Язык Си для МК

(Чуток, но и этого будет достаточно ...)

Я расскажу об устройстве и структуре программы на языке Си и опишу используемые в МК конструкции языка.

По умолчанию компилятор **CVAVR.** В других компиляторах могут быть незначительные отклонения, нюансы не связанные с языком Си а обусловленные стараниями и предпочтениями разработчиков этих компиляторов.

Я покажу вам, что Си это ДЕЙСТВИТЕЛЬНО ОЧЕНЬ ПРОСТО, если у вас реальные для начинающего цели!

Кстати БЭЙСИК тоже не плох ! **Вот скачайте архив** (5 Кб всего) с перечнем АпНоутов - великолепных и интересных устройств в которых программы для МК AVR написаны на языке Бэйсик. Скачать BASCOM-AVR-1.11.8.3 full - полная версия можно там

Минимальная программа на Си может быть такой:

```
main(){}
```

Эта программа не делает ни чего полезного - но это уже программа и она показывает что **в программе на языке Си - должна быть главная функция main - обязательно !Реальные программы на Си конечно больше.**

Скачайте и Распечатайте - Памятка Си для МК на ОДНОЙ странице !

Рассказывая про МК я говорил вам что, задача программы МК: читать числа из регистров МК, делать что-то с ними и записывать числа в регистры. Только так программа может общаться с МК. Как это делать на языке Си

Регистры МК в программе на Си имеют названия как и в ДШ и так как числа в большинстве из них можно менять - для программы регистры
являются по сути переменными.

1) Чтобы поместить число в переменную (в регистр) в Си есть оператор присваивания - это знак = ("равно") . **В Си этот знак не означает равенство ,** = в Си означает вычислить результат того что справа от оператора присваивания и поместить этот результат в переменную находящуюся **левее** оператора присваивания.

```
PORTB = PINB + 34;/* Эта строчка на Си означает
Взять (прочитать, считать) значение переменной (регистра) PINB, затем прибавить к нему
число 34 и поместить результат в переменную PORTB */
```

```
ПЕРЕМЕННАЯ = PINC; /* Эта строчка на Си означает
Взять (прочитать, считать) значение переменной (регистра) PINC и поместить результат в
переменную с именем ПЕРЕМЕННАЯ */
```

Чтобы в Си взять (прочитать) число из регистра или значение переменной нужно написать его название НЕ непосредственно с лева от оператора присваивания ! **во загогулина понимашшшь... Ж)**

примеры :

1) Строка где переменная стоит слева от = но через знак &

PORTB & = **0x23;**

на Си означает - прочитать содержимое переменной **PORTB**, затем выполнить "поразрядное (побитное) логическое И" между прочитанным значением и числом **0x23** и поместить (записать, присвоить) результат в переменную **PORTB**

2) Строка где переменная стоит **непосредственно** слева от =

PORTB = **0x23;**

на Си означает - **не читая** содержимое переменной **PORTB** присвоить ей значение **0x23**

Вместо & "И" (AND - только 1 и 1 дают 1) могут быть и другие побитные логические операции:

| "ИЛИ" (OR только 0 и 0 дают 0)

^ "Исключающее ИЛИ" (XOR изменить бит напротив "1")

~ "инвертирование битов" (INV изменить биты регистра)

и арифметические операции: **+ - * / %**

С оператором присваивания
используются вот такие сокращения:

ДЛИННАЯ ЗАПИСЬ		СМЫСЛ		СОКРАЩАЕТСЯ ДО
x = x + 1;		добавить 1		x++; или ++x;
x = x - 1;		вычесть 1		x--; или --x;
x = x + y;		прибавить y		x += y;
x = x - y;		вычесть y		x -= y;
x = x * y;		умножить на y		x *= y;
x = x / y;		поделить на y		x /= y;
x = x % y;		остаток от деления		x %= y;
x--;		вычесть 1		x -= 1;
x++;		добавить 1		x += 1;

примеры :

```
10010 | 1001111 // "ИЛИ" - только 0 и 0 дают 0
                //     англ. название  OR

  1011111  // это результат

// только биты_5 в обоих числах были нули
```

```
10010 &  1001111 // "И" - только 1 и 1 дают 1
                //    англ. название   AND

     10  // это результат

// только биты_2 в обоих числах были единицы
```

```
10010 ^ 1001111
/* "исключающее ИЛИ" - результат любое из пары чисел в котором инвертированы биты
напротив битов равных "1" в другом числе.

англ. название  XOR*/

       1011101  // это результат

/* изменились биты во втором числе напротив
   установленных битов 4 и 1 первого числа. */
```

```
~  1001111 /* инвертировать биты
те что были "1" станут "0" и наоборот  */

  110000  // это результат
```

Запомните ! Результатом поразрядных (побитных) логических операций : & | ^ ~ является число, которое может быть интерпретировано компилятором как "истина" если оно не ноль и "ложно" если число ноль.

Числа

В компиляторе можно записывать в виде указанном в его Help'e ! Раздел - константы - Constants.

например - Целые числа могут быть записаны :

- - в десятичной форме - 1234
- - в двоичной форме с префиксом 0b так: 0b101001
- - в шестнадцатеричной форме с префиксом 0x так: 0x5A
- - в восьмеричной форме с префиксом 0 так: 0775

Числа с плавающей точкой имеют в записи эту точку и какое либо число после этой точки, так: 61.234 или так: 73.0 и так: .786 и могут иметь в конце F вот так: 61.234F

Цвета я применил УСЛОВНО для лучшей читаемости!

Различные представления числа

D3h равно 0xD3 равно 0b1101 0011 равно 211

шестнадцатеричное число 0xD3									
0	x	D				3			
двоичное представление - число 0b1101 0011									
0	b	1	1	0	1	0	0	1	1
номера бита		7	6	5	4	3	2	1	0
		два в степени равной номеру бита							
		128	64	32	16	8	4	2	1

число 211 в десятичном виде это сумма степеней двойки где биты равны "1"									
Сложите		+128	+64		+16			+2	+1

Четыре бита это 1 нибл или 1 символ в 16-ричной системе или десятичное число от 0 до 15.

"В уме" удобно оперировать ниблами:

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

Для перевода чисел из одного вида в другой можно использовать калькулятор Windows в инженерном виде.

Есть в Си операции которые изменяют значение переменной и без оператора присваивания :

```
PORTA++;   /* Эта строчка на Си означает
Взять значение переменной PORTA добавить к ней 1 и записать результат обратно в PORTA

говорят: Инкрементировать регистр PORTA */

PORTC--;      /* Эта строчка на Си означает
```

обратное действие!

Декрементировать - вычесть 1 из значения регистра PORTC */

Инкремент и **декремент** удобно использовать для изменения значения различных переменных счетчиков.
Важно помнить что они имеют очень низкий приоритет - поэтому чтобы быть уверенными в порядке выполнения желательно писать
их отдельной строчкой программы !

Обратите внимание !

В конце выражения или конструкции в программе на Си ставят точку с запятой. Длинные выражения можно писать в несколько строк.

/* ЗЕЛЕНЫМ я пишу комментарий к программе
в Си он может быть написан в несколько
строк */

// или в одну после двух черточек

Компилятор игнорирует все что написано в комментариях. Вы не компилятор , не игнорируйте, пишите и читайте !

Когда инкремент или декремент используется в выражении то важно где стоят два знака + или - перед переменной или после переменной :

```
a=4;
b=7;

a = b++;   /* Эта строчка на Си означает Взять значение переменной b присвоить его переменно a затем добавить 1 к переменной b
и сохранить результат в b. В результате a будет содержать число 7 а b будет содержать число 8  */

a=4;
b=7;

a = ++b;   /* Эта строчка на Си означает Взять значение переменной b затем добавить к нему 1 и сохранить результат в b и этот же результат присвоить a. Теперь  aбудет содержать число 8  и b будет содержать число 8      */
```

2) Арифметические операции в Си

x + y // сложение
x - y // вычитание
x * y // умножение
x / y /* деление.

Если числа целые результат - целое число с отброшенной дробной частью - **не округленное** ! т.е. если в результате деления на калькуляторе получается 6.23411 или 6.94 то результат будет просто целое число 6 - **запомните** ! Если числа с плавающей точкой, то есть float или double и записываются с точкой и числом после точки, то и результат будет число с плавающей точкой */

x **%** y // вычислить остаток от деления нацело

// примеры:
5 / 2 // даст 2
5 % 2 // даст 1

3) Операторы сравнения (или отношения): **используются для сравнения переменных, чисел (констант) и выражений.**

x < y // X меньше Y
x > y // больше
x <= y // меньше или равно
x >= y // больше или равно
x == y // равно
x != y /* не равно

Результат выполнения этих операторов:

"истина" это "1" (точнее "не ноль")

"ложно" это "0"

Значения хранимые в переменных (в регистрах) **x** и **y** НЕ изменяются.

Берутся (считываются) значения хранящиеся (или содержащиеся) в переменных и сравниваются */

! /* "НЕ" - логическое отрицание */

4) Логические операции **:**

```
|| // "ИЛИ" - только "ложь" и "ложь"
   //          дают "ложь"


&&  //  "И" - только "истина" и "истина"
        //         дают "истина"

!   // "НЕ" - логическое отрицание

/*   Правило - в Си считается:

"Ложь" (False) только ноль.

"Истина"(True)- не ноль. или так:  (!0)

*/

!(истина) // дает "ложь"

!(ложь)    // дает "истина"
```

В результате логической операции вы получаете НЕ ЧИСЛО, а логическое значение "истина" или "ложь"

Для логических операций && и || берутся результаты выражений слева и справа от знака операции **преобразованные в "истину" или "ложь"** и определяется логический результат операции. Компилятор, для определенности наверно, результат "истина" превращает в 1 а не в любое отличное от 0 число.

Совет: Используйте скобки
(() + (() * ()))
чтобы точно знать порядок выполнения операций программой !

Логические операции могут объединять несколько проверяемых условий.

Например:

```
if((выражение1)&&((выражение2)||(выражение3)))
{/*
Код программы здесь будет выполняться если:
```

Выражение1 "Истина" (значит не ноль) и хотя бы одно из выражений 2 и 3 тоже "Истина" (значит не ноль).
};

Подробнее о логических операциях обязательно прочитайте по линку в низу 2-й части этой страницы !

Самое интересное - Ходовые конструкции на Си

5) **if**(){}**else**{}; идеальная конструкция если вам нужно выполнить какую то часть программы при наличии каких либо условий :

```
if (выражение) { /* делать этот код если выражение "истина" - т.е. результат его
вычисления не ноль */
                }
else { /* делать этот код если выражение "ложь" - т.е. результат его вычисления равен
нулю */
            };
```

} **else** { это не обязательный элемент конструкции :

```
if (выражение) { /* делать этот код если выражение "истина" - т.е. результат его
вычисления не ноль */
              };
```

6) **while**(){}; условный цикл - используйте если вам нужно выполнять какой то код программы пока выполняется (существует, справедливо, не ноль) некоторое условие :

```
while (выражение) { /* делать этот код если выражение "истина" - т.е. результат его
вычисления не ноль.

Пока выполняется этот код выражение не проверяется на истинность !

После выполнения кода происходит переход к строке while снова проверять истинность
выражения */
                 };
```

Цикл while имеет вариант do - while при котором код в { } выполняется по меньшей мере один раз не зависимо от истинности условия в скобках :

do { /* сделать этот код один раз

затем, если выражение есть "истина" - т.е. результат его вычисления не ноль - опять делать код с начала, и так до тех пор пока **выражение** истина */ }
while (выражение**);**

7) **for**(;;){}; - этот цикл позволяет выполнить часть программы нужное число раз:

char i; /* объявление переменной для **for**

это обычная переменная и значит может иметь любое допустимое имя по вашему желанию */

for (i=5;i<20;i+=4) { /* код цикла **for**

i=5 - это начальное выражение

Число 5 просто для примера, может быть таким, как позволяет объявление переменной i, в нашем случае от 0 до 255

i<20 - контрольное выражение

Может быть с разными операторами отношения, **важно лишь** чтобы по ходу цикла оно становилось когда-то "ложью" - иначе цикл "зациклится" т.е. ни когда не кончится.

i+=4 - счетчик

Обычно это **i++** т.е.к переменной добавляется 1 каждый "прогон" цикла. Но опять же может быть таким какое вам требуется, важно лишь достижение когда либо условия абзацем выше ! Иначе цикл станет бесконечным.

Код цикла **for** будет первый раз выполнен для i=5, затем по выражению i+=4, i станет 9

теперь будет проверено контрольное выражение i<20

и так как 9<20 код цикла **for** будет выполнен еще раз.

Так будет происходить до тех пор пока контрольное выражение "истино"

Когда оно станет "ложно" цикл **for** закончится и программа пойдет дальше. */
};

Начальным условием - может быть любое допустимое в Си выражение результатом которого является целое число.

Контрольное выражение - определяет до каких пор будет выполнятся цикл.

Счетчик - показывает как изменяется начальное выражение перед каждом новом выполнении цикла .

циклы **for**(;;) и **while**() **часто используют вот так:**

```
while(1);

for (;;);

/* Так написанные эти циклы означают :

МК выполнять эту строчку пока есть питание, нет сброса и нет прерывания.

Когда возникает прерывание, программа переходит на обработчик прерывания и (если в обработчике нет перехода в другое место программы)по завершении кода обработчика опять возвращается в такой цикл.  */
```

8) **switch**(){}; - оператор множественного выбора,
позволяет вам сделать выбор из нескольких вариантов.

```
switch (выражение) {
case 5:
/* этот код будет выполняться если результат вычисления выражения равен числу 5

на этом работа оператора switch закончится */
break;

case -32:
/* этот код будет выполняться если результат вычисления выражения равен отрицательномц числу -32

на этом работа оператора switch закончится */
break;

case 'G':
/* этот код будет выполняться если результат вычисления выражения равен числу соответствующему символу G в таблице ASCII

на этом работа оператора switch закончится */
break;

default:
/* этот код будет выполняться если результат вычисления выражения не равен ни 5 ни -32 ни 'G'
```

```
на этом работа оператора switch закончится */
};

/* switch закончен - выполняется дальнейший код программы */
```

case - может быть столько сколько вам нужно, чтобы программа работала быстрее старайтесь наиболее вероятные варианты располагать выше!

default - не обязателен.

break; - лучше писать, иначе найдя нужный вариант программа будет проверять и следующие условия **case** - напрасно тратя время.

Скачайте и Распечатайте Таблицу символов ASCII на ОДНОЙ странице !

9) **goto** - оператор безусловного (немедленного) перехода.

```
mesto_5: /* сюда мы попадем после выполнения строки программы goto mesto_5 */

goto mesto_1; /* перейти в то место программы где в начале строки
написано mesto_1: */

goto mesto_5; /* перейти в то место программы где в начале строки
написано mesto_5: */

mesto_1: /* сюда мы попадем после выполнения строки программы goto mesto_1 */
```

goto - существует наверно во всех языках и в ассемблере в том числе. **Используйте его с осторожностью!** Думайте к чему может привести выполнение функций или конструкций вашей программы не до конца.

Например: Если вы покинете функцию - обработчик прерывания по **goto** не завершив ее, то не произойдет автоматического включения прерываний глобально - т.е. не установится бит I в регистре SREG, **Этот бит устанавливается автоматически после полного выполнения функции обработки прерывания и "естественного" выхода из неё.**

Ну вот - ПОЧТИ всё что нужно нам из Си ! Как использовать описанное выше вы можете посмотреть в примерах к компилятору ! Примеры в папке :

C:\CVAVR\EXAMPLES

Открывайте файлы .c и разбирайте текст программ - что делает каждая строчка! Это великолепный способ само- обучения программированию !

Структура программы на языке Си

Программа на языке Си это текстовый файл с расширением .c

Текст программы называют исходным или "исходником" или "сурцом" от анг. **source code** - это вам **ключевые слова для поиска** !

"Исходник" на Си имеет определенную структуру :

- 1) заголовок
- 2) включение необходимых внешних файлов
- 3) ваши определения для удобства работы
- 4) объявление глобальных переменных

Глобальные переменные

- объявляются вне какой либо функции.
т.е. не после фигурной скобки **{**

- доступны в любом месте программы - значит можно читать их значения и присваивать им значения там где требуется.

5) описание функций - обработчиков прерываний

6) описание других функций используемых в программе

7) функция **main** **- это единственный обязательный пункт !**

Это не жесткий порядок а ориентировочный !

Иногда п.6 это прототипы функций, а сами функции описываются полностью после п.7

Прототип функции - показывает образец того как применять функцию в программе, какие значения в нее передаются и если она возвращает какое-то значение то прототип указывает тип возвращаемых данных. Прототип не имеет скобок { } а после скобок ( ) ставится **;**

Функция - имеет { "тело" } в фигурных скобках. Тело это код на Си определяющий то что делает функция.

; после функции не ставится.

Программа на Си начинает работу с функции main(), по необходимости из main()вызываются другие функции программы, по завершении работы функции программа возвращается в main() в то место от куда функция была вызвана.

```
main(){

... какой то код программы ...

вызов функции_1; //программа перейдет в функцию_1

строка программы; // будет выполнятся после
                  // возврата из функции_1

... какой то код программы ...

}
```

функции могут вызываться не только из main() но и из других функций.

Пример программы на Си

с описанной выше структурой я буду писать на голубом фоне.

По мере надобности я буду разрывать голубой фон обычным текстом, затем голубой фон и программа будет продолжаться.

```
/* п.1 заголовок программы

Он оформляется как комментарий, и обычно содержит информацию

- о названии, назначении, версии и авторе программы
- краткое описание алгоритма программы
- пояснения о назначении выводов МК
- другие сведения которые вы считает полезным указать
*/

// комент. после двух косых черт пишут в одну строку!



//п.2 включение внешних файлов

#include <mega16.h> /* перед компиляцией, препроцессор компилятора вставит вместо
этой строчки содержимое (текст) заголовочного файла "хидера" mega16.h - этот файл
содержит перечень регистров имеющихся в МК ATmega16 и соответствие их названий их
физическим адресам в МК.

Посмотрите его содержание

CVAVR\inc\mega16.h          */
```

```
//delay functions
#include <delay.h>
/* перед компиляцией, препроцессор компилятора вставит вместо этой строчки текст
"хидера" delay.h - этот файл содержит функции для создания пауз в программе.

Теперь чтобы сделать паузу вам нужно лишь написать :

delay_ms(x); // сделать паузу x милиСек
delay_us(x); // сделать паузу x микроСек

x - число от 0 до 65535 (тип unsigned int)      */



//п.3 определения пользователя

// AD7896 control signals PORTB bit allocation
#define ADC_BUSY PINB.0
#define NCONVST PORTB.1
/* после этих двух строк, перед компиляцией, препроцессор компилятора заменит в тексте
программы ADC_BUSY на PINB.0 и NCONVST на PORTB.1

Таким образом вместо того что бы помнить что вывод занятости AD7896 подключен у вас к
ножке PB0 вы можете проверять значение осмысленного понятия ADC_BUSY - "АЦП занят"

а вместо управления абстрактной ножкой PB1 (через PORTB.1) вы можете управлять
"НьюКонвешнСтат" - NCONVST - "стартовать новое АЦ преобразование"

#define   -   Это   удобно !
но ВОВСЕ не обязательно.


*/

#define INIT_TIMER0 TCNT0=0x100L-F_XTAL/64L/500L
// этот пример показывает что определения
// могут быть и сложней !
```

Определения (соответствие номера бита в регистре его названию по ДШ) отдельных битов есть в "хидерах" .h в ICC, IAR и других компиляторах, но их нет в хидерах CodeVisionAVR Поэтому я сделал для вас **файл - заголовок** **m8_128.h** **скачайте его и добавьте в программу вот так:**

```
#include <mega16.h> //сперва обычный хидер

#include <m8_128.h> //мой хидер для битов
```

Теперь вы можете использовать примеры на Си из ДШ на соответствующий МК ! Мой файл m8_128.h содержит определения битов для микроконтроллеров **ATmega8** **ATmega16** **ATmega32** **ATmega64** **ATmega128**

Мастер начального кода программы в компиляторе ICC умеет по вашему желанию автоматически делать #define для ножек МК !Подробней про это и с картинкой смотри в соответствующей задаче курса.

#define - может содержать и некоторые переменные, вместо которых в тексте программы могут быть подставлены и числа и слова.

Например:

```
#define invbit(p,n) (p=p^bit(n))
```

Здесь переменные величины это 'p' и 'n'. Кроме того в самой правой части эти переменные величины могут быть связаны и арифметическими операциями и таких переменных может быть

много.