Особенности и рабочая среда HI-TECH

PICC

Содержание

Введение

Данный документ является вольным переводом “PICC Manual” Copyright © 2002 HI-TECH Software (Email: hitech@htsoft.com; Web: http://www.htsoft.com; FTP: ftp://ftp.htsoft.com).

В документе рассматриваются основные особенности реализации и способы использования специфических возможностей микроконтроллеров. Для микроконтроллеров фирмы Microchip существует несколько различных реализаций компилятора языка программирования С, однако имеено эта наиболее близка к стандарту. Лично для меня является очень удобной возможность отлаживать значительные фрагменты кода на ПК, а затем компилировать программу для контроллера. Кроме этого, нет необходимости разбираться с дополнительными командами и библиотеками которые реализуют управление контроллером.

Автор перевода Яловой Илья (www.ar2.mksat.net). Последнюю версию перевода и другие документы, посвященные этой теме, вы можите найти на моем сайте.

Дата последней модификации документа: 15.11.03.

HI-TECH C поддерживает ряд специальных возможностей и расширений языка С и оптимизирован для создания приложений, предназначенных для записи в ПЗУ. После прочтения этого руководства вы должны знать и уметь:

- конфигурирование стандартных процедур ввода – вывода и использовать функции библиотеки <stdio.h> применительно к своему оборудованию;
- обеспечивать поддержку прерываний, используя только С код;
- программировать устройства ввода – вывода, используя только С код;
- объединять С и Ассемблер код в одном проекте, используя внутренние или внешние функции.

Благодарности:

Хочу выразить искреннюю благодарность всем, кто принял участие в подготовке этого перевода:

- Андрей ... - корректура опечаток и орфографических ошибок;
- Земфир ... - обнаружил и помог исправить ерунду в описание 32-х типов данных.

1 Общие сведения

1.1 Соответствие стандарту ANSI C

PIC C отличается от стандарта ANSI C только в плане реализации рекурсивного вызова функций, но это отличие обусловленно исключительно особенностями архитектуры ядра контроллеров фирмы Microchip. Если более конкретно, то в силу аппаратных ограничений в работе стека и малого объема оперативной памяти, в PIC C рекурсия не поддерживается.

1.2 Поддерживаемые процессоры

PIC C поддерживает огромное количество процессоров фирмы Microchip. Полный список можно найти на сайте фирмы разработчика компилятора (см. выше). Добавить поддержку более новых процессоров можно редактируя файл *picinfo.ini* в каталоге *lib*. Этот файл разделен на секции, описывающие процессоры начального, среднего и крутого уровня. А вот процессоры, определенные пользователем должны размещаться в конце файла. Дополнительные пояснения по описанию новых процессоров приводятся в заголовке файла.

1.3 Стандартные библиотеки

В состав PIC C входит несколько стандартных библиотек. Каждая из них содержит функции, описанные в документе “Библиотечные функции”. На рисунке 1 показаны правила именования файлов стандартных библиотек. Рассмотрим подробнее назначение каждого поля:

- тип процессора всегда равен 'pic';
- класс процессора может принимать следующие значения: '2' для начального уровня, '4' для среднего уровня и '7' для крутых процессоров (имеется в виду 17-я серия :-);
- количество банков ОЗУ, деленное на 2;
- количество банков ПЗУ, деленное на 2;
- тип double определяет точность расчетов: '-' – 24-х битное представление чисел, и 'd' – 32-х битное;
- тип библиотеки равен 'c' для стандартной библиотеки, 'l' – для библиотеки, связанной исключительно с работой функции 'printf()' с поддержкой печати чисел типа long, 'f' – для библиотеки, связанной исключительно с работой функции 'printf()' с поддержкой печати чисел типа long и float;

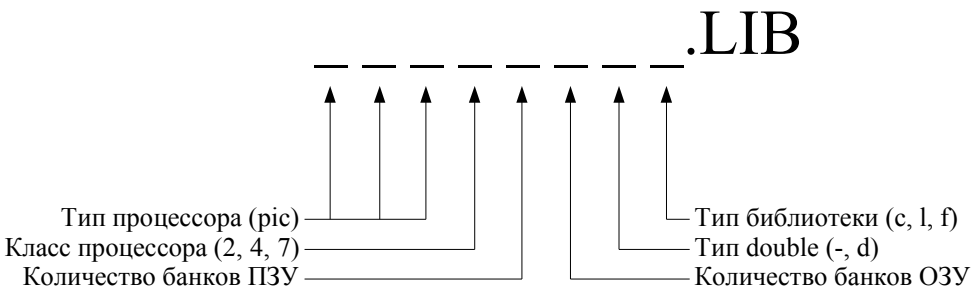

Рис. 1 Правила именования стандартных библиотек

Функция printf() реализована, но имеет некоторые ограничения. Подробнее смотрите далее по тексту.

1.4 Формат выходного файла

Компилятор непосредственно поддерживает несколько форматов выходного файла. Это форматы, которые наиболее часто используются в ППЗУ программаторах и внутрисхемных отладчиках.

Если вы используете интегрированную среду разработчика HPDPIC, то вы можите, с помощью меню “Options”, установить для компилятора один из перечисленных далее форматов:

- Motorola Hex;
- Intel Hex;
- Binary;
- UBROF;
- Tektronix Hex;
- American Automation symbolic hex;
- Bytecraft .COD;
- Библиотечный файл.

По умолчанию компилятор создает выходные файлы в формате Bytecraft .COD и Intel Hex. Если явно не указано имя выходного файла, то в качестве последнего используется имя первого исходного файла.

Дополнительно может создаваться символьный файл, который используется отладчиками и симуляторами для обеспечения отладки на уровне символов и исходных текстов.

Пример создания символьного файла с названием “test.sym”:

```
PICC -16c84 -Gtest.sym test.c
```

Символьный файл, созданный этой командой, может быть использован при работе с внутрисхемным отладчиком.

Компилятор предоставляет некоторые макросы, которые однозначно определяют тип процессора, модель памяти и пр. (см. Таблица 1)

Таблица 1 Предопределенные макросы (символы)

Символ	***Установлен***	***Использование***
HI_TECH_C	Всегда	Указывает на использование HI-TECH C
_HTC_VER_MAJOR_	Всегда	Целая часть версии компилятора
_HTC_VER_PATCH_	Всегда	Десятичная часть версии компилятора
MPC	Всегда	Код компилируется для контроллеров PIC
_PIC12	Для 12-битных	Указывает на контроллер начального класса
_PIC14	Для 14-битных	Указывает на контроллер среднего класса
_PIC17	Для 16-битных	Указывает на контроллер крутого класса
COMMON	Если имеется общая память	Указывает на наличие общей памяти
BANKBITS	0,1 или 2	Равен количеству банков ОЗУ
GPRBITS	0,1 или 2	Равен количеству банков ОЗУ общего назначения
PROGMEM	Если устройство может писать-читать память программ. (1 или 2)	Равен 1, если устройство может читать память программ. Равен 2, если устройство может читать и писать память программ.
MPLAB_ICD	Если используется ICD	Указывает на то, что код был создан для MPLAB In-Circuit Debugger

1.5 Специфика контроллера

Специфические особенности конкретного процессора отражены в файле заголовка. Собственно любую вашу программу целесообразно начинать с указания стандартного файла заголовка:

```
#include <pic.h>
```

Компилятор сам загрузит файл заголовка, предназначенный для указанного в проекте контроллера. Поддержка процессора заключается в назначении семантических имен служебным регистрам, отдельным битам и описании сервисных макросов. Для контроллеров, оснащенных ЭСПЗУ, предоставляются макросы для чтения и записи ЭСПЗУ:

```
EEPROM_WRITE( <адрес> , <значение> );// Запись байта
<переменная> = EEPROM_READ( <адрес> );// Чтение байта
```

Для удобства программиста константа EEPROM_SIZE содержит полный размер ЭСПЗУ.

Если контроллер позволяет писать и читать память программ, то облегчат эту задачу следующие макросы:

```
FLASH_WRITE( <адрес> , <значение> );// Запись байта
<переменная> = FLASH_READ( <адрес> );// Чтение байта
```

Настроить параметры контроллера при его программировании поможет макрос:

```
__CONFIG( x );
```

Где “x” – слово конфигурации контроллера. Допустим нам необходимо запретить сторожевой таймер, отключить защиту кода и данных и настроить контроллер на работу с генератором типа XT. Для этого нам придется записать одну очень наглядную строчку:

```
__CONFIG( WDTDIS & XT & UNPROTECT);
```

Однако следует помнить, что этот макрос должен выполняться в начале программы, но после директивы #include <pic.h>, так как именно эта директива подгружает файл заголовка, содержащий описание конфигурационных констант для данного контроллера.

Некоторые процессоры оборудованы дополнительной памятью, которая находится за пределами адресного пространства и может быть использована для сохранения некой константы необходимой для работы программы, например серийного номера изделия. Для доступа к этой памяти используется макрос:

```
__IDLOC( x );
```

Иногда возникает необходимость непосредственно при прошивке контроллера заполнить имеющееся ЭСПЗУ или его часть, если оно вообще есть, некоторыми данными, например таблицей, описывающей какую-то сложную зависимость. Сделать это не просто, а очень просто! Достаточно записать следующий код:

```
__EEPROM_DATA (0,1,2,3,4,5,6,7);
```

Этот макрос ожидает 8 параметров, которые он последовательно размещает в ЭСПЗУ. Каждое значения должно быть типа байт. Рекомендуется размещать этот макрос за пределами описания функций. Но программист должен отдавать себе отчет в том, что эти данные будут занесены в ЭСПЗУ не во время выполнения (запуска) программы, а при прошивке. То есть, если в случае несчастного случая или взрывов на солнце (магнитных бурь) будет поврежден или изменен участок ЭСПЗУ, то предсказать поведение программы, да и всего устройства будет сложно... А вот для оперативного доступа к этой памяти непосредственно из программы следует использовать описанные выше макросы EEPROM_WRITE() и EEPROM_READ().

Отдельно хочется остановиться на операциях с битами, так как использовать их приходится очень часто при программировании на языке ассемблера. При решении этой проблемы компилятор PICC проявляет завидную догадливость. Собственно при анализе С – кода он автоматически определяет где можно задействовать битовые операции ассемблера. Вот небольшой пример:

```
int a;
a |=0x40;
```

после компиляции будет получен следующий код:

```
bsf _a,6
```

Для особого удобства рекомендуется самостоятельно определить два чертовски полезных макроса:

```
#define bitset (var,bitno) ((var) |= 1<< (bitno))
```

```
#define bitclr (var,bitno) ((var) &= ~(1<< (bitno)))
```

Тогда можно будет записывать выражения типа:

bitset(a,6);

что есть суть то же, что и в примере выше, но более наглядно.

Имеются так же и другие специфические команды, для поддержки контроллеров начального класса и хитрых контроллеров типа PIC14000, но с ними можно легко разобраться самостоятельно, тому кому это интересно.

2 Простые типы данных и переменные

2.1 Поддерживаемые типы данных

Компилятор PICC поддерживает базовые типы данных размером в 1, 2, 4 байта. При размещении многобайтовых значений используется формат известный как little endian, то есть наименее значащий байт размещается первым. Таким образом значение размером в слово будет размещено таки образом, что наименее значащий байт окажется в ячейке памяти с меньшим адресом. Аналогично размещаются и значения размером в двойное слово. В таблице 2 приведено описание основных типов данных, которые поддерживает компилятор.

Таблица 2 Типы данных

Тип	***Размер***	***Арифметический тип***
bit	1	Булевый тип
char	8	Целое со знаком или без
unsigned char	8	Целое без знака
short	16	Целое со знаком
unsigned short	16	Целое без знака
int	16	Целое со знаком
unsigned int	16	Целое без знака
long	32	Целое со знаком
unsigned long	32	Целое без знака
float	24	Действительное число
double	24 или 32	Действительное число

2.2 Форма записи литералов

PICC поддерживает формат записи литералов в соответствии со стандартом ANSI. Практика показывает, что при программировании микроконтроллеров, часто возникает необходимость записывать литералы в двоичном формате. И, хотя в стандарте такая возможность отсутствует, ее добавили в PICC. При записи двоичных и шестнадцатиричных литералов можно использовать как заглавные, так и строчные буквы (см. Табл. 3).

Таблица 3 Форма записи литералов

Литерал	***Формат***	***Пример***
Двоичный	0b<число> или 0B<число>	0b10011001
Восьмеричный	0<число>	0345
Десятичный	<число>	129
Шестнадцатиричный	0x<число> или 0X<число>	0x2F

Необходимо четко представлять себе как компилятор работает с литералами. Получив число он соотносит его с тем типом данных в размер которого он вписывается без переполнения и потери точности. Иногда это приводит к ошибкам преобразования типов или вообще ошибочным результатам вычислений. Поэтому существует механизм явного указания типовой принадлежности литерала. Для этого используются специальные

суффиксы. Так суффикс “l” или “L” показывает, что литерал должен рассматриваться как принадлежащий к типу long. Суффикс “u” или “U” указывает компилятору, что рассматриваемое целое значение не имеет знака (тип целое без знака). Возможно использовать комбинацию суффиксов. Результат, получаемый при этом, вполне предсказуем.

Если константа не может быть соотнесена ни с одним из целых типов, то она рассматривается как double. Чтобы явно указать ее принадлежность к типу float, что кроме прочего экономит память, необходимо воспользоваться суффиксом “f” или “F”.

Символьные константы (char) записываются в одинарных кавычках типа 'a' и могут быть размещены в переменных типа char. многобайтные символьные константы и переменные не поддерживаются.

Строковые литералы (константы) записываются в двойных кавычках, например “мама мыла раму”. Строковые константы сохраняются как **const char *** в ПЗУ. Рассмотрим следующие примеры:

```
char * cp         = “one”;
const char * ccp  = “two”;
char ca[]         = ”two”;
```

В первом случае указатель не const в результате чего будет выведено соответствующее предупреждение. Второй пример совершенно корректен и приведет к формированию в ПЗУ указанной строчки, а доступ к ней можно будет получить посредствам указателя **ccp**. Последний пример надо рассмотреть особо. Так в этом случае мы хотим получить строковую переменную (строку, размещенную в ОЗУ и позволяющую производить модификации). В этом случае будет создан код, который при запуске программы будет выделять в ОЗУ массив требуемого размера и переносить в него из ПЗУ указанную строку.

2.3 Тип данных – бит

Как уже несколько раз отмечалось, операции с битами имеют важное значение и часто используются при программировании микроконтроллеров. Чтобы обеспечить максимальное удобство программиста при решении задач, связанных с манипуляциями отдельными битами, был введен дополнительный тип данных – бит (bit). Переменные этого типа описываются как обычно:

```
static bit over_flag;
```

переменные типа бит не могут иметь модификатор auto и не могут передаваться в функцию в качестве аргумента, но функция может возвращать значение типа бит. По большому счету переменные этого типа ведут себя как обычные переменные типа unsigned char, но при этом могут принимать значение 0 или 1. Поэтому их удобно использовать для определения различных логических переменных и флагов. Такой подход значительно экономит оперативную память. Компилятор не позволяет создавать указатели типа бит или статически инициализировать переменные этого типа. Все операции с переменными типа бит выполняются с помощью бит-ориентированных инструкций ассемблера, насколько это вообще возможно, при этом создается очень эффективный и компактный код.

Если попытаться присвоить переменной типа бит целое значение, то будет сохранен только младший бит. Если необходимо присвоить переменной типа бит 1 или 0 в зависимости от того равна нулю некоторая переменная или нет, то необходимо использовать следующий код:

```
bitvar = char_var != 0;
```

Все переменные этого типа упаковываются таким образом, что 8 переменных займут до одного байта.

2.4 Использование бит-адресуемых регистров

Возможность создавать битовые переменные может быть использована, для удобного доступа к отдельным битам служебных регистров. Дело в том, что мы можем описывать переменные с указанием абсолютного адреса:

```
static unsigned char STATUS @ 0x03;
```

При описании битовых переменных с указанием абсолютного адреса необходимо указывать абсолютный номер бита в непрерывном битовом поле, которое образуется из последовательности регистров. Например 3-тий бит в ячейке памяти с адресом 5 будет иметь абсолютный номер равный 5*8+3. Таким образом, если мы хотим эксклюзивно обращаться к биту PD регистра STATUS, то достаточно совершить следующий реверанс:

```
static bit PD @ (unsigned)&STATUS*8 + 3;
```

Справедливости ради, хочу напомнить, что все стандартные регистры и служебные биты уже описаны до нас и за нас в файле заголовка <pic.h>.

Лично от себя могу посоветовать использовать для описания подобных вещей следующий макрос:

```
#define     PORTBIT(adr, bit) ((unsigned)(&adr)*8+(bit))
```

Используя этот скрипт, вы упрощаете себе жизнь и повышаете читабельность ваших программ:

```
static bit PD @ PORTBIT(STATUS, 3);
```

или более жизненный пример:

```
static unsigned char ptKeys @ &PORTB;
static bit btLeft  @ PORTBIT(ptKeys, 0);
static bit btRight @ PORTBIT(ptKeys, 1);
static bit btDown  @ PORTBIT(ptKeys, 2);
static bit btUp    @ PORTBIT(ptKeys, 3);
```

2.5 8-бит – целые

Компилятор PIC C поддерживает символьный тип как с указанием знака так и без. Для микроконтроллеров среднего класса этот тип является основным. По-умолчанию символьный тип (char) считается без учета знака. Чтобы это изменить можно воспользоваться опцией компилятора SIGNED_CHAR. Если использован указанный параметр, то возможно использование целых с учетом знака в диапазоне от -128 до +127 включительно или без учета знака (unsigned) от 0 до 255 включительно. В ламерской среде существует предрассудок, что тип данных **char** предназначен исключительно для хранения кодов символов ASCII. Но это глубокое заблуждение, тем более теперь, когда повсеместно применяются 16-и битовые кодировки. Реально же тип **char** это наименьший из существующих целых типов и ничем не отличается от своих больших собратьев. Исторически сложилось так, что на компьютерах, где C получил наибольшее распространение использовались кодировки, для которых тип char был идеальным хранилищем символов. Использовать же его для хранения чисел было не выгодно, так как адресация памяти была как минимум словами (16 бит), да и диапазон маловат для прикладных ПС программ. А вот для контроллера (конкретно того семейства о котором идет речь в этом труде) char – это родной тип и с ним он работает очень эффективно. Так что, если вы планируете использовать C для разработки встроенных систем на базе PIC-контроллеров, то привыкайте использовать тип **char** как основной.

2.6 16-бит – целые

Для представления 16-и битных данных в PIC C есть 4 типа **short**, **int**, **unsigned short**,

unsigned int. Котрые могут вмещать числа из диапазонов от -32768 до 32767 с учетом знака и от 0 до 65535 без учета знака. Как мы видим, в **PIC C** два типа **int** и **short** практически не имеют отличий. Это связано со стремлением соответствовать стандартам. И это очень серьезный довод, тем более что 8-ми битные целые числа поддержаны в полном объеме посредствам типа **char**, поэтому последний рекомендуется использовать вместо **int**, где это только возможно.

2.7 32-бит – целые

Компилятор PIC C поддерживает два вид целых 32 битных чисел – **long** и **unsigned long**. Они могут содержать числа от -2,147,483,648 до 2,147,483,647 включительно и от 0 до 4,294,967,295 соответственно. В соответствии со стандартом ANSI на язык C, 32 бита – это наименьший размер для типа long.

2.8 Числа с плавающей запятой

Числа с плавающей запятой реализованы в соответствии со стандартом IEEE 754 (32 бита) и модифицированный IEEE 754 (24 бита). Для представления чисел с плавающей запятой используется два типа **float** и **double**. По умолчанию оба типа реализуют урезанный стандарт. Если требуется повышенная точность в вычислениях, то необходимо воспользоваться опцией компилятора **-D32**. В этом случае для типа double бедет использован полный IEEE 754.

В соответствии со стандартом число с плавающей запятой представлено тремя полями (см. табл. 4):

- **знак** – это один бит определяющий знак числа);
- **показатель степени** – это 8-и битное дополнение до 127 показателя степени, если реальный показатель степени равен 0 то в это поле будет записано число 127;
- **мантисса** – это мантисса числа (правая часть от десятичной точки), значащая часть числа с левой стороны от десятичной точки всегда равна 1 за исключением случая, когда само число равно нулю. Индикатором равенства всего числа нулю является равенство нулю показателя степени.

Зная чему равны отдельные поля, можно легко расчитать чему равно сохраненное число, воспользовавшись следующей формулой:

$$Число = (-1)^{Знак} \cdot 2^{(Показатель\,степени - 127)} \cdot 1.мантисса \quad .$$

Несколько примеров записи чисел с плавающей запятой в разных форматах приведены в таблице 5.

Таблица 4 Форматы записи чисел с плавающей запятой

Формат	***Знак***	***Смещенный показатель степени***	***мантисса***
IEEE 754 (32 бита)	x	xxxx xxxx	xxx xxxx xxxx xxxx xxxx xxxx
Уркзанный IEEE 754 (24 бита)	x	xxxx xxxx	xxx xxxx xxxx xxxx

Таблица 5 Примеры записи чисел с плавающей запятой

Формат	***Число***	***Смещенный показатель степени***	***1.мантисса***	***Десятичное число***
IEEE 754 (32 бита)	7DA6B69Bh	11111011b (251)	1.01001101011011010011011b (1.302447676659)	2.77000e+37
Уркзанный IEEE 754 (24 бита)	42123Ah	10000100b (132)	1.001001000111010b (1.142395019531)	36.557

Необходимо запомнить, что самый старший бит мантиссы (бит, который рсположен слева от десятичной точки) не сохраняется и считается равным 1, кроме случая, когда показатель степени равен нулю, что означает, что все чило равно нулю.

2.9 Статические переменные

Глобальные или статические переменные могут располагаться по абсолютному адресу. Для этого в описании переменной используется специальная конструкция "@ <адрес>". Например:

```
volatile unsigned char Portvar @ 0x06;
```

Такое описание дает возможность обращаться к абсолютному адресу 0x06 посредствам переменной **Portvar**. Надо учитывать, что в этом случае компилятор не резервирует память под размещение этой переменной. С точки зрения ассемблера это описание выглядит следующим образом:

```
_Portvar equ 06h
```

Запомните, что ни компилятор ни компоновщик не производят никаких проверок на счет перекрытия абсолютных переменных. Другими словами, ответственность за правильное размещение абсолютных переменных полностью ложиться на плечи программиста.

Основной целью данной конструкции является дать возможность напрямую связывать служебные регистры контроллера с переменными. Чтобы разместить переменные, описанные пользователем, по абсолютным адресам, необходимо поместить их в одну секцию и указать компоновщик разместить эту секцию по определенному адресу. Подробнее смотрите описание директивы **#pragma psect**.

3 Структурные типы данных и строки

3.1 Структуры и строки

HI-TECH C поддерживает структурные типы данных **struct** и **union** любого размера от одного байта и более, но не стоит забывать и о физических размерах доступной памяти. Переменные структурного типа могут быть переданы в качестве аргумента в функцию и возвращены ей в качестве результата. Позволю себе заметить, что, в соответствии со стандартом, аргументы передаются в функцию по значению, то есть намного гуманнее по отношению к объему кода и расходу памяти передавать не сами структуры а указатели (ссылки) на них. Тем более, что PIC C полностью поддерживает указатели на структурные типы.

3.2 Битовые поля

Внутри структурных типов могут быть организованы, так называемые, битовые поля. Такое поле начинается с младшего бита того слова в котором оно будет размещаться. Битовые поля размещаются внутри 8-ми битных слов. Если поле не удается разместить в текущем байте структуры, то размещение продолжается в следующем байте. Битовые поля никогда не пересекают границу между 8-ми битовыми областями, выделенными для данной структуры. Например:

```
struct {
unsigned hi    : 1;
unsigned dummy : 6;
unsigned lo    : 1;
} foo @ 0x10;
```

В данном случае структура будет занимать один байт по адресу 0x10. При этом поле **hi** будет соответствовать нулевому биту, а **lo** – 7-му биту регистра по адресу 0x10. Младший бит поля dummy будет располагаться в первом, а старший – в 6-м битах регистра 0x10. Если структура описана как показано выше, то она будет размещена по абсолютному адресу и память под ее размещение выделяться не будет.

Если необходимо определить битовое поле, которое содержит неиспользуемые области, то можно указать безымянные поля:

```
 struct {
unsigned hi : 1;
unsigned    : 6;
unsigned lo : 1;
} foo @ 0x10;
```

3.3 Строки в ОЗУ и ПЗУ

Анонимные неизменные строки (строковые литералы) всегда располагаются в ПЗУ и могут быть доступны только через **const-**указатели. В следующем примере строка “Мама мыла раму” является строковой константой и размещена в ПЗУ. Поэтому к ней можно обращаться только через константный указатель:

```
#define MATHER “Мама мыла раму”
SendBuff(MATHER);
```

Массивы переменных (в том числе изменяемые строки) должны быть проинициализированы как в следующем примере:

```
char fred[] = “Мама мыла раму”;
```

В этом случае мы получаем массив в ОЗУ. Этот массив инициализируется строкой “Мама мыла раму” при запуске программы. Образец строки хранится в ПЗУ. Анонимные неизменные строки в другом контексте будут квалифицированы как константы и доступны

непосредственно в ПЗУ.

Если необходимо передать строку в качестве аргумента функции, или сослаться на нее с помощью указателя, то указатель должен быть описан как **const char ***:

```
void SendBuff(const char * ptr)
{
...
}
```

В этом случае вы вольны передавать в качестве аргумента как указатель на строку в ПЗУ, так и на строку в ОЗУ. Данные будут корректно обработаны в обоих случаях.

4 Управление размещением переменных

4.1 Квалификаторы типа const и volatile

Для эффективного размещения и доступа к переменным компилятору необходимо сообщить дополнительную информацию. К такой информации информации относятся следующие сведения:

- будет ли переменная изменятся программно (константы);
- будет ли переменная изменяться помимо программы (порты и прочие регистры специального назначения);
- должна ли переменная обнулиться (инициализироваться) при перезапуске или запуске программы;
- в каком банке памяти должна размещаться переменная.

В соответствии с этими дополнительными данными осуществляется в том числе и оптимизация окончательного кода.

PIC C компилятор поддерживает такие стандартные квалификаторы, как **const** и **volatile**. Квалификатор типа const указывает на неизменность объекта. Любая попытка изменить содержимое переменной, помеченной квалификатором const, приведет к выдаче компилятором соответствующего сообщения. Все объекты, помеченные пользователем как const размещаются в специальной секции, которая называется “строка в ПЗУ”. Все такие объекты должны инициализироваться при описании, так как в дальнейшем их значений не может быть изменено. Пример:

```
const int version = 3;
```

Квалификатор **volatile** сообщает компилятору, что нельзя гарантировать сохранность (неизменность) значения этой переменной между двумя удачными обращениями к ней из программы. Имеются в виду служебные регистры, значение которых может меняться помимо программы. Это предотвращает удаление “лишних” ссылок на этот объект на этапе оптимизации кода, что, в данном случае, может изменить поведение программы. Кроме всего, этим квалификатором надо помечать и те переменные, которые изменяются процедурами обработки прерываний. Пример:

```
volatile unsigned char P_A @ 0x05;
```

К таким объектам организуется процедура доступа отличная от обычной. Например, если вы в программе присвоете обычной переменной занчение 1, то переменная будет очищена и инкрементирована, та же операция с переменной, отмеченной как **volatile**, произведет запись единицы в рабочий регистр с последующим переносом значения из рабочего регистра в память.

4.2 Специальные квалификаторы типов

Контроллерная ориентация компилятора требует наличия дополнительных (специальных) квалификаторов. К таким квалификаторам относятся: **persistent**, **bank1**, **bank2**, **bank3**. Если использована опция компилятора -STRICT, то эти квалификаторы надо записывать в виде: **__persistent**, **__bank1**, **__bank2**, **bank3**. Эти квалификаторы могут применяться и к указателям, но они несовместимы с переменными класса **auto**. Если вы хотите их использовать для переменных локальных по отношению к функции, то необходимо добавить ключевое слово **static**. Пример:

```
void func (void)
{
     persistent int intvar1; // Неправильно (класс auto)!!!
     static persistent int intvar2; // Правильно
     ...
     //  Тело функции.
     ...
```

```
}
```

По умолчанию любая переменная, если она не была явно инициализирована, обнуляется. Это не противоречит стандарту языка С. Однако, бывают случаи, когда данные сохраняются после рестарта или даже цикла включения-выключения. Чтобы исключить очистку переменной при старте программы нужно воспользоваться квалификатором **persistent**. Все переменные помеченные таким образом размещаются отдельно. Существует несколько библиотечных функций, которые позволяют проверять и инициализировать такие переменные.

Квалификаторы **bank1**, ... – используются для размещения переменных в указанной странице памяти. Они неприменимы для указателей в микроконтроллерах начального класса.

Не существует квалификатора **bank0**, так как переменные по умолчанию, в том числе аргументы функций и объекты auto, размещаются в нулевой странице. Примеры:

```
static bank3 unsigned char Masha;
bank3 unsigned char * ptrMasha;
static bank3 unsigned char * bank2 ptrMasha2;// Собственно указатель будет
размещен во второй странице.
```

4.3 Указатели

Все указатели в контроллерах начального класса 8-ми битные, поэтому подробно останавливаться на них не буду.

Рассмотрим особенности реализации указателей для контроллеров среднего класса:

- *Указатели на ОЗУ*

Поскольку 8-ми битный указатель может адресовать только 256 байт, то они могут дать доступ только к 0-й и 1-й страницам ОЗУ.

- *Указатели на 2-ю и 3-ю страницы ОЗУ*

Для доступа к верхним страницам ОЗУ необходимо использовать соответствующие модификаторы (Bank2, Bank3). Заметьте, что для микроконтроллеров этого класса невозможно определить указатель на ОЗУ, который бы давал доступ к более чем двум страницам или парам страниц, отличных от указанных выше.

- *Константные указатели*

Константные указатели – 16-ти битные и могут давать доступ как к ОЗУ так и к ПЗУ. Если старший разряд указателя равен нулю, что адресуется ОЗУ, иначе – ПЗУ. Такие указатели могут использоваться только для чтения данных, независимо от типа адресуемой памяти (ОЗУ или ПЗУ). Другими словами, осуществлять запись в ОЗУ с помощь. Константного указателя нельзя, что собственно следует из его определения. Но польза от них, тем не менее, очень существенная. Особенно удобно их использовать в функциях, связанных с обработкой строк. Передавая в функцию константный указатель на строку, вам не надо беспокоиться о ее местоположении (ОЗУ или ПЗУ). В любом случае доступ будет прозрачен.

- *Указатели на функции*

С помощью таких указателей удобно ссылаться на функцию. Вызов функции осуществляется в соответствии с адресом, который содержится в указателе.

Чтобы четче определить поведение и особенности указателя и объекта на который он указывает можно комбинировать различные квалификаторы. К наиболее часто употребимым квалификаторам относятся **const**, **volatile**, **persistent**. Применяя комбинации этих квалификаторов необходимо следить за соответствием (отсутствием противоречий) квалификаторов, оказывающих воздействие на свойства собственно указателя и объекта на котрый он указывает. Правила, которые помогут сделать все правильно, просты: если квалификатор находится слева от “*” в описании указателя, то он воздействует на объект, адресуемый указателем. Если квалификатор находится справа, то он воздействует на собственно указатель. Проиллюстрируем это примерами:

```
volatile char * nptr;
```

объявляет указатель на **volatile** символ. Другими словами в этом примере квалификатор воздействует на объект, адресуемый указателем **nptr**.

```
char * volatile ptr;
```

так как квалификатор располагается справа от “*”, то он будет воздействовать на собственно указатель **ptr**, а не на объект. И заключительный пример по этому поводу:

```
volatile char * volatile nnptr;
```

теперь будет описан **volatile** указатель на **volatile** переменную.

Рассмотрим некоторые аспекты применения константных указателей. Они применяются для косвенного обращения к переменным описанным как **const**. В общем их поведение (константных указателей) не отличается от поведения обычных указателей, но компилятор препятствует выполнению операций записи с использованием этих указателей. Вот несколько примеров:

```
const char * cptr;
```

при этом выражение:

```
char ch = *cptr;
```

абсолютно легально, в то время как выражение:

```
*cptr =  ch;
```

недопустимо и вызовет ошибку.

5 Особенности реализации

5.1 Поддержка прерываний

Компилятор PICC полностью поддерживает прерывания контроллера. Таким образом вам не придется писать ни строчки ассемблерного кода при написании полноценных обработчиков прерываний. Для описания функции, которая и будет являться обработчиком, необходимо воспользоваться квалификатором **interrupt**. Эта функция будет напрямую вызвана в случае возникновения прерывания. Но компилятор оформит эту функцию специальным образом. Прежде всего будут сохранены и восстановлены все регистры, которые были задействованы в обработчике и, кроме того, для выхода из функции бкдет использоваться оператор **retfie**.

Функция обработчик прерываний должна возвращать значение типа **void** и не должна иметь аргументов. Ее нельзя вызывать непосредственно из программы, но она может обращаться к другим функциям, с чем надо проявлять особую осторожность.

Опуская поддержку прерываний в контроллерах базового класс, сразу переходим к контроллерам среднего класса. Вот пример простейшего обработчика прерываний:

```
long tick_count;
void interrupt tc_int(void)
{
     ++tick_count;
}
```

Поскольку в контроллерах этого класса реализован только один вектор прерываний, то и функцию обработчик можно описать только одну, что звучит вполне логично. В самой функции вы можите анализировать какое именно прерывание имело место и выполнять соответствующие действия. Код реализующий эту функцию будет автоматически расположен в положенном месте. Должен обратить ваше внимание, насколько удобный и простой код получается в этом случае. Конечно при профессиональном подходе и ассемблерный код можно оформить вполне читабельно, но такой прозрачности, пожалуй, добиться будет трудно.

Следует отдельно остановиться на сохранении контекста при выполнении обработчика прерывания.

Все знают, что при возникновении прерывания контроллер автоматически сохраняет только регистр PC, чего явно недостаточно для работы нормальной программы. Поэтому программист должен сам позаботиться о сохранении всех важных регистров и объектов. Поскольку система прерываний контроллеров среднего класс не поддерживает приоритеты, то особых проблем с сохранением контекста не возникает. Более того, компилятор PICC сам отслеживает какие переменные и объекты были использованы в обработчике прерываний и сохраняет их. Это очень удобно при написании тривиальных обработчиков прерываний. Когда требуется тонкое управление процедурой сохранения-восстановления контекста, то можно, к примеру, воспользоваться встроенным ассемблером, так как фрагменты ассемблерного кода, которые компилятор встречает в обработчике прерываний, не сканируются на предмет выявления переменных, нуждающихся в сохранении.

Думаю на этом можно закончить рассмотрение вопроса сохранения контекста при обработке прерываний, хотя здесь много тонких вопросов, но они вызывают интерес только при необходимости вмешаться в процесс сохранения-восстановления переменных.

В заключении хочется рассказать, как собственно управлять прерываниями (разрешать и запрещать). С PICC это делается очень просто:

```
ADIE = 1;   // Разрешить прерывание от АЦП
PEIE = 1;   // Разрешить все прерывания от периферийных устройств
ei();       // Разрешить все прерывания
di();       // Запретить все прерывания
```

5.2 Совмещение кода C и Assembler

Конечно, вы можите включать фрагменты ассемблерного кода непосредственно в свою программу на С. Для этого воспользоваться следующими механизмами: оформить ассемблерный код как внешнюю функцию или включить фрагмент в программу.

В первом случае, определенная функция может быть написана полностью на ассемблере как внешний файл с расширением **.as**, скомпилирована с помощью **ASPIC** и включена в двоичный образ с помощью компоновщика. Эта технология позволяет передавать аргументы и принимать результат между С и ассемблерной программами. Для этого, прежде всего, необходимо включить адекватное описание внешней функции в С-программе. Допустим, нам необходима функция сдвига в лево через флаг переноса, и мы решили реализовать ее на ассемблере:

extern char rotate_left(char);

В примере, приведенном выше, мы описали внешнюю функцию под названием **rotate_left ( )**, которая получает в качестве аргумента символ (**char**) и, соответственно, возвращает символ, только сдвинутый влево в соответствии с условим задачи. Текст этой архи-сложной функции поставляется в виде отдельного файла (.**as**), который, к тому же, и компилируется отдельно с помощью **ASPIC**. Вот вам возможная реализация этой функции:

```
processor 16C84

psect text0,class=CODE,local,delta=2
global      _rotate_left
signat      _rotate_left,4201
_rotate_left
; Аргумент передается через регистр W
      movwf       ?a_rotate_left

; Результат сдвига помещаем опять в W
      rlf         ?a_rotate_left,w

; Результат должен находиться в W
      return

FNSIZE      _rotate_left,1,0
global      ?a_rotate_left
end
```

Название ассемблерной функции должно совпадать с именем, упомянутым в С-программе, но с добавленным лидирующим знаком подчеркивания. Директива **global** является аналогом **extern**, а signat – необходима для проверки соответствия на этапе сборки. Подробнее этот вопрос будет рассмотрен далее.

ВНИМАНИЕ! Чтобы все работало правильно ассемблерная функция должна брать аргументы из правильного места и правильно оформлять результат. Механизм выделения памяти под локальные переменные (через **fnsize**), аргументы и результат детально обсуждается в руководстве и его таки надо хорошо понимать, прежде чем лезть со своим ассемблером в С-программу!!!

5.3 Передача аргументов в функцию

Метод передачи функции аргументов определяется их размерами и количеством.

Если передается только один аргумент типа **char**, то используется регистр W.

Если передается один аргумент, но большего размера, то он помещается в “область аргументов” вызываемой функции. Если имеются еще аргументы, то они отправляются туда же.

Если передается несколько аргументов, но первый из них однобайтный, то он помещается в авто-переменную, а все остальные аргументы – в область аргументов вызываемой функции.

В случае передачи списка аргументов, создает временный список аргументов и передает указатель на него.

Для примера рассмотрим простой вызов функции:

```
void test(int a, int b, int c)
{
     ...
}
```

Очевидно, что эта функция получит все аргументы в свою область аргументов. В случае вызова *test(0x65af, 0x7288, 0x080c)* будет получен следующий код:

```
     movlw 0AFh
     movwf (((?_test))&7fh)
     movlw 065h
     movwf (((?_test+1))&7fh)
     movlw 088h
     movwf ((0+((?_test)+02h))&7fh)
     movlw 072h
     movwf ((1+((?_test)+02h))&7fh)
     movlw 0Ch
     movwf ((0+((?_test)+04h))&7fh)
     movlw 08h
     movwf ((1+((?_test)+04h))&7fh)
     lcall (_test)
```

Начало области аргументов определяется меткой состоящей из знака вопроса, знака подчеркивания и названия функции (**?_test**). Эта величина принимается за базу, к которой прибавляется смещение для доступа ко всем аргументам. Таким образом первый аргумент будет доступен по адресам **?_test** и **?_test+1**.

Бывает полезно создавать пустые функции с заданными аргументами, компилировать программу с включенным флагом **-S** для контроля генерируемого кода.

5.4 Возвращение результата из функции

Передача результата из функции осуществляется следующим образом:

Значения размером 8-бит

В микроконтроллерах среднего класса, байтовые значения возвращаются через рабочий регистр W. Пример:

```
char return_8(void)
{
     return 0;
}
```

В результате компиляции будет получен следующий код:

```
     movlw 0
     return
```

Значения размером 16-бит

Значения размером 16 и 32 бита возвращаются через память. Пример:

```
int return_16(void)
{
     return 0x1234;
}
```

В результате компиляции будет получен следующий код:

```
    movlw low 01234h
    movwf btemp
    movlw high 01234h
    movwf btemp+1
    return
```

Значения – структура

Если функция возвращает структуру или объединение (struct, union) размером менее 4 байт, то результат упаковывается как 16-и или 32-х битное значение (см. ранее). Если размер результата превышает 4 байта, то он копируется в специально отведенную область памяти:

```
struct fred
{
    int ace[4];
}

struct fred return_struct(void)
{
    struct fred wow;
    return wow;
}
```

В результате компиляции будет получен следующий код:

```
    movlw ?a_return_struct+0
    movwf 4
    movlw structret
    movwf btemp
    movlw 8
    GLOBAL structcopy
    lcall structcopy
    return
```

5.5 Вызов функции

Как можно использовать С для процессоров начального уровня лично я не представляю, поэтому и освещать это извращение не буду. В контроллерах среднего и крутого уровня наличествует стек возвратов нормального размера (или около того), что уже делает допустимым полноценное использование процедурных языков программирования, но никак не объектно-ориентированных!

Локальные переменные

С поддерживает два класса локальных переменных: автоматические переменные, память под которые выделяется динамически из нулевого банка, и статические переменные, которые всегда располагаются в фиксированных адресах памяти.

Автоматические переменные

По умолчанию все переменные, описанные в теле функции относятся к этому классу. Исключение составляют те переменные в описании которых явно присутствует ключевое слово **static**. К автоматическим переменным не могут применяться квалификаторы, так как отсутствует контроль над выделением памяти. Исключение составляют только такие квалификаторы как **const** и **volatile**.

Все автоматические переменные располагаются в нулевом банке памяти.

Каждая функция имеет собственную область памяти в которой размещаются автоматические переменные. Эти области могут пересекаться для функций, которые, по мнению компоновщика, никогда не вызываются одновременно.

Статические переменные

Статические переменные всегда располагаются в фиксированных адресах памяти и

никогда не пересекаются с другими областями. При этом, они могут быть непосредственно доступны только из той функции, в которой описаны. Хотя к ним можно получить косвенный доступ из любого места программы, воспользовавшись указателями. Для статических переменных гарантируется сохранение их значения между вызовами функции. Исключение составляет случай косвенного изменения содержимого статической переменной с помощью указателя.

Статические переменные, которые инициализируются в описании, реально инициализируются только один раз, при запуске программы. В этом смысле они отличаются от автоматических переменных, которые инициализируются при каждом вызове функции.

5.6 Управление запуском программы

Далее идет довольно сложный фрагмент в котором объясняются принципы запуска программы при подаче питания. При запуске должны выполняться некоторые служебные процедуры связанные с инициализацией переменных, выделением и распределением памяти, настройкой системы.

Стартовым адресом программы является младший адрес (0x0). В этой точке располагается вызов служебной функции, которая собственно и занимается инициализацией. После общей инициализации вызывается функция **_main**(). Имеется в виду имя функции с точки зрения ассемблерного кода. В тексте С-программы эта функция описывается как **main**().

Стартовый код находится в стандартном модуле в каталоге **lib** и называется **picrt*XXX*.obj**. Исходный код модуля находится в каталоге **sources** и называется **picrt66x.as**. Кроме этого, могут потребоваться процедуры копирования данных или очистки памяти. Исходные тексты этих дополнительных функций вы также можите найти в каталоге **sources**, например **clrbank0.as** или **cpybank1.as**.

Некоторые устройства требуют специальной инициализации при подаче питания. Для реализации этих специальных функций нет необходимости изменять стандартные модули. Компилятор PICC предусматривает возможность выполнения при запуске программы действий необходимых в подобной ситуации. Программист должен создать специальный ассемблерный модуль, который будет выполняться непосредственно после рестарта или подачи питания. Обычно этот модуль внедряют непосредственно файл с С-программой. Пустая процедура запуска (заглушка) находится в файл **powerup.as**. Этот файл может быть скопирован в ваш проект, модифицирован и использоваться в дальнейшем в качестве исходного кода вашей процедуры запуска. В этом случае она заменит собой процедуру принятую по умолчанию.

При написании собственной процедуры запуска необходимо иметь в виду следующие обстоятельства: минимально использовать память, а лучше вообще не использовать; перед использованием любых системных ресурсов, они должны быть проверены и подключены.

Вот пример стандартной процедуры запуска:

```
    GLOBAL powerup,start
    PSECT powerup,class=CODE,delta=2
powerup
    ljmp start
```

По сути своей процедура запуска должна быть маленького размера. Это требование обусловлено еще и физическим расположением кода. Дело в том, что процедура запуска размещается в непосредственной близости вектора прерываний. Увеличение размера процедуры может привести к конфликту с обработчиком прерываний.

Если, все таки, возникает необходимость написание большой процедуры запуска, то избежать конфликта поможет следующая техника:

```
GLOBAL powerup,start,big_powerup
PSECT powerup,class=CODE,delta=2
powerup
    ljmp big_powerup
```

```
PSECT big_powerup,class=CODE,delta=2
big_powerup
      ...
      powerup code
      ...
      ljmp start
```

5.7 Директивы компилятора

Любой солидный компилятор содержит, так называемый, препроцессор, который делает возможным использовать такие необходимые технологии как макросы, условная компиляция и проч. Такой препроцессор имеется и в PICC. Давайте рассмотрим, какие команды (директивы) могут выполняться этим препроцессором (см. таблицу 6).

Таблица 6 Директивы препроцессора

Директива	***Описание***	***Пример***
#	Пустая директива – ничего не делает	#
#assert	Генерирует ошибку если результат не истина	#assert SIZE > 10
#asm	Позволяет внедрять ассемблерный код в программу на С	#asm movlw 10h #endasm
#define	Описание макроса	#define SIZE 5 #define FLAG #define add(a,b) ((a)+(b))
#elif	Сокращение для #else#if	see #ifdef
#else	Включает строки исходного текста по условию	see #if
#endasm	Замыкает внедренный ассемблерный код	see #asm
#endif	Замыкает строки исходного текста, включаемые по условию	see #if
#error	Генерирует сообщение об ошибке	#error Size too big

#if	Включает строки исходного текста если условие истинно	#if SIZE < 10
		c = process(10)
		#else
		skip();
		#endif

#ifdef	Включает строки исходного текста если если символ определен	#ifdef FLAG
		do_loop();
		#elif SIZE == 5
		skip_loop();
		#endif

#ifndef	Включает строки исходного текста если если символ не определен	#ifndef FLAG
		jump();
		#endif

#include	Подключает внешний текстовый файл к исходному тексту	#include <stdio.h> #include "project.h"
#line	Указывает номер строки и имя файла для листинга	#line 3 final
#**nn**	(где **nn** – номер) сокращение для #line **nn**	#20
#pragma	Специальные опции компилятора	См. далее.
#undef	Удаляет определенный ранее символ	#undef FLAG
#warning	Выводит предупреждение	#warning Possible conflict

Особый интерес представляет директива **#pragma**, которая позволяет указывать специальные опции компилятора.

Эта директива имеет следующий формат:

```
#pragma keyword options
```

где **keyword** – ключевое слово из таблицы 7.

Таблица 7 Варианты использования директивы #pragma

Директива	***Описание***	***Пример***
interupt_level	Позволяет функции обработки прерываний быть вызванной из основной программы.	#pragma interupt_level 2
jis	Включает поддержку символов jis в обработчике строк.	#pragma jis
nojis	Запрещает поддержку символов jis в обработчике строк.	#pragma nojis
printf_check	Включает проверку формата строк в стиле printf	#pragma printf_check(printf)
psect	Переименование секцию, определенную компилятором.	#pragma psect text=mytext
regsused	Явно указывает регистры которые используются в обработчике прерываний.	#pragma regsused w

Рассмотрим подробнее каждую из этих директив.

#pragma jis и nojis

Если в вашей программе встречаются литералы, содержащие двухбайтные символы в кодировке JIS для Японии и проч., то данная директива управляет обработкой этих символов. (Как для меня, так не актуально)

#pragma prinf_check

Некоторые библиотечные функции принимают в качестве аргумента строку формата с последующим списком аргументов переменной длины. Количество аргументов может быть разным в зависимости от строки формата наподобие функции **ptintf**. Несмотря на то, что строка формата интерпретируется во время выполнения программы, ее корректность может быть проверена на этапе компиляции. Эта директива как раз и включает такую проверку. Она по умолчанию включена в заголовочный файл **<stdio.h>**. Вы можите ее также включить и для ваших собственных функций, которые в качестве аргумента принимают список переменной длины. Учтите что уровень предупреждений компилятора должен быть установлен как минимум в -1, чтобы директива возымела действие.

#pragma psect

Обычно компилятор разбивает объектный код на фрагменты, которые размещены в именованных секциях, описанных в соответствующем разделе документации (я его не переводил – слишком глубоко...). Это поведение удовлетворяет потребностям большинства обычных приложений. Но иногда возникает необходимость перенаправить фрагмент в другую секцию. Когда возникнет, тогда и переведу этот фрагмент :-)

#pragma regsused

Компилятор автоматически сохраняет контекст при возникновении прерывания. Но он определяет все те переменные и объекты, которые явно присутствуют в функции обработки прерываний. Чтобы ограничить количество сохраняемых переменных и объектов можно воспользоваться этой директивой.

Директива определяет поведение только первой по тексту программы функции обработки прерываний. В крутых контроллерах может быть несколько таких обработчико. В этом случае необходимо использовать эту директиву для каждой из функций.

5.8 Стандартные функции ввода-вывода

Несколько функций ввода-вывода поставляются в составе стандартной библиотеки. Основным их предназначением является ввод и вывод форматированного текста в стандартные потоки. Подробнее эти функции рассмотрены в разделе руководства “Функции стандартной библиотеки”. Перечень поддерживаемых функций приведен в таблице.

Функция	***Описание***
prinf(char * s, ...)	Форматированная печать в **stdout**
sprintf(char * buf, char * s, ...)	Запись форматированного текста в **buf**

Перед тем, как станет возможным осуществлять запись-чтение текста необходимо описать служебные функции **putch()** и **getch()**, кроме них могут потребоваться **getche()** и **kbhit()**.

В каталоге с примерами есть программы, где вы можите посмотреть реальную реализацию этих и других функций.