Начальный курс программирования МК ATMEL AVR.

Цель курса:

дать начальные понятия о цифровой технике и навыки программирования на си. Материал рассчитан на тех, кто не имеет ни малейшего понятия о программировании, цифровой логике. Все представлено и описано простым языком и предложениями.

Для освоения МК необходимо время от 2х часов ежедневно, терпение и порядка 300мб места на диске. Покупать МК не нужно, но желательно. По началу ваш код будет большой. Это требует большого объема программной памяти. Для этого лучше всего подходит современный atmega16, который имеет 16к памяти под вашу программу, что на начальном этапе должно хватить с головой. Цена его до 5 баксов.

Требования

Даташит , Симулятор AVR Studio 4 ,
Симулятор с визуальными компонентами PROTEUS Рабочая версия 6.7 сп3
(прокрутите страницу вниз, нажмите free, подождите около минуты, введите появившиеся буквы в поле рядом. Пароль на архив www.sonsivri.com)
Компилятор, отладчик, редактор кода, программатор WINAVR
Программатор (чертежи в формате eagle 4.13 прилагаются)
Калькулятор значений для регистров , Продвинутый калькулятор Hexelon
Калькулятор для работы с логическими операциями
Среда разработки приложений для обычного компьютера Dev c++
Драйвер ядра для доступа к порту LPT DLPortIO
Программы для работы с этим портом

Логические анализаторы для этого порта и соответствующий софт 1 , 2 , 3
Для версии ДОС нужен отдельный компьютер с быстрым процессором (от 400мгц) и самим досом.

Кратко о семействе AVR

Высокопроизводительные RISC микроконтроллеры семейства AVR
В отличие от MICROCHIP, компания ATMEL Corp. — один из мировых лидеров в производстве широкого спектра микросхем энергонезависимой памяти, FLASH-микроконтроллеров и микросхем программируемой логики, взяла старт по разработке RISC-микроконтроллеров в середине 90-х годов, используя все свои технические решения, накопленные к этому времени.

Концепция новых скоростных микроконтроллеров была разработана группой разработчиков исследовательского центра ATMEL в Норвегии, инициалы которых затем сформировали марку AVR. Первые микроконтроллеры AVR AT90S1200 появились в середине 1997 г.и быстро снискали расположение потребителей.

AVR-архитектура, на основе которой построены микроконтроллеры семейства AT90S, объединяет мощный гарвардский RISC-процессор с раздельным доступом к памяти программ и данных, 32 регистра общего назначения, каждый из которых может работать как регистр- аккумулятор, и развитую систему команд фиксированной 16-бит длины. Большинство команд выполняются за один машинный такт с одновременным исполнением текущей и выборкой следующей команды, что обеспечивает производительность до 1 MIPS на каждый МГц тактовой частоты.

32 регистра общего назначения образуют регистровый файл быстрого доступа, где каждый регистр напрямую связан с АЛУ. За один такт из регистрового файла выбираются два операнда, выполняется операция, и результат возвращается в регистровый файл. АЛУ поддерживает арифметические и логические операции с регистрами, между регистром и константой или непосредственно с регистром.

Регистровый файл также доступен как часть памяти данных. 6 из 32-х регистров могут использоваться как три 16-разрядных регистра-указателя для косвенной адресации. Старшие микроконтроллеры семейства AVR имеют в составе АЛУ аппаратный умножитель.

Базовый набор команд AVR содержит 120 инструкций. Инструкции битовых операций включают инструкции установки, очистки и тестирования битов.

Все микроконтроллеры AVR имеют встроенную FLASH ROM с возможностью внутрисхемного программирования через последовательный 4-проводной интерфейс.

Периферия МК AVR включает: таймеры-счётчики, широтно-импульсные модуляторы, поддержку внешних прерываний, аналоговые компараторы, 10-разрядный 8-канальный АЦП, параллельные порты (от 3 до 48 линий ввода и вывода), интерфейсы UART и SPI, сторожевой таймер и устройство сброса по включению питания. Все эти качества превращают AVR-микроконтроллеры в мощный инструмент для построения современных, высокопроизводительных и экономичных контроллеров различного назначения.

Отличительные особенности:

- 8-разрядный высокопроизводительный AVR микроконтроллер с малым потреблением
- Прогрессивная RISC архитектура
- 130 высокопроизводительных команд, большинство команд выполняется за один тактовый цикл
- 32 8-разрядных рабочих регистра общего назначения
- Полностью статическая работа
- Производительность приближается к 16 MIPS (при тактовой частоте 16 МГц)
- Встроенный 2-цикловый перемножитель
- Энергонезависимая память программ и данных
- 16 Кбайт внутрисистемно программируемой Flash памяти
- Обеспечивает 1000 циклов стирания/записи
- Дополнительный сектор загрузочных кодов с независимыми битами блокировки
- Внутрисистемное программирование встроенной программой загрузки
- Обеспечен режим одновременного чтения/записи (Read-While-Write)
- 512 байт EEPROM
- Обеспечивает 100000 циклов стирания/записи
- 1 Кбайт встроенной SRAM
- Программируемая блокировка, обеспечивающая защиту программных средств пользователя

Встроенная периферия:

1. Два 8-разрядных таймера/счетчика с отдельным предварительным делителем, один с режимом сравнения
2. Один 16-разрядный таймер/счетчик с отдельным предварительным делителем и режимами захвата и сравнения
3. Счетчик реального времени с отдельным генератором
4. Четыре канала PWM
5. 8-канальный 10-разрядный аналого-цифровой преобразователь
6. 8 несимметричных каналов
7. 7 дифференциальных каналов (только в корпусе TQFP)
8. 2 дифференциальных канала с программируемым усилением в 1, 10 или 200 крат (только в корпусе TQFP)
9. Байт-ориентированный 2-проводный последовательный интерфейс
10. Программируемый последовательный USART
11. Последовательный интерфейс SPI (ведущий/ведомый)
12. Программируемый сторожевой таймер с отдельным встроенным генератором
13. Встроенный аналоговый компаратор

Специальные микроконтроллерные функции:

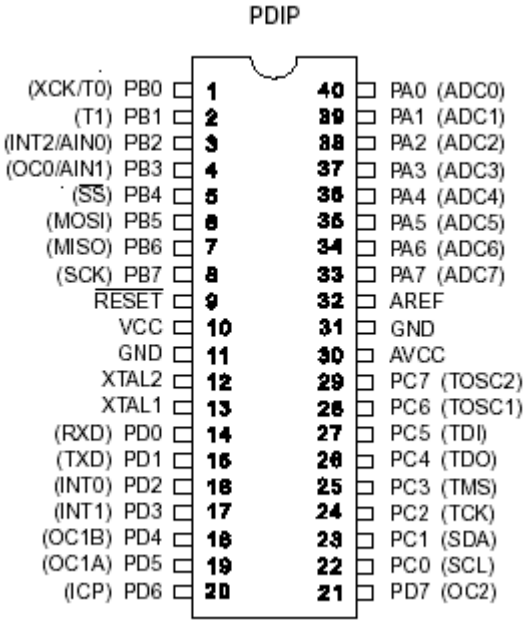
- Сброс по подаче питания и программируемый детектор кратковременного снижения напряжения питания
- Встроенный калиброванный RC-генератор
- Внутренние и внешние источники прерываний
- Шесть режимов пониженного потребления: Idle, Power-save, Power-down, Standby, Extended Standby и снижения шумов ADC

Выводы I/O и корпуса

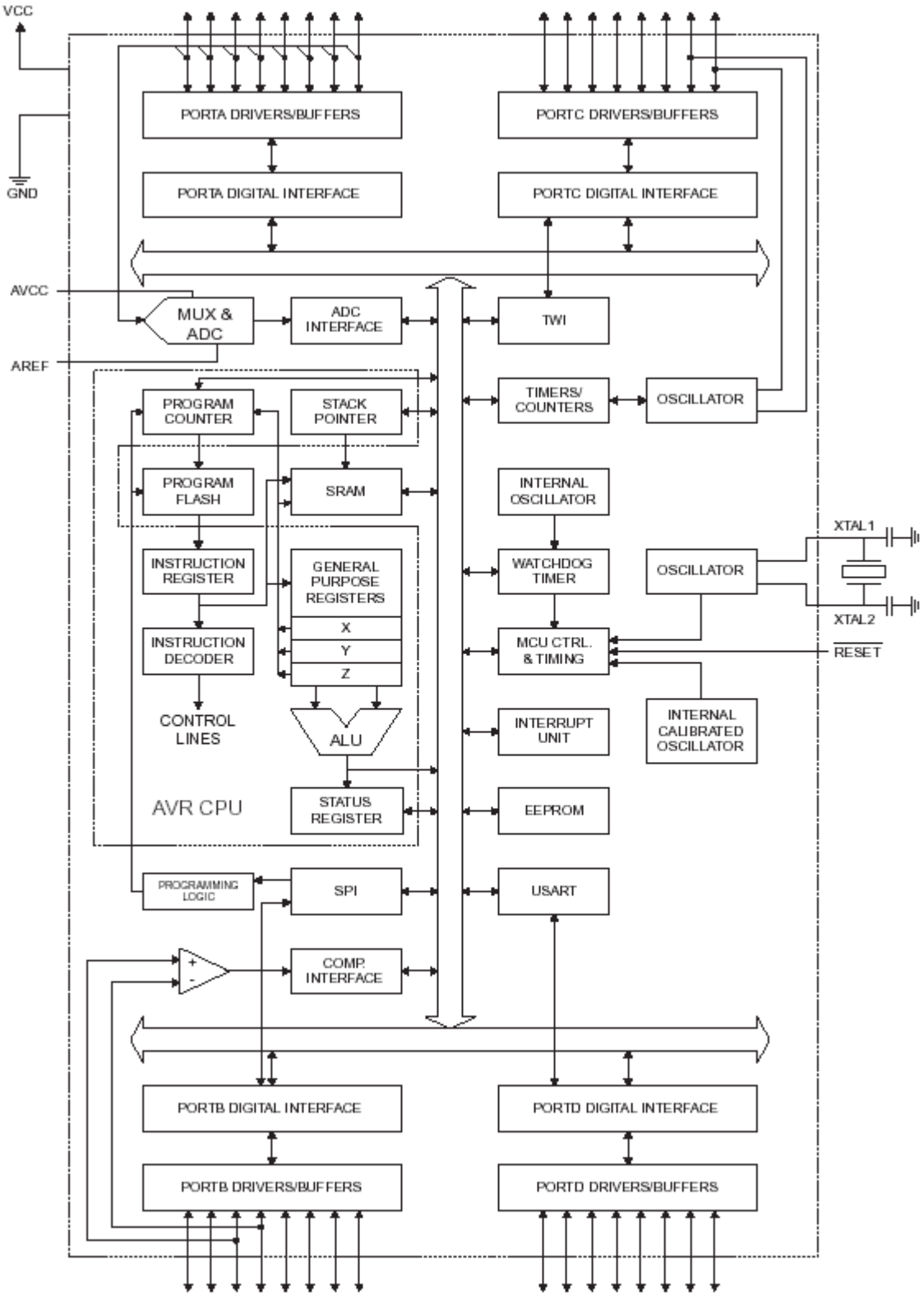
32 программируемые линии ввода/вывода
40-выводной корпус PDIP и 44-выводной корпус TQFP

Рабочая частота

0 - 8 МГц (ATmega16L)
0 - 16 МГц (ATmega16)



Блок- схема ATmega16



Как и что делать

Для начала не надо будет ничего из аппаратных компонентов. Потом, когда вы уже сможете писать небольшие программы, полезным будет подключить что-либо к лпт порту и посмотреть, как оно будет работать. Этот порт имеет 8 выходов, как и обычный мк. Во избежании повреждения выходов, желательно сделать небольшой драйвер на каких-либо мс типа 74244. Либо можно использовать старые ISA платы. Использовать переходник USB – LPT нежелательно. Особенно для программаторов.

В качестве подключаемого железа нам надо будет один индикатор LCD, один семи сегментный, 8 светодиодов и резисторы по 200ом к ним, 5 кнопок.

Для выполнения упражнений по передаче с последовательным интерфейсом понадобится еще микросхема 40094 (мс14094) – регистр с последовательной загрузкой.
Для подключения мк и компьютеру через последовательный порт, потребуется преобразователь уровней rs232-TTL - max232 или аналогичная (шнур от телефона).
Для отладки программ желательно иметь осциллограф. Еще лучше цифровой с памятью.

Т.к. программа для мк очень похожа на программу для компьютера, начнем с написания для последнего. В этом случае вам не надо будет держать еще открытый симулятор и тем более прошивать каждый раз мк.

Для начала стоит ознакомиться с цифровой техникой вообще: какие блоки обычно есть в таких устройствах, как они работают и можно ли их заменить на другие. А так же системы представления данных. Это особо важно, т.к. в дальнейшем это будет встречаться на каждом шагу.
Для начала подойдет калькулятор hexelon, а потом вы должны будете уметь представлять двоичные числа и конвертировать их в десятичные или шестнадцатеричные. Хотя, можно их будет записывать и в двоичном формате. Но 16-битное число будет содержать 16 знаков, вместо 4х в шестнадцатеричном.

Весь код в уроках 100% рабочий и тестируется. Некоторые уроки содержат файл проекта для соответствующего компилятора, все необходимые файлы с кодом и исполняемый .exe или .hex файлы.
Перед тем как сделать изменения в этих файлах, советую сделать полную копию всех уроков.

Системы счисления. Основы цифровой техники.

В самом начале компьютерной эры существовали аналоговые вычислительные устройства: механические, электрические и электронные.
Последние базировались на операционных усилителях и их свойствах складывать и вычитать амплитуды сигналов. Однако, такая техника имела один недостаток - неточность результатов, а так же устройств индикации - стрелочных микроамперметров. Каким бы точным ни был ОУ, у него всегда будет процент погрешности.
Современный же калькулятор за 10 баксов на 10 разрядов способен извлечь корень и возвести это же число в квадрат с точностью в 100% в пределах 10 знаков.

Для достижения такого результата система ввода данных была упрощена. Нужно лишь всего 2 состояния системы: 1 и 0. Лампочка либо горит, либо не горит. Гореть в вполнакала она не может никак. Ошибка ввода сведена к нулю.
Осталось теперь научиться представлять обычные для нас числа в виде последовательности нулей и единиц. Для этого есть специальные математические схемы.
Самая простая из них - перевод из двоичной в десятичную.
В цифровой технике операции производятся над т.н. 8, 16, 32, 64 и т.д. битными числами.

Бит - минимальная единица представления информации.
Один бит может быть либо логическая 1 либо логический 0.

Есть еще третье состояние Z, когда вывод как бы никуда не подключен и не влияет на схему. Так сделано в программаторе, предложенным на первой странице.
В режиме, когда не надо программировать, его выходы находятся в Z состоянии и не влияют на работу МК.

За логическую единицу обычно принимают напряжение в 5В для TTL либо оно равно напряжению питания в CMOS технике. Логический ноль может быть до 0.7В.

Для представления числа нам нужно как минимум восемь бит.
Например, обычная цифра 0 в десятичной системе будет выглядеть так: 0000 0000.
Как же узнать, сколько комбинаций можно представить из 8 бит?
Для этого есть формула:

$$2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0=128+64+32+16+8+4+2+1=256$$

Итого 256 комбинаций. Или же числа от 0 до 255. Больше число представить одним байтом нельзя. Для этого нам нужно добавить еще один байт. Такое число уже будет содержать 16 бит. Максимальных комбинаций из нулей и единиц будет: $2^{16}=65536$

Цифра 2 - это максимальное число состояний бита. Цифра в степени определяет порядковый номер бита в байте и придаем ему вес. Чем больше число, тем более высокий вес имеет бит. Есть старший бит MSB и младший LSB. В дальнейшем в даташитах вы будете встречаться с ними. Одни компоненты при их инициализации требуют передавать с MSB (слева направо), другие же наоборот.

Перевод из двоичной системы в десятичную

Возьмем наше число 10101001 и распишем его побитно в таблице.

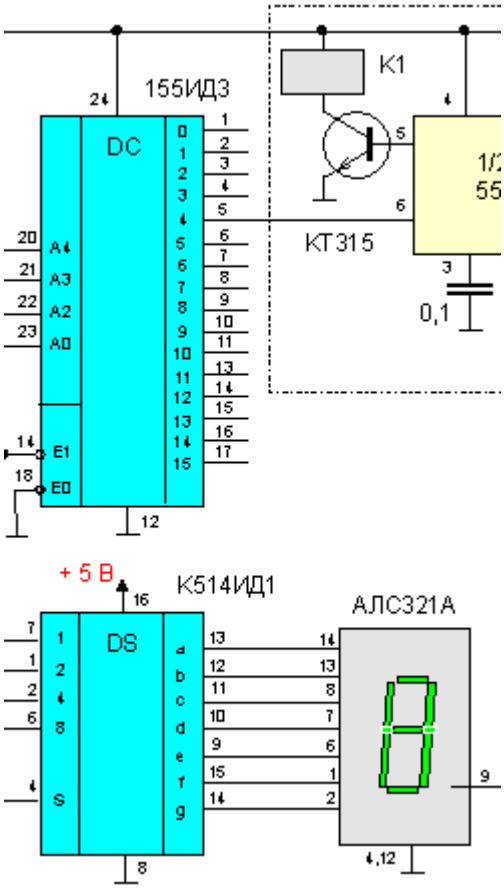
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	**0**	**1**	**0**	**1**	**0**	**0**	**1**
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128		32		8			1
169							

Там, где в двоичной системе стоит 0, этот разряд выкидываем, его "полезность" равна нулю.
Там, где стоит единица, берем эти десятичные значения и складываем друг с другом. Полученное число и будет искомое значение.
Другими словами: для нас число 169, а для компьютера это последовательность 10101001, причем читает он ее справа налево: сначала LSB, а потом MSB.
Не все процессоры работают в такой последовательности. Например моторолла читает их слева направо.

Все современные компиляторы имеют функции преобразования из одной системы в другую и вам не надо заботится об этом. Для ручной конвертации используйте hexelon.

BCD код

Этот код позволяет представить десятичное число. Он появляется на выходе любого двоично-десятичного счетчика. Если вы посмотрите на списки компонентов в программах для рисования схем, то там будет стоять нечто типа 4 Bit BCD Counter.
Такой счетчик переводит последовательности импульсов тактового генератора в числа. Иначе говоря - считает их. В МК так же есть счетчики 8 и 16 бит. Они отмеряют время, через которое должно что-либо произойти: опросить порт, зажечь светодиод, выполнить преобразование и т.д.
Давайте посмотрим, как импульсы преобразовываются в понятные нам цифры.



Порядковый номер	BCD код	16 ричный код
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Тут представлен 4х битный код счетчика, который считает до 15 и сбрасывается в нуль. Числа больше 9 представлены в виде букв. В МК счетчики считают до 255 и до 65535, хотя можно задать число, от которого начнется отсчет. При достижении конечного числа, счетчик в МК генерирует прерывание, а обычный дискретный - импульс переноса, который служит тактовым импульсом для счетчика следующего разряда. В компиляторах есть функция преобразования числа в BCD код, что позволит вам подключить к четырем выводам МК специальную микросхему-декодер. Есть 2 типа таких декодеров - в код для семи сегментного индикатора или же в виде обычных выводов 1-16 для управления реле или лампочками.

Типы данных

Для правильного выделения столь дефицитной памяти в мк, компилятору надо указать, какой же длины твои данные.
Самый популярный тип данных - целочисленный без знаковый байт. поесть обычные числа от 0 до 255.
Записывается это так: unsigned char <имя переменной>;
Например мы хотим хранить в переменной значение температуры. Для этого нам надо сначала прикинуть, в каком диапазоне может быть наша температура. Вообще она может быть отрицательной и положительной. Обычно все датчики с цифровым выходом представляют температуру в виде числа 0...255. Но это не значит, что 0 будет равно нулю градусов, а 255 - 255 градусов Цельсия.
Для того, чтобы сконвертировать это число в нормальную температуру, надо смотреть даташит на конкретный датчик. Там будет точная методика или формула.
А если нам надо хранить частоту приемника? Для УКВ это 5 десятичных разрядов, что на 2 разряда больше, чем может поместится в unsigned char.
Значит нам надо уже для переменной "частота" брать 16 бит. Такой тип называется unsigned int.

Тут надо сделать небольшое отступление. В данном примере частота представлена как целое число. Например 104.50МГц будет как 10450. Это правильно с точки зрения программирования и самого синтезатора частоты, т.к. он не понимает дробные числа и тогда все равно придется перед посылкой их в него конвертировать в целое. Для отображения на ЖК индикаторе достаточно будет поделить это число на 100 и передать результат в функцию, где ей указать, что передаваемое число - десятичное. А для отображения на светодиодном индикаторе - это число вообще придется раскладывать на цифры.

Тип char происходит от character - одна буква. то есть для представления всех букв, цифр и знаков латинского алфавита нам хватит с головой 256 комбинаций. Сюда даже влезут еще и буквы из других алфавитов. На основе этого и была создана ASCII таблица. Когда текст вводится в компьютер, вводятся не буквы, а коды из этой таблицы. А программа уже смотрит в таблицу кодировок и показывает тот символ, код которого указан в этой таблице.
В китайском и японском языках символов больше, чем 256, поэтому была разработана новая таблица универсальных кодов, где каждый символ представлен уже двумя байтами, а два байта могут представлять уже 65536 символов. Этого хватит на все языки планеты и даже на все старые языки.
Число 65535 в шестнадцатеричной системе записывается так: FF FF, а для того чтобы компилятор понял, что это шестнадцатеричная система, добавляется еще префикс - 0xFFFF. Для десятичных чисел он не нужен.

Еще есть знаковые целые и дробные числа. Например -2754 или 3.3453434.
Если вначале мы писали unsigned, то для хранения отрицательных чисел нужно писать signed char или signed int.
Типы char и int могут быть только целыми. Дробные числа хранить в них нельзя. При компилировании вы сразу получите ошибку о несоответствии типам данных.
Для хранения чисел с плавающей запятой есть **float**. Этот тип так же может быть отрицательным.
Если вы выбираете со знаком, то максимальное положительное число будет:
тип данных / 2, а отрицательное - тип данных / 2 - 1.
Например если вы выбрали **signed** char, который может иметь 256 комбинаций,

то 256 / 2 = 128. Для максимального отрицательного числа: 256 / 2 - 1 = 127.
Один бит тут уже резервируется под знак.
Более подробную информацию вы найдете в мануале к конкретному компилятору.

Операции с числами
и какие типы данных надо резервировать для результата.

Очень важно на этапе программирования предусмотреть переменные и их тип для того, чтобы хранить в них результат от математических (есть еще логические) действий.
Дело в том, что если у вас две переменные типа char (в дальнейшем просто char означает без знаковый тип 0-255), в каждой мы храним цифры 5 и 2. Если мы их сложим и результат сохраним в третью переменную char, то это будет 7.
7 меньше 255 и не отрицательное число, значит оно подходит под тип char.
Если мы умножим 2 на 5, мы получим 10. Результат все еще в пределах допустимого.
Стоит нам только из 2 вычесть 5, как произойдет непредвиденное: результат будет 0. Или же компилятор выдаст ошибку. Если мы поделим 5 на 2, то будет 3.5. Это - уже дробное число и опять будет ошибка, или же результат просто округлится до целого 3.
Такие ошибки очень трудно отловить. Поэтому надо в самом начале все четко запланировать.
Еще есть редкий тип данных - bit. Он может быть либо 1 либо 0. Не рекомендуется к использованию, т.к. еще 7 бит будут просто не задействованы.

Массивы (arrays)

Это - тоже тип данных однотипного типа данных(!). Другими словами - это ящичек, в котором могут лежать только кубики определенного размера, которые заполняют ящичек без остатка. Шарики, конечно, тоже туда влезут, но не так эффективно, как кубики.
Чтобы поместить шарики, надо вместо ящичка взять шар.
Проще говоря, если массив типа char и длинной в 5 элементов, то в него влезут только пять чисел от 0 до 255. Запихнуть туда 6 чисел из этого диапазона, или же 5 чисел больше 255 или дробные - не удастся - компилятор выдаст ошибку.
Записывается это так:

```
unsigned char chans[5]; // сюда влезут только 5 чисел 0-255 (будем хранить тут каналы)
unsigned int freq[20]; // сюда влезут 20 значений 0-65535 (а тут частоты)
```

На данный момент мы зарезервировали только память. Давайте посчитаем, сколько же памяти ушло? Итак 5 раз по одному байту, и 20 раз по 2 байта - 45 байт.

В версиях компиляторов для МК значения всех переменных проставляются (переменная инициализируется) равно 0. Компиляторы для PC ничего не делают. Поэтому при запуске программы там будут находится случайные числа из памяти (если эта область памяти еще не использовалась, то там будут нули).
Поэтому лучше сразу инициализировать массив и записать туда 0. Бывают случаи, когда надо взять значение и прибавить к нему значение из другой переменной и результат положить в первую переменную. Если первая переменная не была

инициализирована и там оказался мусор, то этот мусор сложится со значением второй переменной и только потом сохранится вместо мусора. А вы будете думать, почему результат при каждом запуске разный. Или же ваша программа сразу после включения компьютера работает нормально, а спустя какое-то время начинает выдавать неверные числа.
Инициализация массива выполняется так:

```
unsigned char chans[5] = {0};
unsigned int freq[20] = {0};
```

Не забывайте точку с запятой!
По началу обычно путают длину массива с номером первого элемента.
Так вот 5 и 20 - это длина. В каждом массиве лежит 5 и 20 элементов.
Первый элемент начинается с 0, номер последнего же будет 4 и 19 соответственно.

Теперь запишем частоты и номера каналов.

```
unsigned char chans[5] = {1,3,45,255,102};
unsigned int freq[20] = {8750,8800,8930,10100,10150,
                         0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};//87,50 - 101,50мГц
```

Заметьте, что 5 каналов и 20 частот никак не связаны между собой.
К тому же частоты 6- вообще неизвестны, но их можно изменить во время работы программы. Правда, после выключения питания - они будут потеряны. Но при повторном включении все будет так, как написано во второй строчке.
Кстати изменить частоты можно все, а не только те, которые 0.
Номера каналов тоже можно изменить.

А если мне не надо менять частоты совсем, но я хочу иметь и каналы и частоты?
Тогда эти переменные можно сделать неизменяемыми. Это немного оптимизирует программу в плане выделения оперативной памяти. то есть для неизменяемых переменных оперативная память не выделяется. Значения берутся напрямую из FLASH, но не загружаются в SRAM.
Для этого надо перед unsigned дописать еще const

```
const unsigned char chans[5] = {1,3,45,255,102};
const unsigned int freq[20] = {8750,8800,8930,10100,10150,
                               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
```

А если я хочу просто нумерацию каналов с 1 по 20, на которые я сам хочу повесить нужные мне частоты и потом записать все в память?
Тогда программа упрощается!

```
unsigned int freq[20];
```

Теперь, чтобы записать в определенный канал нужную частоту, нам надо выбрать этот канал.
Это делается проще, чем если бы мы имели 2 разных массива - для каналов и для частот.

Так, нам надо записать в первый канал частоту 100мГц.

unsigned char freq[0] = 10000;

В 20й канал 98,8МГц

unsigned char freq[19] = 9880;

Все рассмотренные выше массивы являлись одномерными. Есть еще многомерные, когда в одном элементе находится не данные,а еще один массив.

char mysuperarray[2] [3] = { {123,123,123}, {123,123,123} };

В данном случае 2 массива, каждый из которых содержит по 3 элемента. Такой массив может содержать до 255 однобайтных массивов.

Компиляторы

С этого места начинается практические упражнения, для которых вам уже потребуется компилятор для обычного ПК.
Почти во всех курсах по программированию процессу компиляции уделяется много внимания. Я считаю, что это не так важно, но основы нужно знать обязательно, чтобы понять, на какой стадии сборки произошла ошибка.
Процесс компиляции состоит из нескольких шагов.
Сначала происходит анализ синтаксиса - не опечатался ли где программист, не забыл ли где скобку закрыть или поставить точку с запятой.
Некоторые среды разработки имеют такую функцию. Либо же вы все равно увидите ошибку syntax error на стадии компиляции. Достаточно 2 раза кликнуть по ошибке, и вы попадете на ту самую строчку. Смотрите тип ошибки. Это может быть пропущенная точка с запятой или незакрытая скобка или же опечатка в функции или операторе.
Потом препроцессор обрабатывает директивы типа #define, #include - заменяет константы на числа и подключает готовые библиотеки или части кода из них к вашей программе. Так же он убирает комментарии и пустые строки.
Если ошибок нет, то дальше идет анализ кода на предмет соответствия типам данных.
После этого сишный код переводится в ассемблерный и подается компилятору для этого языка. Перевод сишного кода в асм код - непростая задача, и каждый компилятор дает свой асм код. Кто-то лучше, кто-то хуже. Поэтому критические ко времени части программ пишутся на асм и вставляются в си код.

Если ваша программа состоит из нескольких частей-файлов, то каждый файл компилируется в отдельный промежуточный объектный код и потом линкуется линкером в один файл и переводится в двоичный исполняемый файл. Это позволяет не компилировать каждый раз все файлы, а только тот, в который вы внесли изменения. Процесс компиляции занимает много времени, и таким образом вы экономите 90%.
Для того, чтобы указать как собирать ваши файлы, есть makefile - файл с командами для программы make, которая уже запускает нужный компилятор и указывает ему нужные флаги оптимизации и прочее. Такой makefile содержит в себе метки, которые являются указателями на часть инструкций.

Например типичные инструкции
make build
make clean
make programm

Запуская make с этими метками, вы говорите тем самым, что нужно сделать с вашими файлами: скомпилировать, очистить от бинарных и промежуточных файлов или же зашить в МК.
Естественно для этих меток необходим соответствующий код в makefile. Такой файл сам генерируется средой разработки, либо программой MFile [WinAVR] из компилятора WinAVR. Бывают компиляторы, где таких файлов нет и весь процесс сборки скрыт от пользователя, предоставляя ему удобные меню для конфигурации.

Для отладки программ существуют специальные приложения - дебаггеры (debugger). Чтобы он мог работать - надо указать в проекте, чтобы при компиляции в ваш файл включалась отладочная информация. Тогда ваш исполняемый файл будет занимать много больше места на диске. Когда же вы отладите программу, то надо переключиться в режим release в вашей среде разработке или иным путем указать, чтобы отладочная информация не генерировалась.
Для МК отладочная информация содержится в отдельных файлах
с расширением .cof. Именно эти файлы нужно указывать симуляторам в качестве бинарных, а не .hex.

Структура программ

Программы на си для ПК и МК немного различаются, но общий принцип у них один. Программа на ПК выполняется в бесконечном цикле или же выполняет последовательность операций и завершается. Конечно же программа в бесконечном цикле имеет условие выхода из него при нажатии определенной кнопки. Или же ее можно насильно прервать CTRL+C.
Программа для МК выполняется в бесконечном цикле без какого-либо выхода из него. Но и тут есть свой особенный выход - прерывания. Они прерывают основную программу и происходит переход в подпрограмму - обработчик прерывания.
После завершения действий в обработчике, основная программа снова продолжает работать с той точки, где она была остановлена.
Более подробно о прерываниях будет рассказано позже.

Простая программа состоит из нескольких частей:
инклюдов, главной функции main, пользовательских функций и констант.
Самая простая программа:

```
#include <stdio.h>

void main (void){
	printf("the simple prog");
}
```

Разберем ее по частям.
Директива препроцессора #include говорит, что надо подключить библиотеку standard input/output, где определена стандартная функция вывода на экран printf. Треугольные скобки говорят, что файл stdio.h - является частью компилятора и лежит в папке include. Вы можете сами определять свои инклюды, в которых будут описаны прототипы ваших функций и константы. Для этого ваш файл должен быть в одной папке с файлом .c (где код вашей программы).
Только теперь вместо скобок - двойные кавычки " ".

#include "myinclude.h"

Void - пустота. Первый раз оно говорит, что функция main не возвращает параметров. Во второй раз оно говорит, что в главную функцию main не передаются параметры. Сама функция main() является главной и обязательной для всех си программ. В фигурных скобках, собственно, сами инструкции вашей программы.

В некоторых случаях главная функция может иметь как входные параметры, так и выходной результат. Это действует только для ПК, но в данном случае нам не нужно.

Функция printf - означает печатать с форматированием. В данном случае у нас нет никакого форматирования, но в будущем оно будет обязательно. Примерами форматирования может быть резервирование знакомест под числа с переменной длиной - частоты 88.5 или же 105.5мгц - разной длины и если не резервировать заранее 5 знакомест, то строка на индикаторе будет прыгать.

Работа с переменными

Небольшое введение в переменные было уже в самом начале. Теперь перейдем в более конкретным действиям.
Итак, что же можно делать с переменными?
Оказывается с ними можно делать не только арифметические действия, но и логические. Сначала рассмотрим арифметические.
Возьмем две переменные целочисленные без знака.
char a,b;
Теперь присвоим им некоторые значения

a = 5;
b = 2;

Значения можно присвоить им и в самом начале

char a=5, b=2;

или же, если две переменные имеют одинаковое значение, то записываем так

char a=b=2;

Теперь мы хотим сложить их. Для этого нам надо либо еще одну переменную, либо результат можно сохранить в одной из них. Соответственно ранее записанное туда значение пропадет.

```
a = a + b;
```

В этом случае сначала будут произведены действия справа от знака =, а потом результат сохраниться в переменную a.
Если же нам и в дальнейшем нужны будут оригинальные значения переменных, то для результата следует зарезервировать еще одну переменную c.

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

Имена переменных не должны быть ключевыми словами языка и не могут начинаться с цифр. Единственный разрешенный знак в имени - это символ пробела _ .

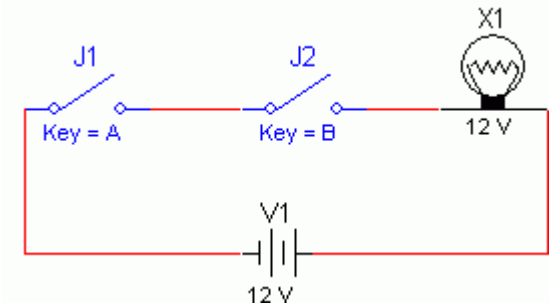
Пример: some_long_variable123 .

Действия с переменными:
сложение +
вычитание -
деление /
деление с остатком %
умножение *
Так же можно использовать возведение в степень при помощи функции pow(), извлекать корень sqr() и прочие синусы и косинусы. Для этого нужно подключить библиотеку math.h .

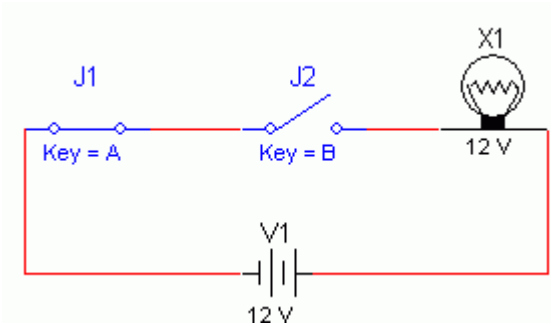
Помните, что при делении целых чисел образуются дробные, а при вычитании - отрицательные. Не забывайте о типах переменных для хранения результатов.

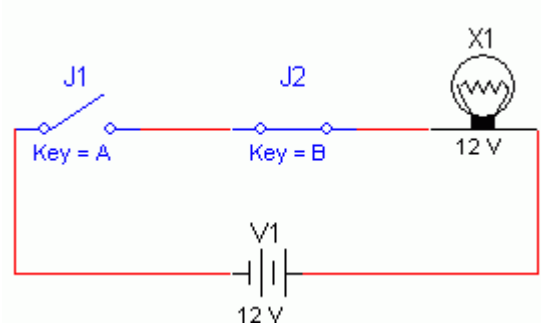
Битовые операции

Очень важным при программировании МК являются операции с отдельными битами. При помощи них вы сможете управлять отдельными битами, не затрагивая все остальные. Например у вас на одном из портов висит 4 светодиода и 4 кнопки. Как определить, какой светодиод надо зажечь и какая из кнопок нажата?
Вот тут-то и понадобятся операции сдвига и логические И, ИЛИ и прочие.
Разберем сначала логическое И (AND).
Представим, что эти два выключателя - 2 бита. Как уже говорилось раньше - бит имеет 2 состояния: лампочка либо горит, либо не горит.

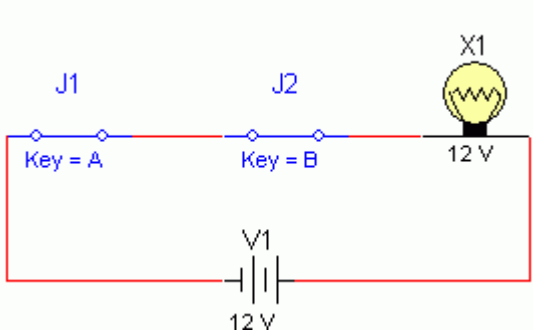


Выключатели разомкнуты, значит оба бита имеют лог. 0.
Замыкаем один из них.

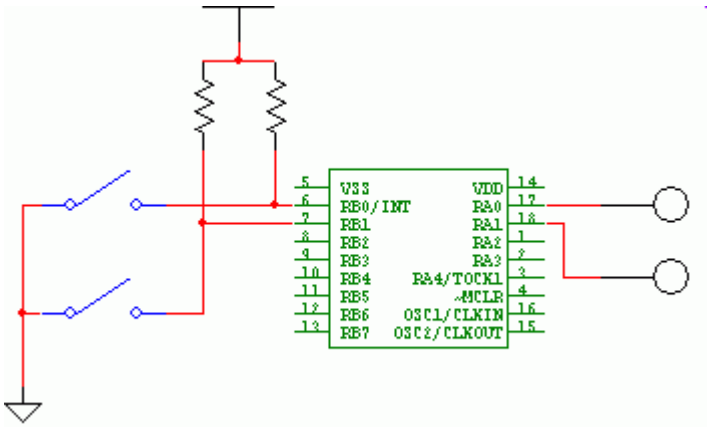




Если хотя бы один из битов 0, то результат будет 0.



Записывается это так 1&0 = 0 или 0&1 = 0. Нетрудно понять, что 0&0 = 0.
Если же оба бита лог. 1, то 1&1 = 1 - лампочка горит.
Но как узнать, какой из восьми битов имеет лог.1 или лог. 0?
Для этого надо применить оператор сдвига >> или же <<.
Не забываем, что обычно отсчет идет справа налево.
Итак, нам надо определить, нажата ли кнопка на первом выводе. Это - нулевой бит.
Битов у нас 8. Тогда нам надо сдвинуть на 0 и применить &1.



Сдвиг на 0 не нужен, т.к. никакого сдвига не будет. Но сначала разберем схему. Два вывода подключены через резисторы к плюсовому проводу. Если кнопка не нажата, то на входах будет лог 1. Если нажата - то напряжение упадет до нуля. Представим наши восемь бит в виде последовательности нулей и единиц:

128	64	32	16	8	4	2	1
0x80	0x40	0x20	0x10	0x08	0x04	0x02	0x01
PB7	**PB6**	**PB5**	**PB4**	**PB3**	**PB2**	**PB1**	**PB0**
0	0	0	0	0	0	1	1

На схеме pic16f84 порты имеют маркировку RB, но все AVR МК вместо R имеют P. Итак, мы видим, что два последних бита благодаря резисторам имеют лог 1. Такая схема выбирается потому, что входные пины не должны висеть в воздухе: либо подтянуты к плюсу, либо к земле.
Теперь самое главное - проверить, какой уровень имеет PB0.
Для этого делаем так:

```
if (!(PINB & (1<<0))) {
        ... кнопка нажата
} else {
        ... кнопка не нажата
}
```

Пропустим пока if ... else и посмотрим внимательней. PINB - это имя регистра, на который приходит информация из внешнего мира. С него только можно читать.

1<<0 - 0 в данном случае это - PB0 или номер бита. Такое выражение вернет 1. Если подставить 2, то будет 2. Но если подставить 3, то результат уже будет 4. Попробуйте запустить "чиста калькулятор" и набрать 1 slh(0). Цифра в скобках и есть номер бита.
Если посмотреть внимательно на таблицу, то мы увидим готовые значения и операция сдвига нам не нужна. Поэтому мы можем записать:

```
if ( ! (PINB & 0x01)) // первая кнопка
if ( ! (PINB & 0x02)) // вторая
if ( ! (PINB & 0x04)) // третья
if ( ! (PINB & 0x20)) // шестая
```

Между скобками затесался восклицательный знак. Это тоже одна из битовых операций - инвертирование.

(PINB & [любое число, не ноль]) вернет нам логическое 1.
Т.к. пин0 и пин1 подтянуты резисторами к плюсу питания, то они будут 1.
Накладываем маси 0x01 или 0x02. Это выглядит так:

Наш порт	00000011
Наша маска для 0x01	00000001
Получим	00000001 - некое число, больше нуля

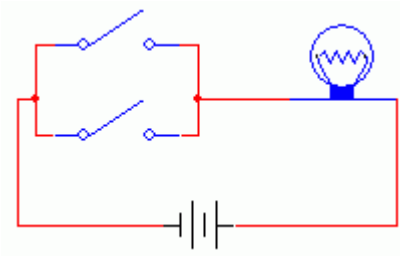
Оператор if (если) проверяет условие. Если оно не равно нулю, то оно верно. В нашем случае оно не равно нулю. Но нам нужна реакция на нажатую кнопку, поэтому это выражение можно инвертировать, т.е. сделать из него логический 0.
Else можно опустить, если при не нажатой кнопке ничего особого делать не надо.
Конечно, сперва смотрится немного запутанным. Тогда это можно упростить, но это не совсем правильный путь.

```
If ((PINB & 0x01) == 0){
        ... кнопка нажата
}
```

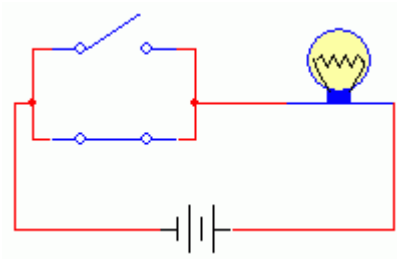
Тут мы сравниваем с нулем, хотя сам оператор if уже предусматривает по умолчанию: все, что не 0 - есть 1. Обратите внимание на двойной знак равенства. Это знак сравнения. Так же частая ошибка.

Пока что мы определяли нажатие кнопок. А как же зажечь светодиод на другом порту?
И тут нам понадобятся операции сдвига, но уже не логическое И, а логическое ИЛИ.

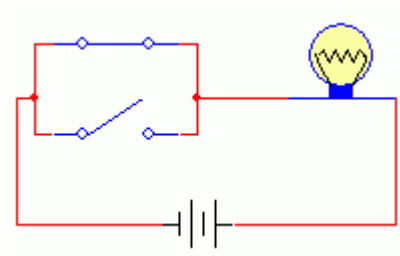
0 | 0 = 0



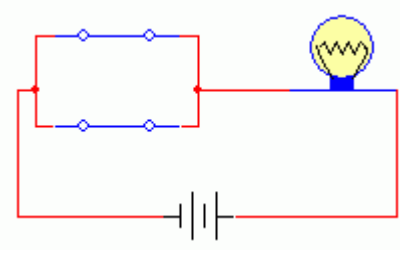
1 | 0 = 1



0 | 1 = 1



1 | 1 = 1



Как видно, операция ИЛИ противоположность И: лампочка горит при трех разных комбинациях, когда при И она горела только если 1 и 1.

Отсюда следует, что если мы при операции ИЛИ оставим единицу, то нужный нам бит установится в 1, а если там была уже лог 1, то ничего плохого не случится.
Светодиоды у нас висят на порту С. Изначально они не горят и это выглядит так:

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
0	0	0	0	0	0	0	0

Сравнивать нам тут уже ничего не надо, а надо записать новое состояние в порт.
Делается это так же легко, как и присвоение значения переменной.
Сперва мы хотим зажечь первый светодиод, который висит на бит0.
PORTA = PORTA | 1;
Пояснение: берется сначала значение (некоторое число, в данном случае оно будет 0 в десятичном и в шестнадцатеричном 0x00), сдвигать на ноль бит нам не надо т.к. это и есть нулевой бит и применяем операцию ИЛИ.
Потом результат записываем обратно в порт. Теперь состояние изменилось:

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
0	0	0	0	0	0	0	1

И оно так и останется дальше, пока снова бит0 не будет установлен в 0. Чуть позже мы рассмотрим, как это сделать при помощи другой логической операции. А сейчас нам надо включить и второй светодиод.

PORTA = (PORTA << 1) | 1;

Новое состояние:

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
0	0	0	0	0	0	1	1

Скорей всего у вас уже возник вопрос, а как сразу зажечь 2 светодиода? Конечно, можно выполнить две команды по очереди, но это займет много времени. Но и тут есть выход.
Это работа с масками или маскИрование.
Давайте разберемся подробней и с использованием таблиц.
Изначально наши светодиоды не горят.

PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
0	0	0	0	0	0	0	0

Нам надо изменить в регистре сразу два бита. Два изменяемых бита и шесть неизменяемых - это уже число. Если набрать это число в двоичном режиме и переключиться в десятичный (калькулятор hexelon), то мы увидим цифру 3. Вот ее-то нам и надо записать в порт с учетом состояния других битов.

это наша маска 0x03							
0	0	0	0	0	0	1	1

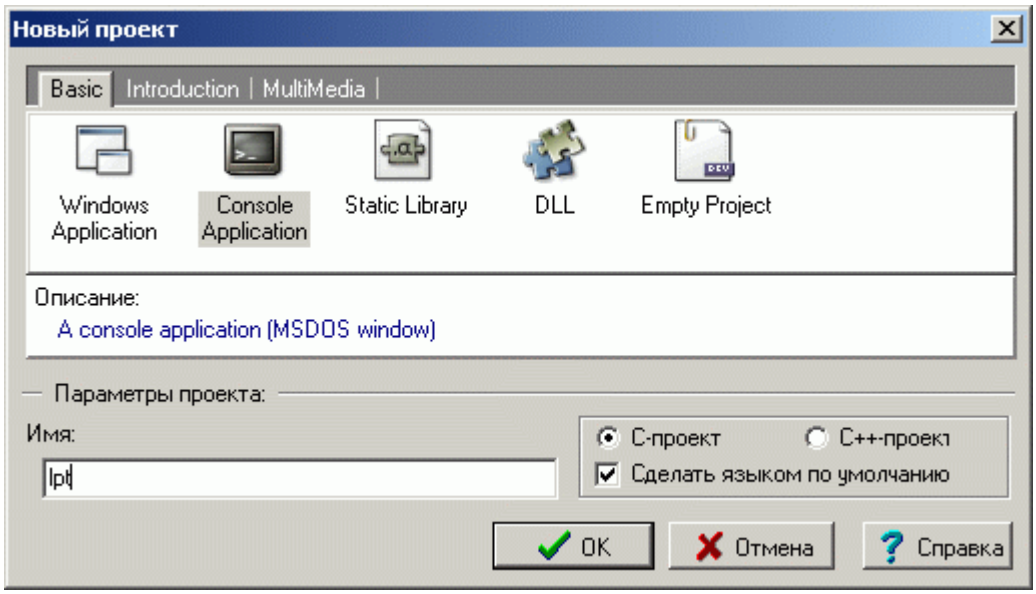
PORTA = PORTA | 0x03;

Проще можно было бы записать и так PORTA = 0x03;
Но тогда вы запишете не только 2 бита в лог 1, но и шесть остальных в лог 0. Это уничтожит содержимое этих шести битов. В нашем случае там лежат нули и поэтому ничего страшного тут не произойдет, А если бы там были другие светодиоды, которые нам совсем не надо было трогать?
Поэтому надо быть осторожным и помнить, что прямая запись в порт - это не маскирование, а именно прямая запись в порт числа.
А маскирование - это изменение (так же сравнение и инвертирование) конкретных битов по маске.

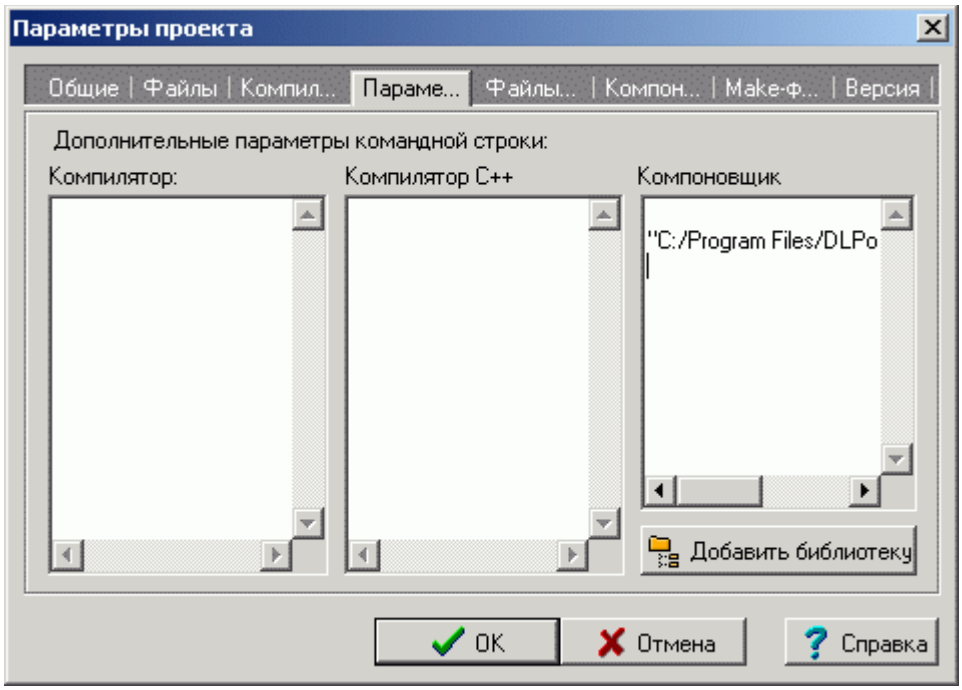
Теперь, когда мы знакомы с переменными, операциями с ними и со структурой программы, мы напишем простую программу для мигания светодиодами для МК и для ПК.
Для этого вам потребуются:

1. PROTEUS
2. dlportio
3. Компилятор winavr
4. Компилятор dev c++
5. Разъем DB25 и кабель для лпт порта с 8 светодиодами и резисторами на 200ом

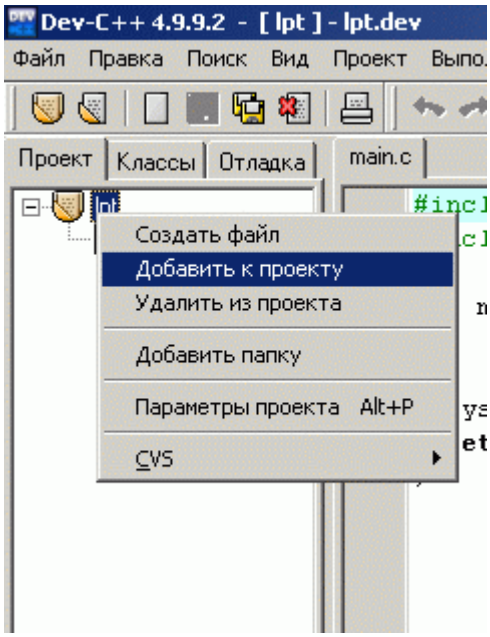
Запускаем dev c++, создаем новый проект lpt и сохраняем файл проекта lpt.dev в новую папку lpt.



Заходим в проект -> параметры проекта, выбираем нужную вкладку и указываем путь до библиотеки "C:/Program Files/DLPortIO/API/DLPORTIO.lib"



В папке C:\Program Files\DLPortIO\API есть еще один файл Dlportio.h, который копируем в папку lpt и еще в папке lesson1 есть файл fakeio.c, который тоже копируем в папку lpt.
Далее добавляем fakeio.c к проекту.



Сам fakeio.c - просто дает нам короткие имена функций и особо затрагивать его не будет, хотя все будет работать и без него. Сохраните все файлы, при этом вам предложат сохранить и main.c - главный файл нашей программы. Удалите в редакторе все строки - мы будем писать все с нуля.
В файле fakeio.c уже есть #include "DLPORTIO.h", который говорит, что надо использовать функции из библиотеки DLPORTIO.lib, которую мы еще в самом начале указали компилятору.
Как уже было сказано выше, fakeio.c будет скомпилирован один раз и потом прилинкован к исполняемому файлу. На этом мы его забываем совсем.

Наша программа начинается с указания компилятору, что мы хотим использовать стандартный ввод-вывод на экран. Это так же описывалось выше.

#include <stdio.h>
Далее нам нужна главная функция main без параметров и без возврата результата.
void main(void){

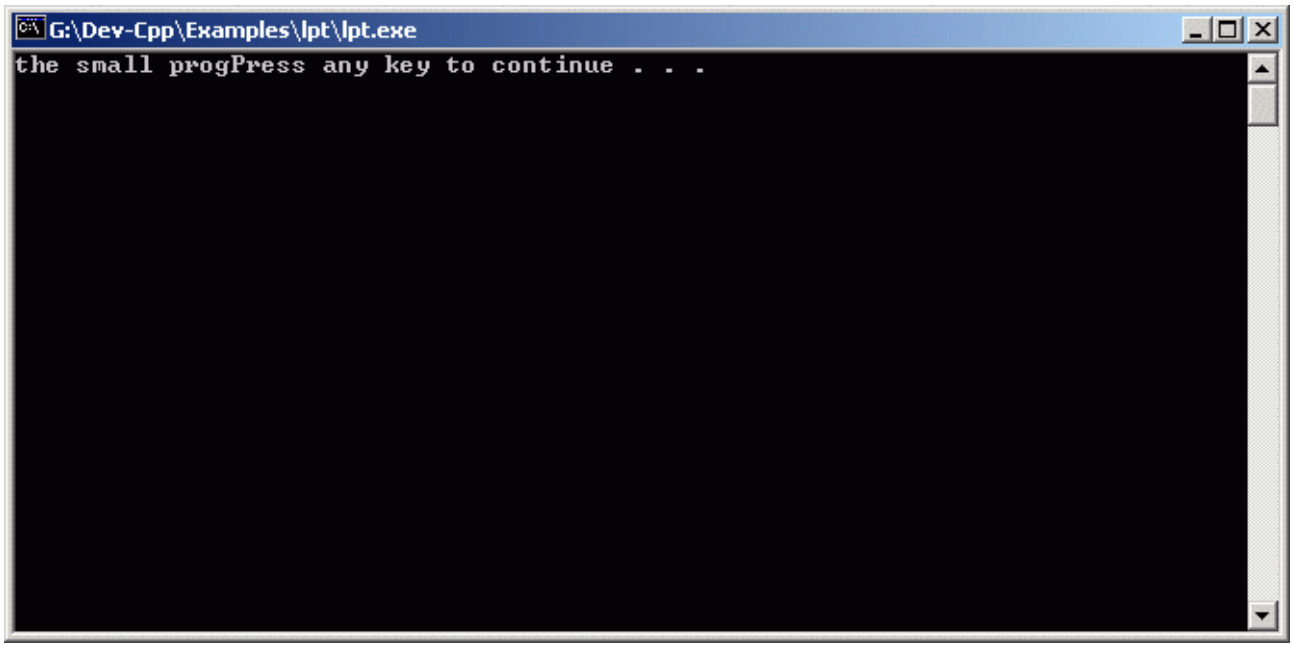
}

Внутри фигурных скобок и будет наш код. Сперва мы выведем на экран просто строку.

printf("the small prog");

Жмем "скомпилировать и выполнить" F9. Через некоторое время вы увидите, как на доли секунды мелькнет консольное окно и закроется. Или же компилятор выдаст ошибки. Посмотрите, точно ли вы указали библиотеку в свойствах проекта, есть ли в проекте файл fakeio.c, нет ли опечаток?
Чтобы окно не закрывалось само, можно в конце (но в пределах фигурных скобок!) добавить строчку

system ("pause");



Чтобы переносить строки, нужно в конце этой строки ставить \n. Это специальный зак переноса на новую строку. Обратный слеш говорит компилятору, что это не просто буква n, а некий код или escape sequense.

ASCII Name	Description	C Escape Sequence
nul	null byte	\0
bel	bell character	\a
bs	backspace	\b
ht	horizontal tab	\t
np	formfeed	\f
nl	newline	\n
cr	carriage return	\r

Попробуйте написать еще одну строчку printf с другими словами и подставьте вместо \n другие коды, например \t.

Теперь самое главное в этом уроке - зажечь светодиоды. Для начала самое простое - потушим все, т.к. они изначально все горят.
Восемь светодиодов - это восемь нулей или 00000000, или в хексе 0x00.
В файле fakeio.c у нас есть функция записи в порт outportb(), которая требует 2 аргумента:
адрес порта и данные.
Немного об адресах. У каждого порта как в ПК, так и в МК есть свой адрес. Для удобства работы с ними в МК им были присвоены буквенные имена типа PORTA, PORTB.
В ПК такого нет и следует обращаться к ним по определенному адресу. Адрес лпт порта у нас обычно 0x378. Это вы можете посмотреть в биосе.
Итак, запишем в лпт порт по адресу 0x378 число 0x00.

```
outportb(0x378, 0x00);
```

Светодиоды погаснут. Теперь зажгем все сразу. Для этого надо записать восемь единиц или число 0xFF. Запустите калькулятор hexelon, нажмите кнопку bin, наберите 11111111, щелкните вверху справа на hex.

```
outportb(0x378, 0xFF);
```

Это вторая строчка. Первую стирать не надо.

Теперь поработаем со сдвигом байтов.
Оставим первую строчку без изменений. А во второй попытаемся зажечь один светодиод.
Пока просто без маски (без учета состояния остальных битов).
Значит нам надо снова послать восемь бит 00000001. Воспользуйтесь калькулятором, чтобы перевести это число в хекс. Получится 01 или с префиксом 0x01.
Теперь представьте, что мы не знаем, горит ли первый светодиод или нет, но нам надо зажечь и второй. Для этого закомментируйте первую строчку, где мы гасили все светодиоды.
Пока у нас должен гореть только первый.
Теперь появляется еще одна функция чтения состояния из порта. Она возвращает некое число. В данном случае она возвратит 0x01, т.к. у нас пока горит первый светодиод.

Зная состояние порта, мы можем изменить его на новое, но сохранить состояние битов, которые нам не нужны.
Функция называется inportb() и в качестве аргумента требует только адрес порта.
Итак, у вас должна быть только одна строка во всей программе.

```
outportb(0x378,  inportb(0x378) << 1 | 1);
```

После выполнения, у вас должно гореть два светодиода подряд.
Попробуйте теперь вместо << 1 подставить другие цифры. Так же попробуйте поменять направления сдвига.

Пришла пора познакомится вплотную с мк и перенести нашу программу на него и посмотреть, чем она отличается от программы для пк. Для этого вам понадобится пакет winavr и proteus, а так же урок lesson2.
Создадим новую папку для проекта. Запускаем Programmers Notepad [WinAVR], создаем новый проект и указываем, куда сохранять файл проекта. Потом создаем новый файл и сохраняем его как led.c. Так же этот файл надо добавить к проекту, как это делали в dev c++. Добавление файлов в проект позволит быстро их открывать, однако это не влияет на процесс компиляции. Для создания makefile используется специальная утилита MFile [WinAVR], с которой мы ознакомимся чуть позже. Перед тем, как начать писать программу для мк, вам необходимо ознакомится с некоторыми особенностями.

Любой мк содержит в себе как минимум один 8-битный порт, выводы которого могут быть входами или выходами, причем каждый пин может быть сконфигурирован отдельно. Так же в процессе работы программы, пины можно переключать с входа на выход или же подключать к ним внутреннюю периферию. При подключении такой периферии, порт автоматически конфигурируется на вход или выход. Например, модуль UART для связи с пк имеет 2 пина RX и TX, для приема или передачи. Можно задействовать только прием или только передачу, или же оба сразу. Если же вы включите АЦП, то на такой вывод уже нельзя будет подключить светодиод или кнопку.
Для того, чтобы сконфигурировать мк, надо записать 1 или 0 в соответствующие регистры и соответствующие места этих регистров.
Поясню: не все биты регистров могут быть доступны на запись и даже на чтение. Некоторые биты зарезервированы и их невозможно использовать. Более подробно смотрите даташит на конкретный мк.
Но в данном случае нам надо указать, что все пины PORTD будут выходами. Для этого есть специальный регистр DDR - data direction register. Его размер 8 бит. Каждый бит отвечает за один пин. Если бит установлен в 1, то этот вывод будет выходом. Подробнее в даташите на стр. 50 таблица 20.
Для порта D регистр будет иметь название DDRD. Мы хотим все 8 пинов задействовать как выходы. Следовательно, надо записать в этот порт число 11111111 или 0xFF.
Но перед этим мы должны указать компилятору, что будем использовать эти регистры. Для этого нам надо подключить файл avr/io.h. Не подключайте iom16.h, он сам будет использован, если надо.

Итак:
```
#include <avr/io.h>
```

Дальше мы должны сконфигурировать направление порта на выход. Все остальное - уже является частью си-программы и должно находится внутри главной функции main.

```
viod main(void){
	DDRD = 0xFF;


}
```

По умолчанию светодиоды должны быть погашены. Как вам уже известно, для этого нам надо установить все биты в ноль на порту D. Для этого есть регистр PORT.
PORTD = 0x00;
У вас должно получиться следущее:

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

void main(void){

	DDRD = 0xFF;
	PORTD = 0x00;

}
```
Осталось скомпилировать это и протестить в протеусе.

Компиляция и makefile

Ранее уже упоминалось о makefile. Напомню, что этот файл нужен для программы make, и представляет собой скрипт с инструкциями, как компилятор и линкер должны обработать и собрать бинарный файл из разных файлов с исходными кодами. Для облегчения написания этого файла есть утилита MFile [WinAVR]. Опишем некоторые из его частей.

Code generation
main file name - имя конечного файла-прошивки. Назовем его led.

MCU type - тип мк, в самом тексте найдите чуть ниже F_CPU = 8000000. Это частота кварца в герцах. Обычно используют 4000000 или 4МГц. Не забудьте указать эту же частоту и в протеусе. От нее зависят и временнЫе задержки, которые мы будет позднее использовать.
Оставим ее 8000000.

output format - ihex стандартный формат от интела.

optimization level - уровень оптимизиции. Стандартно s - оптимизирует по размеру прошивки. Оставим стандартно.

debug format - генерирует файл с отладочной информацией, который используется симуляторами AVR Studio и другими. Пока оставим как есть.

c standard level - gnu99

C/C++ source files - если у вас один .c файл, который вы указали в директиве main file, то теперь его указывать не надо. Если же ваша программа состоит из нескольких файлов, то их надо перечислить через пробел. Указывать полный путь не надо, если же они лежат в одной папке. Или же полный путь не должен иметь пробелов в названии файлов и папок. Всегда называйте ваши папки и файлы без пробелов.
В нешм случае это выглядит так:

```
# Target file name (without extension).
TARGET = led
# List C source files here. (C dependencies are automatically generated.)
SRC = $(TARGET).c
```

assembler source files - исходники на асм

printf, scanf - функции, требующие много памяти. Можно включить оптимизацию, если вы не используете некоторые возможности. Например, не работаете с числами с плавающей запятой. Тогда оставте стандартно. Минимальный вариант исключает вообще какое-либо форматирование.

AVR dude
programmer - ваш аппаратный программатор. Рекомендованный в начале программатор тут называется pony-stk200
port - lpt1

Последний пункт - разрешает редактирование уже открытого makefile.

Пробный пуск

Вернемся в редактор кода. В меню tools выберите make all.
Если все верно, то вы увидите следущее:

```
> "make.exe" all

-------- begin --------
avr-gcc (GCC) 3.4.5
Copyright (C) 2004 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.


Linking: led.elf
avr-gcc -mmcu=atmega16 -I. -gdwarf-2 -DF_CPU=8000000UL  -Os -funsigned-char
-funsigned-bitfields -fpack-struct -fshort-enums -Wall -Wstrict-prototypes -Wa,-
adhlns=led.o  -std=gnu99 -MD -MP -MF .dep/led.elf.d led.o --output led.elf -Wl,-
Map=led.map,--cref    -lm

Creating load file for Flash: led.hex
avr-objcopy -O ihex -R .eeprom led.elf led.hex
```

```
Creating load file for EEPROM: led.eep
avr-objcopy -j .eeprom --set-section-flags .eeprom=alloc,load \
--change-section-lma .eeprom=0 -O ihex led.elf led.eep

Creating Extended Listing: led.lss
avr-objdump -h -S led.elf > led.lss

Creating Symbol Table: led.sym
avr-nm -n led.elf > led.sym

Size after:
AVR Memory Usage
----------------
Device: atmega16

Program:     162 bytes (1.0% Full)
(.text + .data + .bootloader)

Data:        0 bytes (0.0% Full)
(.data + .bss + .noinit)



-------- end --------


> Process Exit Code: 0
> Time Taken: 00:01
```

Обратите внимание на выделенный жирным текст. Наша программа заняла 162 байта из 16 килобайт.
Второе - показывает файл EEPROM, отдельные данные, например различная информация, записанная в процессе работы МК (настройки частоты, каналы, громкость и т.д.)

Теперь откройте файл led.dsn со схемой, проверьте правильность пути к .hex файлу. Для этого щелкните на черной стрелке в левой части экрана, потом на МК, в поле program file укажите путь. Не забудьте выставить частоту 8Мгц.
Внизу есть кнопка, похожая на кнопку воспроизведения. При нажатии на нее должна запустится симуляция без ошибок. Вы увидите, что PORTA все выводы с красными квадратиками, а PORTD - синие. Так и должно быть.
Наша программа отработала и завершилась. В ней нет бесконечного цикла while(1). Теперь сделаем так, что зажгем просто светодиод и завершим программу.

Представим снова линейку из битов, где первый бит - справа. Его-то мы и выставим в 1.

```
PORTD = 0x01;
```

А что будет, если мы сделаем бесконечный цикл? А если постоянно будем выполнять одну и ту же команду?
Давайте посмотрим.

```
while(1){
	PORTD = 0x01;
}
```

С виду ничего не изменилось - светодиод горит так же. Однако теперь МК выполняет несколько действий, пока не будет сброшен. Что он там делает - на данный момент не важно.
Суть в том, что инициализация портов и другой периферии должна происходить до вызова while(), но внутри главной функции main.
Теперь нам осталось только сделать так, чтобы светодиод мигал через нужные нам промежутки. Для этого нам его надо зажечь и погасить. Как это сделать - уже было описано. Но есть еще пара операций, изменяющих состояние битов.
Так же нам нужна какая-то функция задержки. Сначала мы ее найдем в помощи по компилятору avr-libc Manual [WinAVR].
Находится это на стр.113
5.22 <util/delay.h>: Busy-waitdelayloops

```
#define F_CPU 1000000UL // 1 MHz
//#define F_CPU 14.7456E6
#include <util/delay.h>
```

Note:
As an alternative method, it is possible to pass the F_CPU macro down to the compiler from the Makefile. Obviously, in that case, no #define statement should beused.

Т.к. мы уже в makefile задали частоту, задавать повторно ее не надо. Оставим только инклюд.
Прочитаем внимательно о всех функциях...

Нас интересует только
5.22.2.3 void **_delay_ms** (double__ms)
Perform a delay of __ms milliseconds, using_delay_loop_2().
The macro F_CPU is supposed to be defined to a constant defining the CPU clock frequency(in Hertz).
The maximal possible delay is 262.14ms/F_CPU in MHz.
Отсюда следует, что максимальная задержка при 8МГц будет не более 32мс.
Значит нам надо снизить частоту кварца в makefile и в протеусе.

В конечном итоге у вас должно получиться:

```
while(1){
	PORTD = 0x00;
	_delay_ms(260);
	PORTD = 0x01;
	_delay_ms(260);
}
```

Теперь вернемся к операциям с битами и оптимизируем программу.

```
PORTD = PORTD ^ 1;
	_delay_ms(260);
```

Операция ^ XOR - инвертирует биты. Не путать с !.
Задержки, подобные этой, - есть обычные тики процессора. Во время выполнения их больше ничего нельзя делать параллельно. Поэтому использовать эти функции надо там, где требуются короткие задержки, а для длинных - надо использовать аппаратные таймеры-счетчики.
И под конец еще 2 урока, показыающие сдвиг битов и работу счетчика.
Представим себе, что у нас все биты порта сброшены (лог 0). Что будет, если мы будем прибавлять по единице к порту за каждый повторный проход цикла?
То есть PORTD = 0x00;, а потом PORTD = PORTD + 1;?
Давайте посмотрим...

Оказалось, вопреки ожиданиям, что светодиоды не загораются поочередно или друг за другом, а как бы "накапливаются" и гаснут. Вспомните про BCD код. Так работает любой аппаратный счетчик. Эти 8 светодиодов всего могут иметь 256 комбинаций. Потом происходит переполнение счетчика. В данном случае он сбрасывается в ноль и светодиоды гаснут, другие же счетчики-таймеры генерируют прерываение или же тактовый импульс для следущего счетчика.
В протеусе есть ошибка: 6 и 7 светодиоды на порту C не горят. Так же некоторые компоненты в версии 6.7 сп3 имеют ошибки. На порту D все работает.

На последок сделаем бегущие огни.

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

int main(void){
unsigned char i;

DDRD = 0xFF;
PORTD = 0x00;

while(1){

	i=0;

	while(i<8){
		PORTD =  1 << i ;
		_delay_ms(260);
		i++;
	}

}// конец while(1)

}// конец main()
```