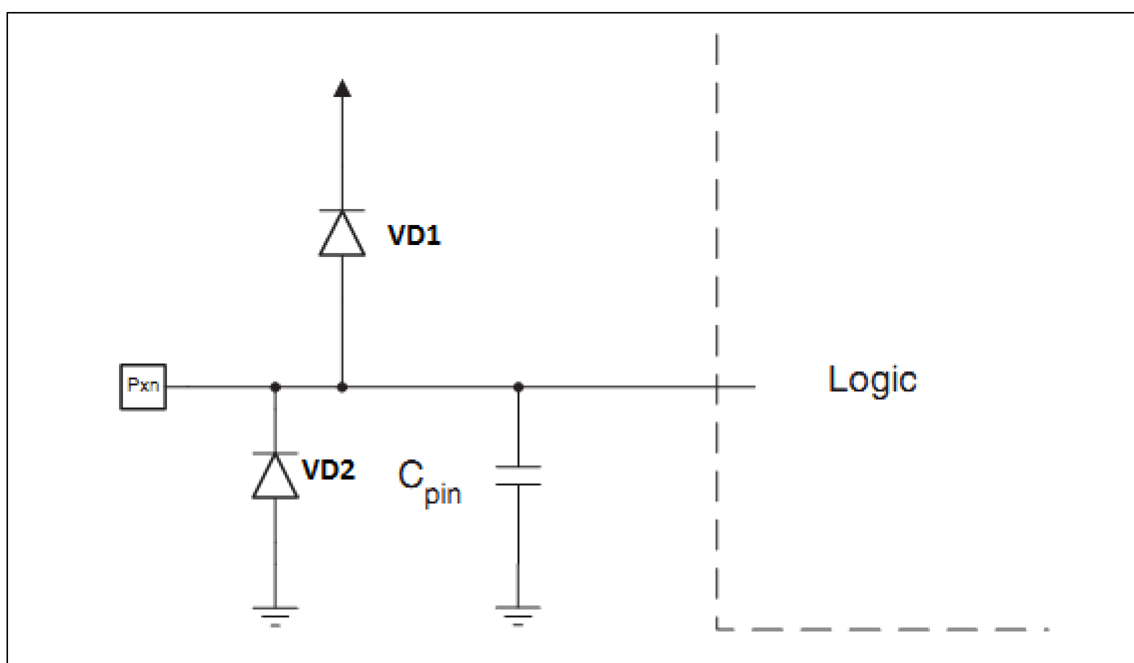


рис. 6 Входной каскад вывода порта

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

```
#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;
}
```

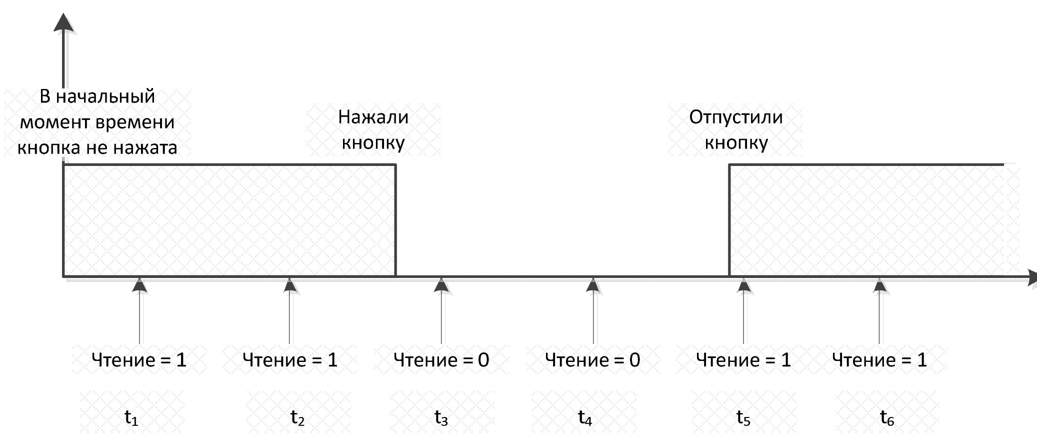
Внимательно прочитайте комментарии в коде. Конфигурирование портов должно быть понятно из сказанного выше. Бесконечный цикл while (цикл выполняется бесконечно, поскольку условие продолжения всегда истинно) необходим, поскольку опрос порта необходимо производить в непрерывном режиме (Подумайте, что произойдет, 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 == 0xFF), светодиоды будут гореть. Как только будет нажата и удерживаться любая кнопка (т.е. port_value != 0xFF), часть светодиодов будет потушена.

Техника программирования при работе с портами ввода-вывода.

Регистрация нажатия кнопки

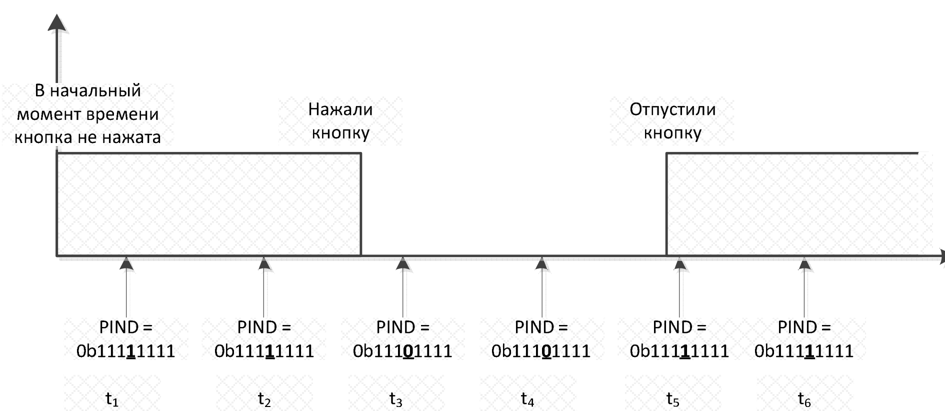
Временная диаграмма, из которой следует, что необходимо постоянно опрашивать порт, причем надо обеспечить считывание как текущего, так и предыдущего значения. Нельзя считать только один бит, приходится читать весь порт.



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



В момент нажатия кнопки (момент времени t_3) переменная curr_value равна 0b11101111, а prev_value равна 0b11111111. Для 4 бита регистра PIND необходимо реализовать функцию логическую функцию со следующей таблицей истинности:

Бит prev_value, отвечающий за кнопку	Бит curr_value, отвечающий за кнопку	Результат	Смысл состояния
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
```

Убедимся, что это условие действительно реализует требуемую таблицу истинности.
Теперь можно собраться все вместе:

```
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 нескольких 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 (Рисунок 1). 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.

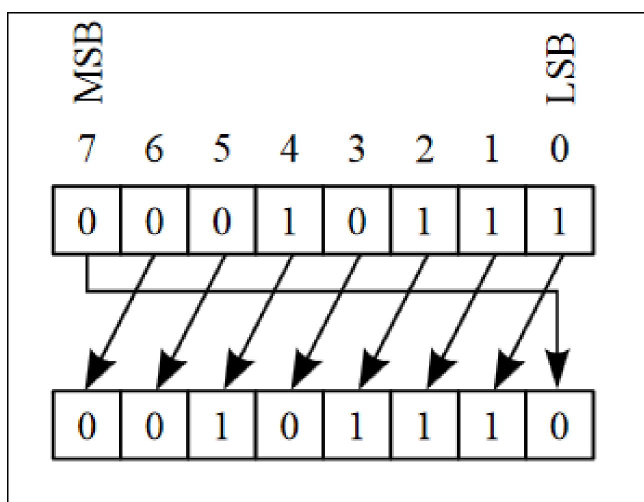


Рисунок 1. Операция циклического сдвига

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. Удобно определить 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 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 просто 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).

Временная диаграмма. Показать, что может оказаться нужно держать кнопку довольно долго.

Недостаток такого кода – нельзя обрабатывать кнопки. Во время паузы обработку вести нельзя. 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 идею что если добиться того, что на один шаг светодиода осуществлять несколько опросов кнопок, то можно добиться ожидания времени с параллельной обработкой нажатий кнопок.

```
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 1000 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:

```
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;
}
```