

Часть 1.

Первая попытка (моя) сделать FAQ по программированию на С для контроллеров PIC. Использованы материалы сайта [MicrochipC](#), техническая документация от [Microchip Tech Inc](#), [High Tech Software](#). Предложения и вопросы жду [почтой](#). В ближайших планах - расширение FAQ и небольшой цикл статей для начинающих.

- *Все представленные материалы являются некоммерческими и могут использоваться при условии сохранения ссылок на источник.*

- [1. Как я могу из своей программы на С обращаться к ресурсам процессора?](#)
- [2.Как в выражении указать число в десятичной, шестнадцатеричной и двоичной форме?](#)
- [3. Что неправильно в следующем примере?](#)
- [4.Что неправильно в следующем примере?](#)
- [5.Какие типы данных поддерживает HT PICC компилятор?](#)
- [6.Как разместить переменную в нужном мне банке?](#)
- [7.Как я могу изменить значение одного бита в переменной?](#)
- [8.Как проверить состояние бита в переменной?](#)
- [9.Как умножить или разделить на 2 наиболее эффективно?](#)
- [10.Если целое 16-битное \(два байта\), как получить значение одного из байтов в нем?](#)
- [11.Эффективен ли данный код? Делает ли выражение \$x \gg= 8\$ восемь сдвигов?](#)
- [12.Взамен использования \(unsigned char\)\(x>>8\), как в предыдущем вопросе, почему я не могу использовать указатели?](#)
- [13.В чем отличие между этими примерами?](#)
- [14.Могу я использовать inline code для функции?](#)
- [15.В чем ошибка при инициализации порта?](#)
- [16.Что такое переменная типа bit?](#)
- [17.Как описать слово конфигурации процессора в программе?](#)
- [18.Как предварительно занести данные в EEPROM?](#)

- [1.Как я могу из своей программы на С обращаться к ресурсам процессора?](#)

О. Прежде чем вы начнете писать код, ознакомьтесь с соответствующим header файлом. Его можно найти в каталоге include компилятора. Как правило, имя файла достаточно легко ассоциируется с процессором, и вдобавок, в его начале всегда пишется, для каких процессоров он предназначен. Например, для процессора 16F627/628 файл называется pic16f6x.h (я обычно распечатываю этот файл и держу его под рукой, пока не запомню всю мнемонику для регистров конкретного процессора). Дальше все очень просто. Например:

```

TRISB = 0b00001111; // установить разряды RB0...3 на ввод, RB4...7 - на вывод

RB5 = 1; // установить значение порта RB5 в '1'

if (RB0 == 0) // установить значение порта RB4 в зависимости от значения RB0
RB4 = 1;

// функция начальной инициализации (пример)

void Init (void)
{
  INTCON = 0; // никаких прерываний, настраиваемся...
  PIE1 = 0;
  CMCON = 0b00000101; // один компаратор, RA1: Vin-, RA2: Vin+
  VRCON = 0b11000000; // настройка источника опорного напряжения
  T0CS = 0;
  PORTA = 0;
  PORTB = 0;
  TRISA = 0b00000110;
  TRISB = 0b00000111; // RB0...2 - ввод, RB3...7 - вывод
  ...
  ...
} // end Init

```

Причем можно работать как с регистром целиком, так и с отдельными битами. Обратите внимание, что всю возню с переключением банков памяти компилятор делает сам.

2. Как в выражении указать число в десятичной, шестнадцатеричной и двоичной форме?

О. Используйте следующий код:

```

unsigned char x; // 8-битовое целое без знака
x = 255; // десятичное
x = 0377; // восьмеричное
x = 0xFF; // шестнадцатеричное число
x = 0b11111111; // двоичное

```

3. Что неправильно в следующем примере?

```

unsigned char i;
for ( i = 7; i >= 0; i -- ) // будет крутиться всегда !
{
  ...
}

```

О. Ааа! Это очень коварная ошибка. Когда *i* будет равно нулю, цикл уменьшит *i* на единицу и будет проверять условие выхода из цикла. Но, конечно, *i* является беззнаковым целым и после нуля его следующее значение будет равно 255, то есть всегда положительным. Этот цикл будет крутиться до следующей голубой луны! Надо изменить объявление на следующее:

```

signed char i;
for ( i = 7; i >= 0; i -- ) // это работает !
{
  ...
}

```

Предупреждение: цикл, который считается от нуля в сторону увеличения, будет работать всегда при любом типе *i*.

4. Что неправильно в следующем примере?

```
unsigned char i;
for ( i = 0; i <= 255; i ++ ) // будет крутиться всегда !
{
...
}
```

О. Это тоже очень хитрая ошибка. Как и в предыдущем примере забыли о том, что после 255 переменная *i* примет значение 0, а не 256! Надо использовать двухбайтовое целое, например:

```
unsigned int i;
for ( i = 0; i <= 255; i ++ ) // это работает !
{
...
}
```

5. Какие типы данных поддерживает HT PICC компилятор?

О. Следующие типы:

Тип данных	Синтаксис	Размер, байт	Диапазон
однобитовая переменная	bit s;	-	0 или 1
беззнаковое целое 8 бит	unsigned char i;	1	от 0 до 255
знаковое целое 8 бит	signed char i;	1	-127 до 127
беззнаковое целое 16 бит	unsigned int k;	2	от 0 до 65535
знаковое целое 16 бит	signed int k;	2	от -32768 до 32767
беззнаковое целое 32 бит	unsigned long m;	4	от 0 до 4294967295
знаковое целое 32 бит	long m;	4	от -2147483648 до 2147483647
знаковое с плавающей запятой 24 бита	float x;	3	
знаковое с плавающей запятой 24/32 бита	double x;	3/4 *	

* - может быть 24 или 32 бита, в зависимости от опции -D24 или -D32 компилятора.

6. Как разместить переменную в нужном мне банке?

О. По умолчанию компилятор пытается все разместить в банке 0. Конечно, это у него получается не всегда. Желательно относительно редко используемые переменные размещать в других банках. В ряде случаев места в нулевом банке просто не хватает. Например, для размещения кольцевого буфера приемника USART необходимо 64 байта, а свободно в нулевом банке всего 30 и программа еще будет писаться дальше, то есть место потребуется для не менее важных вещей. Пишем так:

```
#define RX_BUFF_SIZE 64 // размер приемного кольцевого буфера

unsigned char RxCount; // переменная по умолчанию в банке 0
bank1 unsigned char RxBuff[RX_BUFF_SIZE]; // переменная в банке 1
```

При дальнейшей работе вы обращаетесь ко всем этим переменным, не задумываясь о банках памяти, компилятор делает переключение банков за вас.

7. Как я могу изменить значение одного бита в переменной?

О. Вы можете использовать следующий код:

```
unsigned char x = 0b0001;
bit_set ( x, 3 ); // сейчас x = 0b1001;
bit_clr ( x, 0 ); // сейчас x = 0b1000;
// определение функций
#define bit_set ( var, bitno ) ((var) |= 1 << (bitno))
#define bit_clr ( var, bitno ) ((var) &= ~ (1 << (bitno)))
```

Можно использовать следующий пример для изменения нескольких битов по маске

```
unsigned char x = 0b1010;
bits_on ( x, 0b0001 ); // сейчас x = 0b1011
bits_off ( x, 0b0011 ); // сейчас x = 0b1000
// определение функций
#define bits_on (var, mask) var |= mask
#define bits_off (var, mask) var &= ~ 0 ^ mask
```

8. Как проверить состояние бита в переменной?

О. Используйте следующий код:

Для проверки одиночного бита в переменной. Помните, что биты в 8-битной переменной всегда считаются справа налево, от 0 до 7. Для примера, бит #0 в 0b00000001 равен 1. Бит #1 равен 0, а наибольший значащий бит #7 - слева.

```
x= 0b1000; //десятичное 8 или шестнадцатеричное 0x8
if ( testbit ( x, 3 ) )
a ( );
else
b ( ); // функция a() будет выполнена

if ( testbit ( x, 0 ) )
a ( );
else
b ( ); // функция b() будет выполнена

if ( ! testbit (x, 0 ) )
b ( ); // функция b() будет выполнена

// определение функции
#define testbit_on ( data, bitno ) ( ( data >> bitno ) & 0x01 )
```

9. Как умножить или разделить на 2 наиболее эффективно?

О. Для умножения на 2 можно использовать операцию сдвига: $x = x \ll 1$ или $x \leq 1$. Второй вариант короче (в написании, но не в генерируемом коде!). Если $x = 2$, или 0v10 в двоичном виде, сдвиг влево даст 0b100 или 4. Для умножения на 4, используйте выражение $x \leq 2$, для умножения на 8 - $x \leq 3$. Для деления на 2 направление сдвига надо поменять: $x = x \gg 1$, или $x \geq 1$. Деление на четыре выглядит так: $x \geq 2$.

В дополнение, многие компиляторы в процессе оптимизации заменяют процедуру деления на процедуру сдвига, что значительно короче по размеру кода.

Примечание: операции сдвига можно применять только с целыми. Не пытайтесь это сделать с float или double, компилятор будет недоволен!

10. Если целое 16-битное (два байта), как получить значение одного из байтов в нем?

О. Используйте следующий код:

```
unsigned int x;  
unsigned char y;  
#define hbyte ( x ) ( unsigned char ) ( x >> 8 )  
#define lbyte ( x ) ( unsigned char ) ( x & 0xFF )
```

Пример:

```
x = 0x1234;  
y = hbyte ( x ); // итак y = 0x12  
y = lbyte ( x ); // итак y = 0x34
```

Примечание: если у компилятора включена оптимизация, то операция сдвига вправо на 8 выполнена не будет, а будет возвращено прямо значение байта. Обязательно нужно использовать операцию приведения типа (см. учебники по C). Применение объединений (union) не всегда желательно, потому что передача функции объединения в качестве параметра более трудное.

Вышеуказанный метод хорош для просмотра переменных в любых банках. Но для их изменения нужно использовать указатели. Для каждого банка необходимы свои определения:

```
#define lbyte_atbank0(x) (unsigned char)((unsigned char *)&x+0)  
#define hbyte_atbank0(x) (unsigned char)((unsigned char *)&x+1)  
#define lbyte_atbank1(x) (unsigned char)((bank1 unsigned char *)&x+0)  
#define hbyte_atbank1(x) (unsigned char)((bank1 unsigned char *)&x+1)
```

и так далее.

11. Эффективен ли данный код? Делает ли выражение `x >>= 8` восемь сдвигов?

```
unsigned int x;  
unsigned char y;  
x = 0xFF00;  
y = (unsigned char) (x>>8); // теперь y = 0xFF
```

О. Нет. Это очень эффективный код - компилятор Hi-Tech C ссылается прямо нужный байт в переменной x. Проследите, чтобы была включена глобальная и локальная оптимизация. Но конечно, выражение `y = x >> 7` заставить компилятор сделать семь сдвигов, что займет время и место.

12. Взамен использования `(unsigned char)(x>>8)`, как в предыдущем вопросе, почему я не могу использовать указатели?

О. Это возможно, но для доступа к различным банкам требуется различные `#define` для переменных в каждом банке. Проще и эффективнее использовать `(unsigned char)(x>>8)`, если только вы не пытаетесь модифицировать различные байты в этих переменных.

13. В чем отличие между этими примерами?

```
x ++;  
if ( x >= 4)  
x = 0;
```

```
x ++;  
if ( x == 4 )  
x = 0;
```

О. Отличие малозначительное, но первый пример несколько лучше. Если шальной указатель или пропадание напряжения (brownout) изменят значение x, то проверка на строгое равенство может привести к более печальным последствиям.

14. Могу я использовать inline code для функции?

О. Для многих компиляторов выражение inline при объявлении функции означает, что каждый раз при вызове этой функции будет подставляться ее код. Это уменьшает количество переходов и экономит стек, но увеличивает размер кода. Hi-Tech C не поддерживает 'inline'.

Однако, если нельзя, но очень хочется, то можно. Для этого используется #define. Если текст планируемой "inline" функции не помещается в одну строку, то используйте знак '\' после каждой строки. Для примера:

```
#define DelayUs(x) { unsigned char _dcnt; \
    _dcnt = x; \
    while(--_dcnt != 0) \
    {asm ("nop");\
    continue; }}
```

Каждый раз при вызове функции DelayUs (0x55) будет вставляться непосредственно ее код:

```
:DelayUs (0x55);
0007C5 MOVLW 0x55
0007C6 MOVWF _dcnt
0007C7 NOP
0007C8 DECFSZ _dcnt, 0x1
0007C9 GOTO 0x7c7
```

Достаточно много примеров использования #define для вставки кода имеется в h-файла компилятора.

15. В чем ошибка при инициализации порта?

```
//эти четыре строки правильные
#define INPUT 1
#define OUTPUT 0
#define DATA RB1
#define DATA _DIRECTION TRISB0
```

```
// метод 1 - вначале направление, потом значение в порт (правильно!)
DATA _DIRECTION=OUTPUT;
DelayUs(10); // задержка 10 мкс
DATA =1;
```

```
// метод 2 - вначале значение порта, потом направление (НЕправильно!)
// ошибка - значение порта может остаться в 0 !
DATA =1;
DelayUs(10); // задержка 10 мкс
DATA _DIRECTION=OUTPUT;
```

Два способа установки описаны выше, первый правильный, а второй нет. Если используется второй способ, порт DATA должен быть установлен в 1, но пока порт остается в режиме ввода, он может быть изменен опять в 0, если на входе порта присутствует 0. Когда порт будет настроен на вывод, его значение может быть как 0, так и 1.

Эта ситуация достаточно подробно описана в руководствах по микроконтроллерам PIC. Правило: если меняется направление работы порта ввод => вывод, всегда вначале устанавливается направление порта, а затем значение.

16. Что такое переменная типа bit?

Среди типов, разрешенных к использованию, есть битовые переменные. Для их хранения компилятор создает специальную секцию rbit_n (где n - номер банка памяти). Применение битовых переменных очень удобно для представления двоичных флагов, но существуют некоторые ограничения. Значение ее может быть 0 или 1. Битовая переменная не может быть класса auto и не может быть параметром функции, но функция может возвращать переменные битового типа. Нельзя объявлять указатель на битовую переменную и выполнять статическую инициализацию (то есть нельзя объявить `bit test_flag = 1;` это ошибка), инициализацию нужно выполнять в начале вашего кода. Если битовой переменной присваивают значение переменной другого типа, из нее берется только младший значащий бит. При объявлении внутри функции обязательно применение ключевого слова `static`, а при объявлении вне функции - нет.

```
bit my_flag; // объявление глобальной битовой переменной
```

```
void main (void)
{
    static bit flag; // объявление внутри функции
    . . .

    . . .
}
```

Битовые переменные обнуляются автоматически по старту программы при исполнении начального кода, который всегда генерирует компилятор. Эффективно ли использование битовых переменных? Однозначно, да. В один регистр компилятор упаковывает до 8 битовых переменных, значит на восемь флагов тратится только один байт. Проигрываем ли мы в скорости? Нет, так как процессор имеет эффективные команды изменения и проверки одиночных битов.

17. Как описать слово конфигурации процессора в программе?

Хорошим стилем программирования является привычка не полагаться на умолчания. Все что можно описать - описывается.

Описать слово конфигурации в тексте программы (лучше это делать в начале) можно с помощью макроса `__config` следующим образом (пример):

```
__CONFIG (XT &WDTDIS &PWRTEEN &MCLRDIS &BORDIS &LVPPDIS &UNPROTECT);
```

^^ два символа подчеркивания!!!

В этом случае слово конфигурации будет вставлено в результирующий hex-файл и будет записано программатором по нужному адресу.

В предыдущих версиях компилятора для объединения параметров использовался знак операции "или" - `|`, начиная с версии 8 и далее - используется знак `&`. Как узнать перечень аббревиатур для параметров конфигурации - очень просто, из 'h' файла для выбранного процессора (обычно ближе к концу файла), файлы с описанием процессоров находятся в каталоге "include" компилятора.

18. Как задать нужные значения для eeprom?

В ряде случаев нужно предварительно заполнить EEPROM константами, которые будут использоваться в дальнейшем при работе программы. Для этого в исходный текст

программы нужно включить следующее определение
(например для заполнения 32 байт, начиная с адреса 0):

```
__EEPROM_DATA (0x41, 0xA0, 0x42, 0xA1, 0xE0, 0x45, 0xA3, 0xA4);  
__EEPROM_DATA (0xA5, 0xA6, 0x4B, 0xA7, 0x4D, 0x48, 0x4F, 0xA8);  
__EEPROM_DATA (0x50, 0x43, 0x54, 0xA9, 0xAA, 0x58, 0xE1, 0xAB);  
__EEPROM_DATA (0xAC, 0xE2, 0xAD, 0xAE, 0x20, 0xAF, 0xB0, 0xB1);
```

^^ два символа подчеркивания!!!

Этот текст нужно включать вне какой-либо функции. Память заполняется последовательно, поэтому, если нужно вставить данные по определенному адресу, предыдущие адреса нужно заполнить любой константой. В результирующий hex-файл будет включена необходимая для записи EEPROM информация и программатор запишет ее.

Есть альтернативный способ, позволяющий записывать данные по любому адресу (код любезно предоставлен Владимиром Раздолиным, rasdolin@mail.ru):

```
#define __EEPROM_FILL(w) asm("\tpsect eedata,delta=2,abs,ovrld");\  
asm("\torg 2100h");\  

```

```
asm("\trept 256"); \  
asm("\tdb " #w); \  
asm("\tendm")
```

```
#define __EEPROM_STRING(off,s,w) asm("\tpsect  
eedata,delta=2,abs,ovrld");\  
asm("\torg 2100h+"#off);\  
asm("\tirpc\t__arg, "#w); \  
asm("\tdb  
'__arg'"); \  
asm("\tendm")
```

далее в тексте пишем:

```
__EEPROM_FILL( 0 ); // всю EEPROM заполнили нулями  
__EEPROM_STRING( 0x10, my_string ); // по смещению 0x10 записали строку
```

Не забудьте, при необходимости, изменить размер EEPROM в определении `asm("\trept 256");`, где 256 - размер в байтах.