CC7A

Си-компилятор
для микроконтроллеров серии 1886

Версия 1.0

Руководство пользователя

B Knudsen Data
Trondheim Norway

ЗАО «ПКК Миландр»
Зеленоград Россия

Перевод и адаптация: Шумилина Анна Сергеевна.
Оригинальный текст на английском языке: http://www.bknd.com/cc7a-10.pdf

Оглавление

1. ВВЕДЕНИЕ

Знакомьтесь с Си-компилятором CC7A для семейства микроконтроллеров серии 1886. Компилятор CC7A позволяет программировать, используя подмножество языка Си. Ассемблер более не требуется. Переход на язык Си обусловлен простотой. Язык ассемблера достаточно трудно читается, и поэтому легко ошибиться.

У языка Си существуют следующие преимущества перед ассемблером:

- Исходные стандартные коды
- Быстрая разработка программ
- Улучшает читабельность кода
- Упрощает документирование
- Упрощает поддержку
- Переносимость кода

Компилятор CC7A был разработан для создания компактного и оптимального кода. Оптимизатор автоматически сжимает код до минимума. Это позволяет написать код, который может быть скомпилирован напрямую в одиночные инструкции, но с Си синтаксисом. Это означает, что исходный Си код может быть оптимизирован путем исправления неэффективных выражений.

Преимущество разработки заключается не в том, чтобы обеспечить полную поддержку ANSI C, а для того, чтобы позволить лучше использовать ограниченные ресурсы кода и ОЗУ. Если компилятор генерирует менее оптимальный код, то можно использовать вставки кода на ассемблере.

Возможности CC7A:

- Локальные и общие переменные 8, 16, 24 и 32 битные, плюс битовые переменные.
- Эффективное повторное использование пространства локальных переменных.
- Генерирование сжатого и оптимального кода.
- Производит бинарные, ассемблерные, листинговые, COD, список ошибок, функциональные схемы и файлы переменных.
- Автоматическое обновление битов выбора банка.
- Улучшает и компактно поддерживает битовые операции, включая битовые функции.
- Переменные с плавающей и фиксированной запятой, размером до 32 бит
- Математическая библиотека включает такие функции, как sin(), log(), exp(), sqrt(), и т.д.
- Поддерживает стандартные Си константы и строки в памяти программ
- Модель указателей 8 и 16 битная, допускается смешение размеров в рамках одного приложения
- Указатели на ПЗУ и/или ОЗУ
- Размер отдельного указателя может быть выбран самим компилятором
- Расширенный уровень вызовов использует GOTO вместо CALL там, где это возможно
- Доступ к основным ассемблерным инструкциям через соответствующие выражения
- Ассемблерные вставки
- Интегрированная поддержка прерываний
- Конфигурирование микросхемы в исходном коде

Размеры переменных (в битах), поддерживаемые в различных редакциях компилятора

	STANDARD	EXTENDED
integer	8+16+24	8+16+24+32
fixed	8+16+24	8+16+24+32
float	24+32	16+24+32

1.1 Поддерживаемые устройства

Компилятор CC7A поддерживает все микроконтроллеры серии 1886 и позволяет:
- создавать приложения с объемом кода программы 65536 слов в 1- 8 страницах
- использовать до 16 банков ОЗУ.

1.2 Установка и системные требования

Компилятор CC7A запускается на IBM-PC совместимых машинах, использующих Windows (NT / 95 / 98 / me / 2000 / XP / 7 / 10). Для установки CC7A достаточно создать папку на жестком диске, где будут расположены файлы компиляторов. Большинство программ приложения находятся в "Program Files" на диске C. Создайте, например, папку CC7A там. Компилятор обычно поставляется как ZIP файл.

Теперь CC7A готов компилировать Си файлы. Заголовочные и исходные C файлы могут быть созданы и редактироваться в любом редакторе, например, в среде IDE1886.

Файлы компилятора CC7A могут быть удалены без каких-либо специальных процедур.

Поддержка длинных имен файлов

CC7A поддерживает длинные имена файлов. Также возможно использовать пробелы в именах файлов и включать имена папок.

Пример задания пути в параметрах вызова:

```
-I"C:\Program Files\cc7e"
-IC:\progra~1\cc7e
```

Пример задания пути в директиве #include

```
#include "C:\Program Files\cc7a\C file"
#include "C:\progra~1\cc7a\Cfile~1"
```

Альтернативой длинным именам файлов является усеченный короткий формат. Усеченная форма определяется файловой системой. Наилучшее предположение состоит из 6 первых цифр длинных имен плюс ~1. Последнее число может быть различным (~2), если первые 6 знаков пути нескольких папок равны друг другу.

Интерфейс пользователя

Компилятор CC7A выполняется из командной строки. Поэтому требует задание перечня командных опций для компилирования исходного файла Си и выводом необходимых файлов.

Для запуска CC7A в Windows необходимо выбрать Start->Run menu. Затем напишите полный путь включая cc7a и нажмите ввод. Список параметров командной строки компилятора будет выведен на экран. Стандартное использование CC7A - это его запуск из интегрированной среды, например IDE1886.

Для компилирование программы требует указать имя компилируемого файла и задать параметры в командной строке:

```
cc7a -a sample1.c <enter>
```

Поддержка интегрированных сред разработки CC7A

Интегрированная среда разработки IDE1886 имеет встроенные средства взаимодействия с компилятором CC7A.

Список файлов входящих в комплект поставки

CC7A.EXE	: исполняемый файл компилятора
INSTALL.TXT	: руководство по установке и поддержке setup
CHIP.TXT	: руководство по определению новых микросхем
CDATA.TXT	: информация по директивам #pragma cdata
CONFIG.TXT	: биты конфигурирования микросхем
MATH.TXT	: руководство по библиотеке математических функций
INT17XXX.H	: заголовочный файл для поддержки прерываний
HEXCODES.H	: 16-тиричные коды инструкций
OP.INC	: опции командной строки
RELOC.INC	: опции для создания объектных файлов для объединения с помощью MPLINK
DEMO.C	: пример программы для демонстрации синтаксиса
DEMO-VAR.C	: пример определение переменных
DEMO-MAT.C	: пример целочисленной математики
DEMO-FPM.C	: пример математики с плавающей запятой
DEMO-FXM.C	: пример математики с фиксированной запятой
DEMO-ROM.C	: пример использования const
DEMO-PTR.C	: пример использования указателей
DEMO-INS.C	: пример вызова одиночных инструкций
MATH16.H	: 8-16 битная целочисленная математическая библиотека
MATH24.H	: 8-24 битная целочисленная математическая библиотека
MATH32.H (1)	: 8-32 битная целочисленная математическая библиотека
MATH16X.H	: 16 битная математическая библиотека с фиксированной запятой
MATH24X.H	: 24 битная математическая библиотека с фиксированной запятой
MATH32X.H (1)	: 32 битная математическая библиотека с фиксированной запятой
MATH16F.H (1)	: 16 битная математическая библиотека с плавающей запятой
MATH24F.H	: 24 битная математическая библиотека с плавающей запятой
MATH32F.H	: 32 битная математическая библиотека с плавающей запятой
MATH24LB.H	: 24 битная математическая библиотека функций с плавающей запятой(log,sqrt,cos,..)

MATH32LB.H : 32 битная математическая библиотека функций с плавающей запятой(log,sqrt,cos,..)
VE2_CC.H : заголовочные файлы различных микросхем серии 1886
VE7_CC.H
NEWS.TXT : список различных дополнений
README.TXT

(1) не доступны в редакциях *DEMO* и *STANDARD*

Пример простой программы

```
/* глобальные переменные */
char a;
bit b1, b2;
/* присвоение имен выводам порта */
#pragma bit in @ PORTB.0
#pragma bit out @ PORTB.1
void sub( void)
{
      char i; /* локальные переменные */
      /* генератор 20 импульсов */
      for ( i = 0; i < 20; i++)
      {
            out = 1;
            nop();
            out = 0;
      }
}
void main( void)
{
// if (TO == 1 && PD == 1 /* power up */) {
// WARM_RESET:
// clearRAM(); // clear all RAM
// }
/* первоначально определяется уровень на выходных портах,
затем определяется направление input/output порта.
Определение может быть задано для каждого вывода отдельно */
      PORTA = 0b.0010; /* out = 1 */
      TRISA = 0b.1111.0001; /* xxxx 0001 */
      a = 9; /* присвоение значения для глобальной переменной */
      do
            {
            if (in == 0) /* остановится если на входе 0 */
            break;
            sub();
            } while ( -- a > 0); /* выполнение 9 итераций */
      // if (some condition)
      // goto WARM_RESET;

}
```

Задание типа микросхемы

CC7A имеет три способа для выбора типа микросхемы для реализации приложения:

1. Через командную строку
   -p1886BE5

2. Через директиву #pragma в исходном коде. Задание через командную строку переопределять выбор через #pragma.
   #pragma chip 1886BE5

3. Через использование директивы #include с соответствующим заголовочным файлом. Это не рекомендуется делать если тип микросхемы задается в командной строке.
   #include “VE5_CC.h”

Примечание 1.
Когда используется директива #pragma или заголовочный файл, помните что они должны быть заданы в начале Си программы, что бы были скомпилированы первыми. Так же выбор микросхемы может быть определен директивами *#define* и *#if* поверх директив *#include/#pragma*
Примечание 2.
CC7A автоматически подключит соответствующий заголовочный файл при использовании опции –p<device> или #pragma.
Примечание 3.
Если заголовочный файл не лежит в папке с проектом, то необходимо указать полный путь к нему, либо указать путь к папке с заголовочными файлами в опции командной строки (-I<path>).
Примечание 4.
Новые заголовочные файлы могут быть созданы как описано в файле chip.txt

Что будет далее…

Важно познакомится с семейством микроконтроллеров 1886 и соответствующим инструментарием. Наиболее простой способ для начала работы это прочесть доступные документы по данной тематике и изучить примеры. Затем создать простой свой проект.

- изучить простые примеры программ
- скомпилировать отдельные фрагменты программ и изучить отчеты созданные компилятором
- изучить ассемблерный файл созданный компилятором

Типовые шаги при разработке программ:

- определите систему и опишите требования,
- предложите решения, которые удовлетворяют этим требованиям
- напишите детальный код на языке Си
- скомпилируйте программу с помощью компилятора CC7A
- отладьте эту программу на микросхеме или эмуляторе

Написание программ для микроконтроллеров 1886 требует осторожного планирования. Память программ и ОЗУ ограничены и вы должны задуматься сможет ли это приложение разместится в выбранной микросхеме.

2. ПЕРЕМЕННЫЕ

Компилятор печатает информацию на экран при компиляции. В основном это сообщения об ошибках и как много ОЗУ и памяти программ потребовалось для программы. Так же это информация записывается в файл *.occ.

Пример:

```
CC7A Version 1.0, Copyright (c) B Knudsen Data, Norway 2003-2009
--> EXTENDED edition, 8-32 bit int, 16-32 bit float, 32k code words
17\demo.c:
Chip = 17C42
RAM: 00h : -------- -------- -------- ========
RAM: 20h : ======== ======== =====... ........
RAM: 40h : ...6**** ******** ******** ********
RAM: 60h : ******** ******** ******** ********
RAM: 80h : ******** ******** ******** ********
RAM: A0h : ******** ******** ******** ********
RAM: C0h : ******** ******** ******** ********
RAM: E0h : ******** ******** ******** ********
Bank 0:188 bytes free
RAM usage: 44 bytes (29 local), 188 bytes free
Optimizing - removed 9 code words (-2 %)
File 'demo.var'
File 'demo.asm'
File 'demo.lst'
File 'demo.occ'
Origin statement skips 7 words in codepage 0
File 'demo.hex'
Total of 207 code words (10 %)
```

2.1. Распределение памяти

Приоритет распределения памяти под переменные

1. Переменные, которым принудительно заданы адреса
2. Локальные переменные, адреса задаются компилятором
3. Глобальные переменные, адреса задаются компилятором

Компилятор отображает информацию о распределении памяти ОЗУ. Полная карта распределения отображается для небанкированной области и банка 0, которая позволяет определить какой объем ОЗУ не задействован. Детальная информация о распределении памяти сохраняется в файле <src>.var, когда в командной строке указан ключ -V

Символы:

- * : свободная память
- \- : предварительно заданные переменные (регистры ядра и переферии)
- = : локальные переменные
- . : глобальные переменные
- 7 : 7 свободных бит

Таблицы или структуры не могут быть больше размера банка (224 байта). Переменные располагаются с начиная с начала банка.

2.2.Определение переменных

CC7A поддерживает целочисленные переменные и переменные с фиксированной и плавающей точкой. Размер переменных может быть 1, 8, 16, 24 и 32 бита. По определению размер типа int составляет 8 бит, long составляет 16 бит. Символьные переменные являются беззнаковыми и расположены в диапазоне от 0 до 255. В демоверсии компилятора не поддерживаются 32 битные целочисленные переменные.

Математические библиотеки должны быть подключены для математических операций (Раздел 6.5. Поддержка Библиотек)

CC7A использует LOW ORDER FIRST (little-endian младшие вперед) последовательность бит в переменных. Это так же означает, что более младший байт располагается по более младшему адресу. Все переменные расположенные в ОЗУ располагаются с младших адресов и вверх. Каждая ячейка памяти ОЗУ может содержать 8 бит. Диапазон адресов используемых для отображения регистров ядра и периферии не доступен для распределения переменных. При невозможности выделить достаточного места для расположения всех переменных выводится соответствующие сообщение об ошибке.

Все переменные по возможности располагаются в банке 0. За более подробной информацией обратитесь к разделу 2.3. Использование банков памяти ОЗУ.

Регистры ядра определены в компиляторе и не требуется их определение в заголовочном файле (WREG, IDF0, FSR0, PCL, PCLATH, ALUSTA и т.п.).

Целочисленные переменные

```
unsigned a8;            // 8 bit unsigned
char a8;                // 8 bit unsigned
unsigned long i16;      // 16 bit unsigned
char varX;
char counter, L_byte, H_byte;
bit ready;              // 0 or 1
bit flag, stop, semafor;
int i;                  // 8 bit signed
signed char sc;         // 8 bit signed
long i16;               // 16 bit signed
uns8 u8;                // 8 bit unsigned
uns16 u16;              // 16 bit unsigned
uns24 u24;              // 24 bit unsigned
uns32 u32;              // 32 bit unsigned
int8 s8;                // 8 bit signed
int16 s16;              // 16 bit signed
int24 s24;              // 24 bit signed
int32 s32;              // 32 bit signed
```

Битовая разрядность так же может быть задана следующим образом:

```
unsigned x : 24;        // 24 bit unsigned
int y : 16;             // 16 bit signed
```

Диапазон значений переменных:

```
TYPE        SIZE        MIN         MAX
```

```
----        ----        ---         ---
int8        1           -128        127
int16       2           -32768      32767
int24       3           -8388608    8388607
int32       4           -2147483648 2147483647
uns8        1           0           255
uns16       2           0           65535
uns24       3           0           16777215
uns32       4           0           4294967295
```

Переменные с плавающей запятой

Компилятор поддерживает 16,24 и 32-х битные переменные с плавающей запятой. 32-х битные переменные с плавающей запятой могут быть преобразованы в и из формата IEEE754 за три инструкции (макрос в math32f.h)

Поддерживаемые типы с плавающей запятой

```
float16             : 16 bit floating point
float, float24      : 24 bit floating point
double, float32     : 32 bit floating point

Format    Resolution          Range
16 bit    2.4 digits          +/- 3.4e38, +/- 1.1e-38
24 bit    4.8 digits          +/- 3.4e38, +/- 1.1e-38
32 bit    7.2 digits          +/- 3.4e38, +/- 1.1e-38
```

16-ти битные переменные с плавающей запятой предназначены для использования когда точность не так важна. Больше деталей о типе данных с плавающей запятой можно найти в math.txt. Информация о библиотеке с плавающей запятой можно найти в разделе 6.5. Поддержка библиотек.

Флаги исключений для типов с плавающей запятой

Флаги типов данных с плавающей запятой доступны в приложении. При запуске программы, эти флаги должны быть проинициализированы.

```
FpFlags = 0;      // reset all flags, disable rounding
FpRounding = 1;   // enable rounding
```

Так же, после того, как исключение произошло, и флаг был выставлен, он должен быть сброшен, для того что бы было возможно определить новое исключение. Исключения могут быть проигнорированы, если они не важны. Новые операции не влияют на старые исключения. Так же допустимо отложенная обработка исключений. Флаги исключений могут быть сброшены только программным путем.

```
char FpFlags;                       // contains the floating point flags
bit FpOverflow    @ FpFlags.1;      // fp overflow
bit FpUnderFlow   @ FpFlags.2;      // fp underflow
bit FpDiv0        @ FpFlags.3;      // fp divide by zero
bit FpDomainError @ FpFlags.5;      // domain error
bit FpRounding    @ FpFlags.6;      // fp rounding
// FpRounding=0: truncation
// FpRounding=1: unbiased rounding to nearest LSB
```

Представление в стандарте IEEE754

Используемый тип данных с плавающих запятой не соответствует стандарту IEEE754. Но отличия не существенны. Основной причиной отличия в способе представления заключен в более эффективном кодировании операций. Представление в стандарте IEEE необходим когда осуществляется обмен данными с плавающей запятой с другими устройствами. Макрос для преобразования типов доступен:

```
Math32h:
            // before sending a floating point value:
float32ToIEEE754(floatVar);
            // change to IEEE754 (3 instr.)
            // before using a floating point value received:
IEEE754ToFloat32(floatVar);
            // change from IEEE754 (3 instr.)

math24f.h:
float24ToIEEE754(floatVar);
            // change to IEEE754 (3 instr.)
IEEE754ToFloat24(floatVar);
            // change from IEEE754 (3 instr.)
```

Данные с фиксированной запятой

Данные с фиксированной запятой могут быть применены вместо данных с плавающей запятой, в основном для уменьшения размера программы. Математика с фиксированной запятой использует формат, где десятичная точка жестко установлена между различными байтами. Например, формат fixed8_8 использует один байт для целочисленной части и один байт для десятичной части. Операции с фиксированной запятой выполняются обычными целочисленными операциями, за исключением умножения и деления, которые выполняются специальными библиотечными функциями. Информацию по библиотеке с поддержкой фиксированной запятой можно найти в разделе 6.5.

```
fixed8_8 fx;
fx.low8     : Least significant byte, decimal part
fx.high8    : Most significant byte, integer part

MSB   LSB         1/256 = 0.00390625
07    01    : 7 + 0x01*0.00390625 = 7.0039625
07    80    : 7 + 0x80*0.00390625 = 7.5
07    FF    : 7 + 0xFF*0.00390625 = 7.99609375
00    00    : 0
FF    00    : -1
FF    FF    : -1 + 0xFF*0.00390625 = -0.0039625
7F    00    : +127
7F    FF    : +127 + 0xFF*0.00390625 = 127.99609375
80    00    : -128
```

Расшифровка типов:
Fixed<S><I>_<D>:
<S> : ‘U’ unsigned
<node> : signed
<I> : число бит целочисленной части
<D> : число бит десятичной части

Таким образом, fixed16_8 использует 16 бит для целочисленной части и 8 бит для десятичной, всего 24 бита. Разрешение значений для fixed16_8 будет 1/256=0.0039. Это эквивалентно 2.4 десятичным знакам после запятой.

Встроенные типы данных с фиксированной запятой.

```
Type:        #bytes       Range                     Resolution
fixed8_8     2 (1+1)      -128, +127.996            0.00390625
fixed8_16    3 (1+2)      -128, +127.99998          0.000015259
fixed8_24    4 (1+3)      -128, +127.99999994       0.000000059605
fixed16_8    3 (2+1)      -32768, +32767.996        0.00390625
fixed16_16   4 (2+2)      -32768, +32767.99998      0.000015259
fixed24_8    4 (3+1)      -8388608, +8388607.996    0.00390625

fixedU8_8    2 (1+1)      0, +255.996               0.00390625
fixedU8_16   3 (1+2)      0, +255.99998             0.000015259
fixedU8_24   4 (1+3)      0, +255.99999994          0.000000059605
fixedU16_8   3 (2+1)      0, +65535.996             0.00390625
fixedU16_16  4 (2+2)      0, +65535.99998           0.000015259
fixedU24_8   4 (3+1)      0, +16777215.996          0.00390625
```

(дополнительные типы, без целочисленной части, только десятичные)

```
fixed_8      1 (0+1)      -0.5, +0.496              0.00390625
fixed_16     2 (0+2)      -0.5, +0.49998            0.000015259
fixed_24     3 (0+3)      -0.5, +0.49999994         0.000000059605
fixed_32     4 (0+4)      -0.5, +0.4999999998       0.0000000002328
fixedU_8     1 (0+1)      0, +0.996                 0.00390625
fixedU_16    2 (0+2)      0, +0.99998               0.000015259
fixedU_24    3 (0+3)      0, +0.99999994            0.000000059605
fixedU_32    4 (0+4)      0, +0.9999999998          0.0000000002328
```

Точность:

1. Все типы оканчивающиеся на _8 имеют 2 корректных десятичных знаков после запятой.
2. Все типы оканчивающиеся на _16 имеют 4 корректных десятичных знаков после запятой.
3. Все типы оканчивающиеся на _24 имеют 7 корректных десятичных знаков после запятой.
4. Все типы оканчивающиеся на _32 имеют 9 корректных десятичных знаков после запятой.

Константы с фиксированной запятой

Формат 32-х битных чисел с плавающей запятой используется для записи и вычислений перед компиляцией.

```
fixed8_8 a = 10.24;
fixed16_8 a = 8 * 1.23;
fixed8_16 x = 2.3e-3;
fixed8_16 x = 23.45e1;
fixed8_16 x = 23.45e-2;
fixed8_16 x = 0.;
fixed8_16 x = -1.23;
```

Округление констант с фиксированной запятой:
Константа: `0.036`
Тип: `fixed16_8`
Ошибка округления:
`0.036*256=9.216.` Но целое число 9.

Таким образом, ошибка составляет `(9/256-0.036)/0.036 = -0.023.`
Компилятор выведет предупреждение о возникновении ошибки при приведении типа

Приведение типов

Фиксированная запятая является частным случаем чисел с плавающей запятой. Поэтому данный тип используется не так часто.

Взаимодействия различных типов с фиксированной запятой

Рекомендуется использовать один тип с фиксированной запятой для переменных в программе. Основная проблема при взаимодействии типов возникает при использовании различных типов в различных комбинациях, что требует значительной поддержки от библиотек. Однако, большинство операций с различными типами разрешены в компиляторе CC7A за счет перевода типа с фиксированной запятой в целочисленный код, но иногда это не возможно:

```
fixed8_16 a, b;
fixed_16 c;
a = b + c; // OK, код сгенерируется правильно
a = b * 10.22; // OK: функция поддерживается в библиотеке

a = b * c; // нет поддержки в библиотеке
// необходимо приведение типов, поддержка которых есть в библиотеке:
a = b * (fixed8_16)c;
```

Задание адресов переменных в памяти ОЗУ

Все переменные, включая структуры и массивы могут быть созданы с заданием адреса расположения. Это так же применимо для портов и для составных типов (по аналогии с union). Переменные могут содержать часть других переменных, таблиц или структур. Допустимы многоуровневые составные типы. Синтаксис приведен ниже:

```
<variable_definition> @ <address | (constant_expression)>;
<variable_definition> @ <variable_element>;
```

Пример:

```
char th @ 0x25;
//bit th1 @ 0x25.1; // предупреждение о составном типе
bit th1 @ th.1; // нет предупреждения
char tty;
bit b0;
char io @ tty;
bit bx0 @ b0;
bit bx2b @ tty.7;
//char tui @ b0; // превышение размерности
//long r @ tty; // превышение размерности
char tab[5];
long tr @ tab;
struct {
long tiM;
long uu;
} ham @ tab;
char aa @ ttb[2]; // char ttb[10];
bit ab @ aa.7; // следующий уровень составной переменной
bit bb @ ttb[1].1;
size2 char *cc @ da.a; // 'da' - структура
```

```
char dd[3] @ da.sloi[1].pi.ncup;
uns16 ee @ fx.mid16; // float32 fx;
TypeX ii @ tab; // TypeX структура определенная через typedef
```

Адрес расположения переменной может быть задан через выражения. Это облегчает сбор переменных.

```
char tty @ (50+1-1+2);
bit tt1 @ (50+1-1+2+1).3;
bit tt2 @ (50+1-1+2+1).BX1; // enum { .., BX1, .. };
```

Директивы так же могут быть применены для задания адресов (только для битов и байтов)

```
#pragma char port @ PORTC
#pragma char varX @ 0x23
#pragma bit IOpin @ PORTA.1
#pragma bit ready @ 0x20.2
```

Если компилятор обнаружит назначение двух переменных по одному адресу, это будет отображено как предупреждение при компиляции. Предупреждений можно избежать за счет переназначения имени одной переменной через имя другой переменной для которой задан адрес (*#pragma char var2 @ var1*).

Как альтернативу можно использовать директиву #define

```
#define PORTX PORTC
#define ready PA2
```

Применение shadowDef можно применять для задания адреса глобальных и локальных переменных и параметров функции без затрагивания нормального распределения переменных. Компилятор будет игнорировать эти переменные при распределении обычных переменных.

```
shadowDef char gx70 @ 0x70; // global or local variable
```

Приведенные выше определения местоположения переменной по адресу 0x70 позволит читать и изменять эту ячейку с помощью переменной 'gx70'. Параметрам функциям так же могут быть заданы адреса. Никакие другие переменные не будет назначен компилятором на эти места. Такое ручное распределение может быть полезно, когда необходимо повторное использование ОЗУ задать в вручную.

```
void writeByte(char addr @ 0x70, char value @ 0x71) { .. }
```

Этот синтаксис так же доступен для прототипов функций

Поддерживаемые модификаторы типов

static `char a; /* a global variable; known in the current module only, or having the same name scope as local variables when used in a local block */`

extern `char a; // global variable (in another module)`

auto `char a; // local variable`
`// 'auto' is normally not used`

register `char a; // ignored type modifier`

```
const char a; /* ‘const’ tells that compiler that the data is not
modified. This allows global data to be put in program memory. */

volatile char a; /* ignored type modifier. Note that CC8E use the
address to automatically decide that most of the special purpose
registers are volatile */

page0 void fx(void); // IGNORED by CC8E
// page0,page1,page2,page3

bank0 char a; // variable ‘a’ resides in RAM bank 0
// bank0,bank1,bank2,..,bank15
// shrBank,accessBank : access bank (unbanked)

size2 char *px; // pointer px is 16 bit wide
// size1,size2

shadowDef char gx70 @ 0x70; /* a variable can be assigned to a
location without affecting normal allocation */
```

Локальные переменные

Локальные переменные поддерживаются компилятором. Компилятор выполняет сжатие объема переменных безопасным путем за счет проверки сферы действия переменных и повторное использование места, если это возможно. Ограниченное пространство в оперативной памяти используется эффективно. Параметры функций располагаются вместе с локальными переменными. Переменные должны быть определены в одном банке, поскольку это позволяет наилучшим образом использовать их повторно. Также можно добавить внутренние блоки (часть кода выделенная скобками {}) только для того, чтобы сократить место действия переменных, как показано в следующем примере:

```
void main(void)
{
char i; /* no reuse is possible at the
outermost level of 'main' */
i = 9;
{ // an inner block is added
char a;
for (a = 0; a < 10; a++)
i += fx(PORTB,0);
}
sub(i);
{ // another inner block to enable better reuse
char b = s + 1;
int i1 = -1, i2 = 0;
// more code
}
}
```

Локальные переменные могут иметь одно и то же имя. Однако, компилятор добавляет расширение создавая уникальные имена этим переменным при формировании ассемблерного файла, листинга и COD файла. Если функция не вызывается (определена, но не используются), то все ее параметры и локальные переменные располагаются в одной не использованной области.

Локальные переменные, как правило, живут в одном банке и не пересекают его границ, но есть возможность определить большой стек, который может пересекать границы. Компилятор не передвигает переменные из небанковой области в банк 0.

Стек для локальных переменных, параметров и временных переменных, как правило, выделяемых отдельно в каждом банке. Банк, как правило, определяется так же, как глобальные переменные через # Pragma rambank или банковского типа модификаторов. Это позволяет разделить стек на несколько независимых стеков. Используя единый стек, как правило, рекомендуется, но иногда это не возможно когда размер стека слишком велик.

Использование увеличенного стека

Можно использовать один основной стек для локальных переменных. Основной стек не является дополнительным блоком, но сообщает компилятору, где основной стек будет находится (в каком банк). Основной стек может быть больше, чем один банк, и определяется через директиву Pragma:

```
#pragma mainStack 3 @ 0x110 // set lower main stack address
```

Использование директивы Pragma означает, что локальные переменные, параметры и временные переменные размером 3 байта и больше (в том числе таблицы и структуры) будут храниться в одном стеке начиная с адреса не меньше, чем адрес 0x110. Меньшие переменные и переменные с модификатором указанием банка будет храниться по умолчанию / другим правилам. Использование размера 0 означает, что это определение действует на все переменные, в том числе битовые.

Помните, что #pragma rambank игнорируется для переменных сохраненных в основном стеке. Адресный диапазон с 0x100 до 0x1FF эквивалентен модификатору типа для банка 1.

В ряде случаев это позволяет увеличить эффективность использования небанковой области или задания банков для локальных переменных с целью экономии места. Для этого возможно использование следующей pragma:

```
#pragma minorStack 1 @ 0x1F
```

В этом случае, локальные переменные, параметры и временные переменные размером до 1 байта будут располагаться в небанковой области с адресов 0x18 до 0x1F. Большие переменные и переменные с модификатором банка будут располагаться из стандартных правил. Использование размера 0 означает область действия только на битовые переменные. Эта pragma может быть использована с основным стеком. Размер типа определенного в minor стеке имеет приоритет над основным стеком.

Для эффективного использования ОЗУ желательно использовать один стек. Разделение на несколько стеков увеличивает общий объем используемой памяти ОЗУ, и должен избегаться по возможности.

Временные переменные

Операции такие как умножение, деление, взятие модуля и подобные для своего выполнения требуют дополнительных временных переменных. Однако, компилятору не

требуется постоянного выделенного пространства в ОЗУ для этих временных переменных.

Временные переменные располагаются аналогично локальным переменным, но в более узком диапазоне. Это означает что эта область ОЗУ может быть использована в других частях программы. Это более эффективная стратегия и не требует дополнительного места в приложении.

Массивы, структуры и объединения

Допустимы только одномерные массивы.

```
char t[10], i, index, x, temp;
uns16 tx[3];
tx[i] = 10000;
t[1] = t[i] * 20; // ok
t[i] = t[x] * 20; // not allowed
temp = t[x] * 20;
t[i] = temp;
```

Нормальные структуры языка С так же могут быть определены, даже с включением различным типов. Объединения (unions) так же допустимы.

```
struct hh {
long a;
char b;
} vx1;
union {
struct {
char a;
int16 i;
} pp;
char x[4];
uns32 l;
} uni;
// accessing structure elements
vx1.a = -10000;
uni.x[3] = vx1.b - 10;
```

Как эквивалент многомерных массивов допустимы структуры с элементами в виде одномерных массивов. Однако, при этом в качестве индекса может выступать только одна переменная.

```
struct {
char e[4];
char i;
} multi[5];
multi[x].e[3] = 4;
multi[2].e[i+1] += temp;
```

Битовые поля

Битовые поля в структурах допустимы их размер может быть 1,8,16,24 и 32 бита

```
struct bitfield {
      unsigned a : 1;
      bit c;
      unsigned d : 32;
```

```
      char aa;
} zz;
```

Компилятор допускает через синтаксис задания битовой размерности и для обычным переменных

```
      int x : 24; // a 24 bit signed variable
```

Typedef

Typedef применяется для задания новых типов данных, включая структуры и другие типы.

```
typedef struct hh HH;
HH var1;
typedef unsigned ux : 16; // equal to uns16
ux r, a, b;
```

2.3. Использование банков

Банки ОЗУ определены следующим образом

Регистры ядра : 0x000 – 0x00F
Регистры периферии банка 0: 0x010 – 0x017
Регистры периферии банка 1: 0x110 – 1x017
…
Регистры периферии банка 15: 0xF10 – 0xF17

Регистры общего назначения: 0x018 – 0x01F
Банка 0 ОЗУ: 0x020 – 0x0FF
Банка 1 ОЗУ: 0x120 – 0x1FF
…
Банка 15 ОЗУ: 0xF20 – 0xFFF

В зависимости от типа кристалла физически могут быть реализованы не все банки.

При использовании более одного банка расположение переменных можно задавать через активный rambank

```
/* variables proceeding the first rambank statement are placed in the
access bank. This is also valid for local variables and parameters */
#pragma rambank 1
char a,b,c; /* a,b and c are located in bank 1 */

/* parameters and local variables in functions placed here are also
located in bank 1 ! */
#pragma rambank 0
char d; /* located in bank 0 */
```

Компилятор автоматически найдет первое свободное место для создаваемой переменной в выделенном банке.

Примечание: Локальные переменные и параметры функций также должны быть расположены. Для этого может быть необходимо использовать # Pragma rambank между некоторыми функциями и даже внутри функции. Рекомендуемая стратегия заключается в том, чтобы разместить локальные переменные и параметры

функции доступа в один банк. Небанковое расположение выбирается:

Pragma rambank –

Модификаторы типа банка
Так же можно использовать модификаторы типа банка для выбора банка ОЗУ

```
bank0..bank15, shrBank/accessBank : can replace #pragma rambank
// shrBank and accessBank is the access bank
bank1 char tx[3]; // tx[] is located in bank 1
```

Модификаторы типа банка определяют банк ОЗУ для расположения переменных. Так же можно задавать банк для расположения глобальных переменных, параметров функций и локальные переменные.

Модификатор типа банка применим непосредственно к переменным, но не к данным. Это отличие важно для указателей.

Примечание 1: Модификатор типа банка имеет больший приоритет чем #pragma rambank.
Примечание 2: использование модификатора 'extern' позволяет определить переменную несколько раз. Однако первое определение с использованием rambank и последующие должны иметь один и тот же банк.
Примечание 3: Когда определяется прототип функции, при этом обычно не задается размещение параметров. Однако, когда добавляется модификатор типа данных в прототипе, это задает банк используемый для этих переменных.

Если переменные определены в небанкируемой области ОЗУ, эти переменные доступны во всех банках. Ручное использование задания банков требует особого планирования. Это позволит сократить размер кода за счет уменьшения задания битов выбора банков.

Некоторые рекомендации:

1. Все локальные переменные и параметры функций предпочтительно сгруппированы в одном банке.

2. Наиболее часто используемые переменные должны быть помещены в небанкируемую область ОЗУ.

3. Переменные используемые для вычислений желательно располагать в одном банке.

4. Переменные используемые в одной функции желательно располагать в одном банке.

Выбор банка ОЗУ

ОЗУ и регистры периферии могут располагаться в 16 банках. Специальные инструкции используются для задания нужного банка.

Биты выбора банка автоматически проверяются и обновляются компилятором, поэтому добавление задания банка напрямую в исходном коде может быть удалено компилятором.

Эта функция может быть отключена, но при этом корректное задание банков должно быть задано в исходном коде.

Компилятор использует глобальные оптимизацию для минимизации размера кода необходимого для обновления битов выбора банков. Удалить все ненужные обновления очень трудно, но это позволяет удалить некоторые повторяемые инструкции.

Примечание: Компилятор УДАЛЯЕТ попытки использовать инструкции задания банков в исходном коде. Это опция может быть отключена с использованием команды –b в командной строке или заданием директивы в исходном коде

Задание локального региона действия ограничений

Автоматическое обновление может быть выключено локально. Это осуществляется с помощью директив:

```
#pragma updateBank 0 /* OFF */
#pragma updateBank 1 /* ON */
```

Эти определения могут быть установлены в любом месте, но желательно что бы они обрамляли минимальный размер кода. Проверьте сгенерированный ассемблерный код для того, что бы убедится в его корректности.

Другие использования #pragma updateBank определяет алгоритм обновления банков по критериям выбора. Подробное описание в разделе *#pragma updateBank* в разделе 4.1. определения *pragma*.

Примечание:
Для безопасного кодирования желательно не использовать специфические конструкции выбора банков. Компилятор использует полный комплекс правил для выбора банков.

2.4. Указатели

Поддерживаются одноуровневые указатели.

```
char t[10], *p;

p = &t[1];
*p = 100;
p[2] ++;
```

Обычно компилятор использует 8-ми битные указатели, когда все использования этого указателя ограничены в одном банке. Банк автоматически определяется и используется. В некоторых случаях банк необходимо указать напрямую. В этом случае при компиляции будет выведено сообщение об ошибке задания значения указателя адресом из другого банка ОЗУ.

```
bank1 char t[10];
bank3 char *pi;
#pragma assume *pi in rambank 1
..
pi = &t[2];
```

Модели указателей

Использование 8-ми битных указателей позволяет уменьшить объем кода и ОЗУ. Компилятор задает один размер всех указателей, который определяется автоматически. Однако указатели на структуры и массивы должны определятся заранее, с помощью задания модели памяти, опций командной строки или модификатора размера типа. Отметим, что оператор sizeof(указатель) зафиксирует размер в соответствии с выбранной моделью. Использование sizeof(указатель) обычно не требуется и их следует избегать.

Заданный размер указателя по определению используется только тогда, когда размер указателя не выбран динамически. Приоритет при определении размера указателя:

1. Модификаторы типа размера указателя
2. Автоматический выбор размера указателя (одиночные указатели)
3. Размер указателя выбирается в соответствии с определенной моделью.

Опции командной строки:
-mc1 : Указатель 'const' размером в 1 байт
-mc2 : Указатель 'const' размером в 2 байта
-mr1 : Указатель в ОЗУ размером в 1 байт
-mr2 : Указатель в ОЗУ размером в 2 байта
-mm1 : Универсальный указатель размером в 1 байт (для всех типов)
-mm2 : Универсальный указатель размером в 2 байт (для всех типов)

Модификаторы типа размера указателей

size1: Указатель размером в 1 байт
size2: Указатель размером в 2 байт

```
bank1 size2 float *pf;
```

Поддерживаемые типы указателей:

А) 8-ми битный указатель в ОЗУ. Компилятор автоматически обновлять ????
Б) 16-ти битный указатель в ОЗУ. Требуется когда необходимо обращаться к данным расположенным в разных банках.
В) 8-ми битный указатель в памяти программ. Этот указатель позволяет получить доступ до 256 байт данных.
Г) 16-ти битный указатель в памяти программ. Позволяет получить доступ более чем к 256 битам.
Д) 16-ти битный указатель в памяти программ. Старший бит определяет тип памяти: ОЗУ или память программ.

2.5.Поддержка констант

Компилятор поддерживает не изменяемые данные-константы хранимые в памяти программ. Ключевое слово языка Си «const» сообщает компилятору, что эти данные не будут изменяться. Пример:

```
const char *ps = "Hello world!";
const float ftx[] = { 1.0, 33.34, 1.3e-10 };
..
t = *ps;
ps = "";
fx = ftx[i];
```

Обычно компилятор вставляет «const» данные в конец кода программы (старше адреса). С помощью директивы #pragma можно вставить данные между пользовательскими функциями или по заданному адресу:

```
#pragma insertConst
```

Реализация «const» данных поддерживает следующие особенности:

- все 8-ми и 16-ти битные указатели на «const» данные можно использовать в одной программе
- размер одиночного указателя на «const» данные может быть изменен автоматически
- «const» указатели могут использоваться для доступа к ОЗУ и памяти программ
- компилятор не хранит все константные данные в одной таблице, но уменьшает таблицы если это уменьшает размер кода
- дублируемые строки и другие данные автоматически объединяются для экономии места

Рекомендации:
Рекомендуется использовать маленькие таблицы данных и структуры. Это позволяет компилятору объединить эквивалентные данные и строить более оптимальные блоки константных данных.

Ограничения

1. Компилятор не инициализирует переменные в ОЗУ при запуске процессора
2. Данные более 16 бит в структурах с размером более 256 байт должны быть выровнены

Данные с размером более 16 бит

Компилятор позволяет получить доступ к 8-ми, 16-ти, 24-х и 32-х битным данным, включая данные с фиксированной и плавающей запятой. Когда используются массивы или структуры с размером более чем 256 байт, одиночные данные должны быть выровнены. Выравнивание обозначает, что не должно возникать остатка при делении размера структуры на размер элемента. Это только проблема при определении структур содержащих данные различных размеров.

```
const long tl[5] = { 10000, -10000, 0, 30000, -1 };
const uns24 th[] = { 1000000, 0xFFFFFF, 9000000 };
const int32 ti[] = { 1000000000, 0x7FFFFFFF,
-900000000 };
const fixed8_8 tf[] = { -1.1, 200.25, -100.25 };
const float tp[] = { -1.1, 200.25, 23e20 };
const double td[] = { -1.1, 200.25, 23e-30};
const float16 ts[] = { -1.1, 200.25, 23e-30};
..
l = tl[i]; // reading a long integer
d = td[x]; // reading a double float constant
```

Объединение данных

Компилятор автоматически объединяет одинаковые строки, части строк, и другие элементы. Использование небольших таблиц повышает шанс найти данные, которые могут быть объединены. Но объединены могут быть только данные расположенные в

одной памяти. Данные расположенные в ОЗУ и памяти программ не могут быть объединены. Пример:

1. Строка «world!» является частью строки «Hello world!». Это означает что для хранения этой строки не требуется дополнительного места. Компилятор автоматически вычислит адрес начала этой строки в таблице второй строки. При этом символ окончания строки «\0» так же будет объединен. Но например строка «world» не может быть объединена с этими строками, так как не заканчиваются одинаково.
2. Объединение доступно для всех данных. Данные сравниваются байт с байтом. Это позволяет объединить первые две таблицы с последней.

```
const char a1[] = { 10, 20, 30 };
const char a2[] = "ab";
const char a3[] = { 5, 10, 20, 30, 'a', 'b', 0 };
```

Пример
Таблицы указателей на структуры

```
const struct {
const char *s;
} tb[] = {
"Hello world",
"Monday",
"",
"world" // automatically merged with first string
};
p = tb[i].s; // const char *p; char i;
t = *p++; // char t;
t = p[x]; // char x;
```

Но «const struct» требует хранить указатель на массив в памяти программ. Использование «const char *tx[];» означает что указатели хранятся в памяти программ, но сами таблицы «tx[]» хранятся в ОЗУ.

Строки как параметры

```
myfunc(“Hello”); // void myfunc(const char *str);
myfunc(&tab[i]); // char tab[20]; // string in RAM
myfunc(ctab); // const char ctab[] = “A string”;
```

3. СИНТАКСИС

Операторы

Операторы языка Си разделяются точкой с запятой и определяются фигурными скобками.
```
{ <statement>; .. <statement>; }
```

Обычные операторы:
```
// if, while, for, do, switch, break, continue,
// return, goto, <assignment>, <function call>
      while (1) {
            k = 3;
            X:
            if (PORTA == 0) {
                  for (i = 0; i < 10; i++) {
                        pin_1 = 0;
                              do {
                                    a = sample();
                                    a = rr(a);
                                    s += a;
                              }
                              while (s < 200);
                        }
                  reg -= 1;
                  }
            if (PORTA == 4)
                  return 5;
            else if (count == 3)
                  goto X;
            if (PORTB.3)
            break;
      }
```

Оператор IF
```
if (<condition>)
      <statement>;
else if (<condition>)
      <statement>;
else
      <statement>;
```

else if и else могут не применяться

Оператор WHILE
```
while (<condition>)
      <statement>;
while (1) { .. } // бесконечный цикл
```

Оператор FOR
```
for (<initialization>; <condition>; <increment>)
<statement>;
```

Initialization: задание начального состояния или пусто
Condition: условие завершения или пусто
Increment: закон изменения или пусто

```
for (v = 0; v < 10; v++) { .. }
for (; v < 10; v++) { .. }
for (v = 0; ; v--) { .. }
for (i=0; i<5; a.b[x]+=2) { .. }
```

Оператор DO

```
do
<statement>;
while (<condition>);
```

Оператор SWITCH

Оператор switch поддерживает переменные до 32-х бит. Генерируемый код при этом более компактный и быстрый чем при реализации этой конструкции через if else.

```
switch (<variable>) {
      case <constant1>:
            <statement>; .. <statement>;
            break;
      case <constant2>:
            <statement>; .. <statement>;
            break;
      ..
      default:
            <statement>; .. <statement>;
      break;
}
```

<Variable>: все переменные от 8-ми до 32-х бит.
Break: опционально
Default: опционально

```
switch (token) {
      case 2:
            i += 2;
            break;
      case 9:
      case 1:
      default:
            if (PORTA == 0x22)
            break;
      case 'P':
            pin1 = 0; i -= 2;
            break;
}
```

Оператор BREAK

Оператор break используется внутри циклов (for, while, do) для завершения цикла. Так же может применяться в операторе switch.

```
while (1) {
      ..
      if (var == 5)
      break;
      ..
      }
```

Оператор CONTINUE

Оператор Continue используется внутри циклов для ускоренного перехода на следующий цикл итераций, при этом весь последующий код не будет выполняться.

```
for (i = 0; i < 10; i++) {
      ..
      if (i == 7)
```

```
            continue;
      ..
}
```

Оператор RETURN

```
return <expression>; /* exits the current function */

return; /* no return value */
return i+1; /* return value */
```

Оператор GOTO

goto <label>;

Обеспечивает переход на метку кода вверх или вниз.

```
goto XYZ;
..
XYZ:
..
```

Присвоение и условия

Основные присвоения приведены в примере:

```
var1 = x + y;
i = x - 100;
y ^= 'A'; // y = y ^ 'A';
W |= 0x10; // W = W | 0x10;
a = b = c + 1; // multiple assignment
// operations: + - & | ^ * / % << >>
flag = 1; // set bit variable
i++; /*or*/ ++i; /*or*/ i = i + 1;
i--; /*or*/ --i; /*or*/ i = i - 1;
```

Примеры специальных особенностей синтаксиса

```
#define mx !a
if (!mx) ..

W = W - 3; // ADDLW 256-3
b = fx() - 3;
// Post- and pre-incrementing of pointers
char *cp;
t = *--cp;
t |= *++cp;
*cp-- = t;
t = *cp++ + 10;

// pre-incrementing of variables
t = ++b | 3;
sum( --b, 10);
t = tab[ --b];
```

Условия

```
[ ++ | -- ] <variable> <cond-oper> <value>
[ && condition ]
[ || condition ]
cond-oper : == != > >= < <=
```

```
if (x == 7) ..
if (Carry == 1 && a1 < a2) ..
if (y > 44 || Carry || x != z) ..
if (--index > 0) ..
if (bx == 1 || ++i < max) ..
if (sub_1() != 0) ..
```

Битовые переменные

```
bit a, b, c, d;
char i, j, k;
bit bitfun(void) // bit return type (using Carry bit)
{
return 0; // Clear Carry, return
return 1; // Set Carry, return
nop();
return Carry; // return
return b; // Carry=b; return
return !i;
return b & PORTA.3;
}
..
b = bitfun2(bitfun(), 1);
if (bitfun()) ..
if (!bitfun()) ..
if (bitfun() == 0) ..
b = !charfun();
b = charfun() > 0;
b = !bitfun();
Carry = bitfun();
b &= bitfun();

if (bitfun() == b) ..
if (bitfun() == PORTA.1) ..
i += b; // conditional increment
i -= b; // conditional decrement
i = k+Carry;
i = k-Carry;
b = !b; // Toggle bit (or b=b==0;)
b = !c; // assign inverted bit
PORTA.0 = !Carry;
a &= PORTA.0;
PORTA.1 |= a;
PORTA.2 &= a;
// assign condition using 8 bit char variables
b = !i;
b = !W;
b = j == 0;
b = k != 0;
b = i > 0;
// assign bit conditions
b = c&d; //also &&, |, ||, +, ^ , ==, !=, <, >, >=, <=
// conditions using bit variables
if (b == c) .. // also !=, >, <, >=, <=
// initialized local bit variables
bit bx = cx == '+';
bit by = fx() != 0xFF;
```

Умножение, деление и модуль

```
multiplication : a16 = b16 * c16; // 16 * 16 bit
```

Основной алгоритм умножения реализован в компиляторе. Поддерживается все комбинации переменных различного размера. Включение математической библиотеки позволяет библиотечные вызовы встроить непосредственно в код. Операция умножения удаляется когда это возможно, например при умножении на 2, так как это эквивалентно более простой операции сдвигу влево.

```
division : a16 = b16 / c8; // 16 / 8 bit
modulo : a32 = b32 % c16; // 32 % 16 bit
```

Деление так же поддерживает все комбинации переменных различного размера. И так же по возможности удаляется, например, при делении на 2, так как это эквивалентно сдвигу вправо.

Приоритет операций

Наивысший:

```
( )
++ --
* / %
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
= += -= *= /= etc.
```

Наименьший

Действия над данными различных размеров

```
a32 = (uns32) b24 * c8;       // 24 * 8 bit, result 32 bit
a16 = a16 + b8;               // 16 + 8 bit, result 16 bit
```

Большинство комбинаций переменных допустимы, компилятор учитывает знак числа если это необходимо. Множественные операции допустимы при использовании 8-ми битных переменных.

```
      a8 = b8 + c8 + d8 + 10;
```

Константы

```
x = 34;                     /* decimal */
x = 0x22;                   /* hexadecimal */
x = 'A';                    /* ASCII */
x = 0b010101;               /* binary */
x = 0x1234 / 256;           /* 0x12 : MSB */
x = 0x1234 % 256;           /* 0x34 : LSB */
x = 33 % 4;                 /* 1 */
x = 0xF & 0xF3;             /* 3 */
x = 0x2 | 0x8;              /* 10 */
x = 0x2 ^ 0xF;              /* 0b1101 */
x = 0b10 << 2;              /* 8 */
x = r1 + (3 * 8 - 2);       /* 22 */
x = r1 + (3 + 99 + 67 - 2); /* 167 */
x = ((0xF & 0xF3) + 1) * 4; /* 16 */
```

Скобки необходимы в некоторых случаях.

Выражения с константами

Размер integer в компиляторе по определению 8 бит (другие Си компиляторы обычно считают integer 16-ти или 32-х битные в зависимости от возможности процессора). При потере знаковых бит при превышении допустимых значений будут обозначены как ошибки при компиляции.

```
char a;
a = (10 * 100) / 256;              // an error is printed
a = (10L * 100) / 256;             // no error
a = ((uns16) 10 * 100) / 256;      // no error
a = (uns16) (10 * 100) / 256;      // error again
a = (10 * 200) / 256;              // no error, 200 is a long int
```

Добавление символа L означает преобразование в тип long (16 бит).

В командной строке опции –cu задает 32 битные константы, таким образом, никакие значимые биты не теряются.

Допустимые значения для различных типов:

```
TYPE        SIZE                   MIN         MAX
----        ----                   ---         ---
int8 :      8 bit signed 1         -128        127
int16:      16 bit signed 2        -32768      32767
int24:      24 bit signed 3        -8388608    8388607
int32:      32 bit signed 4        -2147483648 2147483647
uns8 :      8 bit unsigned 1       0           255
uns16:      16 bit unsigned 2      0           65535
uns24:      24 bit unsigned 3      0           16777215
uns32:      32 bit unsigned 4      0           4294967295
```

Константы по определению создаются наименьшими знаковым типом integer. Добавление символа U означает, что константа без знаковая. Но константа 0x7FFFFFFF по определению без знаковая (не зависимо от наличия символа U)

Перечисляемые типы

Перечисляемые типы это способ поименовать целочисленные константы. Это так же можно реализовать с помощью директив #define. Нумерация начинается с 0, но это может быть изменено.

```
enum { A1, A2, A3, A4 };
typedef enum { alfa = 8, beta, zeta = -4, eps, } EN1;
EN1 nn;
enum con { Read_A, Read_B };
enum con mm;
mm = Read_A;
nn = eps;
```

Функции

Определение функций могут выглядеть так:

```
void subroutine2(char p) { /* C statements */}
bit function1(void) { }
long function2(char W) { }
```

```
void main(void) { }
```

Вызовы функций:

```
 subroutine1();
subroutine2(24);
bitX = function1();
x = function2(W);
y = fx1(fx3(x));
```

Компилятору необходимо знать определение функций перед ее вызовом, для разрешения правил преобразования типов. По этому может быть создан прототип функции. Прототип функции используется для реализации вызовов функции до ее непосредственного определения.

```
char function3(char);
void subroutine1(void);
```

Возвращаемые значения функций

Функции могут вернуть переменные размером до 4 байт. Возвращаемые значения могут быть присвоены какой либо переменной, либо проигнорированы. Обработка и использование возвращаемых значений автоматически обрабатывается компилятором.

Младщий знаковый байт всегда размещается в регистре WREG. Знаковые переменные и переменные более 8 бит так же используют временные переменные в стеке.

Функций может вернуть любой тип. WREG регистр используется для возврата 8-ми битных переменных, если это возможно. Флаг переноса может быть использован для возврата битовых переменных. Компилятор автоматически распределяет временные переменные для других возвращаемых типов. Функции, которые не возвращают значений должны иметь тип void.

Параметры при вызове функций

Нет ограничений на число параметров при вызове функций. Память для параметров выделяется аналогично памяти для локальных переменных. Битовые переменные так же могут выступать в качестве параметров. Помните, если используется регистр WREG, то он должен выступать в качестве последнего параметра

```
char func(char a, uns16 b, bit ob, char W);
```

Встроенные функции

Встроенные функции позволяют получить прямой доступ к определенным инструкциям

```
btsc(Carry);      // void btsc(char); - BTFSC f,b
btss(bit2);       // void btss(char); - BTFSS f,b

clrwdt(); // void clrwdt(void); - CLRWDT
clearRAM(); // void clearRAM(void); clears all RAM
i = decsz(i); // char decsz(char); - DECFSZ f,d
W = incsz(i); // char incsz(char); - INCFSZ f,d
nop(); // void nop(void); - NOP
nop2(); // void nop2(void); - branch (2 cycles)
retint(); // void retint(void); - RETFIE
W = rl(i); // char rl(char); - RLF i,d
```

```
i = rr(i); // char rr(char); - RRF i,d
sleep(); // void sleep(void); - SLEEP
skip(i); // void skip(char); - computed goto (single word)
k = swap(k); // char swap(char); - SWAPF k,d
W = addWFC(k);// char addWFC(char); - ADDWFC k,d
i = subFWB(k);// char subFWB(char); - SUBFWB k,d
W = rlnc(i); // char rlnc(char); - RLNCF i,d
i = rrnc(i); // char rrnc(char); - RRNCF i,d
i = decsnz(i);// char decsnz(char); - DCFSNZ f,d
W = incsnz(i);// char incsnz(char); - INFSNZ f,d
b = negate(b);// char negate(char); - NEGF f
W = decadj(W);// char decadj(char); - DAW
multiply(i); // void multiply(char); - MULWF f
multiply(50); // void multiply(char); - MULLW literal
skipIfEQ(a); // void skipIfEQ(char); - CPFSEQ f
skipIfLT(a); // void skipIfLT(char); - CPFSLT f
skipIfGT(a); // void skipIfGT(char); - CPFSGT f
skipIfZero(a);// void skipIfZero(char); - TSTFSZ f
a = readLUInc(void); // char readLUInc(void); - TABLRD 0,1,a
a = readLU(void); // char readLU(void); - TABLRD 0,0,a
a = readHUInc(void); // char readHUInc(void); - TABLRD 1,1,a
a = readHU(void); // char readHU(void); - TABLRD 1,0,a
a = readL(void); // char readL(void); - TLRD 0,a
a = readH(void); // char readH(void); - TLRD 1,a
writeLUInc(char); // void writeLUInc(char); - TABLWT 0,1,a
writeLU(char); // void writeLU(char); - TABLWT 0,0,a
writeHUInc(char); // void writeHUInc(char); - TABLWT 1,1,a
writeHU(char); // void writeHU(char); - TABLWT 1,0,a
writeL(char); // void writeL(char); - TLWT 0,a
writeH(char); // void writeH(char); - TLWT 1,a
```

Встроенные функции rotate (rl, rr) так же доступны для переменных больших размеров:

```
a16 = rl(a16); // 16 bit left rotation
a32 = rr(a32); // 32 bit right rotation
```

Встроенные функции nop2() реализуется через инструкцию GOTO на следующий адрес. Nop2 может быть применена для создания более компактного кода. Применение функций nop() и nop2() обусловлено создание дополнительных временных задержке в критических к временам частям программы.

Приведение типов

Константы и переменные в выражениях могут быть различных типов. Компилятор автоматически приведет все типы к общему по заданным правилам. Например:

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

Выражение содержит 2 различных операции. Первая – суммирование, вторая – присвоение. Преобразование типов сначала выполняется для суммирования a + b. Затем производится приведение типов для результата суммирования и переменной a.

Компилятор поддерживает различные типы данных (целочисленные, с фиксированной и плавающей запятой). Приведение типов выполняется с максимальным соответствием стандарту языка Си (который обычно использует 16-ти и 32-х битные integer).

Правила приведения типов:

1. Если один операнд double, остальные так же преобразуются в double

2. Если один операнд float, остальные так же преобразуются в float
3. Если один операнд 32 бита, остальные так же преобразуются в 32 бита
4. Если один операнд 24 бита, остальные так же преобразуются в 24 бита
5. Если один операнд long, остальные так же преобразуются в long
6. Если один операнд беззнаковый, остальные так же преобразуются в беззнаковые

Примечание:
- Знак сохраняется до тех пор, пока не будет преобразования в беззнаковое число
- Присвоение так же является операцией
- Константы всегда знаковые, если нет символа U
- Битовые типы преобразуются в unsigned chat
- Переменные с фиксированной точкой приводятся к плавающей точке.

Преобразования типов в Си очень сложно. Если возникают трудности при компиляции, компилятор отображает предупреждение с указанием места проблемы. Присвоение так же является операцией. В ниже следующем примере операции пронумерованы цифрами (1:), (2:) и (3:) и так далее.

```
uns16 a16;
uns8 b8, c8;
int8 i8, j8;
```

a16 = b8 * c8;
(1:) В данном случае обе переменные b8 и c8 являются 8-ми битными беззнаковыми. Таким образом результат операции умножения будет 8-ми битным числом.
(2:) Результат присваивается 16-ти битной беззнаковой перемененной a16. Преобразование 8-ми битного беззнакового результата к 16-ти битной беззнаковой означает очистку старших бит a16. Компилятор выведет предупреждение об ошибке поскольку произойдет потеря знаковых бит при преобразовании типов.

a16 = (uns16) (b8 * c8);
(1:) Принудительного преобразования типов отделено от самой операции умножения скобками. По этому результат умножения все равно будет 8-ми битным.
(2:) Принудительное преобразование (uns16) в данном случае не требуется, так как преобразование произойдет автоматически при выполнении операции присвоения. Компилятор выведет предупреждение об ошибке поскольку произойдет потеря знаковых бит при преобразовании типов.

a16 = (uns16) b8 * c8;
(1:) Преобразование типа в 16-ти битную беззнаковую переменную произойдет до выполнения операции умножения и таким образом будет получен правильный 16-ти битный результат.
(2:) Результат и переменная a16 теперь имеют одинаковый размер и преобразования типов при присвоении не требуется

a16 = (uns8) (b8 * c8);
(1:) Результат умножения будет 8-ми битным.
(2:) Принудительное преобразование (uns8) сообщает компилятору что результат и должен быть 8-ми битным. И по этому компилятор не будет выдавать предупреждения об потере знаковых бит.

a16 = b8 * 200;
(1:) Константа 200 является знаковой 16-ти битной константой (200U является 8-ми битной беззнаковой константой, а так это число более 127 8-ми битной знаковой константой). Аргумент b8 тем не менее автоматически преобразуется к 16 битам. Таким образом результат будет 16-ти битным беззнаковым.
(2:) Результат и переменная a16 теперь имеют один и тот же размер и преобразования типов не требуется.

a16 = (int16) i8 * j8;
(1:) Оба аргумента преобразуются к 16-ти битным знаковым типам. Результат так же будет 16-ти битным знаковым.
(2:) Результат преобразуется к беззнаковому перед присвоением. Но это не означает каких либо изменений в числе, таким образом число -1 превратится в 0xFFFF.

a16 = (uns16) (uns8)i8 * (uns8)j8;
(1:) Для получения 8*8 битного беззнакового умножения необходимо преобразовать оба аргумента преобразовать к беззнаковому типу перед расширением до 16-ти битного беззнакового типа. В противном случае будет выполнятся знаковое умножение, которое требует большего кода и циклов исполнения.
(2:) Результат и переменная a16 теперь имеют один тип и преобразования не требуется.

a16 = ((uns16) b8 * c8) / 3;
(1:) Преобразование одного из аргументов до 16-ти битного беззнакового типа до умножения даст 16-ти битный результат.
(2:) Деление в следующей использует 16-ти битный беззнаковый результат выполнения. Константа 3 является 8-ми битной знаковой, таким образом она будет автоматически преобразована сначала в 16-ти битную знаковую, а затем и беззнаковую. Результат деления будет 16-ти битным беззнаковым.
(3:) Результат деления и переменная a16 имеют один тип и дополнительного преобразования не требуется.

Доступ к частям переменных

Каждый бит переменой может быть использован отдельно.

```
uns32 a;
a.7 = 1; // set bit 7 of variable a to 1
if (a.31 == 0) // test bit 31 of variable a
t[i].4 = 0; // bit 4 of the i'th element
Bit 0: least significant bit
Bit 7: most significant bit of a 8 bit variable
Bit 15: most significant bit of a 16 bit variable
Bit 23: most significant bit of a 24 bit variable
Bit 31: most significant bit of a 32 bit variable
```

Так же могут использоваться и части переменной:

```
uns16 a;
uns32 b;
a.low8 = 100; // set the least significant 8 bits
a = b.high16; // load the most significant 16 bits
low8 : least significant byte
high8 : most significant byte
mid8 : second byte
midL8 : second byte
midH8 : third byte
```

```
low16 : least significant 16 bit
mid16 : middle 16 bit
high16: most significant 16 bit
low24 : least significant 24 bit
high24: most significant 24 bit
```

Следующая таблица показывает какие биты используются в переменных при различных размерах исходных переменных.

	1	2	3	4
low8	0-7	* 0-7	* 0-7	* 0-7
high8	0-7	* 8-15	* 16-23	* 24-31
mid8	0-7	8-15	* 8-15	8-15
midL8	0-7	8-15	8-15	* 8-15
midH8	0-7	8-15	16-23	* 16-23
low16	0-7	0-15	* 0-15	* 0-15
mid16	0-7	0-15	8-23	* 8-23
high16	0-7	0-15	* 8-23	* 16-31
low24	0-7	0-15	0-23	* 0-23
high24	0-7	0-15	0-23	* 8-31

Расширение языка Си

Компилятор имеет ряд расширений синтаксиса стандарта языка Си

1. Тип переменных bit.
2. Тип функций interrupt
3. C++ стиль комментариев: // комментарий
4. Переменные могут быть объявлены между выражениями как в C++.
5. Бинарные константы: 0b0101010 или bin(0101010). Отдельные биты могут быть отделены «.», например 0b010101 = 0b01.01.0.1

6. Директивы препроцессора могут быть размещены в макросах. Такая директива препроцессора не распространяет свое действие на несколько строк. Встроенные директивы выполняются только при выполнении директив более высокого уровня.

```
#define MAX \
      { \
      a = 0; \
      #if AAA == 0 && BBB == 0 \
      b = 0; \
      #endif \
      }
```

7. Некоторые модификаторы типов не являются стандартными для Си. Например bank0, bank15, size1 и так далее.

Предопределенные символы

Основные регистры микроконтроллера определены в компиляторе

```
INDF0, FSR0, PCL, PCLATH, ALUSTA, T0STA, CPUSTA, INTSTA,
INDF1, FSR1, W, WREG, TMR0L, TMR0H, TBLPTR, TBLPTRL, TBLPTRH,
BSR, BSRL, BSRH, PRODH, PRODL, Carry, DC, Zero_, Overflow
```

Следующие имена встроенный функций напрямую транслируется в ассемблерные инструкции

```
btsc, btss, clearRAM, clrwdt, decsz, incsz, nop, nop2, retint, rl,
rr, sleep, skip, swap, decsnz, incsnz, addWFC, subWFB, rlnc, rrnc,
negate, decadj, multiply, skipIfEQ, skipIfLT, skipIfGT, skipIfZero
```

Расширения стандартного набора ключевых слов языка Си

```
bank0, .. bank15, bit, fixed8_8, .. fixed24_8, float16, float24,
float32, int8, int16, int24, int32, interrupt, page0, .. page7,
shrBank, size1, size2, uns8, uns16, uns24, uns32
```

Стандартные ключевые слова языка Си:

```
auto, break, case, char, const, continue, default, double, enum,
extern, do, else, float, for, goto, if, inline, int, long, return,
short, signed, sizeof, static, struct, switch, typedef, union,
unsigned, void, while,
define, elif, ifdef, ifndef, include, endif, error, pragma, undef
```

Остальные стандартные ключевые слова обнаруживаются и компилируются. Некоторые из них игнорируются (register), другие приводят к отображению предупреждений (volatile, line)

Оператор sizeof

Оператор sizeof() возвращает размер в байтах аргумента. Аргументом может быть имя типа, имя переменной, указателя, структуры, массива, символа, строки или константы. Sizeof может быть использован директивах препроцессора.

Например:
sizeof(char) = 1
sizeof(bit) = 0
sizeof(“abc”) = 4
sizeof(int24) = 3.

Функция offsetof(struct_type, struct_member)

Функция offsetof() возвращает смещение члена в структуре. Первым аргументом выступает имя типа структуры, второй аргумент член структуры. Функция так же может использоваться в директивах препроцессора.

```
typedef struct sStx {
char a;
uns16 b;
} Stx;
x = offsetof( Stx, b);
x = offsetof( struct sStx, a);
x = offsetof( struct_x, member_n.sub2.q[3]);
```

Автоматические предопределенные макросы и символы

Следующие символы автоматически определяются компилятором и используются как предпроцессорные макросы:

```
__CC7A__ := Integer version number: 1000 means version 1.0
                 * first 2 digits : main version
                 * last 2 digits : minor release (01='A', 02='B', etc.)

__CoreSet__ := 1700 always for all 1886 devices
```

Макросы __FILE__ и __LINE__

Макрос __FILE__ заменяется именем (символьная строка) текущего исходного файла. Макрос __LINE__ заменяется текущим номером строки (десятичная константа) текущего компилируемого исходного файла.

Макросы__DATA__ и __TIME___

Макросы для даты и времени так же определены в компиляторе

```
Macro       Format           Example
__TIME__    HOUR:MIN:SEC     "23:59:59"
__DATE__    MONTH DAY YEAR   "Jan 1 2005"
__DATE2__   DAY MONTH YEAR   " 1 Jan 2005"
```

4. ДИРЕКТИВЫ ПРЕПРОЦЕССОРА

Препроцессор распознает следующие ключевые слова:

```
#define, #undef, #include
#if, #ifdef, #ifndef, #elif, #else, #endif
#error, #warning, #message
#pragma
```

Строка препроцессорной директивы может расширена на следующую строку с помощью символа «\» в конце строки. Но при этом перед символом «\» не должно быть пробела.

#define

```
#define counter v1
#define MAX 145
#define echo(x) v2 = x
#define mix() echo(1) /* nested macro */
```

Определения сделанные с помощью #define являются глобальными и действуют во всех функциях.
Другие препроцессорные директивы так же могут быть сделаны в #define

Макрос конкатенации

Оператор конкатенация ## могут использоваться для расширения макросов. Пример:

```
#define CONCAT(NAME)    NAME  ##    _command()
CONCAT(quit)            =>          quit_command()
CONCAT()                =>          _command()
CONCAT(dummy(); help);  =>          dummy(); help_command()

#define CONCAT2(N1,N2)  N1    ##    _comm ## N2()
CONCAT2(help, and)      =>          help_command()

#define CONCAT3(NBR)          0x    ##    NBR
CONCAT3(0f);                  =>          0x0f

#define CONCAT4(TKN)          TKN   ##    =
CONCAT4(+)                    =>          +=

#define mrg(s)                s     ##    _msg(s)
#define xmrg(s)                           mrg(s)
#define foo                               alt

mrg(foo)                      =>          foo_msg(alt)
xmrg(foo)                     =>          alt_msg(alt)

#define ILLEGAL1()            ##          _command
#define ILLEGAL2()            _command          ##
```

Макрос стрингификации

Оператор стригификации # позволяет преобразовать аргумент в строковую константу. Например:

```
#define STRINGI1(ARG) #ARG
STRINGI1(help)          => "help"
```

```
STRINGI1(p="foo\n";)    => "p=\"foo\\n\";"

#define STRINGI2(A1,A2) #A1 " " #A2
STRINGI2(x,y)           => "x" " " "y" (equivalent to "x y")

#define str(s) #s
#define xstr(s) str(s)
#define foo 4
str(foo)          => "foo"
xstr(foo)         => "4"

#define WARN_IF(EXP) \
do { if (EXP) \
      warn("Warning: " #EXP "\n"); } \
while (0)
WARN_IF (x==0); => do { if (x==0)
warn("Warning: " "x==0" "\n"); } while (0);
```

#include

```
#include "test.h"
#include <test.h>
```

#include могут быть вложенными. Когда используется #include “test.h” то поиск первоначально проводится в текущей директории, и если в текущей директории файл не найден, то поиск продолжается в выбранных директориях библиотек, в том порядке как указаны эти директории в командной строке в опции –I (-I<dir>). Текущая директория исключается из поиска когда используется #include <test.h>.

Макросы могут использоваться совместно с #include. Следующий пример иллюстрирует эту возможность. Но это не является стандартом языка Си.

```
#include "file1" ".h"
#define MAC1 "c:\project\"
#include MAC1 "file2.h"
#define MAC2 MAC1 ".h"
#include MAC2
#define MAC3 <file3.h>
#include MAC3
```

Правила для макросов в #include

1. Строки использующие “” могут быть объединены, строки использующие <> не могут быть объединены.
2. Только первая часть строки может быть макросом.
3. Вложенные макросы допустимы
4. Только один макрос доступен на каждом уровне.

#undef

```
#define MAX 145
..
#undef MAX /* removes definition of MAX */
```

#undef отменяет действие #define. Если символ в #undef не был ранее определен, то при этом не будет выводится никаких сообщений об ошибке.

#if

```
#if defined ALFA && ALFA == 1
..
      /* statements compiled if ALFA is equal to 1 */
      /* conditional compilation may be nested */
#endif
```

Анализ сборных константных выражений может быть применен. Выражения разбираются таким же образом, как и условия в языке Си. Однако, все константы изначально преобразуются в 32-х битные знаковые константы.

1) Макросы автоматически раскрываются
2) defined(SYMBOL) и defined SYMBOL заменяется единице1 если символ определен, иначе 0
3) допустимы константы: 1234, -1, ‘a’ ‘\\’
4) допустимы операции: + - * / % >> << == != < <= > >= || && ! ~ ()

#ifdef

```
#ifdef SYMBOL
..
/* Statements compiled if SYMBOL is defined.
Conditional compilation can be nested. SYMBOL
should not be a variable or a function name. */
#endif
```

#ifndef

```
#ifndef SYMBOL
/* statements compiled if SYMBOL is not defined */
#endif
```

#elif

```
#ifdef AX
..
#elif defined BX || defined CX
/* statements compiled if AX is not
defined, and BX or CX is defined */
#endif
```

#else

```
#ifdef SYMBOL
..
#else
/* statements compiled if SYMBOL is not defined */
#endif
```

#endif

```
#ifdef SYMBOL
..
#endif /* end of conditional statements */
```

#error

```
#error This is a custom defined error message
```

Компилятор генерирует сообщение об ошибке с текстом найденным за #error.

#warning

```
#warning This is a warning
```

Производится вывод следующего сообщения.
`Warning test.c 7: This is a warning`
Но это не является стандартом языка Си.

#message

```
#message This is message 1
```
Производится вывод следующего сообщения.
`Message: This is message 1`
Но это не является стандартом языка Си.

Директивы PRAGMA

Директивы pragma используются для задания специфических процессорных особенностей.

#pragma alignLsbOrigin <a> [to <b>]

Эта pragma позволяет задать начало выравнивания. Компилятор проверит, если младший значимый байт на начало с адреса эквивалентного <a>, или на нахождение в диапазоне от <a> до <b>. Если это не так, начало будет увеличено до тех пор пока условие не станет выполнятся. Оба <a> и <b> могут быть в диапазоне от -254 до 254, и должны быть четными числами.

```
#pragma alignLsbOrigin 0
#pragma alignLsbOrigin 6 to 100
#pragma alignLsbOrigin 0 to 190 // [-254 .. 254]
#pragma alignLsbOrigin -100 to 10
```

Некоторые выравнивания используются для того что бы вычисляемые GOTO не пересикали 256 байтную адресную границу. Для более подробной информации обратитесь в раздел Выравнивание в части 9.1. Вычисляемые GOTO

#pragma asm2var 1

Разрешает EQU для различных преобразований. Это подробнее расписано в разделе 6.6. Inline Assembly

#pragma assume *<pointer> in rambank <n>

#pragma assume сообщает компилятору что 8-ми битные указатели в ОЗУ используются в ограниченном адресном пространстве. Подробнее в разделе 2.4. Указатели.

```
#pragma assume *p in rambank 3
```

#pragma bit <name> @ <N.B or variable[.B]>

Определяет глобальную битовую переменную <name>. Которая используется для присвоения точных адресов битовым переменным. Допустимы только существующие адреса:

```
#pragma bit bitxx @ 0x20.7
#pragma bit rx @ FSR0.1
#pragma bit C_bit @ Carry
```

Примечание: Если компилятор обнаруживает двойное присвоение одному расположению в ОЗУ, то об этом будет сообщено в виде предупреждения. Предупреждение может быть пропущено, если второе присвоение использует имя переменной от первого присвоения в качестве адреса (*#pragma bit var2 @ var1*)

#pragma cdata[ADDRESS] = <VXS>, .., <VXS>

#Pragma cdata позволяет сохранить в памяти программ 16-ти битные данные по фиксированным адресам. Более подробно в разделе 6.9. CDATA.

```
#pragma cdata[ADDRESS] = <VXS>, .., <VXS>
#pragma cdata[] = <VXS>, .., <VXS>
#pragma cdata.IDENTIFIER = <VXS>, .., <VXS>
ADDRESS: 16 bit word address
VXS : < VALUE | EXPRESSION | STRING>
VALUE: 0 .. 0xFFFF
EXPRESSION: any valid C constant expression,
i.e. 0x1000 | (3*1234)
STRING: "Valid C String\r\n\0\x24\x8\xe\xFF\xff\\\""
```

#pragma char <name> @ <constant or variable>
Определяет глобальную переменную <name>. Используется для доступа к переменной по фиксированному адресу. Допустимы только существующие адреса:

```
#pragma char i @ 0x20
#pragma char PORTX @ PORTC
```

Компилятор обнаруживает двойные присвоения в ОЗУ и выводит предупреждения об этом. Предупреждение может быть продушено, если второе присвоение использует имя переменной от первого присвоения в качестве адреса.

#pragma chip [=] <device>

Определяет тип микроконтроллера. Это позволяет компилятору правильно выбрать настройки размеров памяти программ, памяти ОЗУ и имен переменных. Так же тип микроконтроллера может быть выбран через командную строку.

```
#pragma chip PIC1886BE5
```

#pragma computedGoto [=] <0,1>

Используется для задания региона действия вычисляемого GOTO. Подробнее в разделе 9.1. Вычисляемое GOTO.

```
#pragma computedGoto 1 // start region
#pragma computedGoto 0 // end of region
```

#pragma inlineMath <0,1>

Компилятор может генерировать inline реализацию целочисленных математических библиотечных функций. Это позволяет увеличить скорость, но увеличивает размер кода.

```
#pragma inlineMath 1
a = b * c; // inline integer code is always generated
#pragma inlineMath 0
```

#pragma insertConst

Компилятор обычно вставляет константы в конец программы (старшие адреса). Данная pragma указывает что константы должны быть вставлены между пользовательскими функциями или по заданному адресу (если используется #pragma origin)

```
#pragma insertConst
```

#pragma library <0/1>

Компилятор автоматически удаляет не используемые библиотечные функции

```
#pragma library 1
// functions that are deleted if unused
// applies to prototypes and function definitions
#pragma library 0
```

#pragma mainStack <minVarSize> @ <lowestStartAddr>

Это определение задает основной стек для локальных переменных, параметров и временные переменные. Основной стек не является дополнительным стеком, но сообщает компилятору где он физически расположен. Основной стек может пересекать границы банков при необходимости. Только переменные меньшие или равные <minVarSize> будут автоматически положены в основной стек. <lowestStartAddr> является минимально возможным стартовым адресом для стека (стек растет вверх)

```
#pragma mainStack 3 @ 0x110
```

Использование этой pragma означает что локальные переменные, параметры и временные переменные размером не менее 3 байт (включая таблицы и структуры) будут сохранены в одном стеке расположенном не ниже адреса 0x110. Переменные меньшего размера и переменные с модификатором банка сохраняются в соответствии с другими правилами. Использования размера 0 соответствует всем переменным, включая битовые переменные.

Примечание: #pregma rambank игнорируется для переменных сохраненных в основном стеке. Адресный диапазон с 0x100 по 0x1FF эквивалентен банку 1 модификатора типа.

#pragma minorStack <maxVarSize> @ <lowestStartAddr>

Это определение позволяет задать минорный стек для локальных переменных, параметров и временных переменных. Основным критерием по которому может использоваться минорный стек, это повышение эффективности использования небанковой области для локальных переменных. Только переменные с размером меньше или равным <maxVarSize> будут автоматически сохранены в минорном стеке. <lowestStartAddr> является минимальным стартовым адресом для стека (стек растет вверх).

```
#pragma minorStack 1 @ 0x18
```

В данном случае локальные переменные размером до 1 байта будут размещены в небанковой области с адреса 0x18 по 0x1F. Переменные и параметры большего размера сохраняются согласно другим правилам. Размер 0 означает только битовые переменные. Эта pragma может использоваться совместно с основным стеком. Размер типа указанного для минорного стека имеет приоритет над размером указанным для основного стека.

#pragma optimize [=] [N:] <0,1>

Это определение позволяет задать оптимизацию для локального региона. Глобальная оптимизация при этом может быть включена или выключена. По определению она включена

1. перенаправление GOTO на GOTO
2. удаление излишних GOTO
3. удаление GOTO через инструкцию пропуска.
4. удаление инструкций которые заведомо не выполняются.
5. замена INCF и DECF на INCFSZ и DECFSZ
6. удаление излишний обновлений PA0 и PA1
7. удалений излишних инструкций
8. удалений излишних загрузок в WREG
9. вставление TSTFSZ и CPFSEQ

Пример:

```
#pragma optimize 0 /* ALL off */
#pragma optimize 1 /* ALL on */
#pragma optimize 2:1 /* type 2 on */
#pragma optimize 1:0 /* type 1 off */
/* combinations are also possible */
#pragma optimize 3:0, 4:0, 5:1
#pragma optimize 1, 1:0, 2:0, 3:0
```

Примечание: В командной строке опция –u выключает глобальную оптимизацию, при этом установки в исходном коде игнорируются.

#pragma origin [=] <expression>

Допустимый байтовый адрес: 0x0000 - <верхняя граница>

Это определение задает байтовый адрес последующего кода. Текущая активная область расположения не будет сдвинута назад, если в этой области не будет кода. Значении адреса расположения не может быть изменено в функции.

```
#pragma origin 8 // high priority interrupt start address
#pragma origin 0x700 + 2
```

#pragma rambank [=] <-,0,1,2,..,15>

```
- => unbanked: 0x018/0x1A – 0x01F
0 => bank 0: 0x020 – 0x0FF
1 => bank 1: 0x120 – 0x1FF
2 => bank 2: 0x220 – 0x2FF
..
15 => bank 15: 0xF20 – 0xF7F
```

Определяет регион где компилятор будет располагать переменные. Компилятор выдаст ошибку если расположение в данной области не возможно.

Не все банки ОЗУ доступны в различных микроконтроллерах, не используемые банки отображаются на нулевой банк.

#pragma rambase [=] <n>

Определяет стартовый адрес где декларируются глобальные переменные. Использование rambank и rambase очень схоже. Адрес должен попадать в диапазон физически реализованной памяти микроконтроллера. Примечание: стартовый адрес не подходит для локальных переменных, но rambase может быть использован для выбора заданного необходимого банка ОЗУ.

#pragma resetVector <n>

Вектор прерывания при сбросе в некоторых микроконтроллерах может быть переопределен. Обычно это не реализуется и в качестве вектора прерывания по сбросу используется нулевой адрес.

```
#pragma resetVector 0 // at byte address 0
#pragma resetVector 10 // at byte address 10
#pragma resetVector - // NO reset-vector
```

#pragma return[<n>] = <strings or constants>

Реализует возможность многократного RETURN. Это определение может быть расположено только в след за skip(). Компилятор может оптимизировать основную часть RETURN. Константа <n> опциональна, но позволяет компилятору вывести предупреждение, если число возвратов не равно константе <n>. Подробнее в разделе 9.1 Вычисляемое GOTO. Примечание: константные типа данных должны быть использованы для констант.

```
skip(W);
#define NoH 11
#pragma return[NoH] = "Hello world"
#pragma return[5] = 1, 4, 5, 6, 7
#pragma return[] = 0 1 2 3 44 'H' \
"Hello" 2 3 4 0x44
#pragma return[]= 'H' 'e' 'l' 'l' 'o'
#pragma return[3] = 0b010110 \
0b111 0x10
#pragma return[9] = "a \" \r\n\0"
#pragma return[] = (10+10*2), (0x80+'E') "nd"
#pragma return[] = 10000 : 16 /* 16 bit constant */ \
                   0x123456 : 24 /* 24 bit constant */ \
                   (10000 * 10000) : 32 /* 32 bit constant */
```

#pragma sharedAllocation

Это определение позволяет функции содержащей локальные переменные и параметры использовать общие независимые деревья вызовов (прерывания и основная программа). Однако, когда это создает риск перезаписи этих общих переменных. Более подробная информация представлена в разделе 6.2.

#pragma stackLevels <n>

Задает число уровней стека. Микроконтроллеры серии 1886 имеют аппаратный 16-ти уровневый стек.

```
#pragma stackLevels 14 // max 64
```

#pragma unlockISR

Вектора прерываний расположены по адресам 0x08, 0x10, 0x18 и 0x20. Данное определение позволяет перенести вектора на любые другие места. При этом компилятор не генерирует переходы с векторов 0x08, 0x10, 0x18 и 0x20 на реальные обработчики прерываний.

```
#pragma unlockISR
```

#pragma updateBank [entry | exit | default] [=] <0,1>

Это определение позволяет отключить автоматическое обновление выбора банков. Это определение может быть задано вне функции, но желательно задавать его для как можно более мелких регионов.

```
#pragma updateBank 0 /* OFF */
#pragma updateBank 1 /* ON */
```

Другой режим использования этого определения, это задание алгоритму обновления банков жестких правил выбора банка.

```
#pragma updateBank entry = 0
/* The 'entry' bank force the bank bits to be set
to a certain value when calling this function */

#pragma updateBank exit = 1
/* The 'exit' bank force the bank bits to be set
to a certain value at return from this function */

#pragma updateBank default = 0
/* The 'default' bank is used by the compiler at
loops and labels when the algorithm gives up
finding the optimal choice */
```

#pragma versionFile [<file>]

Позволяет добавлять версию к концу файла после каждой компиляции. Подробное описание можно найти в разделе 5.2.

5. ОПЦИИ КОМАНДНОЙ СТРОКИ

Компилятор использует имена исходных Си файлы для компиляции. Другие аргументы могут понадобиться если это необходимо.

```
CC7A [options] <src>.c [options]
```

-a[<asmfile>]
Создать ассемблерный файл. Имя по определению <scr>.asm

-A[scHDftumiJRN+N+N]
Опции создания ассемблерного файла
s: Символьные аргументы заменены числами
c: не добавлять исходный Си код
H: только 16-ти-ричные числа
D: только 10-ти-ричные числа
f: не добавлять директивы объектного формата
t: не добавлять символы табуляции, только пробелы
u: не добавлять расширенной информации в конец файла
m: одиночная строка исходного кода
i: не добавлять исходное представление, выравнивать влево.
J: добавлять исходные данные после инструкции для компактности ассемблерного файла
R: детализированное раскрытие макросов.
N+N+N: пространство между меткой, мнемоникой и аргументами. По определению 8+6+10

-b: не выполнять обновления битов выбора банка
-bu: не оптимизировать обновление битов выбора банков

-B[pims]
Создать файл с настройками препроцессора <scr>.cpr
p: частичный препроцессор
i: без include файлов
m: модифицировать символы
s: модифицировать строки

-CC[<file>]
Создать COD файл, в режиме Си

-CA[<file>]
Создать COD файл, в режиме ASM

-cd: располагать cdata вне памяти программ (только предупреждения)

-cu: использовать 32-х битное представление констант
-cxc: не искать в текущей директории include файлы

-dc: не создавать выходной файл компилятора <scr>.occ

-D<name>[<token>xxx]
Определяет макрос, эквивалентный #define name xxx

-e: Вывод сообщений об ошибке в одну строку.

-ed: не печатать детали по ошибкам
-ew: не печатать детали по предупреждениям
-eL: список деталей ошибок и предупреждений в конце файла
-E<N>: остановиться после <N> ошибок (по определению 4)

-f<hex-file-format>
INHX8M, INHX8S, INHX16, INHX32. По определению INHX32.
INX8S использует выходные файлы: <file>.HXH или <file>.HXL

-F: создает файл <scr>.err со списком ошибок

-g: не заменять call на goto

-GW: динамически выбирать skip() формат, предупреждение на длинный форматы
-GD: динамически выбирать skip() формат (по определению)
-GS: всегда использовать короткий skip() формат (предупреждение если превышена граница в 256 байт)
-GS: всегда длинный skip() формат.

-I<directory>
Каталог в котором размешены заголовочные файлы. До 5 директорий могут быть указаны через –I. Когда используется #include “test.h” текущая директория просматривается в первую очередь. Если искомый файл не найден, то поиск продолжается в указанных через -I директориях. Текущая директория не просматривается, если используется #include<test.h>

-li<ENVI>: директория с заголовочными файлами из переменных окружения
-lh<ENVD>: текущая директория из переменных окружения

-L[<col>,<lin>]
Создать файл листинга <src>.lst
Максимальное число столбцов в строке <col> и число строк на странице <lin>.
По определению –L80,60

-mc1: по определению “const” указатель размером 1 байт
-mc2: по определению “const” указатель размером 2 байта
-mr1: по определению указатель ОЗУ размером 1 байт
-mr2: по определению указатель ОЗУ размером 2 байта
-mm1: по определению все указатели размером 1 байт
-mm2: по определению все указатели размером 2 байта

-Ma : убрать все автоматически созданные метки в ассемблерном файле и файле листинга

-o<name>: создать HEX файл с именем name

-O<directory> : директория для создаваемых файлов. Файлы создаваемые компилятором будут размещены в этой директории, за исключением случаев, когда указан полный путь к файлу.
-p<device> : определение целевого микроконтроллера, при этом он должен иметь заголовочный файл.

-p- : очистить любые настройки заданные -p<chip> для задания нового целевого микроконтроллера.

-q<N>: запретить прерывания на <N> уровне вложенных вызовов через CALL. Например, -q1 задает, что в любых функциях, кроме main запрещены все прерывания. Данный механизм предназначен, для программной защиты стека от переполнения.

-Q: создать файл дерева вызовов CALL в файл <src>.fcs

-S: не выводить никаких сообщений от компилятора

-u: не оптимизировать

-V[rnuD] : создать файл переменных <src>.var. По определению сортированный по адресам.
r: только переменные определенные в коде
n: сортировать по имени
u: не сортировать
D: десятичные числа

-wC: предупреждать о проблемах совместимости вверх
-wE: не предупреждать о округлении чисел с фиксированной запятой
-wi: не предупреждать о inline реализации целочисленного умножения
-wm: не предупреждать о одиночных вызовах целочисленных функций
-wO: предупреждать о вызовах библиотечных операций
-wr: не предупреждать о рекурсивных вызовах
-wS: предупреждать, когда размерность константы не учитывает знаковый бит
-wU: предупреждать о невызываемых функциях

-W: ждать нажатия клавиши после компиляции

-zZ: оптимизация по размеру кода
-zD: оптимизация по скорости (по определению)

Двойные кавычки “ ” выделяют путь при опциях.
```
-I"C:\Program Files\cc7a"
```

Путь может быть так же разделен прямым слешем “/”
```
c:/compiler/lib/file.h
```

Настройки компилятора по умолчанию
- HEX файл сохраняется в файле <name>.hex
- оптимизация включена
- расширенный уровень вызовов разрешен
- обновление битов выбора банков разрешено
- отдельные комментарии разрешены
- символы являются беззнаковыми

5.1.Параметры в файле

Параметры вызова компилятора могут быть оформлены в виде файла.
```
cc7a [..] +<filename> [..]
```

Множество файлов с параметрами может быть применено. Вложенность файлов с параметрами может достигать 5 уровней. Число параметры в файле неограниченно. Параметры могут разделятся концом строки, пробелом или символом табуляции. Комментарии в файле параметров допустимы и отделятся “//”

```
// the rest of the line is a comment
```

Пробелы могут быть добавлены к каждому параметру, если между “-“ и самим параметром так же есть пробел. Синтаксис не позволяет использовать более одного параметра в каждой строке. Например:

```
- D MAC = 1 + OP
- p 17C42 // comment
-p 17C42 // this will not work
- p 17C42 -a // not this either
```

Примечание: Путь к файлу может потребоваться, если он не расположен в текущей директории

Правила разбор строки в параметрах в файле

1. Двойные кавычки удаляются
2. использование \” означает задание одиночного символа кавычек “

```
-I"C:\Program Files\cc7a"          ==> -IC:\Program Files\cc7a
-IC:"\Program Files"\cc7a          ==> -IC:\Program Files\cc7a
-DMyString="\"Hello\n\""           ==> -DMyString="Hello\n"
-DQuote='\\"'                      ==> -DQuote='\"'
```

5.2.Автоматическое увеличения номера версии в файле

Компилятор позволяет автоматически увеличивать один или несколько номеров версии при каждой компиляции. Допустимо три вида синтаксиса:

```
1. Option : -ver#verfile.c
#include "verfile.c" // or <verfile.c>

2. Option : -ver
#pragma versionFile // next include is version file
#include "verfile.c" // or <verfile.c>

3. Option : -ver
#pragma versionFile "verfile.c" // or <verfile.c>
```

Для увеличения номера версии необходим соответствующий параметр при компиляции. Если находится десятичный номер в конце подсоединенного файла, то он увеличивается. Измененный файл сохраняется до того как он будет скомпилирован. Нет специального синтаксиса для файла с номером версии.

```
#define MY_VERSION 20
#define VER_STRING "1.02.0005"
/* VERSION : 01110 */
```

Если десятичное число равно 99, то новое число будет 100, при этом длина увеличится на 1 символ. Если число 099, то длина увеличена не будет. Файл версии не должен быть слишком большим (до 20 кБайт), противном случае будет выдаваться ошибка.

Форматы 2 и 3 допускают наличие более одного файла с версиями. Это рекомендовано для использования в условиях условной компиляции для управления различными редакциями одной программы.

5.3.Переменные окружения

Переменные окружения могут быть использованы для задания папок с заголовочными файлами.

Переменная CCINC является альтернативой для параметра –I<path>. Компилятор будет использовать только эту переменную или другую заданную, если в командной строке в параметрах были заданы следующие настройки:

```
-li         : read default environment variable CCINC
-li<ENVI>   : read specific environment variable
```

Переменная CCHOME может быть использована для задания основной папки для компиляции. Компилятор будет использовать только эту переменную или другую заданную, если в командной строке в параметрах были заданы следующие настройки:

```
-lh : read default environment variable CCHOME
-lh<ENVP> : read specific environment variable
```

6. ПРОГРАММНЫЙ КОД

Страницы памяти программ

Микроконтроллеры серии 1886 могут иметь до 8 страниц кода, каждая страница имеет размер 8К слов. Использование более 1 страницы требует использования механизма выбора страниц. Например, все функции расположенные за #pragma codepage будут расположены в указанную страницу. Страница 0 выбрана по умолчанию.

```
/* functions proceeding the first codepage statement are placed on
codepage 0 */

#pragma codepage 2
char fx(void) { .. }
/* this function is placed on codepage 2 */

#pragma codepage 1
/* following functions are placed on codepage 1 */
```

При переключение на другу страницу, компилятор сохраняет указатель на свободное место для каждой страницы. Использование различных страниц это всего лишь вопрос оптимизации. Оптимизатор может удалить некорректные настройки и очистить настройки битов выбора страниц.

Переключение между страницами может занять от 1 до 2 инструкций. Компилятор вставляет эти инструкции автоматически.

Компилятор выдаст сообщение об ошибке, когда лимит страниц будет исчерпан. Несуществующие страницы отображаются на первую существующую.

Другой путь задания расположения функций

Определение #pragma location позволяет задавать расположение как прототипов функции так и их реализации. Это определение используется когда функции определены в библиотечных файлах, или когда функции располагаются в больших программах. Это нормальное использование в условиях ограниченного пространства в заголовочных файлах. Правила, по которым применяется #pragma location

1. Прототип функции должен разместить функции в желаемой странице, если она отличается от текущей страницы, когда функции реализации компилируется
2. #pragma location имеет больший приоритет чем #pragma codepage
3. #pragma location – восстанавливает текущую активную страницу установленную последним #pragma codepage или #pragma origin

```
#pragma location 1 // codepage 1
void f1(void); // assigned to codepage 1
void f2(void);
void f3(void);

#pragma location 3 // codepage 3
void f4(void);

#pragma location - // return to the active codepage
void f5(void); // this prototype is not located
```

Примечание:

1. Определения места расположения должны быть определены до функции реализации
2. Функции не обязательно будут расположены в текущей активной странице
3. Предупреждения будут выданы в случае конфликов

Определение #pragma location должно применяться если требуется. Например, когда функции из одного исходного файла должны быть расположены в различных страницах или это позволит задать более оптимально расположение. Определения #pragma codepage обычно достаточно.

Модификатор типа страниц

Модификатор типа страниц page0…page7 может заменить #pragma location/codepage

```
page2 void fx(void) { .. } // in codepage 2
page1 char f2(char a); // in codepage 1
```

Модификатор типа страниц задает страницу расположения как для прототипа функции так и для ее реализации

Примечание 1. Модификатор типа страницы имеет больший приоритет над #pragma location.
Примечание 2. Когда страница определена с модификатора типа страниц или #pragma location/codepage, то расположение не может быть изменено в реализации функции или через использования второго прототипа.

Биты выбора страницы

Страница выбирается через биты выбора в регистре PCLATH (3 бита). Обновление битов выполняется автоматически компилятором, и попытка установить или сбросить эти биты в исходном коде будут удалены оптимизатором. Но эта функция оптимизатора может быть отключена через параметр –j в командной строке.

Глубина уровней вызова CALL

Для микроконтроллеров серии 1886 глубина вызовов функций не может превышать 16. Компилятор автоматически проверяет это ограничение.

Компилятор может заменить CALL на GOTO для увеличения глубины вызова функций.

1. Если функция вызывается однократно. CALL будет заменен на GOTO, команды возврата так же будут заменены на GOTO. CALL может быть заменен на GOTO только если необходимо увеличить глубину вызова функций. Так же CALL не заменяется на GOTO когда:
   а) счетчик команд (PCL) модифицируется в пользовательском коде (вычисляемое GOTO) в функции типа char
   б) Число возвращаемых значений превосходит 10

2. CALL следующий за RETURN так же заменяется на GOTO.

Контроль уровня стека при использовании прерываний

Компилятор подразумевает, что прерывания могут произойти в любом месте программы, а так же на самом глубоком месте вызова функций. Сообщение об ошибке будет выведено если будет возможно переполнение стека. Это не всегда может привести к ошибке, так как могут применяться механизмы защиты через биты разрешения прерываний, но этого необходимо избегать. Так же бывают случаи, когда необходима вся глубина стека для вызова функций основной программы.

Параметр –q<N> позволяет задать компилятору настройки, позволяющие вывести предупреждение о функциях, при вызове которых в стеке не останется места для прерываний. Параметр –q2 означает что для обработчика прерываний необходимо минимум 2 уровня стека. –q1 минимум один уровень стека.

Функции, используемые различными вызовами в дереве вызовов.

Сообщение об ошибке будет выведено если компилятор обнаружит функции которые вызнаются из основного тела программы и например из обработчика прерывания. Это так же применимо к математическим библиотечным функциям. Проблема заключается в том, что локальные переменные этой функции расположены статически и если при прерывании будет ее повторный вызов, то возможно изменение ее локальных переменных при выполнении прерывания и после возвращения из обработчика обратно в эту функцию ее дальнейшее функционирование будет некорректным.

Сообщение об ошибке будет заменено на предупреждение при применении следующего определения

```
#pragma sharedAllocation
```

Это определение обозначает, что локальные переменные и параметры в этой функции описаны корректно и их перезаписи не будет. Сам компилятор никак не разрешает данную ситуацию.

Рекурсивные функции

Рекурсивные функции допустимы. При этом условия окончания рекурсии должны быть детерминированы и не превышать допустимую глубину стека. При этом компилятор не допускает наличие локальных переменных в рекурсивных функциях. Локальные переменные и параметры должны быть написаны так, что бы они эмулировали стек.

Предупреждение вырабатывается, когда компилятор обнаруживает функцию, которая вызывает себя или вызывает себя через другие функции. Это предупреждение можно отключить с помощью параметра –wr в командной строке.

Прерывания

Микроконтроллеры серии 1886 имеют низкоприоритетные и высокоприоритетные прерывания.

Структура обработчика прерываний приведена ниже:

```
#include "int17XXX.h"
#pragma origin 0x8
interrupt iServer( void)
```

```
{
multi_interrupt_entry_and_save
PERIPHERAL_service:
// save on demand: PRODL,PRODH,TBLPTRH,TBLPTRL,FSR0,FSR1
// the right peripheral interrupt flag must be cleared manually
/* process periferal interrupt */
if (RC1IF) {
RC1IF = 0; // application code to be inserted here
}
if (TX1IF) {
TX1IF = 0; // application code to be inserted here
}
if (CA1IF) {
CA1IF = 0; // application code to be inserted here
}
if (CA2IF) {
CA2IF = 0; // application code to be inserted here
}
if (TMR1IF) {
TMR1IF = 0; // application code to be inserted here
}
if (TMR2IF) {
TMR2IF = 0; // application code to be inserted here
}
if (TMR3IF) {
TMR3IF = 0; // application code to be inserted here
}
if (RBIF) {
RBIF = 0; // application code to be inserted here
}
// restore on demand: PRODL,PRODH,TBLPTRH,TBLPTRL,FSR0,FSR1
goto RESTORE_and_return;
TMR0_service:
// save on demand: PRODL,PRODH,TBLPTRH,TBLPTRL,FSR0,FSR1
/* process Timer 0 interrupt */
// T0IF is automatically cleared when the CPU vectors to 0x10
// application code to be inserted here
// restore on demand: PRODL,PRODH,TBLPTRH,TBLPTRL,FSR0,FSR1
goto RESTORE_and_return;
T0CKI_service:
// save on demand: PRODL,PRODH,TBLPTRH,TBLPTRL,FSR0,FSR1
/* process T0CKI pin interrupt */
// T0CKIF is automatically cleared when the CPU vectors to 0x18
// application code to be inserted here
// restore on demand: PRODL,PRODH,TBLPTRH,TBLPTRL,FSR0,FSR1
goto RESTORE_and_return;
INT_service:
// save on demand: PRODL,PRODH,TBLPTRH,TBLPTRL,FSR0,FSR1
/* process INT pin interrupt */
// INTF is automatically cleared when the CPU vectors to 0x8
// application code to be inserted here
// restore on demand: PRODL,PRODH,TBLPTRH,TBLPTRL,FSR0,FSR1
RESTORE_and_return:
interrupt_exit_and_restore
}
```

Ключевое слово interrupt позволяет указать обработчику что бы функция заканчивалась инструкцией RETFIE. Это так же позволяет вызывать функцию из обработчика прерываний (это определяется так же в прототипе функции определенном первым)

Обработчик прерываний требует как минимум одного свободного уровня в стеке. Это автоматически проверяется компилятором. Однако, если программы имеет рекурсивные функции, данная ситуация не может быть проверена компилятором.

Векторы прерывания установлены по адресам 0x08, 0x10, 0x18 и 0x20. Обработчики прерывания могут быть установлены только на эти адреса. Определение #pragma origin может быть использована для перехода через не используемую область программ.

Следующее определение позволяет разместить обработчик где угодно. Но компилятор не создаст переходы с адресов 0x08, 0x10, 0x18 и 0x20 на обработчики прерываний.

```
#pragma unlockISR
```

Основные регистры, такие как WREG, ALUSTA, PCLATH и BSR должны быть сохранены и затем восстановлены в обработчике прерываний. Однако, если эти регистры не модифицируются в обработчике, то они могут не сохранятся. В файле “int17xxx.h” содержится рекомендованная программная последовательность для сохранения и восстановления регистров. Остальные регистры должны сохраняться вручную. Обработчик прерываний так же может содержать локальные переменные. Сохранение локальных переменных должны производится отдельно, так как прерывания могут произойти в любой момент времени.

Важно: Компилятор автоматически проверяет, что основные регистры сохраняются и восстанавливаются в обработчике прерываний. Это применимо к:

```
Group 1: W/WREG, ALUSTA, PCLATH, BSR : most frequent used
Group 2: FSR0, FSR1 : indirect access
Group 3: TBLPTR : reading 'const' data
Group 4: PRODL, PRODH : multiplication instructions
```

Примечание: Возможно созадть обработчик прерыаний в котором не требуется сохранять регистры перед обработкой прерывания. Секция Пользовательское сохранения и восстановления при прерывании, показывает исходный код обработчика, который не беспокоит основные регистры

Так же обычно не требуется сохранения PCLATH если используется только одна страница кода (не более 8192 инструкций). Однако, PCLATH должен быть сохранен, если используется в коде программы вычисляемые GOTO.

Так же возможно ограничить сохранение и восстановление специальных регистров в небольшом регионе обработчика прерываний, если эти регистры модифицируются в этом регионе.

Компилятор позволяет реализовывать пользовательские алгоритмы сохранения и восстановления регистров при прерываниях. Если вы хотите реализовать собственный алгоритм сохранения и восстановления регистров, ознакомьтесь с разделом Пользовательское сохранение и восстановление при прерывании

Компилятор обнаруживает если ранее использованные регистры модифицируются в обработчике прерываний без предварительного сохранения и последующего восстановления. Приведенные макросы сохраняют только WREG, ALUSTA, PCLATH и BSR. Другие регистры должны быть сохранены и восстановлены в пользовательском коде программы.

Например, если FSR0 изменяется при табличном доступе или через указатель, или прямой записью, компилятор проверит как FSR0 сохранен и восстановлен. Так же FSR0 может сохраняться и восстанавливаться в область памяти через индексный доступ, в этом случае он не должен сохранятся.

Предупреждения выводятся если регистры из группы 2, 3, 4 были сохранены, но не изменялись. Ошибки и предупреждения могут не выводится, если использовать следующие определения

```
#pragma interruptSaveCheck n // no warning or error
#pragma interruptSaveCheck w // warning only
#pragma interruptSaveCheck e // error and warning (default)
```

Данные определения изменяют проверку всех регистров.

Пользовательское сохранение и восстановление при прерывании

Если не требуется использование стандартных макросов сохранения и восстановления, то компилятор так же позволяет реализовывать другие структуры.

А) Вы можете использовать собственные алгоритмы сохранения и восстановления, это может быть реализовано встроенным ассемблером. Если компилятор будет выдавать предупреждения и ошибки на ваш код, то их можно запретить (но на ваш риск)

```
#pragma interruptSaveCheck n // no warning or error
```

Б) Регистры не требуют сохранения когда используется следующий алгоритм обработчика прерываний. Проверки сохранения регистров при этом НЕ ДОЛЖНЫ БЫТЬ ВЫКЛЮЧЕНЫ.

```
bx2 = 1; // BSF 0x1F,bx2 ; unbanked locations only
bx1 = 0; // BCF 0x1F,bx1 ; unbanked locations only
bx3 = !bx3; // BTG 0x1F,bx3 ; unbanked locations only
btss(bx1); // BTFSS 0x1F,bx1 ; unbanked locations only
btsc(bx1); // BTFSC 0x1F,bx1 ; unbanked locations only
vs = swap(vs); // SWAPF vs,1 ; unbanked locations only
vs = incsz(vs); // INCFSZ vs,1 ; unbanked locations only
vs = decsz(vs); // DECFSZ vs,1 ; unbanked locations only
vs = incsnz(vs);// INFSNZ vs,1 ; unbanked locations only
vs = decsnz(vs);// DCFSNZ vs,1 ; unbanked locations only
a = b; // MOVFP a,b ; unbanked locations only
a = 0; // CLRF a,1 ; unbanked locations only
a = 0xFF; // SETF a,1 ; unbanked locations only
vs = rrnc(vs); // RRNC vs,1 ; unbanked locations only
vs = rlnc(vs); // RRNC vs,1 ; unbanked locations only
skipIfZero(vs); // TSTFSZ vs ; unbanked locations only
nop(); // NOP
clrwdt(); // CLRWDT
```

В) Так же допустимы разрешение обработки прерываний в специальных областях программы (например циклы ожидания), в этом случае регистры могут быть модифицированы в обработчике без влияния на основную программу. В этом случае необходимо отключить проверку сохранения регистров

```
#pragma interruptSaveCheck n // no warning or error
```

Обработка прерывания может быть очень сложна и может иметь очень много подвохов.

Код первоначальной инициализации и завершения

Первоначальный код содержит переход на main(). Никакие переменные не инициализируются. Вся инициализация должна производится в пользовательском коде. Это упрощает программу, когда используется сторожевой таймер или MCLR для функций пробуждения.

Так же это позволяет разместить main() в любой отключении вектора сброса. Это может быть сделано через определение.

```
#pragma resetVector -
```

В этом случае инициализационный код должен быть вставлен вручную, когда автоматический вектор сброса удален, например через определение cdata[] (подробнее cdata.txt)

Инструкция SLEEP выполняется когда процессор выходит из main(). Это останавливает выполнение программы и переводит микроконтроллер в энергосберегающий режим. Программа может быть перезапущена сторожевым таймером или через внешний сброс.

Микроконтроллеры 1886 так же могут быть перезапущены через прерывания. Дополнительная инструкция GOTO вставляется, если main() завершен и переведен в SLEEP режим. Это позволяет перезапустить программу после прерывания. Дополнительные GOTO не вставляются, если команды sleep() вставлены еще где либо в программе.

Очистка всей памяти ОЗУ

Встроенная функция clearRAM() позволяет установить все ячейки ОЗУ в нулевое состояние. Эта функция использует FSR0 регистр. Рекомендуемое использование приведено ниже:

```
void main(void)
{
      if (TO == 1 && PD == 1 /* power up */)
      {
            WARM_RESET:
            clearRAM(); // set all RAM to 0
      }
      ..
      if (condition)
      goto WARM_RESET;
}
```

Размер кода и время выполнения зависит от выбранного чипа, обычно требуется 4 инструкции для каждой ячейки памяти. На 4 МГц каждая инструкция выполняется 1 микросекунду. Для микроконтроллеров серии 1886 с размером ОЗУ 902 байта потребуется 902*4+31 = 3639 инструкций или 3,6 миллисекунды на 4 МГц.

Поддержка библиотек

Библиотеки включают поддержку стандартной математики и поддержку пользовательских функций. Файлы библиотек должны быть подключены в начале программы, но после обработчика прерываний.

```
// ..interrupt routines
#include “math16.h” // 16 bit integer math
#include “math24f.h” // 24 bit floating point
#include “math24lb.h” // 24 bit math functions
```

Компилятор автоматически удалит не используемые библиотечные функции. Эта особенность может быть применена и к не используемым функциям основной программы.

```
#pragma library 1
// library functions that are deleted if unused
#pragma library 0
```

Математические библиотеки

```
Integer: 8, 16, 24 and 32 bit, signed and unsigned
Fixed point: 20 formats, signed and unsigned
Floating point: 16, 24 and 32 bit
```

Все библиотеки оптимизированы под компактный код. Все переменные (за исключением флагов для типа с плавающей точкой) размещены в стеке для более эффективного использования ОЗУ совместно с другими локальными переменными.

Операции с фиксированной запятой требуют внимательного анализа для получения корректных результатов. Так как накапливаемая ошибка и потеря переносов может привести к потере вплоть до знаковых бит. Иногда стоит применить тип с плавающей запятой для получения более точного результата. Но при этом необходимо помнить, что операции с плавающей запятой требуют большего объема кода. Операции с фиксированной запятой и с плавающей выполняются медленно. Операции умножения и деления с плавающей запятой выполняются быстрей чем с фиксированной, но остальные (сложение, вычитание и т.п.) выполняются медленней.

Операции не найденные в библиотеках встроены в сам компилятор. Так же компилятор будет встраивать inline реализации тех операций которые в этом случае будут более эффективны.

Следующие параметры командной строки допустимы
-we : не выводить предупреждений когда округляются числа с фиксированной запятой
-wO: выводить предупреждение при вызове библиотечных функций
-wi: не выводить предупреждения на inline реализацию операции умножения
-wm: не выводить предупреждение на однократный вызов библиотечной функции

Библиотека операций с целыми числами

Математическая библиотека с целочисленными операциями позволяет проводить оптимизацию по скорости или размеру. Библиотека содержит операции умножения, деления и вычисления остатка от деления.

```
math16.h : basic library, up to 16 bit
```

```
math24.h : basic library, up to 24 bit
math32.h : basic library, up to 32 bit

The min and max timing cycles are approximate only.

Sign: -: unsigned, S: signed
Sign Res=arg1 op arg2         Program Approx. CYCLES

A:math32.h
B:math24.h
C:math16.h
                              Code  min   aver  max
.B.   S     24 = 16 * 16      28    40    40    42
A..   S     32 = 16 * 16      37    47    48    53
A..   -     32 = 16 * 16      25    41    41    41
.B.   -     24 = 24 * 24      28    46    46    46
A..   S     32 = 32 * 16      45    62    62    65
A..   -     32 = 32 * 16      39    65    65    65
A..   S/-   32 = 32 * 32      53    77    77    77
ABC   -     16 = 16 / 8       19    229   229   229
AB.   -     24 = 24 / 8       20    362   362   362
A..   -     32 = 32 / 8       21    511   511   511
ABC   -     16 = 16 / 16      23    248   251   296
.B.   -     24 = 24 / 16      27    429   453   533
A..   -     32 = 32 / 16      28    594   637   738
.B.   -     24 = 24 / 24      30    464   470   584
A..   -     32 = 32 / 32      37    744   754   968
ABC   S     16 = 16 / 8       35    190   195   205
AB.   S     24 = 24 / 8       38    299   305   318
A..   S     32 = 32 / 8       41    424   431   447
ABC   S     16 = 16 / 16      46    257   266   320
.B.   S     24 = 24 / 16      49    398   418   489
A..   S     32 = 32 / 16      52    555   587   674
.B.   S     24 = 24 / 24      57    473   487   612
A..   S     32 = 32 / 32      68    753   772   1000
ABC   -     8 = 16 % 8        18    221   221   221
.B.   -     8 = 24 % 8        19    352   352   352
A..   -     8 = 32 % 8        20    499   499   499
ABC   -     16 = 16 % 16      21    241   243   273
.B.   -     16 = 24 % 16      25    412   439   500
A..   -     16 = 32 % 16      26    567   618   695
.B.   -     24 = 24 % 24      28    456   461   552
A..   -     32 = 32 % 32      35    735   743   927
ABC   S     8 = 16 % 8        31    186   187   192
.B.   S     8 = 24 % 8        34    294   295   302
A..   S     8 = 32 % 8        37    417   419   427
ABC   S     16 = 16 % 16      44    252   257   294
.B.   S     16 = 24 % 16      47    393   405   451
A..   S     16 = 32 % 16      50    549   569   625
.B.   S     24 = 24 % 24      57    467   476   575
A..   S     32 = 32 % 32      70    746   760   954
```

Библиотека операций с фиксированной запятой

math16x.h : 16-ти битная фиксированная запятая, 8_8, со знаком и без знака
math24x.h : 24-х битная фиксированная запятая, 8_16, 16_8, со знаком и без знака
math32x.h : 32-х битная фиксированная запятая, 8_24, 16_16, 24_8, со знаком и без знака

Библиотеки могут использоваться как по отдельности, так и вместе

Времена выполнения операций выражены в циклах (4 такта) и включают передачу параметров, вызов, возврат и присвоение результирующего значения. Временные характеристики вычислены при циклическом выполнении операции над различными аргументами. Время вычисления результата для конкретных аргументов может отличатся.

```
Sign: -: unsigned, S: signed
Sign        Res=arg1 op arg2        Program Approx. CYCLES
math16x.h:                          Code  min   aver  max
S           8_8 = 8_8 * 8_8         46    48    51    56
-           8_8 = 8_8 * 8_8         22    36    36    36
S           8_8 = 8_8 / 8_8         49    445   464   532
-           8_8 = 8_8 / 8_8         31    476   499   580

math24x.h:                          Code  min   aver  max
S           16_8 = 16_8 * 16_8      79    82    87    94
-           16_8 = 16_8 * 16_8      49    70    70    70
S           16_8 = 16_8 / 16_8      60    718   749   897
-           16_8 = 16_8 / 16_8      38    765   797   965
S           8_16 = 8_16 * 8_16      94    97    102   109
-           8_16 = 8_16 * 8_16      64    85    85    85
S           8_16 = 8_16 / 8_16      60    886   935   1105
-           8_16 = 8_16 / 8_16      38    941   999   1197
math32x.h:                          Code  min   aver  max
S           24_8 = 24_8 * 24_8      158   123   130   139
-           24_8 = 24_8 * 24_8      122   111   111   111
S           24_8 = 24_8 / 24_8      71    1054  1097  1357
-           24_8 = 24_8 / 24_8      45    1117  1159  1445
S           16_16= 16_16*16_16      143   146   153   162
-           16_16= 16_16*16_16      107   134   134   134
S           16_16= 16_16/16_16      71    1254  1322  1613
-           16_16= 16_16/16_16      45    1325  1399  1725
S           8_24 = 8_24 * 8_24      120   161   168   177
-           8_24 = 8_24 * 8_24      84    149   149   149
S           8_24 = 8_24 / 8_24      71    1454  1549  1869
-           8_24 = 8_24 / 8_24      45    1533  1639  2005
```

Библиотека операция с плавающей запятой

```
math16f.h : 16 bit floating point basic math
math24f.h : 24 bit floating point basic math
math24lb.h : 24 bit floating point library
math32f.h : 32 bit floating point basic math
math32lb.h : 32 bit floating point library
```

Время вычислений включает передачу параметров, вызов, возврат и присвоение результата.

```
Basic 32 bit math:          Approx. CYCLES
                            Size  min   aver  max
a * b: multiplication       138   132   136   152
a / b: division             117   512   560   657
a + b: addition             168   38    130   205
a - b: subtraction add+     5     45    137   212
int32 -> float32            78    46    69    119
float32 -> int32            84    36    77    143

Basic 24 bit math:          Approx. CYCLES
                            Size  min   aver  max
a * b: multiplication       89    82    86    98
a / b: division             100   309   343   400
```

```
a + b: addition              145   32    111   169
a - b: subtraction add+      5     39    118   176
int24 -> float24             61    36    64    107
float24 -> int24             73    31    72    116

Basic 16 bit math:           Approx. CYCLES
                             Size  min   aver  max
a * b: multiplication        58    50    54    62
a / b: division              84    138   156   174
a + b: addition              118   26    86    131
a - b: subtraction add+      5     33    93    138
int16 -> float16             69    39    71    108
float16 -> int16             53    26    60    98
```

Операции присвоения, сравнения с константами, умножение и деление на степень 2 (например a*0.5 или и*1024.0/4.0) встраиваются в код через inline

Функции библиотеки с плавающей запятой

```
float24 sqrt(float24); // square root
      Input range: positive number including zero
      Accuracy: ME:1, relative error: 1.5*10**-5 (*)
      Timing: min aver max 589 619 657 (**)
      Size: 56 words
      Minimum complete program example: 70 words

float32 sqrt(float32); // square root
      Input range: positive number including zero
      Accuracy: ME:1, relative error: 6*10**-8 (*)
      Timing: min aver max 1026 1110 1183 (**)
      Size: 66 words
      Minimum complete program example: 83 words

float24 log(float24); // natural log function
      Input range: positive number above zero
      Accuracy: ME:1, relative error: < 1.5*10**-5 (*)
      Timing: min aver max 1323 1872 2157 (**)
      Size: 213 words + basic 24 bit math library
      Minimum complete program example: 628 words

float32 log(float32); // natural log function
      Input range: positive number above zero
      Accuracy: ME:1, relative error: < 6*10**-8 (*)
      Timing: min aver max 1840 2549 2898 (**)
      Size: 267 words + basic 32 bit math library
      Minimum complete program example: 791 words

float24 log10(float24); // log10 function
      Input range: positive number above zero
      Accuracy: ME:1-2, relative error: < 3*10**-5 (*)
      Timing: min aver max 1408 1954 2253 (**)
      Size: 15 words + size of log()
      Minimum complete program example: 643 words

float32 log10(float32); // log10 function
      Input range: positive number above zero
      Accuracy: ME:1-2, relative error: < 1.2*10**-7 (*)
      Timing: min aver max 1975 2679 3014 (**)
      Size: 17 words + size of log()
      Minimum complete program example: 808 words

float24 exp(float24); // exponential (e**x) function
```

```
      Input range: -87.3365447506, +88.7228391117
      Accuracy: ME:1, relative error: < 1.5*10**-5 (*)
      Timing: min aver max 951 1652 1850 (**)
      Size: 248 words + 82(floor24) + basic 24 bit math
      Minimum complete program example: 671 words

float32 exp(float32); // exponential (e**x) function
      Input range: -87.3365447506, +88.7228391117
      Accuracy: ME:1, relative error: < 6*10**-8 (*)
      Timing: min aver max 1637 2178 2441 (**)
      Size: 320 words + 125(floor32) + basic 32 bit math
      Minimum complete program example: 872 words

float24 exp10(float24); // 10**x function
      Input range: -37.9297794537, +38.531839445
      Accuracy: ME:1, relative error: < 1.5*10**-5 (*)
      Timing: min aver max 966 1732 1866 (**)
      Size: 255 words + 82(floor24) + basic 24 bit math
      Minimum complete program example: 643 words

float32 exp10(float32); // 10**x function
      Input range: -37.9297794537, +38.531839445
      Accuracy: ME:1, relative error: < 6*10**-8 (*)
      Timing: min aver max 1638 2201 2469 (**)
      Size: 326 words + 125(floor32) + basic 32 bit math
      Minimum complete program example: 878 words

float24 sin(float24); // sine function, input in radians
float24 cos(float24); // cosine function, input in radians
      Input range: -512.0, +512.0
      Accuracy: error: < 3*10**-5 (*)
      The relative error can be larger when the output is near
      0 (for example near sin(2*PI)), but the absolute error is
      lower than the stated value.
      Timing: min aver max 400 1290 1499 (**)
      Size: 214 words + basic 24 bit math library
      Minimum complete program example: 597 words


float32 sin(float32); // sine function, input in radians
float32 cos(float32); // cosine function, input in radians
      Input range: -512.0, +512.0
      * can be used over a much wider range if lower accuracy
      is accepted (degrades gradually to 1 significant
      decimal digit at input value 10**6)
      Accuracy: error: < 1.2*10**-7 (*)
      The relative error can be larger when the output is near
      0 (for example near sin(2*PI)), but the absolute error is
      lower than the stated value.
      Timing: min aver max 551 2417 2667 (**)
      Size: 352 words + basic 32 bit math library
      Minimum complete program example: 847 words
```

(*) Точность математических функций определена с помощью более чем 1000 различных вычислений. Но результат над конкретными аргументами может отличатся. ME=1 означает, что мантисса может иметь ошибку плюс-минус 1. Таким образом относительная ошибка будет 1.5*10-5 для 24-х битной плавающей запятой и 6*10-8 для 32-х битной плавающей запятой. Лишь в небольшой части расчетов может возникнуть эта ошибка

(**) Минимальной и максимальное время вычислений было рассчитано при выполнении более 1000 различных вычислений. Однако при конкретных значениях аргументов время вычисления может отличаться. Времена измеряются в циклах (4 такта).

Быстрые и компактные операции интегрированные в код (INLINE)

Компилятор интегрирует некоторые операции в код (inline) для более эффективной их реализации:

Целочисленные:
- приводимые к сдвигу в право или лево, например: a*8 или a/2
- выделяющие верхний или нижний байт или слово, например: 2/256 или a%256
- заменяемые на логические операции, например на AND : a%64

Фиксированная запятая:
- приводимые к сдвигу в право или лево, например: a*8 или a/2
- все другие операции, за исключением умножения и деления

Плавающая запятая:
- add/sub (inc/decr) над экспонентой, например: a*128.0
- операции «==» и «!=», например: a==b.
- сравнение с константой, например : a>0
- инвертирование знакового бита, например: a = -a

Комбинирование inline операции и вызова функций

Возможно принудительное указание компилятору способа реализации (интегрирование в код inline или вызов функции) той или иной операции. Это может быть актуально когда, части проекта оптимизируются по различным критериям, например программа в целом оптимизирована по размеру кода, а обработчик прерываний по скорости. Функции с параметрами или локальными переменными не реентерабельны, т.е. не могут выполнятся повторно, потому что локальные переменные размещены в памяти и при их повторном вызове, например из обработчика прерываний они повлияют на первоначальный вызов. По этому компилятор не позволяет осуществлять вызов одних и тех же функций из обработчика прерываний и основного тела программы.

```
uns16 a, b, c;
..
a = b * c; // inline code is generated
..
#include "math16.h"
..
a = b * c; // math library function is called
..
#pragma inlineMath 1
a = b * c; // inline code is generated
#pragma inlineMath 0
..
a = b * c; // math library function is called
```

Inline модификатор типа для математических операций

Это позволяет комбинировать встраиваемые и вызываемые функции целочисленной библиотеки без создания специальной версии библиотеки. Для этого достаточно в начале до подключения самой библиотеки для выбранных операций создать прототип с inline

модификатором типа. Это оптимально делать когда необходимо выделить только одну операцию:

```
inline uns24 operator * (uns24 arg1, uns24 arg2);
#include "math24.h"
```

Прототип математической функции будет найден в начале библиотеки. Только не забудьте удалить название оператора, прежде чем добавить inline модификатор типа.

Предупреждение будет выведено если в программе происходит только один вызов функции. Вывод предупреждения может быть заблокирован параметром –wm в командной строке. Inline модификатор типа обычно игнорируется, за исключением математических функций

Обнаружение множества inline целочисленных математических операций

Компилятор выдает предупреждение когда обнаруживает более чем одну inline операцию одного типа. Включение математической библиотеки позволит уменьшить размер кода, но приведет к снижению скорости. Заметим, что анализ ассемблерного кода и его тиражирование может привести к увеличению числа вызовов библиотечных функций

Предупреждение может быть заблокировано параметром –wi в командной строке.

Пример с фиксированной запятой

```
#include "math24x.h"
uns16 data;
fixed16_8 tx, av, mg, a, vx, prev, kp;
void main(void)
{
      vx = 3.127;
      tx += data; // automatic type cast
      data = kp; // assign integer part
if (tx < 0)
      tx = -tx; // make positive
      av = tx/20.0;
      mg = av * 1.25;
      a = mg * 0.98; // 0.980469, error: 0.000478
      prev = vx;
      vx = a/5.0 + prev;
      kp = vx * 0.036; // 0.03515626, error: 0.024
      kp = vx / (1.0/0.036); // 27.7773437
}
```

Размер кода 266 инструкций, включая библиотеку (129 инструкций)

Пример с плавающей запятой

596 инструкций кода, включая библиотеку (424 инструкции). Определение идентично приведенному ранее примеру с фиксированной запятой для сравнения размера кода

```
#include "math24f.h"
uns16 data;
float tx, av, mg, a, vx, prev, kp;
void main(void)
{
InitFpFlags(); // enable rounding as default
```

```
      vx = 3.127;
      tx += data; // automatic type cast
      data = kp; // assign integer part
if ( tx < 0)
      tx = -tx; // make positive
      av = tx/20.0;
      mg = av * 1.25;
      a = mg * 0.98;
      prev = vx;
      vx = a/5.0 + prev;
      kp = vx * 0.036;
      kp = vx / (1.0/0.036);
}
```

Как уменьшить размер кода

Определите, что влияет на код:

1. Какие библиотеки подключены (24 или 32 бита, плавающая или фиксированная запятая)
2. Округление может быть отключено перманентно
   ```
   #define DISABLE_ROUNDING
   #include "math32f.h"
   ```

3. Оптимизация. Оптимизация по скорости выбрана по умолчанию, это не способствует уменьшению размера кода.
   ```
   #define FP_OPTIM_SIZE // optimize for SIZE
   #define FP_OPTIM_SPEED // optimize for SPEED: default
   ```

Рекомендуемая стратегия заключается в выборе основной библиотеки для всех операций. Переход от плавающей запятой к фиксированной и обратно должен быть обусловлен тем, что это позволит увеличить эффективность.

Использование различных типов данных допустимо для уменьшения размера кода и экономии ОЗУ. Например использовать маленькие типы для индексов массивов и большие для арифметических операций.
Определитесь, какая математическая библиотека будет использоваться. Для плавающей запятой основной выбор идет между 24 битной и 32 битной библиотеками. Если вы используете 32 битные операнды, то они могут быть скомбинированы с 24 и 16-ти битными типами с плавающей запятой.

Арифметические преобразования типов
```
integer <-> float/double
integer <-> fixed point
float <-> double
fixed point <-> float/double : requires additional functions
```

В основном, использование маленьких типов данных позволяет уменьшить размер кода и требуемой памяти ОЗУ. Но это требует дополнительного анализа на предмет возникновения переполнений и слишком большой накопленной ошибки в программе. Если размер кода и скорость не важны, то рекомендуется использовать большие типы данных. Дополнительный анализ необходим для выбора оптимального решения.

Так же рекомендуется сократить число вызываемых библиотечных функций на сколько это возможно. Несмотря на то что компилятор выбирает функции автоматически, можно

использовать функции преобразования типов или сделать собственные библиотеки, путем копирования в них требуемые функции из основных библиотек для уменьшения размера кода. Все библиотеки написаны на Си. Компилятор выводит предупреждения для каждой операции которая была вызвана (параметр –wO)

Востренный ассемблер

Компилятор позволяет использовать ассемблерные вставки непосредственно в Си коде. Но есть ряд отличий по сравнению с обычным кодом программы на ассемблере. Первое, допустим только CALL вызов других функций. Второе, GOTO ограничено метками только этой функции

```
#asm
.. assembly instructions
#endasm
```

Особенности:
- различные ассемблерные форматы
- определение EQU может быть преобразовано для объявления переменных
- макросы и условная компоновка допустимы
- вызов Си функций и доступ к Си переменным
- разрешен Си стиль комментариев
- допустима оптимизация
- допустим автоматическое обновление банков

Встроенный ассемблер не является Си описанием, но запускается из Си описания. Не рекомендуется писать код следующим образом:

```
if (a==b)
#asm
nop // this is not a C statement (by definition)
#endasm
a = 0; // THIS is the conditional statement!!!
```

Встроенный ассемблер поддерживает DW. Это может быть использовано для вставки данных или специальных инструкций. Компилятор подразумевает, что вставленные данные так же могут быть инструкциями, но не будет их интерпретировать и анализировать. После выполнения DW инструкций может потребоваться принудительное установление текущего банка.

```
#asm
DW 0xFFFF ; any data or instruction (2 bytes stored)
DW 0xFFFF, 0, 0xC000 ; multiple words
#endasm
```

Ассемблерные инструкции не критичны к заглавным или прописным буквам, но переменные и символы требуют точного указания верхнего или нижнего регистра каждого символа.

```
clrwdt
Nop
NOP
```

Поддерживаемые форматы операндов приведены ниже:

```
k           EXPR
f           VAR + EXPR
f,d         VAR + EXPR, D
f,b         VAR + EXPR, EXPR
fs,fd       VAR + EXPR, VAR + EXPR
f,k         VAR + EXPR, EXPR
a           LABEL or FUNCTION_NAME
EXPR := [ EXPR OP EXPR | (EXPR) | -EXPR ]
EXPR := a valid C constant expression, plus assembly extensions
```

Формат констант:

```
MOVLW 10 ; decimal radix is default
MOVLW 0xFF ; hexadecimal
MOVLW 0b010001 ; binary (C style)
MOVLW 'A' ; a character (C style)
MOVLW .31 ; decimal constant
MOVLW .31 + 20 - 1 ; plus and minus are allowed
MOVLW H'FF' ; hexadecimal (radix 16)
MOVLW h'0FF'
MOVLW B'011001' ; binary (radix 2)
MOVLW b'1110.1101'
MOVLW D'200' ; decimal (radix 10)
MOVLW d'222'
MOVLW MAXNUM24EXP ; defined by EQU or #define
;MOVLW 22h ; NOT allowed
```

Формат, когда результат загружается в WREG:

```
decf ax,0 // load result into W
iorwf ax,w
iorwf ax,W
```

Формат, когда результат записывается в ОЗУ

```
decf ax
decf ax,1
iorwf ax,f
iorwf ax,F
```

Битовые переменные доступны в следующем формате:

```
bcf Carry
bsf Zero_
bcf ax,B2 ; B2 defined by EQU or #define
bcf ax,1
bcf ALUSTA,Carry ; Carry is a bit variable
```

Массивы, структуры и переменные больше чем 1 байт так же могут быть доступны через смещение

```
clrf a32 ; uns32 a32; // 4 bytes
clrf a32+0
clrf a32+3
clrf tab+9 ; char tab[10];
; clrf tab-1 ; not allowed
```

Метки могут быть установлены где угодно

```
goto LABEL4
```

```
LABEL1
:LABEL2
LABEL3:
LABEL4 nop
nop
goto LABEL2
```

Функции вызываются напрямую, простые беззнаковые 8-ми параметры могут передаваться через WREG

```
movlw 10
call f1 ; equivalent to f1( 10);
rcall f1 ; equivalent to f1( 10);
```

Единственный путь передать множество параметров и параметры размером более 8 бит является закончить ассемблерную вставку, использовать Си код для вызова и затем начать новую ассемблерную вставку.

```
#endasm
func( a, 10, e);
#asm
```

Описание EQU может быть использовано для определения констант. Ассемблерные блоки содержащие EQU могут быть определены вне функций. EQU константы могут быть доступны только в ассемблере. Константы определенные как #define могут быть доступны как в Си так и в ассемблере

```
#asm
B0 equ 0
B7 equ 7
MAXNUM24EXP equ 0xFF
#endasm
```

EQU так же могут быть использованы для определения адресов переменных. Однако, компилятор не будет знать об отличии EQU адреса и EQU константы при их использовании в инструкции. Когда EQU символ используется как переменная, это означает что эта ячейка запрещена для других переменных. Если изменить EQU символ на символ переменной, то это сделает его доступным и в Си коде. Это несет некоторую опасность для логики. НЕ ИСПОЛЬЗУЙТЕ последовательность EQU символов для задания массивов. Если один из элементов массива не будет читаться или записываться напрямую, то компилятор не будет знать что он является частью массива и возможно начнет использовать для других целей. Чтение и запись через FSRx и INDFx не используется для преобразования EQU определений. Следовательно, определять массивы стоит через Си синтаксис (или через #pragma char)

```
// enable equ to variable transformation
#pragma asm2var 1
..
A1 equ 0x20
..
CLRF A1
;A1 is changed from an equ constant to a char variable
```

Последующие адресные операции допустимы когда переменные (структуры или массивы) установлены по фиксированным адресам.

```
char tab[5] @ 0x110;
```

```
struct { char x; char y; } stx @ 0x120;
#asm
MOVLW tab
MOVLW &tab[1]
MOVLW LOW &tab[2]
MOVLW HIGH &tab[2]
MOVLW UPPER &tab[2]
MOVLW HIGH (&tab[2] + 2)
MOVLW HIGH (&stx.y)
MOVLW &stx.y
MOVLW &ALUSTA
#endasm
```

Комментарии допустимы в ассемблерной вставке

```
NOP ; a comment
NOP // C style comments are also valid
/*
CLRWDT ;
NOP /* nested C style comments are also valid */
*/
```

Условная компиляций так же возможна, только при применении Си синтаксиса

```
#ifdef SYMBOLA
nop
#else
clrwdt
#endif
```

Директивы препроцессора могут быть использованы в ассемблерной вставке

```
#pragma return[] = "Hello"
```

С стиль макросов может содержать ассемблерные инструкции, и условные определения. Компилятор не проверяет содержимое макросов при их задании

```
#define UUA(a,b)\
clrwdt\
movlw a \
#if a == 10 \
nop \
#endif \
clrf b
UUA(10,ax)
UUA(9,PORTA)
```

Так же необходимо помнить, что если в макросе используются метки, то они должны быть оформлены как параметр, если макрос используется более одного раза. Так же необходимо ставить обратный слеш «\» в том числе и в #endasm, для избежание ошибок когда макрос вызывается из Си кода. Это так же применимо ко всем определениям в макросах.

```
#define waitX(uSec, LBM) \
      #asm \
            LBM: \
                  NOP \
                  NOP \
```

```
                DECFSZ uSec,1 \
                GOTO LBM \
      #endasm \

waitX(i, LL1);
waitX(i, LL2);
```

Компилятор может оптимизировать обновление банков в ассемблерном коде, это не происходит автоматически, но может быть включено в исходном коде. Рекомендуется включить эту оптимизацию по обновлению битов выбора банка. Инструкции обновления битов выбора будут сначала убраны из кода, а затем компилятор вставить свои собственные. Если ассемблерный код имеет критические по времени участки кода, то оптимизация может быть отключена только для этих участков.

```
// default local assembly settings are b- o-
#pragma asm default b+ o+ // change default settings
#asm // using default local settings
#endasm

#asm b- o- // define local settings
#pragma asm o+ // change setting in assembly mode
#endasm // end current local settings
```

Интерпретация:

```
o+ : current optimization is performed in assembly mode
o- : no optimization in assembly mode

b+ : current bank bit updating is performed in assembly mode
b- : no bank bit update in assembly mode
```

Помните, что b+ и o+ означают обновление выполняется, если аналогичная установка сделана и в Си коде. Обновление не производится, если в Си коде отключено автоматическое обновление банков и начинается выполнятся ассемблерная вставка. Параметры в командной строке –b и –u отключают автоматическое обновление для всей программы. Установки в исходном коде после этого будут игнорироваться.

Прямое кодирование инструкций

Файл «hexcodes.h» содержит Си макрос, позволяющий закодировать инструкции напрямую.

Прямое кодирование инструкций отличается от ассемблерных вставок, с точки зрения компилятора. Компилятор видит эти инструкции не как инструкции, а как данные. Поэтому все возможности компилятора не применяются. Оптимизация, обновление банков и прочее не выполняется после DW определения.

Пример:

```
#include "hexcodes.h"
..
// 1. In DW statements:
#asm
DW __SLEEP // Enter sleep mode
DW __MOVWF(__INDF0) // Store indirectly
DW __ANDLW(0x80) // W = W & 0x80;
DW __DECF(__FSR0,__F) // Decrement FSR0
```

```
DW __CLRF(0xFF,1) // Clear ram (banked access)
DW __BCF(__ALUSTA,__Carry) // Clear Carry bit
DW __MOVFP(__INDF0, __INDF1) // Move byte indirectly
DW __GOTO(0) // Goto byte address 0
#endasm
..
// 2. In cdata statements:
#pragma cdata[1] = __GOTO(0x3FF)
```

Создание одиночной инструкции в Си коде.

Компилятор позволяет создать одиночную инструкции напрямую в Си коде. Только после компиляции необходимо проанализировать полученный полный ассемблерный код на предмет корректного места установки одиночной инструкции. Следующий пример иллюстрирует создание одиночной инструкции в Си коде.

```
nop(); // NOP
f = W; // MOVWF f
f = 0; // CLRF f
W = f - W; // SUBWF f,W
f = f - W; // SUBWF f
W = f - 1; // DECF f,W
f = f - 1; // DECF f
W = f | W; // IORWF f,W
f = f | W; // IORWF f
W = f & W; // ANDWF f,W
f = f & W; // ANDWF f
W = f ^ W; // XORWF f,W
f = f ^ W; // XORWF f
W = f + W; // ADDWF f,W
f = f + W; // ADDWF f
W = f; // MOVF f,W
W = f ^ 255; // COMF f,W
f = f ^ 255; // COMF f
W = f + 1; // INCF f,W
f = f + 1; // INCF f
W = decsz(i); // DECFSZ f,W
f = decsz(i); // DECFSZ f
W = rr(f); // RRCF f,W
f = rr(f); // RRCF f
W = rl(f); // RLCF f,W
f = rl(f); // RLCF f
W = swap(f); // SWAPF f,W
f = swap(f); // SWAPF f
W = incsz(i); // INCFSZ f,W
f = incsz(i); // INCFSZ f
b = 0; // BCF f,b
b = 1; // BSF f,b
b = !b; // BTG f,b
btsc(b); // BTFSC f,b
btss(b); // BTFSS f,b
sleep(); // SLEEP
clrwdt(); // CLRWDT
return 5; // RETLW 5
s1(); // CALL s1
goto X; // GOTO X
W = 45; // MOVLW 45
W = W | 23; // IORLW 23
W = W & 53; // ANDLW 53
W = W ^ 12; // XORLW 12
W = 33 + W; // ADDLW 33
W = 33 - W; // SUBLW 33
```

```
return; // RETURN
retint(); // RETFIE
W = addWFC(f); // ADDWFC f,W
f = addWFC(f); // ADDWFC f
W = subWFB(f); // SUBWFB f,W
f = subWFB(f); // SUBWFB f
W = rrnc(f); // RRNCF f,W
f = rrnc(f); // RRNCF f
W = rlnc(f); // RLNCF f,W
f = rlnc(f); // RLNCF f
W = decsnz(i); // DCFSNZ f,W
f = decsnz(i); // DCFSNZ f
W = incsnz(i); // INFSNZ f,W
f = incsnz(i); // INFSNZ f
f = negate(W); // NEGF f,1
a = decadj(W); // DAW a,1
multiply(f); // MULWF f
multiply(50); // MULLW 50
skipIfEQ(f); // CPFSEQ f
skipIfLT(f); // CPFSLT f
skipIfGT(f); // CPFSGT f
skipIfZero(f); // TSTFSZ f
f = refdLUInc(); // TABLRD 0,1,f
f = refdLU(); // TABLRD 0,0,f
f = refdHUInc(); // TABLRD 1,1,f
f = refdHU(); // TABLRD 1,0,f
writeLUInc(f); // TABLWT 0,1,f
writeLU(f); // TABLWT 0,0,f
writeHUInc(f); // TABLWT 1,1,f
writeHU(f); // TABLWT 1,0,f
f = refdL(); // TLRD 0,f
f = refdH(); // TLRD 1,f
writeL(f); // TLWT 0,f
writeH(f); // TLWT 1,f
```

Оптимизация в коде

Компилятор содержит мощный кодогенератор, который разработан для создания компактного кода. Например, для сравнения 32-х битной беззнаковой переменой с 32-х битной константой, обычно требуется 12 инструкций. Когда переменная сравнивается с 0, то требуется всего 5 инструкций. Кодогенератор обнаруживает такие особенности и в зависимости от ситуации обеспечивает генерацию минимально компактного кода.

Большинство кода генерируется как inline реализация, например умножение и деление. Однако, если требуется много математических операций рекомендуется воспользоваться математической библиотекой, в этом случае операции будут заменены на соответствующие функции, что позволит уменьшить размер кода. Но при этом необходимо помнить о двойных вызовах.

Оптимальный синтаксис

Сравнение нескольких бит в 16-ти битной или большей переменной

```
uns16 x;
if (x & 0xF0)
if (!(x & 0x3C))
if ((x & 0xF00) == 0x300)
if ((x & 0x7F00) < 0x4000)
```

Сравнение одиночного бита с использованием оператора “&”

```
if (a & 0x10) // BTFSC/BTFSS a,4
if (!(a & 0x80)) // BTFSS/BTFSC a,7
if ((a16 & 0x200) == 0) // BTFSS/BTFSC a16+1,1
```

Локальная оптимизация

Локальная оптимизация заключатся в отдельном просмотре компилятором кода на предмет удаления повторяющихся инструкций или перезаписи кода с применением других инструкций. Оптимизация может быть отключена параметром –u в командной строке. Элементы подвергающиеся оптимизации представлены ниже:

1. перенаправление GOTO на GOTO
2. удаление излишних GOTO
3. удаление GOTO через инструкцию пропуска.
4. замена INCF и DECF на INCFSZ и DECFSZ
5. удаление заведомо неисполняемых инструкций
6. удаление повторных инструкций задания битов выбора банков
7. удаление других повторяющихся инструкций
8. удалений излишних загрузок в WREG
9. вставление TSTFSZ и CPFSEQ

Примечание: Оптимизация может быть выключена локально, через определение #pragma optimize.

Определение cdata

Данное определение позволяет сохранить 16-ти битные данные в памяти программ

Примечание 1. Константы могут быть сохранены в память программ через модификатор const. Однако, cdata[] обычно используется для сохранения инструкций и данных по фиксированным адресам.

Примечание 2. Не проводится проверки корректности вставляемых данных и адресов. Однако, не возможно перезаписать программный код или другие cdata[] секции (за исключением битов конфигурации микроконтроллера). Данные добавляются в конец ассемблерного и hex файлов в порядке как они описаны в коде.

Синтаксис:

```
#pragma cdata[ADDRESS] = <VXS>, .., <VXS>
#pragma cdata[] = <VXS>, .., <VXS>
#pragma cdata.IDENTIFIER = <VXS>, .., <VXS>

ADDRESS: 16 bit word address
VXS : < VALUE | EXPRESSION | STRING>
VALUE: 0 .. 0xFFFF
EXPRESSION: C constant expr. (i.e. 0x1000+(3*1234))
STRING: "Valid C String\r\n\0\x24\x8\xe\xFF\xff\\\""

String translation: \xHH or \xH : hexadecimal number
      \0 => 0 \1 => 1 \2 => 2 \3 => 3 \4 => 4
      \5 => 5 \6 => 6 \7 => 7 \a => 7 \b => 8
      \t => 9 \n => 10 \f => 12 \v => 11 \r => 13
      \\ => the backslash character itself (0x5C)
```

```
\" => '"' (0x22)
\xHH or \xH : hexadecimal number
"\x1Conflict" is better written as "\x1" "Conflict"
```

Строки сохраняются как 8-ми битные ASCII символы. Младшие 8 бит каждого кода размещаются первыми. Строки выравниваются по адресам для каждого элемента <VXS>. Однако, выравнивание не выполняется когда записываются “abc” “def”

IDENTIFIER: любые неопределенные ранее идентификаторы. Конвертируется в макросы идентификаторы и приравниваются к адресам текущего cdata. Это позволяет автоматически находить адрес сохраненных элементов.

Пустое определение cdata может быть использовано для записи или чтения текущего cdata адреса
#prqagma cdata[ADDRESS] // установить текущий cdata адрес
#prqagma cdata.IDENTIFIER // взять текущий cdata адрес

Только cdata с корректным адресом учитывается при расчете общего объемы кода программ.

Использование cdata

1. Задание специальных алгоритмов запуска

```
#include "hexcodes.h"
#pragma cdata[0] = __NOP
#pragma resetVector 2 // goto main at byte address 2
```

2. Сохранение пакетов строк и данных

cdata определения должны быть оформлены в отдельном файле и подключаться в начале программы. Это позволяет идентификаторы для использования в программе и проверке.

```
#define CDATA_START 0x80
#pragma cdata[CDATA_START] // start of cdata block
#pragma cdata[] = 0xFFFF, 0x2000, 0x1000
#pragma cdata[] = 0x100, (10<<4) + 3456,\
10, 456, 10000
#define D8(l,h) (((l)&0xFF) + ((h)&0xFF)*256)
#define D32(x) x%0x10000, x/0x10000
#pragma cdata[] = D8(10,20), D32(10234543)
#pragma cdata.ID0 = 0x10, 200+3000
#pragma cdata.ID1 = “Hello world\0"
#pragma cdata.ID2 = “Another string\r\n" “merged”
#pragma cdata.ID_TABLE = ID0, ID1, ID2 // store addresses
#pragma cdata.CDATA_END // end of cdata block
..
#pragma origin CDATA_END // program code follow here
void write(uns16 strID);
..
write(ID1);
write(ID2);
```

Все cdata начинаются с адресов определенных в ручную. Определение должно быть следующим.

```
.. cdata definitions
```

```
.. C functions at addresses lower than CDATA_START
// #pragma origin CDATA_START // optional
#pragma origin CDATA_END
.. C functions at addresses higher than CDATA_END
```

#pragma origin CDATA_START не требуется, потому что перекрытие данных определяется автоматически. Однако, компилятор сообщит как много инструкций было пропущено для каждого определения. Cdata слова не учитываются при этом выводе.

Определение #pragma origin CDATA_END позволяет функциям быть сохраненными сразу за областью cdata. Это определение не требуется если все cdata собраны в конце кода.

Определения препроцессора могут быть использованы для проверки размера в процессе компиляции

```
#if CDATA_END - CDATA_START > 20
#error This is too much
#endif
```

7. ОТЛАДКА

Избавится от ошибок найденных компилятором достаточно просто, гораздо сложнее определить ошибки в работе компилируемой программы. Всегда старайтесь проверить полученный ассемблерный код после компилирования исходного Си кода. Использование компилятора не избавляет вас от необходимости знания ассемблера.

Ошибки при компиляции

Компилятор выводит сообщения об ошибках при их обнаружении. Сообщение об ошибке содержит 2 строки исходного кода и маркет указывающий на место где обнаружена ошибка. Вывод исходного кода и маркера может быть отключен параметром –e в командной строке. Так же может быть задано максимальное число выводимых ошибок. Можно выводить до 12 строк используя параметр –E12

Формат строки сообщения об ошибке

```
Error <filename> <line number>: <error message>
```

Некоторые ошибки фатальны и по этому компилятор останавливается немедленно. В другом случае процесс компиляции продолжается, но результирующие файлы не создаются.

Если существует синтаксическая ошибка в макросе, то может быть трудно определить причину ошибки. Возможно в этом случае может помочь анализ других ошибок.

Примечание: когда обнаружена ошибка, компилятор удаляет hex и ассемблерный файлы созданные при предыдущей успешной компиляции

Более подробная информация об ошибках и предупреждениях.

Компилятор отображает короткие описания в сообщениях об ошибках на экран и в файл *.occ, но не в файл *.err. Не все короткие описания могут быть отображены, когда разрешен вывод в файл сообщений об ошибках. Настройка выводимой информации может быть осуществлена с помощью параметров командной строки.

```
-ed : do not print error details (disable)
-ew : do not print warning details (disable)
-eL : list error and warning details at the end
```

Общие проблемы при компиляции

- not enough variable space (нет свободного места для переменных)
Решение: потребуется некоторая переработка программы с целью уменьшения объема требуемой памяти

- the compiler is unable to generate code (компилятор не может сгенерировать код)
Решение: Некоторые описания на языке С должны быть переписаны более просто

- too much code generated (создано слишком много кода)
Решение: Необходимо переписать код программы с целью уменьшения объема кода программы

- too deep call level (слишком большая вложенность вызываемых функций)
Решение: переписать код, раскрыв код вложенных функций

8. СОЗДАВАЕМЫЕ ФАЙЛЫ

Компилятор создает результирующие файлы, в том числе и hex файл, который может быть применен для программирования микроконтроллеров. Hex файл создается только в том случае если не было обнаружено ошибок. Компилятор может создавать и другие файлы:
- ассемблерный, файлы переменных, листинг, список функций, COD, список ошибок.

HEX файл

Стандартный Hex файл иммет формат INHX32. Формат может быть изменен с помощью параметра –f командной строки на форматы INHX8M, INHX8S и INHX32:

```
:BBaaaaTT112233...CC
BB - number of data words of 8 bits, max 16
aaaa - hexadecimal address (byte-address)
TT - type :
00 : normal objects
01 : end-of-file (:00000001FF)
11 - 8 bits data word
CC - checksum - the sum of all bytes is zero.
```

16-ти битный формат при INHX16 имеет формат:

```
:BBaaaaTT111122223333...CC
BB - number of data words of 16 bits, max 8
aaaa - hexadecimal address (of 16 bit words)
TT - type :
00 : normal objects
01 : end-of-file (:00000001FF)
1111 - 16 bits data word
CC - checksum - the sum of all bytes is zero.
```

Ассемблерный файл

Компилятор создает ассемблерный файл. Этот файл может быть использован как обычный ассемблерный код для работы микроконтроллера. Текст из исходного С файла переносится в ассемблерный. Это повышает его читаемость. Переменные по возможности имеют одинаковые имена. 16-тиричные директивы так же переносятся в ассемблерный файл. Это можно отключить при необходимости. Компилятор добавляет в именам переменных расширения что бы все переменные были уникальными.

Параметр командной строки –mA автоматически усечет сгенерированные метки в ассемблерном коде и лист файле.

Так же много параметров для настройки создаваемого ассемблерного файла. Необходимо помнить о разнице между –a и –A. Параметр –a говорит о том что надо создать ассемблерный файл. А параметр –A указывает в каком формате его создавать.

Основной формат параметров следующий –A[scHDftumiJN+N+N]

s: символьные аргументы заменить на числовые.
c: не выводить С код
H: только шестнадцатеричные числа
D: только десятичные числа
f: не отображать объектные директивы
t: не выводить символов табуляции

u: не выводить лишней информации в конце hex файла
m: выводить по одной строке исходного кода.
i: не выводить отступов, выровнять по левому краю
J: выводить исходный код после инструкций
R: детализировать макросы
N+N+N: Отступы между метками, мнемониками и аргументами, по умолчанию 8+6+10

Параметры чувствительны к регистру.
Пример:

```
Default :                    ; x++;
           m001  INCF  x
-AsDJ :    m001  INCF  10    ; x++;
-Ac   :    m001  INCF  x
-AJ6+8+11 : m001  INCF  x     ; x++;
-AiJ1+6+10: m001
                 INCF  x     ;x++;
-AiJs1+6+6: m001
                 INCF  0Ah   ;x++;
```

Файл переменных

Файл со списком всех переменных содержит информацию на объявленные в программе переменные. Переменные сортируются по адресу по умолчанию, но это может быть изменено. Для того что бы компилятор создал файл со списком переменных в командной строке должен быть задан параметр –V. Файл будет создан с именем <src>.var

Формат параметра –V[rnuD]. Дополнительные символы позволяют задать содержимое файла:

r: только переменные определенные в исходном коде
n: сортировать переменные по имени
u: не сортировать переменные
D: использовать десятичные числа

Файл со списком переменных содержит:

```
X     B     Address     Size  #AC   Name
X ->        L : local variable
            G : global variable
            P : assigned to certain address
            E : extern variable
            R : overlapping, directly assigned
            C : const variable
      B ->  - : unbanked RAM
            0 : bank 0
            1 : bank 1
.. etc.

Address ->  0x00A       : file address
            0x00C.0     : bit address (file + bit number)

Size -> size in bytes (0 for bit)

#AC -> 12: number of direct accesses to the variable
```

Пример:

```
X     B     Address     Size  #AC   Name
R     [-]   0x01B       1     : 10  : alfa
P     [-]   0x01B       1     : 12  : fixc
L     [-]   0x01D       1     : 1   : lok
L     [0]   0x022.0     0     : 6   : b1
G     [0]   0x022.1     0     : 16  : bx
G     [0]   0x025       1     : 23  : b
```

Если функция не вызывается (не используется), все ее параметры и локальные переменные размещаются по одному адресу. Например:

```
L     [-]   0x00F       1     : 16  <> pm_2_
```

Файл листинга

Компилятор так же может создать файл листинга. Для этого в командной строке используются параметры –L или –L[<col>,<lin>]. Максимальное число столбцов <col> и строк <lin> на одной странице могут быть заданы. По умолчанию это –L200,60. Содержимое файла листинга может быть задано с помощью параметра –A аналогично файлу ассемблера.

Файл структуры вызовов функций

Структура вызовов функций может быть сохранена в файле <src>.fcs. Это может помочь при анализе проблем с превышением уровня стека. Может быть использовано два формата при создании файла: первый, просто список всех функций, второй, рекурсивное отображение структуры вызова функций. Параметр –Q для двух форматов.

Простой формат:

```
      F:    function1   :#1
            func2       : #5
            delay       : #2
            func3       : #3
```

Это означает что:

1. func2, delay и func3 вызываются из функции function1
2. #1: function1 вызывается однократно
3. #3: func3 вызывается трижды (один раз из function1)

Структура вызовов раскрывается рекурсивно. Отступ показывает вложенность вызова функции в исходном коде. Истинный уровень вызовов отображается в начале строки. Истинный уровень может отличаться от уровня отображенного с помощью отступов, если вызов через CALL был заменен на GOTO. В этом случае в конце строки отображается специальный маркер. Уровень вложенности прерываний проверяется автоматически и отображается отдельно.

```
L0    main
L1          function1
L2                func2
L2                delay
L2                func3
L1          function1 ..
```

Символы обозначают:

- L1 : уровень стека 1 (максимум 16). Это истинный уровень стека. С учетом того что CALL мог быть заменен на GOTO.

- .. : только первый вызов раскрыт полностью, последующие вызовы имеют такуюже структуру и не раскрываются
- [CALL->GOTO] : CALL заменен на GOTO
- [T-GOTO] : CALL + RETURN заменены на GOTO
- [RECURSIVE] : рекурсивный вызов функций

Выходной файл препроцессора

Компилятор создает выходной файл препроцессора с именем <src>.cpr если указан параметр –B в командной строке. Повторные директивы препроцессора удаляются или сокращаются. Идентификаторы макросов заменяются на содержимое макросов. Этот файл может быть использован для анализа раскрытия макросов, например когда компилятор выдает ошибку при использовании макросов.

Формат параметра –B[pims] использует дополнительные символы для вывода дополнительной информации.

p: неполный препроцессор
i: без подколюченный файлов
m: модифицированные символы
s: модифицированные строки

Компилятор остановиться после создания этого файла, если параметра был с какими либо дополнительными символами.

9. ПРИМЕРЫ ПРИЛОЖЕНИЙ

Вычисляемое GOTO

Вычисляемое GOTO это компактный и элегантный путь реализовать выборку из многих. Это может быть использовано для сохранения таблиц с константами. Однако, модификатор типа “const” это более удачный путь, так как константы хранятся в памяти программ.

Предупреждение: При разработке вычисляемого GOTO пути не описанные в этом разделе могут вызвать ошибки. Сгенерированный ассемблерный файл должен быть внимательно изучен, так как после оптимизации и обновления битов выбора банков могут возникнуть ошибки.

Регистр PCLATH должен быть корректно обновлен перед считыванием PCL. Компилятор может выполнить все обновления и проверки автоматически. Изучите следующий пример кода:

```
char sub0(char i)
{
      skip(i); // jumps 'i' instructions forward
      #pragma return[] = "Hello world"
      #pragma return[] = 10 "more text" 0 1 2 3 0xFF

/*    This is a safe and position-independent method
      of coding return arrays or lookup constant
      tables. It works for all PICmicro devices. The
      compiler handles all checking and code
      generation issues. It is possible to use return
      arrays like above or any C statements. */

      return 110;
      return 0x2F;
}

char sub01(char W)
{
      skip(W); // using W saves one instruction
      #pragma return[] = "Simple, isn't it" 0
      // skip(W) is allowed for the first 256 addresses of each page
}
```

Встроенная функция skip() для вычисляемого GOTO

Функция skip() так же использует 16-ти битный параметр. Если используется 8-ми битный параметр, перенос автоматически сгенерируется (3 дополнительных инструкции), если таблица пересекает 256 словную адресную границу. Допустимы следующие параметры:

- GD : динамически выбирать формат skip (по умолчанию)
- GW : динамически выбирать формат skip, при long выдавать предупреждение
- GS : всегда использовать короткий формат, ошибка при пересечении границы
- GL : всегда использовать длинный формат.

Когда используется –GS параметр, компилятор генерирует сообщение об ошибке если таблица пересекает границу в 256 байт. Короткий формат позволяет создать более

компактный код, но потребует ручного размещения таблицы в исходном файле, если возникнет ошибка

Выравнивание при ORIGIN

Возможно применение #pragma origin для создания вычисляемого GOTO внутри функции при не пересечении 256 байтной границы. Однако, это может потребовать некоторых изменений при разработке программы. Как альтернатива, можно использовать #pragma alignLsbOrigin для автоматическое выравнивания младших значимых байт

Пример:
Функция содержит вычисляемое GOTO. После проверки созданного файла листинга было обнаружено 16 инструкций между началом функции и началом адреса назначения (смещение 0) сразу после расположения инструкции ADDWF PCL,0, которая применяется для вычисления GOTO. Последний адрес назначения (смещение 10) расположен через 10 инструкций после первого адреса. Для быстрого и компактного вычисления GOTO необходимо что бы первая и последняя инструкции находились в границах одной «байтовой сраницы». Это можно сделать с помощью следующего определения

```
#pragma alignLsbOrigin -16 to 255 – 10 – 16
```

Определения для выравнивания не критичны. Компилятор выдаст ошибку или предупреждение если вычисляемого GOTO будет расположено на границе, что говорит об ошибочном расположении. Более простой способ заключается в том, что бы выравнять младшие знаковые байты на одну величину (если размер кода не критичен)

```
#pragma alignLsbOrigin 0 // align on LSB = 0
#pragma alignLsbOrigin 0 to 190 // [-255 .. 255]
#pragma alignLsbOrigin -100 to 10
```

Область вычисляемого GOTO

Компилятор вводить GOTO регион при обнаружении функции skip(). В этом регионе оптимизация немного изменяется и проводится некоторая проверка адресов. Регион вычисляемого GOTO заканчивается при окончании функции.

Регион вычисляемого GOTO может быть так же задан через определение:

```
#pragma computedGoto 1 // start c-goto region
// useful if PCL is written directly
```

А так же завершен через определение:

```
#pragma computedGoto 0 // end of c-goto region
/* recommended if the function contains code
below the goto region, for instance when the
table consists of an array of goto
statements (examples follow later). */
```

Регион вычисляемого GOTO влияет на:

1. Оптимизацию
2. Регистр выбора банков
3. Границу 256 словных областей.

Пример:

```
void sub3(char s)
      {
            /* the next statements could also be written as
            a switch statement, but this solution is
            fastest and most compact. */
      if (s >= 3)
            goto Default;
            skip(s);
            goto Case0;
            goto Case1;
            goto LastCase;
      #pragma computedGoto 0 // end of c-goto region
            Case0:
            /* user statements */
            return;
            Case1:
            LastCase:
            /* user statements */
            return;
            Default:
            /* user statements */
            return;
      }

void sub4(char s)
{
      /* this solution can be used if very fast
            execution is important and a fixed number of
            instructions (2/4/8/..) is executed at each
            selection. Please note that extra statements
            have to be inserted to fill up empty space
            between each case. */
      if (s >= 10)
      goto END;
      s = rlnc(s); /* multiply by 2 */
      s = rlnc(s); /* multiply by 2 */
      skip(s);
      // execute 4 instructions at each selection
      Case0: nop(); nop(); nop(); return;
      Case1: nop(); nop(); nop(); return;
      Case2: nop(); nop(); nop(); return;
      Case3: nop(); nop(); nop(); return;
      Case4: nop(); nop(); nop(); return;
      Case5: nop(); nop(); nop(); goto END;
      Case6: nop(); nop(); nop(); goto END;
      Case7: nop(); nop(); nop(); goto END;
      Case8: nop(); nop(); nop(); goto END;
      Case9: nop(); nop(); nop(); goto END;
#pragma computedGoto 0 /* end of region */
      END:
      ; // More statements
}
```

Операция SWITCH

```
char select(char W)
{
      switch(W)
      {
            case 1: /* XORLW 1 */
            /* .. */
```

```
            break;
            case 2: /* XORLW 3 */
            break;
            case 3: /* XORLW 1 */
            case 4: /* XORLW 7 */
            return 4;
            case 5: /* XORLW 1 */
            return 5;
      }
      return 0; /* default */
}
```

Компилятор реализует последовательность XORLW <const>. Но при этом константы не будут теми что заданы в С коде. Однако, созданный код будет корректным. Если потребуется более компактный код, то его можно переписать через вычисляемое GOTO, это имеет смысл, если выбор имеет много вариантов (например, 1,2,3,4,5,6 и так далее)

ПРИЛОЖЕНИЕ 1

Регистры определенные в самом компиляторе и не требующие повторного определения в заголовочных файлах

```
char W, WREG;
char INDF0, FSR0;
char PCL, PCLATH;
char ALUSTA, T0STA, CPUSTA, INTSTA;
char INDF1, FSR1;
char TMR0L, TMR0H;
char TBLPTR, TBLPTRL, TBLPTRH;
char BSR, BSRL, BSRH;
bit Carry, DC, Zero_, Overflow;
```

ПРИЛОЖЕНИЕ 2

Ассемблерные инструкции

```
Assembly:   ALUSTA:     Operation:
ADDLW k     C,DC,Z,OV   W = k + W; Add literal and W
ADDWF f,d   C,DC,Z,OV   d = f + W; Add W and f
ADDWFC f,d  C,DC,Z,OV   d = f + W + C; Add with Carry
ANDLW k     Z           W = W & k; AND literal and W
ANDWF f,d   Z           d = f & W; AND W and f
BCF f,b     -           f.b = 0; Bit clear f
BSF f,b     -           f.b = 1; Bit set f
BTG f,b     -           f.b = !f.b; Bit toggle
BTFSC f,b   -           Bit test f, skip if clear
BTFSS f,b   -           Bit test f, skip if set
CALL k      -           Call subroutine
CLRF f,d    -           f = 0; Clear f (and WREG)
CLRWDT -    TO,PD       WDT = 0; Clear watchdog timer
COMF f,d    Z,N d        = f ^ 255; Complement f
CPFSEQ f    -           Skip if f=W
CPFSGT f    -           Skip if f>W
CPFSLT f    -           Skip if f<W
DAW f,d     C           Decimal adjust W, store in f (and WREG)
DCFSNZ f,d  -           Decrement f, skip if not zero
DECFSZ f,d  -           Decrement f, skip if zero
DECF f,d    C,DC,Z,OV   d = f - 1; Decrement f
GOTO k      -           Go to address
INCF f,d    C,DC,Z,OV   d = f + 1; Increment f
INCFSZ f,d  -           Increment f, skip if zero
INFSNZ f,d  -           Increment f, skip if not zero
IORLW k     Z           W = W | k; Inclusive OR literal
IORWF f,d   Z           d = f | W; Inclusive OR W and f
MOVFP f,p   -           Move f to p
MOVPF p,f   Z           Move p to f
MOVLB k     -           BSR (bit 0-3) = k
MOVLR k     -           BSR (bit 4-7) = k
MOVLW k     -           W = k; Move literal to W
MOVWF f     -           f = W; Move W to f
MULLW k     -           PRODH,PRODL = W * k; Multiply
MULWF f     -           PRODH,PRODL = W * f; Multiply
NEGF f,d    C,DC,Z,OV   Negate W, store in f (and WREG)
NOP -       -           No operation
RETLW k     -           Return, put literal in W
RETURN -    -           Return from subroutine
RETFIE -    -           Return from interrupt
RLCF f,d    C           Rotate left f through carry bit
RLNCF f,d               Rotate left f
```

```
RRCF f,d    C           Rotate right f through carry bit
RRNCF f,d               Rotate right f
SETF f,d    -           f = 0xFF; Store in f (and WREG)
SLEEP -     TO,PD       Go into standby mode, WDT = 0
SUBLW k     C,DC,Z,OV   W = k - W; Subtract W from literal
SUBWF f,d   C,DC,Z,OV   d = f - W; Subtract W from f
SUBFWB f,d  C,DC,Z,OV   d = W – f - ~C; Subtract with borrow
SUBWFB f,d  C,DC,Z,OV   d = f – W - ~C; Subtract with borrow
SWAPF f,d   -           Swap halves f
TABLRD t,i,f      -     Table read update
TABLWT t,i,f      -     Table write update
TLRD t,f    -           Table read
TLWT t,f    -           Table write
TSTFSZ f,a  -           Skip if f=0
XORLW k     Z           W = W ^ k; Exclusive OR literal
XORWF f,d   Z           d = f ^ W; Exclusive OR W and f
```

Примечание:

```
d = 1 : destination f
d = 0 : destination W
f : file register
Z : Zero bit : Z = 1 if result is 0
C : Carry bit
C = 1 : indicates carry on addition
C = 0 : indicates borrow on subtraction
DC : Digit Carry bit
DC = 1 : indicates carry on addition
DC = 0 : indicates borrow on subtraction
TO : Timeout bit
PD : Power down bit
```

ПРИЛОЖЕНИЕ 3

Время выполнения инструкций.

Большинство инструкций выполняются за 4 такта. За исключением инструкций которые модифицируют счетчик команд. Они выполняются за 8 тактов.

- CALL и GOTO
- инструкции пропускающие следующую инструкцию
- инструкции модифицирующие счетчик команд ADDWF PCL