CC5X

C Compiler for the
PICmicro Devices

Version 3.7

User's Manual

B Knudsen Data
Trondheim - Norway

This manual covers CC5X version 3.7 and related topics. New versions may contain changes without prior notice.

COMPILER BUG REPORTS:

The compiler has been carefully tested and debugged. It is, however, not possible to guarantee a 100 % error free product.

If the compiler generates application code bugs, it is almost always possible to rewrite the program slightly in order to avoid the bug. *#pragma optimize* can be used to avoid optimization bugs. Other *#pragma* statements are also useful.

Please report cases of bad generated code and other serious program errors.

1) Investigate and describe the problem. If possible, please provide a complete C example program that demonstrates the problem. A fragment from the generated assembly file is sometimes enough.
2) This service is intended for difficult compiler problems (not application problems).
3) Language: English
4) State the compiler version.
5) Send your report to support@bknd.com.

Document version: **L**

CONTENTS

1 INTRODUCTION 8

1.1 SUPPORTED DEVICES 9
1.2 INSTALLATION AND SYSTEM REQUIREMENTS 9
Support for long file names 9
User interface 9
1.3 MPLAB SUPPORT 10
1.4 THE SETCC UTILITY 10
1.5 SUMMARY OF DELIVERED FILES 10
1.6 SHORT PROGRAM EXAMPLE 12
1.7 DEFINING THE PICMICRO DEVICE 13
1.8 WHAT TO DO NEXT 14

2 VARIABLES 15

2.1 INFORMATION ON RAM ALLOCATION 15
2.2 DEFINING VARIABLES 16
Integer variables 16
Floating point 17
IEEE754 interoperability 17
Fixed point variables 18
Assigning variables to RAM addresses 20
Supported type modifiers 21
Local variables 22
Temporary variables 23
Arrays, structures and unions 23
Large arrays on the Enhanced 14 bit core 24
Bitfields 24
Typedef 24
2.3 USING RAM BANKS 25
The bank type modifier 25
RAM bank selection bits 26
Manual bank bit update regions 26
2.4 POINTERS 27
Pointer models 27
The 12 bit Core 28
The 14 bit core: the IRP bit 29
2.5 CONST DATA SUPPORT 30
Storing 14 bit data 31
Data of size 16 bits or more 31
Code pages 31
Locating Const Data 32
Merging data 32
Examples 32
Const data stored in dedicated functions 33

3 SYNTAX 34

3.1 STATEMENTS 34
if statement 34
while statement 34
for statement 34
do statement 35
switch statement 35
break statement 35

continue statement *36*
return statement *36*
goto statement *36*
3.2 ASSIGNMENT AND CONDITIONS 36
Special syntax examples *36*
Conditions *37*
Bit variables *37*
Multiplication, division and modulo *38*
Precedence of C operators *38*
Mixed variable sizes are allowed *39*
3.3 CONSTANTS 39
Constant expressions *39*
Enumeration *40*
3.4 FUNCTIONS 40
Function return values *40*
Parameters in function calls *40*
Internal functions *41*
3.5 TYPE CAST 41
3.6 ACCESSING PARTS OF A VARIABLE 43
3.7 C EXTENSIONS 43
3.8 PREDEFINED SYMBOLS 44
Extensions to the standard C keywords *44*
Standard C keywords used *44*
The sizeof operator *45*
Function offsetof(struct_type, struct_member) *45*
Automatically defined macros and symbols *45*
Macros __FILE__ and __LINE__ *46*
Macros __DATE__ and __TIME__ *46*
3.9 UPWARD COMPATIBILITY 46

4 PREPROCESSOR DIRECTIVES 47

#define *47*
Macro concatenation *47*
Macro stringification *47*
#include *48*
#undef *48*
#if *48*
#ifdef *49*
#ifndef *49*
#elif *49*
#else *49*
#endif *49*
#error *49*
#warning *50*
#message *50*
4.1 THE PRAGMA STATEMENT 50
#pragma alignLsbOrigin <a> [to <b>] *50*
#pragma asm2var 1 ..... *50*
#pragma assert [/] <type> <text field> *50*
*#pragma assume *<pointer> in rambank <n>* *50*
#pragma bit <name> @ <N.B or variable[.B]> *50*
#pragma cdata[ADDRESS] = <VXS>, .., <VXS> *51*
#pragma char <name> @ <constant or variable> *51*
#pragma chip [=] <device> *51*
#pragma codepage [=] <0,1,2,3, ..15> *51*
#pragma computedGoto [=] <0,1,2> *52*

#pragma config [/<regNr> <value>] [<id>] = <state> [, <id> = <state>] *52*
#pragma config_def [=] <value> *52*
#pragma config_reg [=] <address> *53*
#pragma config_reg2 [=] <address> *53*
#pragma data_area [=] <start_address> : <last_address> *53*
#pragma inlineMath <0,1> *53*
#pragma insertConst *53*
#pragma interruptSaveCheck <n,w,e> *53*
#pragma library <0/1> *53*
#pragma location [=] <0,1,2,3,.. 15, - > *54*
#pragma mainStack <minVarSize> @ <lowestStartAddr> *54*
#pragma minorStack <maxVarSize> @ <lowestStartAddr> *54*
#pragma optimize [=] [N:] <0,1> *54*
#pragma origin [=] <expression> *55*
#pragma packedCdataStrings <0,1> *55*
#pragma rambank [=] <0,1,2,3,..31, - > *55*
#pragma rambase [=] <n> *55*
#pragma ramdef <ra> : <rb> [MAPPING] *56*
#pragma resetVector <n> *56*
#pragma return[<n>] = <strings or constants> *56*
#pragma sharedAllocation *56*
#pragma stackLevels <n> *56*
#pragma unlockISR *56*
#pragma updateBank [entry | exit | default] [=] <0,1> *57*
#pragma update_FSR [=] <0,1> *57*
#pragma update_IRP [=] <0,1> *57*
#pragma update_PAGE [=] <0,1> *57*
#pragma update_RP [=] <0,1> *57*
#pragma user_ID_addr [=] <address> *58*
#pragma versionFile [<file>] *58*
#pragma wideConstData [<N> | p | r] *58*
4.2 PICMICRO CONFIGURATION 58

5 COMMAND LINE OPTIONS 60

5.1 OPTIONS IN A FILE 63
5.2 AUTOMATIC INCREMENTING VERSION NUMBER IN A FILE 63
5.3 ENVIRONMENT VARIABLES 64

6 PROGRAM CODE 65

6.1 PROGRAM CODE PAGES 65
Another way of locating functions *65*
The page type modifier *66*
Page selection bits *66*
6.2 SUBROUTINE CALL LEVEL CHECKING 66
Stack level checking when using interrupt *66*
Functions shared between independent call trees *67*
Recursive functions *67*
6.3 INTERRUPTS 67
Custom interrupt save and restore *69*
6.4 STARTUP AND TERMINATION CODE 69
Clearing ALL RAM locations *69*
6.5 LIBRARY SUPPORT 70
Math libraries *71*
Integer libraries *71*
Fixed point libraries *73*
Floating point libraries *73*

Floating point library functions *74*
Fast and compact inline operations *76*
Combining inline integer math and library calls *76*
Using prototypes and multiple code pages *77*
Fixed point example *78*
Floating point example *78*
How to save code *79*
6.6 INLINE ASSEMBLY 79
Direct coded instructions *84*
Generating single instructions using C statements *85*
6.7 OPTIMIZING THE CODE 86
Optimized syntax *86*
Peephole optimization *87*
6.8 LINKER SUPPORT 88
Using MPLINK or a single module *89*
Variables and pointers *89*
Enhanced core 14 and bank boundaries *90*
Local variables *91*
Header files *91*
Using RAM banks *91*
Bank bit updating *91*
Functions *91*
Using code pages *92*
Interrupts *92*
Call level checking *93*
Computed goto *93*
Recommendations when using MPLINK *93*
MPASM *94*
The MPLINK script file *95*
Example with 3 modules *97*
6.9 THE CDATA STATEMENT 100
Using the cdata statement *101*
Storing EEPROM data *102*

7 DEBUGGING 103

7.1 COMPILATION ERRORS 103
Error and warning details *104*
Some common compilation problems *104*
7.2 MPLAB DEBUGGING SUPPORT 104
ICD and ICD2 debugging *105*
7.3 ASSERT STATEMENTS 105
7.4 DEBUGGING IN ANOTHER ENVIRONMENT 106

8 FILES PRODUCED 108

8.1 HEX FILE 108
8.2 ASSEMBLY OUTPUT FILE 108
8.3 VARIABLE FILE 109
8.4 LIST FILE 110
8.5 FUNCTION CALL STRUCTURE 110
8.6 PREPROCESSOR OUTPUT FILE 111

9 APPLICATION NOTES 112

9.1 DELAYS 112
9.2 COMPUTED GOTO 113
Built in skip() function for computed goto *114*
Origin alignment *114*

Computed goto regions *114*
Examples *115*
9.3 THE SWITCH STATEMENT 117

APPENDIX 118

A1 USING INTERRUPTS 118
A2 PREDEFINED REGISTER NAMES 119
A3 ASSEMBLY INSTRUCTIONS 120
Additional for the 14 bit core *120*
Additional for the Enhanced 14 bit core *120*
Additional for the Enhanced 12 bit core *121*
Instruction execution time *121*

1 INTRODUCTION

Welcome to the CC5X C compiler for the Microchip PICmicro family of microcontrollers. The CC5X compiler enables programming using a subset of the C language. Assembly is no longer required. The reason for moving to C is clear. Assembly language is generally hard to read and errors are easily produced.

C enables the following advantages compared to assembly:

- Source code standardization
- Faster program development
- Improved source code readability
- Easier documentation
- Simplified maintenance
- Portable code

The CC5X compiler was designed to generate tight and optimized code. The optimizer automatically squeezes the code to a minimum. It is possible to write code that compiles into single instructions, but with C syntax. This means that the C source code can be optimized by rewriting inefficient expressions.

The design priority was not to provide full ANSI C support, but to enable the best possible usage of the limited code and RAM resources. If the compiler generated less optimal code, this would force assembly to be used for parts of the code.

CC5X features

- Local and global variables of 8, 16, 24 and 32 bits, plus bit variables
- Efficient reuse of local variable space
- Generates tight and optimized code
- Produces binary, assembly, list, COD, error, function outline and variable files
- Automatic updating of the page selection bits
- Automatic updating of the bank selection bits
- Enhanced and compact support of bit operations, including bit functions
- Floating and fixed point math up to 32 bits
- Math libraries including functions like sin(), log(), exp(), sqrt(), etc.
- Supports standard C constant data and strings in program memory (const)
- Automatic storing of compressed 2*7 bit data in each code word if possible
- Pointer models of 8 and 16 bits, mixed sizes in same application allowed
- RAM and/or ROM pointers
- The size of single pointers can be automatically chosen by the compiler
- Linker support (MPLINK), interfaces with assembly (MPASM) modules
- Extended call level by using GOTO instead of CALL when possible
- Inserts links to "hidden" subroutines
- Access to all assembly instructions through corresponding C statements
- Inline assembly
- Lookup tables: #pragma return[] = "Hello world"
- Integrated interrupt support
- Device configuration information in source code

Size (in bits) of the variables supported by the different compiler editions:

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

1.1 Supported devices

12 bit core (PIC16F5X, PIC10, PIC12, etc.):
- up to 2048 words of code in 1 - 4 code pages
- up to 73 byte RAM in 1 - 4 banks (132 byte / 8 banks for enhanced 12 bit core)

14 bit core (PIC12, PIC14, PIC16):
- up to 8192 words of code in 1 - 4 code pages
- up to 512 byte RAM in 1 - 4 banks

14 bit enhanced core (PIC16F1xxx):
- up to 32768 words of code in 1 - 16 code pages
- up to 8k (5k) RAM in 1 - 64 banks

1.2 Installation and System Requirements

The CC5X compiler uses 32 bit processing (console application) and runs on PC compatible machines using Microsoft Windows.

Installing CC5X is normally done by running the installation program for the latest version. Multiple versions can be installed.

CC5X is now ready to compile C files. Header and C source files have to be created and edited by a separate editor (not included), for instance in the MPLAB suite.

The UTF-8 representation of the Byte Order Mark is the byte sequence 0xEF,0xBB,0xBF. This sequence is allowed in the start of a source file.

The same installation program can be used to un-install the compiler. Alternatively the CC5X files can be deleted without any un-installation procedure.

Support for long file names

CC5X (WIN32 editions) supports long file names. It is also possible to use spaces in file names and include directory names. Equivalent include directory option formats:

```
-I"C:\Program Files\cc5x"
-IC:\progra~1\cc5x
```

Equivalent include file formats:

```
#include "C:\Program Files\cc5x\C file"
#include "C:\progra~1\cc5x\Cfile~1"
```

The alternative to long names is the truncated short format. The truncated form is decided by the file system. The best guess consists of the 6 first characters of the long name plus ~1. The last number may be different (~2) if the first 6 characters are equal to another name in the same directory.

User interface

The CC5X compiler is a command-line program that can be run in a console window in the Windows environment. It requires a list of command line options to compile a C source file and generate the required files.

Starting CC5X from Windows can be done by clicking on the executable file. The list of compiler command line options is then written to a console window. The normal way of using CC5X is to use it as a tool from an integrate environment like MPLAB X.

Compiling a program requires a file name and command line options:

```
cc5x sample1.c <enter>
```

1.3 MPLAB Support

CC5X can be selected as a tool in MPLAB X and the older MPLAB, which offers an integrated environment including editor and tool support (compilers, assemblers, simulators, emulators, device programmers). Compilation errors are easily handled. MPLAB supports point-and-click to go directly to the source line that needs correction. CC8E supports COFF and COD debugging file format. Please refer to supplied file '**install.txt**' for further information.

1.4 The SETCC Utility

SETCC is a small utility application for the CC5X/CC8E compilers. Note that the SETCC utility is only available for licensed editions. SETCC is useful for:

a) Generating device specific header files
b) Setting config symbols for a device
c) Compiling projects

Reasons for generating header files using SETCC:
- predefined header files may not be ready yet for new devices
- the number of register symbols can be selected
- the bit name format can be selected
- the register and bit names are generated according to the INC and LKR files supplied by Microchip as found in mpasmx assembler in MPLAB X.

Header file options:
- include alternative register names (all register names in INC file)
- include bit names
- include alternative bit names (as defined in INC file)
- include bit names for all register names
- convert bit names on format NOT_NNNNN to NNNNN_

a) bit name format: REGISTERNAME_BITNAME
b) bit name format: BITNAME - limited set only
c) bit name format: combined BITNAME / REGISTERNAME_BITNAME

The device header files supplied with the compiler can alternatively be used instead of generating new header files for each project. However, the supplied header files contains less bit symbols compared to header files generated by SETCC.

Please refer to file '**setcc.txt**' for further information.

1.5 Summary of Delivered Files

```
CC5X.EXE      : compiler

SETCC.EXE  (1): header file and config utility tool
SETCC.TXT     : information on utility tool

INSTALL.TXT   : installation guide
MPLAB-O.TXT   : integration with old MPLAB (version 5 - 8.92)
INLINE.TXT    : information on inline assembly syntax
CHIP.TXT      : how to make new chip definitions
CDATA.TXT     : info on the #pragma cdata statement
```

```
CONST.TXT     : standard C strings and constant data
CONFIG.TXT    : the chip configuration bits
STARTUP.TXT   : special startup sequences
LINKER.TXT    : using MPLINK to link several modules (C or asm)
C-GOTO.TXT    : application notes on computed goto
OPTIONS.TXT   : compiler command line options
ERRATA.TXT    : silicon errata issues
MATH.TXT      : math library support
NEWS.TXT      : recent added features
README.TXT    : this file

DEFMPX16.H    : compiler definitions for the MPLAB X GUI
INT16CXX.H    : interrupt header file
INLINE.H      : C macros for emulating inline instructions
HEXCODES.H    : direct coded instructions

CC5X.MTC      : MPLAB tool configuration file
TLCC5X.INI    : MPLAB tool configuration file

OP.INC        : command line options in a file
RELOC.INC     : options for object modules (MPLINK)

SAMPLE1.C     : minimal program example
SAMPLE2.C     : recommended program structure and syntax samples
SAMPLE3.C     : data stored in program memory and pointers
DEMO-ENH.C    : example syntax for new Enhanced 14 bit Core
DEMO-E12.C    : example syntax for the Enhanced 12 bit Core
IICBUS.C      : IIC-bus interface
IIC-COM.C     : IIC-bus communication
SERIAL.C      : serial communication (RS232, RS485)
STATE.C       : state machines
DELAY.C       : implementing delays
INT16XX.C     : simple interrupt example
DIV16_8.C     : fast division routine
SCALING.C     : compact and fast 16 bit math scaling routine

MATH16.H      : 8-16 bit math library
MATH16M.H     : 8-16 bit multiply, speed

MATH24.H   (1): 8-24 bit math library
MATH24M.H  (1): 8-24 bit multiply, speed
MATH32.H   (1): 8-32 bit math library
MATH32M.H  (1): 8-32 bit multiply, speed

MATH16X.H  (1): 16 bit fixed point library
MATH24X.H  (1): 24 bit fixed point library
MATH32X.H  (1): 32 bit fixed point library

MATH16F.H  (1): 16 bit floating point library
MATH24F.H     : 24 bit floating point library
MATH32F.H  (1): 32 bit floating point library

MATH24LB.H    : 24 bit floating point functions
                (log,sqrt,cos,..)
MATH32LB.H (1): 32 bit floating point functions
                (log,sqrt,cos,..)
```

```
mventil.c  (2): multitasking example
msg.h      (2): multitasking message library
binsem.h   (2): multitasking binary semaphore library
semaphore.h (2): multitasking semaphore library
event.h    (2): multitasking event library
delay.h    (2): multitasking delay and timing library

12C508.H ..   : header files for specific chip support
```

(1) Not available in the FREE edition
(2) Only available in the EXTENDED edition

1.6 Short Program Example

```
/* global variables */
char a;
bit b1, b2;

/* assign names to port pins */
bit in  @ PORTA.0;
bit out @ PORTA.1;

void sub( void)
{
    char i;      /* a local variable */

    /* generate 20 pulses */
    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
    // }

 /* first decide the initial output level
    on the output port pins, and then
    define the input/output configuration.
    This avoids spikes at the output pins. */

    PORTA =      0b.0010;  /* out = 1 */
    TRISA = 0b.1111.0001;  /* xxxx 0001 */

    a = 9;  /* value assigned to global variable */

    do  {
        if (in == 0)  /* stop if 'in' is low */
             break;
```

```
        sub();
    } while ( -- a > 0);  /* 9 iterations */

    // if (some condition)
    //    goto WARM_RESET;

    /* main is terminated by a SLEEP instruction */
}
```

1.7 Defining the PICmicro Device

CC5X offers 3 ways to select the PICmicro device in an application:

1) By a command line option. MPLAB will generate this option automatically.

```
-p16F883
```

2) By a pragma statement in the source code. Note that the –p command line option will override the selection done by #pragma chip. This pragma should not be used in combination with MPLAB.

```
#pragma chip PIC16F883
```

3) By using include to directly select a header file. This is not recommended because there will be an error if the command line option is also used.

```
#include "16f883.h"
```

NOTE 1: When using a pragma statement or include file, remember to use it in the beginning of the C program so that it is compiled first. However, some preprocessor statements like *#define and #if* may precede the *#include/#pragma* statement.

NOTE 2: When using the command line option or the pragma statement, CC5X will use the internal definitions for some devices. If the device is not known internally, **automatic** include of a header file is started. The internal known devices are: 16C54,55,56,57,58, 61,64,65, 71,73,74, 84, 620,621,622.

NOTE 3: If the header file does not reside in the default project folder, then the path name is required. This can be supplied by a command line option as an include folder/directory (-I<path>).

NOTE 4: Debugging means that the debugger may use certain device resources. These resources should not be used by the application during debugging. The debugger and device documentation should be consulted. Reservations for **some devices** are supplied in the device header files. The device header file should also be inspected. Activating the reservations is done by defining a symbol before the header file is compiled:

a) By a command line option:

```
-DICD_DEBUG   or   -DICD2_DEBUG
```

b) By using #define in combination with #pragma chip or #include:

```
#define ICD_DEBUG   // or ICD2_DEBUG
..
#pragma chip PIC16F877  // or #include "16F877.H"
```

1.8 What to do next

It is important to know the PICmicro family and the tools well. The easiest way to start is to read the available documentation and experiment with the examples. Then move on to a simple project. Some suggestions:

- study the supplied program samples
- compile code fragments and check out what the compiler accepts
- study the optional assembly file produced by the compiler

Note that using more than one ram bank or code page requires pragma instructions.

Typical steps when developing programs are as follows:

- describe the system; make requirements
- suggest solutions that satisfy these requirements
- write detailed code in the C language
- compile the program using the CC5X compiler
- test the program on a prototype or a simulator

Writing programs for the PICmicro microcontroller family requires careful planning. Program and RAM space are limited, and the key question is often: *Will the application code fit into the selected device?*

File '**readme.txt**' contains information on how to write code that can be compiled by CC5X.

2 VARIABLES

The compiler prints information on the screen when compiling. Most important are error messages, and how much RAM and PROGRAM space the program requires. The compiler output information is also written to file *.occ. Example:

```
delay.c:
 Chip = 16C74
 RAM: 00h : -------- -------- -------- --------
 RAM: 20h : ==.***** ******** ******** ********
 RAM: 40h : ******** ******** ******** ********
 RAM: 60h : ******** ******** ******** ********
 RAM: 80h : -------- -------- -------- --------
 RAM: A0h : ******** ******** ******** ********
 RAM: C0h : ******** ******** ******** ********
 RAM: E0h : ******** ******** ******** ********
 Optimizing - removed 11 instructions (-14 %)
 File 'delay.asm'
 Codepage 0 has   68 word(s) :   3 %
 Codepage 1 has    0 word(s) :   0 %
 File 'delay.hex'
Total of 68 instructions (1 %)
```

2.1 Information on RAM allocation

Priority when allocating variables:

1. Variables permanently assigned to a location
2. Local variables allocated by the compiler
3. Global variables allocated by the compiler (up to 80 bytes)
4. Global variables greater than 80 bytes (Enhanced 14 bit core only)

The compiler prints information on RAM allocation. This map is useful to check out which RAM locations are still free. The map for the 16C57 chip may look like this:

```
Mapped RAM: 00h : -------- .7.-****
Bank 0 RAM: 10h : ====4==* ********
Bank 1 RAM: 30h : ..6***** ********
Bank 2 RAM: 50h : ******** ********
Bank 3 RAM: 70h : -7****** ********
```

Symbols:

```
 *  :  free location
 -  :  predefined or pragma variable
 =  :  local variable(s)
 .  :  global variable
 7  :  7 free bits in this location
```

Detailed information on memory allocation is written to file <src>.var when using the –V command line option.

2.2 Defining Variables

CC5X supports integer, fixed and floating point variables. The variable sizes are 1, 8, 16, 24 and 32 bit. The default *int* size is 8 bits, and *long* is 16 bits. Char variables are unsigned by default and thus range from 0 to 255. Note that 24 and 32 bit variables are not supported by all CC5X editions.

Math libraries may have to be included for math operations (Chapter *6.5 Library Support* on page 70).

CC5X uses LOW ORDER FIRST (or little-endian) on variables. This means that the least significant byte of a variable is assigned to the lowest address. All variables are allocated from low RAM addresses and upwards. Each RAM location can contain 8 bit variables. Address regions used for special purpose registers are not available for normal allocation. An error message is produced when there is no space left.

Special purpose registers are either predefined or defined in chip-specific header files. This applies to W, INDF, TMR0, PCL, STATUS, FSR, Carry, PD, TO, etc.

Integer variables

```
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, semaphore;

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
```

The bitfield syntax can also be used:

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

The value range of the variables are:

```
 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
```

Floating point

The compiler supports 16, 24 and 32 bit floating point. The supported 32 bit floating point format can be converted to and from the IEEE754 format by 3 instructions (macro in math32f.h).

Supported floating point types:

```
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
```

Note that 16 bit floating point is intended for special use where accuracy is less important. More details on the floating point formats are found in '**math.txt**'. Information on floating point libraries is found in Chapter *6.5 Library Support* on page 70.

Floating point exception flags

The floating point flags are accessible in the application program. At program startup the flags should be initialized:

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

Also, after an exception is detected and handled in the application, the exception bit should be cleared so that new exceptions can be detected. Exceptions can be ignored if this is most convenient. New operations are not affected by old exceptions. This also enables delayed handling of exceptions. Only the application program can clear exception flags.

```
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
```

IEEE754 interoperability

The floating point format used is not equivalent to the IEEE754 standard, but the difference is very small. The reason for using a different format is code efficiency. IEEE compatibility is needed when floating point values are exchanged with the outside world. It may also happen that inspecting variables during debugging requires the IEEE754 format on some emulators/debuggers. Macros for converting to and from IEEE754 are available:

```
math32f.h:
// 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.)
```

Fixed point variables

Fixed point can be used instead of floating point, mainly to save program space. Fixed point math uses formats where the decimal point is permanently set at byte boundaries. For example, fixed8_8 uses one byte for the integer part and one byte for the decimal part. Fixed point operations map to integer operations except for multiplication and division, which are supported by library functions. Information on fixed point libraries is found in Chapter *6.5 Library Support* on page 70.

```
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
```

Convention: fixed<S><I>_<D> :

```
<S> : 'U' : unsigned
<none>:  signed
<I> : number of integer bits
<D> : number of decimal bits
```

Thus, fixed16_8 uses 16 bits for the integer part plus 8 bits for the decimal, for a total of 24 bits. The resolution for fixed16_8 is 1/256=0.0039, which is the lowest possible increment. This is equivalent to 2 decimal digits (actually 2.4 decimal digits).

Built in fixed point types:

```
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

(additional types with decimals only; no integer part)
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
```

To sum up:
1. All types ending on _8 have 2 correct digits after the decimal point
2. All types ending on _16 have 4 correct digits after the decimal point
3. All types ending on _24 have 7 correct digits after the decimal point
4. All types ending on _32 have 9 correct digits after the decimal point

Fixed point constants

The 32 bit floating point format is used during compilation and calculation.

```
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;
```

Constant rounding error example:

```
Constant: 0.036
Variable type: fixed16_8 (1 byte for decimals)

Error calculation: 0.036*256=9.216. The byte values assigned to the
variable are simply 0,0,9. The error is (9/256-0.036)/0.036 = -0.023.
The compiler prints this normalized error as a warning.
```

Type conversion

The fixed point types are handled as subtypes of float. Type casts are therefore infrequently required.

Fixed point interoperability

It is recommended to stick to one fixed point format in a program. The main problem when using mixed types is the enormous number of combinations which makes library support a challenge. However, many mixed operations are allowed when CC5X can map the types to the built in integer code generator:

```
fixed8_16 a, b;
fixed_16 c;
a = b + c;      // OK, code is generated directly
a = b * 10.22;  // OK: library function is supplied
```

```
a = b * c; // a new user library function is required!

// a type cast can select an existing library function:
a = b * (fixed8_16)c;
```

Assigning variables to RAM addresses

All variables, including structures and arrays, can be assigned to fixed address locations. This is useful for assigning names to port pins. It is also possible to define overlapping variables (similar to union). Variables can overlap parts of another variable, table or structure. Multiple levels of overlapping are allowed. The syntax is:

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

Examples:

```
char th @ 0x25;
//bit th1 @ 0x25.1;  // overlap warning
bit th1 @ th.1;      // no warning

char tty;
bit b0;
char io @ tty;
bit bx0 @ b0;
bit bx2b @ tty.7;
//char tui @ b0;   // size exceeded
//long r @ tty;   // size exceeded

char tab[5];
long tr @ tab;
struct  {
    long tiM;
    long uu;
} ham @ tab;

char aa @ ttb[2];   // char ttb[10];
bit  ab @ aa.7;     // a second level of overlapping
bit  bb @ ttb[1].1;

size2 char *cc @ da.a;  // 'da' is a struct
char dd[3] @ da.sloi[1].pi.ncup;
uns16 ee @ fx.mid16;  // float32 fx;
TypeX ii @ tab;   // TypeX is a typedef struct
```

An expression can define the address of a variable. This makes it easier to move a collection of variables.

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

Pragma statements can also be used (limited to bit and char types):

```
#pragma char port @ PORTC
#pragma char varX @ 0x23
#pragma bit IOpin @ PORTA.1
#pragma bit ready @ 0x20.2
#pragma bit ready @ PA2
```

If the compiler detects double assignments to the same RAM location, this will cause a warning to be printed. The warning can be avoided if the second assignment uses the variable name from the first assignment instead of the address (*#pragma char var2 @ var1*).

An alternative is to use the #define statement:

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

The *shadowDef* type modifier allows local and global variables and function parameters to be assigned to specific addresses without affecting normal variable allocation. The compiler will ignore the presence of these variables when allocating global and local variable space.

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

The above definition allows location 0x70 to be inspected and modified through variable 'gx70'.

Function parameters can be assigned to addresses. No other variables will be assigned by the compiler to these locations. Such manual allocation can be useful when reusing RAM locations manually.

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

This syntax is also possible on function prototypes.

Parameter transfer can be omitted for functions sharing overlapping parameters. This also applies to bit parameters:

```
bit sharedBitPar;
bit func2( bit par @ sharedBitPar ) { /*..*/ return Carry; }
bit func1( bit par @ sharedBitPar ) { /*..*/ return func2( par ); }
```

Supported type modifiers

```
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 CC5X uses the
address to automatically decide that most of the special purpose
registers are volatile */

page0 void fx(void);  // fx() resides in codepage 0
// page0,page1,..,page15

bank0 char a;  // variable 'a' resides in RAM bank 0
// bank0,bank1,..,bank31
```

```
// shrBank : unbanked locations, if available

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

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

Local variables

Local variables are supported. The compiler performs a safe compression by checking the scope of the variables and reusing the locations when possible. The limited RAM space is therefore used efficiently. This feature is very useful, because deciding which variables can safely overlap is time consuming, especially during program redesign. Function parameters are located together with local variables.

Variables should be defined in the innermost block, because this allows best reuse of RAM locations. It is also possible to add inner blocks just to reduce the scope of the variables as shown in the following example:

```
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
    }
}
```

Local variables may have the same name. However, the compiler adds an extension to produce a unique name in the assembly, list and COD files. When a function is not called (defined but not in use), then all parameters and local variables are truncated to the same (unused) location.

Local variables will reside in a single block not crossing any bank boundaries. This is a requirement because of the overlapping/reuse performed within the local block allocated.

Using several stacks

The stack for local variables, parameters and temporary variables is normally allocated separately in each bank and the shared RAM area. The bank is normally defined the same way as global variables through #pragma rambank or bank type modifiers. This makes it possible to split the stack into several independent stacks. Using a single stack is normally recommended, but sometimes this is not possible when the stack size is too large.

The following pragma will define a single main stack. The main stack is not an additional stack, but tells the compiler where the main stack is located (which bank).

```
#pragma mainStack 3 @ 0x20  // set lower main stack address
```

Using this pragma means that local variables, parameters and temporary variables of size 3 bytes and larger (including tables and structures) will be stored in a single stack allocated no lower than address 0x20. Smaller variables and variables with a bank modifier will be stored according to the default/other rules. Using size 0 means all variables including bit variables.

Note that the bank defined by #pragma rambank is ignored for variables stored in the main stack. Addresses ranging from 0x20 to 0x6F/0x7F are equivalent to the bank0 type modifier.

In some cases it will be efficient to use shared RAM or a specific bank for local variables up to a certain size. This is possible by using the following pragma:

```
#pragma minorStack 2 @ 0x70
```

In this case, local variables, parameters and temporary variables up to 2 bytes will be put in shared RAM from address 0x70 and upward. Larger variables and variables with a bank modifier will be stored according to the default/other rules. Using size 0 means bit variables only. This pragma can be used in combination with the main stack. The variable size defined by the minor stack has priority over the main stack.

The most efficient RAM usage is to use a single stack. Separation into different stacks increases total RAM usage, and should be avoided if possible.

Temporary variables

Operations like multiplication, division, modulo division and shifts often require temporary variables. However, the compiler needs NO PERMANENT SPACE for temporary variables.

The temporary variables are allocated the same way as local variables, but with a narrow scope. This means that the RAM locations can be reused in other parts of the program. This is an efficient strategy and often no extra space is required in application programs.

Arrays, structures and unions

One dimensional arrays are implemented. Note that indexed arithmetic is limited to 8 bit. Assignment is allowed for 8, 16, 24 and 32 bit.

```
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;
```

Normal C structures can be defined, as can nested types. Unions are allowed.

```
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;
```

The equivalent of a (small) multidimensional array can be constructed by using a structure. However, only one index can be a variable.

```
struct  {
   char e[4];
   char i;
} multi[5];

multi[x].e[3] = 4;
multi[2].e[i+1] += temp;
```

Large arrays on the Enhanced 14 bit core

The enhanced 14 bit core allows arrays and data structures to cross bank boundaries. This is possible because separate RAM banks are mapped into a linear data space starting at address 0x2000.

Arrays greater than 80 bytes are allocated to multiple banks by the compiler. Such a large array does not belong to a specific bank. The compiler will automatically handle the required mapping. Data items up to 80 bytes are allocated first. Then the large data items are allocated. A data item up to 80 bytes can only cross a bank boundary if it is assigned a specific address (type variable @ address;). Large data items can also be assigned to specific addresses.

Bitfields

Bitfields in structures are allowed. The size has to be 1, 8, 16, 24 or 32 bits.

```
struct bitfield {
    unsigned a : 1;
    bit      c;
    unsigned d : 32;
    char     aa;
} zz;
```

The CC5X compiler also allows the bitfield syntax to be used outside structures as a general way of defining variable size:

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

Typedef

Typedef allows defining new type identifiers consisting of structures or other data types:

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

2.3 Using RAM Banks

Using more than one RAM bank is done by setting the active rambank:

```
/* variables preceding the first rambank statement are placed in
mapped RAM or bank 0. 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 */
```

The compiler automatically finds the first free location in the selected bank.

NOTE: Local variables and function parameters also have to be located. It may be necessary to use #pragma rambank between some of the functions and even INSIDE a function. The recommended strategy is to locate local variables and function parameters in *mapped* RAM or bank 0. Mapped/unbanked RAM is selected by:

```
#pragma rambank –
```

The bank type modifier

It is also possible to use the bank type modifier to select the RAM bank.

```
bank0..bank31, shrBank : can replace #pragma rambank
// shrBank is the mapped/unbanked locations, if available

bank1 char tx[3];  // tx[] is located in bank 1
```

The bank type modifier defines the RAM bank to locate the variable. It can locate global variables, function parameters and local variables. The bank type modifier applies to the variable itself, but not to the data accessed. This difference is important for pointers.

NOTE 1: The bank type modifier has higher priority than #pragma rambank.

NOTE 2: Using 'extern' makes it possible to state the variable definition several times. However, the first definition defines the rambank, and later definitions must use the same bank.

NOTE 3: When defining a function prototype, this will normally not locate the function parameters. However, when adding a bank type modifier to a function parameter in a prototype, this will define the bank to be used for this variable.

If variables are located in non-existing RAM banks for a device, these variables are mapped into existing RAM banks (bank 0). This applies to the bank type modifiers and the #pragma rambank statement.

Using RAM banks requires some planning. The optimal placement requires the least code to update the bank selection bits. Some advice when locating variables:

1. Try to locate variables which are close related to each other in the same bank.
2. Try to locate all variables accessed in the same function in the same bank.

3. Switching between bank 0 and 3, or bank 1 and 2 require more instructions than the other combinations. Note that this does not apply to the Enhanced 14 bit core.
4. Use as few banks as possible. Fill bank 0 first, then bank 1, etc.
5. Remember that local variables and function parameters also may require updating of the bank selection bits.

RAM bank selection bits

RAM and special purpose registers can be located in up to 4 banks. The 12 bit core uses bit 5 and 6 in FSR to select the right bank. In the 14 bit core, RP0 and RP1 in the STATUS register are used for this purpose. The enhanced 14 bit core uses the BSR register.

The bank selection bits are automatically checked and updated by the compiler, and attempts to set or clear these bits in the source code are removed by the compiler. This feature can be switched off, which means that correct updating has to be done in the source code.

The compiler uses global optimizing techniques to minimize the extra code needed to update the bank selection bits. Removing all unnecessary updating is difficult. However, there should be few redundant instructions.

The compiler inserts the following instructions:

```
BCF/BSF 04h,FSR_5    // 12 bit core
BCF/BSF 04h,FSR_6    // 12 bit core
CLRF FSR             // 12 bit core

BCF/BSF 03h,RP0      // 14 bit core
BCF/BSF 03h,RP1      // 14 bit core

MOVLB   k            // 14 bit enhanced core
```

NOTE: The compiler REMOVES all bank updating done by the user. However, it is possible to switch to manual updating with the *-b* command line option, or locally by a pragma statement.

Manual bank bit update regions

The automatic updating can be switched off locally. This is done by pragma statements:

```
#pragma update_FSR 0   /* OFF, 12 bit core only */
#pragma update_FSR 1   /* ON,  12 bit core only */

#pragma update_RP 0    /* OFF, 14 bit core (also enhanced core) */
#pragma update_RP 1    /* ON,  14 bit core (also enhanced core) */

#pragma updateBank 0   /* OFF, all cores */
#pragma updateBank 1   /* ON,  all cores */
```

These statements can be inserted anywhere, but they should surround the smallest possible region. Please check the generated assembly code to ensure that the desired result is achieved. Another use of #pragma updateBank is to instruct the bank update algorithm to do certain selections. Refer to Section *#pragma updateBank* on page 57 for more details.

NOTE: The safest coding is to not assume any specific contents of the bank selection bits when a local update region is started. The compiler uses complex rules to update the bank selection bits outside the local regions. Also, all updating inside a local update region is traced to enable optimal updating when the local update region ends.

2.4 Pointers

Single level pointers are implemented. Note that pointer arithmetic is limited to 8 bit. Assignment is allowed for 8, 16, 24 and 32 bit.

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

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

Pointer models

Using 8 bit pointers when possible saves both code and RAM space. CC5X allows the size of all single pointers to be decided automatically. However, pointers in structures and arrays have to be decided in advance, by using the memory model command line options or a size type modifier. Note that the operator 'sizeof(pointer)' will lock the size according to the chosen default model. Using sizeof(pointer) is normally not required and should be avoided.

Default pointer sizes are used only when the pointer size is not chosen dynamically. The priority when deciding the pointer size is:
1) Pointer size type modifiers
2) Automatically chosen pointer size (single pointers)
3) Pointer size chosen according to the default model

Command line options:
-mc1 : default 'const' pointer size is 1 byte (8 bits)
-mc2 : default 'const' pointer size is 2 bytes (16 bits)
-mr1 : default RAM pointer size is 1 byte
-mr2 : default RAM pointer size is 2 bytes
-mm1 : default pointer size is 1 byte (all pointer types)
-mm2 : default pointer size is 2 bytes (all pointer types)

Pointer size type modifiers:
- size1: pointer size is 1 byte (8 bits)
- size2: pointer size is 2 bytes (16 bits)

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

Certain pointer operation will generate warnings. The warnings can be removed by adding a proper type cast. The first warning can be disabled by command line option -wx. The other two warnings be disabled by command line option -wz.

```
// Suspicious pointer conversion    - different sign used
// Incompatible pointer conversion  - different size/type used
// Nonportable pointer conversion   - not a pointer or address
```

The supported pointer types are:
a) 8 bit pointer to RAM. The compiler will automatically update the MSB bits (FSR0H or IRP) if required (when RAM space exceeds 256 bytes).
b) 16 bit pointer to RAM. This format is required only when the same pointer has to access locations in different 256 byte RAM segments.
c) 8 bit pointer to program memory. This pointer can access up to 256 bytes of data.
d) 16 bit pointer to program memory. This pointer can access more than 256 bytes of data.
e) 8 bit pointer to RAM or program memory. This pointer can access up to 128 bytes of data and 128 bytes RAM. Bit 7 is used to detect RAM or program memory access. The compiler will only choose this format if all RAM addresses loaded to the pointer is in the same bank (14 bit core).

f) 16 bit pointer to RAM or program memory. Bit 15 is used to detect RAM or program memory access.

The 12 bit Core

Indirect RAM access on the 12 bit core (16F57/PIC10) requires some care because the RAM bank selection bits resides in the FSR register (bit 5,6). The compiler can do most of the checking and generate error messages if required. Automatic bank bit updating can be switched off globally (*-b* command line option), or locally (#pragma update_FSR 0). Most of the checking described is performed only if the automatic bank bit updating in ON.

Reading and writing arrays is straight forward:

```
bank2 char a, e, t[3], s[3];

a = t[i];
s[i] = e;
s[i+3] = e;
```

The last three statements requires that variable *e* is located in mapped RAM (below 0x10) or in the same bank as array *s[]*. Otherwise an error message is printed to indicate that the compiler cannot update the bank selection bits.

Pointers may need a *#pragma assume* statement:

```
#pragma rambank 3
char *px, r;
#define LTAB 5
char tab[LTAB];
#pragma assume *px in rambank 3

px = &tab[0];

*px = r;
if (++px == &tab[LTAB])
    px = &tab[0];
```

A pointer may access more than one bank. The #pragma assume statement should NOT be used in such cases. The only difference is that the compiler will know the contents of the FSR.5,6 when a variable in a specific bank is accessed. Therefore, a statement like:

```
*pointer_to_any_rambank = e;
```

requires that *e* in located in mapped RAM (address less than 0x10).

Note that the *#pragma assume* statement works for single pointers (and pointers in arrays), but not for pointers located in structures.

Arrays are often more efficient than pointers:

```
i = 0;
// ..
tab[i] = r;
if (++i == LTAB)
    i = 0;
```

Direct use of INDF and FSR is also possible:

```
FSR = px;
INDF = i;
```

Variable *i* have to reside in mapped RAM. The compiler performs the checking when INDF is accessed. The compiler does not try to trace the contents of FSR when it is loaded directly. Therefore, a statement like **px = r;* is normally preferred.

Using *#pragma assume *px in rambank 3* also makes loading of *px* more restrictive. An error message is printed if *px* is loaded with an address in another bank. The following cases are checked:

```
px = tab;      // same as &tab[0]
px = &tab[0];
px = &tab[i];
px = pxx;      // pxx is another pointer
px = &pxx[i];
```

A statement like *px = &tab[i];* may fool the compiler if the value of *i* is too large.

If the above syntax is too restrictive, then a local update region is the solution. All bank updating then have to be done with C statements. Normally, local update regions require inspection of the generated assembly file to avoid problems.

```
/* these statements clears the buffer */
 i = LTAB;
#pragma update_FSR 0   /* OFF */
 FSR = &tab[0];
 do  {
     INDF = 0;
     FSR ++;
 } while (--i > 0);
#pragma update_FSR 1   /* ON */
```

Without a local update region:

```
i = LTAB;
do
   tab[i-1] = 0;
while (--i > 0);
```

In this example, the local update region only has a speed advantage. The same amount of instructions is generated. Note that although no bank updating is required inside the above local region, the compiler does not know the contents of FSR.5,6 at the end of the region, and will therefore update these bits afterwards.

The 14 bit core: the IRP bit

Some 14 bit core devices contain more than 2 banks. This means that register bit IRP have to be updated in user code when working with arrays and tables. Note that Enhanced 14 bit core devices does not use the IRP bit, but have 16 bit indirect registers.

```
#pragma rambank 2
char array[50];
char x;

FSR = &array % 256 + x;   // LSB of  &array[x]
IRP = &array / 256;       // MSB
```

NOTE: IRP is not updated by the compiler if INDF is used directly in the user code. Using *array[x]* instead of INDF enables automatic updating of the IRP bit.

The compiler will trace all loading of pointers to decide how the IRP bit should be updated. This applies to both 8 and 16 bit pointers.

It is also possible to use *#pragma assume* to state the bank directly:

```
bank1 char t[3];
bank3 char i, *pi, *pit;
#pragma assume *pi in rambank 3   // or rambank 2
#pragma assume *pit in rambank 1  // or rambank 0
..
pi = &i;
pit = &t[2];
```

An error message is printed if a pointer is loaded with an address from the wrong RAM half. Note that bank 0 and 1 are grouped together (the lower RAM half, 0 - 0xFF). Bank 2 and 3 are the upper RAM half (0x100 - 0x1FF).

Updating of IRP can be switched off locally. The compiler does not remove superfluous updating of the IRP register. This means that IRP is updated for each pointer or table access.

An efficient strategy may be to locate (most of) the tables in upper or lower RAM (above or below address 0x100), and do all updating of IRP in the user code. Few updates are sometimes sufficient.

```
#pragma update_IRP 0   /* off */
..
IRP = 1;  // updated by user code
..
#pragma update_IRP 1   /* on */
```

2.5 Const Data Support

CC5X supports constant data stored in program memory. The C keyword 'const' tells the compiler that these data do not change. Examples:

```
const char *ps = "Hello world!";
const int16 itx[] = { -10, 2 – 100, 1.34 * 1000 };
const float ftx[] = { 1.0, 33.34, 1.3e-10 };
..
t = *ps;
ps = "";
fx = ftx[i];
```

The implementation of constant data supports the following features:

- both 8 and 16 bit pointers to const data in the same application
- the size of single const pointers can be chosen automatically
- const pointers can access both RAM and program memory
- the compiler will not put all constant data in a single table, but rather make smaller tables if this saves code space
- some devices support 14 bits data (PIC16F87X). The compiler will automatically choose this format if space can be saved. This allows compact storage of 7 bit ASCII strings.
- duplicate strings and other data are automatically merged to save space

Recommendations:

It is recommended to use small data tables and structures. This allows the compiler to merge equal data items and build optimal blocks of constant data.

Limitations:

1) The compiler will not initialize RAM variables on startup
2) Data items of 16 bits or more in structures with more than 256 bytes of data must be aligned

Storing 14 bit data

Most 14 bit core devices support 14 bits data stored in program memory. This allows compact storage of 7 bit ASCII strings and 14 bits data. The code sequence required for accessing these bits is longer than the code for a return table. This means that code is saved when the amount of data exceeds 40-50 bytes. The compiler will automatically choose the optimal storage format.

When a constant table contains less than 256 byte of data, there will be a tradeoff between speed and size. Using a return table executes faster but requires more code when the table contains more than 40-50 bytes of data. If speed is required, the following pragma statement defines a new limit for size optimization.

```
#pragma wideConstData 200 // return table limit
```

. It is also possible to disable 14 bit data storage:

```
#pragma wideConstData 8192  // always use a const return table
```

Data of size 16 bits or more

The compiler allows access of 8, 16, 24 and 32 bit data, including fixed and floating point formats. When using arrays or structures with more than 256 bytes of data, single data items have to be aligned. Alignment means that there should not be any remainder when dividing the offset by the size of the data item. This is only a problem when defining structures containing data of different sizes.

```
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
```

Code pages

When using devices with more than one codepage, the compiler will automatically calculate the optimal codepage for the data. The disadvantage is that the compiler will not know when a codepage is full, so the functions still have to be moved manually between codepages to find allowed and optimal combinations. Also, const data can be located on a specific codepage by using a page type modifier.

```
const page1 char tx[] = "Hello!";
```

The compiler will group the const data (including strings) into "storage classes" depending on data types, optimization and how they are accessed.

Data belonging to the same storage class are accessed by the same _constX access function generated by the compiler.

The general rule is that all const data accessed by the same pointer belongs to the same storage class.

It is sufficient to use a page type modifier on one of the const data items in a storage class, then all data belonging to that storage class will be stored on that codepage. This can be used to split large amount of const data into separate codepages.

Strings are sometimes harder to force into specific codepages. However, since equal strings are merged together by the compiler it is possible to define an unused data item with a known string used somewhere else to force a specific group of strings to a specific codepage. The unused string will not consume any code space. Example:

```
const page2 char unusedString[] = "Temperature:";
```

Locating Const Data

The compiler will normally insert 'const' data at the start of each codepage (after the interrupt routine). The following pragma statement will allow 'const' data for the current codepage to be inserted between user functions, or at a specific address when using #pragma origin first. The current codepage can also be set by using #pragma codepage.

```
#pragma insertConst
```

Merging data

The compiler will automatically merge equal strings and sub-strings, and also other data items. Using small tables will increase the chance of finding data items that can be merged. Note that data containing initialized addresses (ROM and RAM) are not merged. Examples:

1. The string "world!" is identical to the last part of the string "Hello world!". It is therefore not required to use additional storage for the first string. The compiler handles the address calculations so that merged (or overlapping) strings are handled fully automatically. Note that the string termination '\0' also has to be equal, otherwise merging is not possible. For example, the string "world" cannot be merged with the above strings.

2. Merging applies to all kinds of data. Data is compared byte by byte. This allows the first two of the following tables to be merged with the last one.

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

Examples

A table of pointers to strings:

```
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;
```

Note that 'const struct' is required to put the pointer array in program memory. Using 'const char *tx[];' means that the strings reside in program memory, but the table 'tx[]' resides in RAM.

String parameters:

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

Const data stored in dedicated functions

The data type DataInW allows integer data that fits within an instruction word (12/14 bit) to be stored in const data tables that are mapped to dedicated functions containing data elements only (no code or return). Note that it is not possible to use DataInW outside the const table.

It is possible to read and assign the address of the const table, but without any operations. All access of data within this type of tables must be done by an application access function. It is not possible to use a table index read (dataTable[i]), fixed offset (dataTable[5]) or pointers.

```
const DataInW dataTable[] = {
    1234,
    3456,
    0x3FFF,  // 14 bit core: max 14 bit (12 bit for 12 bit core)
    0,
};

uns8 getData( uns16 ix)
{
    uns16 base = (uns16) dataTable;  // get the table start address
    base += ix;

    // NOTE: register names and access procedure are device dependent
    NVMADRL = base;         // LSB of address
    NVMADRH = base >> 8;    // MSB of address

    NVMREGS = 0; // Do not select Configuration Space
    RD = 1;      // Initiate read

    return NVMDATL;  // read LSB
}
..
uns16 ix16;
uns8 x = getData(ix16);

uns8 y = NVMDATH;  // read MSB
```

3 SYNTAX

3.1 Statements

C statements are separated by semicolons and surrounded by block delimiters:

```
{ <statement>; .. <statement>; }
```

The typical statements are:

```
// 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;
}
```

if statement

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

The *else if* and *else* parts are optional.

while statement

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

while (1) { .. }   // infinite loop
```

for statement

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

initialization: legal assignment or empty
condition: legal condition or empty
increment: legal increment or assignment or empty

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

do statement

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

switch statement

The switch statement supports variables up to 32 bits. The generated code is more compact and executes faster than the equivalent 'if - else if' chain.

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

<variable>: all 8-32 bit integer variables including W
break: optional
default: optional, can be put in the middle of the switch statement

```
switch (token)  {
  case 2:
    i += 2;
    break;

  case 9:
  case 1:
  default:
    if (PORTA == 0x22)
        break;

  case 'P':
    pin1 = 0; i -= 2;
    break;
}
```

break statement

The ‘break;’ statement is used inside loop statements (for, while, do) to terminate the loop. It is also used in *switch* statements.

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

continue statement

The ‘continue;’ statement is used inside loop statements (for, while, do) to force the next iteration of the loop to be executed, skipping any code in between. In *while* and *do-while* loops, the loop condition is executed next. In *for* loops, the increment is processed before the loop condition.

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

return statement

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

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

goto statement

```
goto <label>;
```

Jumps to a location, forward or backward.

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

3.2 Assignment and Conditions

Basic assignment examples:

```
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;
```

Special syntax examples

```
#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];
```

Conditions

```
[ ++ | -- ] <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)  ..
```

Bit variables

```
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;
```

Multiplication, division and modulo

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

A general multiplication algorithm is implemented, allowing most combinations of variable sizes. **Including a math library** allows library calls to be generated instead of inline code. The algorithm makes shortcuts when possible, for instance when multiplying by 2. This is treated as a left shift.

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

The division algorithm also allows most combinations of variable sizes. Shortcuts are made when dividing by 2 (or 2*2*..). These are treated as right shifts.

Precedence of C operators

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

Mixed variable sizes are allowed

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

Most combinations of variables are allowed; the compiler performs sign extension as required. Multiple operations in the same expression are allowed when using 8 bit variables.

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

3.3 Constants

```
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 */
```

Please note that parentheses are required in some cases.

Constant expressions

The size of integers is by default 8 bits for this compiler (other C compilers typically use 16 or 32 bits depending on the CPU capabilities). An error is printed if the constant expression loses significant bits because of value range limitations.

```
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
```

Adding an *L* means conversion to long (16 bit).

The command line option -*cu* forces 32 bit evaluation of constants so that no significant bits are lost.

Some new built in types can also be used:

```
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
```

The constant type is by default the shortest signed integer. Adding a *U* behind a constant means that it is treated as unsigned. Note that constants above 0x7FFFFFFF are unsigned by default (with or without a *U* behind).

Enumeration

An enumeration is a set of named integer constants. It can often replace a number of *#define* statements. The numbering starts with 0, but this can be changed:

```
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;
```

3.4 Functions

Function definitions can appear as follows:

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

Function calls:

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

The compiler needs to know the definition of a function before it is called to enable type checking. A prototype is a function definition without statements. Prototypes are useful when the function is called before it is defined. The parameter name is optional in prototypes:

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

Function return values

Functions can return values up to 4 bytes wide. Return values can be assigned to a variable or discarded. Handling and using return values is automated by the compiler.

The least significant byte is always placed in W when using 14 bit core devices. Signed variables and variables larger than 8 bits also use temporary variables on the computed stack. The 12 bit core use the W register when returning 8 bit constants. All other return values are placed in return variables on the computed stack.

A function can return any value type. The W register is used for an 8 bit return value if possible. The Carry flag is used for bit return values. The compiler will automatically allocate a temporary variable for other return types. A function with no return value is of type *void*.

Parameters in function calls

There is no fixed limit on the number of parameters allowed in function calls. Space for parameters is allocated in the same way as local variables, which allows efficient reuse. The bit type is also allowed. Note that if W is used, this has to be the LAST parameter.

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

Internal functions

The internal functions provide direct access to certain inline code:

```
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
f = decsz(f); // char decsz(char);  - DECFSZ f,d
W = incsz(f); // char incsz(char);  - INCFSZ f,d
nop();        // void nop(void);    - NOP
nop2();       // void nop2(void);   - GOTO next address
retint();     // void retint(void); - RETFIE
W = rl(f);    // char rl(char);     - RLF f,d
f = rr(f);    // char rr(char);     - RRF f,d
sleep();      // void sleep(void);  - SLEEP
skip(i);      // void skip(char);   - computed goto
f = swap(f);  // char swap(char);   - SWAPF f,d
```

Additional internal functions are available for the enhanced core 14:

```
W = addWFC(f);// char addWFC(char); - ADDWFC f,d
f = subWFB(f);// char subWFB(char); - SUBWFB f,d
f = lsl(f);   // char lsl(char);    - LSLF f,d
f = lsr(f);   // char lsr(char);    - LSRF f,d
f = asr(f);   // char asr(char);    - ASRF f,d
softReset();  // void softReset(void); - RESET
```

The internal rotate functions are also available for the larger variable sizes:

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

The inline function nop2() is implemented by a GOTO to the next address. Thus, nop2() can replace two nop() to get more compact code. The main use of nop() and nop2() is to design exact delays in timing critical parts of the application.

3.5 Type Cast

Constants and variables of different types can be mixed in expressions. The compiler converts them automatically to the same type according to the stated rules. For example, the expression:

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

consists of 2 separate operations. The first is the plus operation and the second is the assignment. The type conversion rules are first applied to *b*+*c*. The result of the plus operation and *a* are treated last.

The CC5X compiler uses 8 bit int size and contains many data types (integers, fixed and floating point). The type cast rules have been set up to provide best possible compatibility with standard C compilers (which typically uses 16 or 32 bit int size).

The type conversion rules implemented are:

1. if one operand is double -> the other is converted to double
2. if one operand is float -> the other is converted to float
3. if one operand is 32 bit -> the other is converted to 32 bit
4. if one operand is 24 bit -> the other is converted to 24 bit
5. if one operand is long -> the other is converted to long
6. if one operand is unsigned -> the other is converted to unsigned

NOTES:
- The sign is extended before the operand is converted to unsigned.
- Assignment is also an operation.
- The char type is unsigned
- Constants are SIGNED, except if U is added.
- The bit type is converted to unsigned char.
- The fixed point types are handled as subtypes of float.

Type conversion in C is difficult. The compiler may generate a warning if a type cast is required to make the intention clear. Remember that assignment (=) is a separate operation. The separate operations are marked (1:), (2:) and (3:) in the following examples.

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

a16 = b8 * c8; /* (1:) In this case both b8 and c8 are 8 bit unsigned, so the type of the multiplication is 8 bit unsigned. (2:) The result is then assigned to a 16 bit unsigned variable, a16. Converting the 8 bit unsigned result to 16 bit unsigned means clearing the most significant bits of a16. The compiler generates a warning because significant bits of the multiplication are lost due to the type conversion rules. */

a16 = (uns16) (b8 * c8); /* (1:) Adding parentheses just isolates the multiplication and the multiplication result is still 8 bit unsigned. (2:) The (uns16) type cast is not needed because this type cast is done automatically before the assignment. The compiler generates a warning because significant bits of the multiplication are lost due to the type conversion rules. */

a16 = (uns16) b8 * c8; /* (1:) Converting one of the arguments to 16 bit unsigned BEFORE the multiplication is the right syntax to get a 16 bit result. (2:) The result and the destination a16 now have the same type for the assignment and no type conversion is needed. */

a16 = (uns8) (b8 * c8); /* (1:) The multiplication result is 8 bit unsigned. (2:) The (uns8) type cast tells the compiler that the result should be 8 bit unsigned, and no warning is generated even though it looks like significant bits of the multiplication are lost. */

a16 = b8 * 200; /* (1:) Constant 200 is a 16 bit signed constant (note that 200U is an 8 bit unsigned constant, and that 127 is the largest 8 bit signed constant). Argument b8 is therefore automatically converted to 16 bit. The constant is then converted to unsigned and the result is 16 bit unsigned. (2:) The result and the destination a16 now have the same type for the assignment and no type conversion is needed. */

a16 = (int16) i8 * j8; /* (1:) Both arguments are converted to 16 bit signed and the result is 16 bit signed. (2:) The result is converted to unsigned before the assignment, but this does not mean any real change when the size is the same (example: -1 and 0xFFFF have the same 16 bit representation). */

a16 = (uns16) (uns8)i8 * (uns8)j8; /* (1:) To get an 8*8 bit unsigned multiplication it is necessary to cast both arguments to unsigned before extending the size to 16 bit unsigned. Otherwise the sign bit will be extended and the multiplication will need more code and cycles to execute. (2:) The result and the destination a16 now have the same type for the assignment and no type conversion is needed. */

a16 = ((uns16) b8 * c8) / 3; /* (1:) Converting one of the arguments to 16 bit unsigned before the multiplication gives a 16 bit result. (2:) Division is the next operation and is using the 16 bit unsigned multiplication result. Constant 3 is 8 bit signed, and is then automatically converted to 16 bit signed and further to 16 bit unsigned. The result of the division is 16 bit unsigned. (3:) The division result and the destination a16 now have the same type for the assignment and no type conversion is needed. */

3.6 Accessing Parts of a Variable

Each bit in a variable can be accessed directly:

```
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
```

Also, parts of a variable can be accessed directly:

```
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
```

The table shows which bits are accessed depending on the variable size in bytes (1,2,3,4) and the sub-index used. The * indicates normal use of the sub-index:

	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

3.7 C Extensions

CC5X adds some extensions to the standard C syntax:

1. The *bit* variable type

2. The *interrupt* function type

3. Local variables can be declared between statements as in C++. Standard C requires local variables to be defined in the beginning of a block.

4. Binary constants : 0bxxxxxx or bin(xxxxxx)
   The individual bits can be separated by the '.':
```
0b0100
0b.0.000.1.01.00000
bin(0100)
bin(0001.0100)
```

5. Preprocessor statements can be put into macros. Such preprocessor statements are not extended to multiple lines. The inserted preprocessor statements are evaluated when the macro is expanded, and not when it is defined.

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

6. Several of the type modifiers are not standard in C (page0..page3, bank0..bank3, shrBank, size1,size2)

More C extensions are allowed by the #pragma statement.

3.8 Predefined Symbols

The basic PICmicro registers are predefined (header files define the rest):

```
W, INDF, PCL, STATUS, FSR, PORTA, Carry, etc.
```

The following names are defined as internal functions, and are translated into special instructions or instruction sequences.

```
btsc, btss, clearRAM, clrwdt, decsz, incsz, nop, nop2, retint, rl,
rr, sleep, skip, swap, addWFC, subWFB, lsl, lsr, asr, softReset
```

Extensions to the standard C keywords

```
bank0, .. bank63, bit, DataInW, fixed8_8, .. fixed24_8, float16,
float24, float32, int8, int16, int24, int32, interrupt, page0, ..
page15, shrBank, size1, size2, uns8, uns16, uns24, uns32
```

Standard C keywords used

```
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
```

The remaining standard C keywords are detected and compiled. One is ignored (*register*), and the rest cause a warning to be printed (*volatile*, *line*).

The sizeof operator

The operator sizeof() gives the size in bytes of the argument. The argument can be a type name, a variable name, a pointer, a structure name, an array name, a string literal or a constant. Sizeof can also be used in a preprosessor statement. Examples: sizeof(char) is 1, sizeof(bit) is 0, sizeof("abc") is 4, sizeof(int24) is 3.

Function offsetof(struct_type, struct_member)

Function offsetof() returns the offset to a structure member. The first argument must be a struct type, and the second a structure member. The function can also be used in a preprocessor expression.

```
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]);
```

Automatically defined macros and symbols

The following symbols are automatically defined when using the CC5X compiler, and can be used in preprocessor macros:

```
__CodePages__ := 1..16 : depending on the number of code pages
                         on the actual chip

__CC5X__  := Integer version number: 3200 means version 3.2,
             3202 means version 3.2B
             * first 2 digits : main version
             * last 2 digits : minor release (01='A', 02='B', etc.)

__CoreSet__ := 1200 : 12 bit core
               1400 : 14 bit core
               1410 : 14 bit enhanced core

__IRP_SFR__ := 1 if IRP is active when accessing
           special function registers using INDF, 0
           otherwise. Note that '#pragma update_IRP 0'
           will set this macro to 0 until
           '#pragma update_IRP 1' is processed.

__IRP_RAM__ := 1 if IRP is active when accessing RAM
           registers using INDF, 0 otherwise. Using
           '#pragma update_IRP 0' will set this macro
           to 0 until '#pragma update_IRP 1'.

_16CXX     := 1  : always defined (12 and 14 bit cores)

_16C5X     := 1  : when the 12 bit core is selected

__EnhancedCore14__ := 1  : when the Enhanced Core 14 is selected

__EnhancedCore12__ := 1  : when the Enhanced Core 12 is selected

_16C54     := 1  : when the 16C54 is selected, similar for all
                   other devices
```

Macros __FILE__ and __LINE__

Macro __FILE__ is replaced by the name (string literal) of the current source file. Macro __LINE__ is replaced by the current line number (decimal constant) of the source file being compiled.

Macros __DATE__ and __TIME__

Macros for date and time are defined when compilation starts.

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

3.9 Upward Compatibility

The aim is to provide best possible upward compatibility from version to version. Sometimes the generated code is improved. If the application programs contain critical timing parts (depending on an exact instruction count), then these parts should be verified again, for example by using the MSDOS program *fc* (file compare) on the generated assembly files.

Evaluation of constant expression is slightly changed from version 2.x in order to adapt to standard C. An error message is printed if significant bits are lost. The cure is to use type conversion.

```
a = (uns16) 10 * 100;
```

Alternatively will the command line option *-cu* force 32 bit evaluation of constant expressions. The option *-wS* changes the error message to a warning.

4 PREPROCESSOR DIRECTIVES

The preprocessor recognizes the following keywords:

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

A preprocessor line can be extended by putting a '\' at the end of the line. This requires that there are no space characters after the '\'.

#define

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

Note that all *#define's* are global, even if they are put inside a function.

Preprocessor directives can be put into the *#define* statement.

Macro concatenation

The concatenation operator ## allows tokens to be merged while expanding macros. Examples:

```
#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 ##
```

Macro stringification

The stringification operator # allows a macro argument to be converted into a string constant. Examples:

```
#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's can be nested. When using *#include "test.h"* the current directory is first searched. If the file is not found there, then the library directories are searched, in the same order as supplied in the command line option list (*-I<dir>*). The current directory is skipped when using *#include <test.h>*.

Macros can be used in #include files. The following examples show the possibilities. Note that this is not standard C.

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

Rules for macros in #include:

1. Strings using "" can be split, but not strings using <>
2. Only the first partial string can be a macro
3. Nested macros are possible
4. Only one macro at each level is possible

#undef

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

#undef does the opposite of *#define*. The *#undef* statement will not produce any error message if the symbol is not defined.

#if

```
#if defined ALFA  &&  ALFA == 1
    ..
```

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

An arbitrary complex constant expression can be supplied. The expression is evaluated the same way as a normal C conditional statement is processed. However, every constant is converted to a 32 bit signed constant first.

```
1) macros are automatically expanded
2) defined(SYMBOL)and defined SYMBOL are replaced by 1 if the symbol
   is defined, otherwise 0.
3) legal constants :  1234  -1  'a' '\\'
4) legal operations :  + - * / % >> <<
                        == != < <= > >= || &&
                        ! ~  ()
```

#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
```

The compiler generates an error message using the text found behind #error.

#warning

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

The following output is produced. Note that this directive is not standard C.

```
Warning test.c 7: This is a warning
```

#message

```
#message This is message 1
```

The following output is produced. Note that this directive is not standard C.

```
Message: This is message 1
```

4.1 The pragma Statement

The pragma statement is used for processor specific implementations.

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

This pragma statement allows the origin to be aligned. The compiler will check if the least significant byte of the origin address is equal to <a>, or alternatively within the range <a> to <b>. If this is not true, the origin is incremented until the condition becomes true. Both <a> and <b> may range from -255 to 255.

```
#pragma alignLsbOrigin 0
#pragma alignLsbOrigin 2 to 100
#pragma alignLsbOrigin 0 to 190     // [-255 .. 255]
#pragma alignLsbOrigin -100 to 10
```

Such alignment is useful to make sure that a computed goto does not cross a 256 word address boundary. More details are found in Section *Origin alignment* on page 114 in Chapter *9.2 Computed Goto*. This type of alignment does not apply to the 12 bit core.

#pragma asm2var 1

Enable equ to variable transformation. This is defined in Chapter *6.6 Inline Assembly* on page 79.

#pragma assert [/] <type> <text field>

Assert statements allow messages to be passed to the simulator, emulator, etc. Refer to Chapter *7.3 Assert Statements* on page 105 for details.

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

The *#pragma assume* statement tells the compiler that a pointer operates in a limited address range. Refer to Chapter *2.4 Pointers* on page 27 for details.

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

Defines the global bit variable <name>. It is useful for assigning a bit variable to a certain address. Only valid addresses are allowed:

```
#pragma bit bitxx @ 0x20.7
#pragma bit ready @ STATUS.7
#pragma bit ready @ PA2
```

NOTE: If the compiler detects double assignments to the same RAM location, this will cause a warning to be printed. The warning can be avoided if the second assignment uses the variable name from the first assignment instead of the address (*#pragma bit var2 @ var1*).

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

The cdata statement can store 14 bit data in program memory at fixed addresses. It can also be used to store data in EEPROM memory. Refer to Chapter *6.9 The cdata Statement* on page 100 for details.

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

ADDRESS: 0 .. 0x7FFE
VXS : < VALUE | EXPRESSION | STRING>
VALUE: 0 .. 0x3FFF
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>

Defines the global variable <name>. The statement is useful for assigning a variable to a certain address. Only valid addresses are allowed:

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

NOTE: If the compiler detects double assignments to the same RAM location, this will cause a warning to be printed. The warning can be avoided if the second assignment uses the variable name from the first assignment instead of the address (*#pragma char var2 @ var1*).

#pragma chip [=] <device>

Defines the chip type. This allows the compiler to select the right boundaries for code and memory size, variable names, etc. Note that the chip type can also be defined as a command line option.

```
#pragma chip PIC16C55
```

This statement has to precede any normal C statements, but some preprocessor statements, like #if and #define, can be compiled first.

The supported devices are either known internally (16C54,55,56,57,58, 61,64,65, 71,73,74, 84, 620,621,622) or defined in a PICmicro header file (e.g.16F877.h). It is also possible to make new header files. Refer to file '**chip.txt**' for details.

#pragma codepage [=] <0,1,2,3, ..15>

```
    // 12 bit core      14 bit core
0   // 0x000 - 0x1FF    0x0000 - 0x07FF
1   // 0x200 - 0x3FF    0x0800 - 0x0FFF
2   // 0x400 - 0x5FF    0x1000 - 0x17FF
3   // 0x600 - 0x7FF    0x1800 - 0x1FFF
```

Defines the codepage to be used. Code is located at the start of the active codepage, or from the current active location on that page. The codepage cannot be changed inside a function. Non-existing pages for a specific device are mapped into existing ones.

```
#pragma codepage 3
/* following functions are located on codepage 3 */
```

#pragma computedGoto [=] <0,1,2>

This statement can be used when constructing complicated computed goto's. Refer to Chapter *9.2 Computed Goto* on page 113 for details.

```
#pragma computedGoto  1  // start region
#pragma computedGoto  0  // end of region
#pragma computedGoto  2  // start large region
```

#pragma config [/<regNr> <value>] [<id>] = <state> [, <id> = <state>]

The pragma config statement supports device configuration definition and setting and ID register setting. The compiler supports both direct and symbolic setting of the device configuration. It is NOT allowed to combine direct and symbolic config settings. Example symbolic config setting:

```
#pragma config FOSC = ECL   // setting a config symbol
#pragma config PWRTE = ON, CP = ON, WRT = ALL
```

In order to use symbolic config register setting there must be a symbolic config definition. This is normally found in the device header file:

```
#pragma config /1 0x3FFD FOSC = ECL
```

Direct config setting is an alternative:

```
#pragma config = (0x3 | 0x10) & 0x1FF

#pragma config reg1 = 0x2  // equivalent to #pragma config = 0x2
#pragma config reg2 = 0x3 | 0x4
..
#pragma config reg9 = 0x0
```

For the first config register it is also possible use the '&=' and the '|=' operators. The default setting of the config bits are 0 when using these operators:

```
#pragma config |= 0x100  // set bit 8
#pragma config &= 0xFFFC // clear bit 0 and 1
#pragma config &= ~3     // clear bit 0 and 1
#pragma config |= 3      // set bit 0 and 1
```

Setting ID-locations in the source code:

```
#pragma config ID=0x1234   //  all 4 locations, 4 bit in each

#pragma config ID[0] = 0xF  //  location 0
#pragma config ID[1] = 0x0  //  location 1
#pragma config ID[2] = 1, ID[3] = 0x3
```

Refer to Chapter *4.2 PICmicro Configuration* on page 58 for more details.

#pragma config_def [=] <value>

Defines the position and size of the supported config identifiers, and is normally found in PICmicro header files. Refer to file '**chip.txt**' for details.

#pragma config_reg [=] <address>

The default address of config register can be changed. There is normally no need to do this. The following pragma statement will change the default configuration word address:

```
#pragma config_reg 0x3FF
```

Some devices use a different physical address of the configuration word compared to the default logical address established. The chip programming software will normally map the configuration word in these cases.

#pragma config_reg2 [=] <address>

The address of the second config register is normally defined in the header file using:

```
#pragma config_reg2 0x2008
```

#pragma data_area [=] <start_address> : <last_address>

The 14 bit standard core uses normally region 0x2100 - 0x21FF for EEPROM/data storage in the files generated by the compiler. For some devices (12 bit core) it is required to define the area used for such data. The following statement should be located in the header file:

```
#pragma data_area  0x400 : 0x43F
```

#pragma inlineMath <0,1>

The compiler can be instructed to generate inline **integer** math code after a math library is included.

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

#pragma insertConst

The compiler will normally insert 'const' data at the start of each codepage (after the interrupt routine). The #pragma insertConst statement will allow 'const' data for the current codepage to be inserted between user functions, or at a specific address when using #pragma origin first. The current codepage can also be set by using #pragma codepage.

#pragma interruptSaveCheck <n,w,e>

The compiler will automatically check that vital registers are saved and restored during interrupt. Note that the compiler will not check register saving on the enhanced 14 bit core because of the hardware context saving of these devices. Please refer to Chapter *6.3 Interrupts* on page 67 for details (or file ‘int16cxx.h’). The error and warning messages can be removed:

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

#pragma library <0/1>

CC5X will automatically delete unused (library) functions.

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

#pragma location [=] <0,1,2,3,.. 15, - >

This statement can be used to locate the functions on different codepages. Refer to Chapter *6.1 Program Code Pages* on page 65 for more details.

#pragma mainStack <minVarSize> @ <lowestStartAddr>

This statement defines a main stack for local variables, parameters and temporary variables. The main stack is not an additional stack, but tells the compiler where the main stack is located (which bank). Only variables above or equal to <minVarSize> will automatically be put in the main stack. The <lowestStartAddr> is the lowest possible start address for the main stack (the stack grows upwards).

```
#pragma mainStack 3 @ 0x20
```

Using this pragma means that local variables, parameters and temporary variables of size 3 bytes and larger (including tables and structures) will be stored in a single stack allocated no lower than address 0x20. Smaller variables and variables with a bank modifier will be stored according to the default/other rules. Size 0 means all variables including bit variables.

Note that #pragma rambank is ignored for variables stored in the main stack. Addresses ranging from 0x20 to 0x6F/0x7F are equivalent to the bank0 type modifier.

#pragma minorStack <maxVarSize> @ <lowestStartAddr>

This statement defines a minor stack for local variables, parameters and temporary variables. One reason for defining a minor stack is that it may be efficient to use shared RAM or a specific bank for local variables up to a certain size. Only variables below or equal to <maxVarSize> will automatically be put in the minor stack. The <lowestStartAddr> is the lowest possible start address for the minor stack (the stack grows upwards).

```
#pragma minorStack 2 @ 0x70
```

In this case, local variables, parameters and temporary variables up to 2 bytes will be put in shared RAM from address 0x70 and upward. Larger variables and variables with a bank modifier will be stored according to the default/other rules. Size 0 means bit variables only. This pragma can be used in combination with the main stack. The variable size defined by the minor stack has priority over the main stack.

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

This statement enables optimization to be switched ON or OFF in a local region. A specific type of optimization can also be switched on or off. The default setting is on.

N	Function
1.	redirect goto to goto.
2.	remove superfluous gotos.
3.	replace goto by skip instructions.
4.	remove instructions that affect the zero-flag only.
5.	replace INCF and DECF by INCFSZ and DECFSZ.
6.	remove superfluous updating of page selection bits.
7.	remove other superfluous instructions.
8.	remove superfluous loading of W.

```
Examples:
#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
```

NOTE: The command line option -*u* will switch optimization off globally, which means that all settings in the source code are ignored.

#pragma origin [=] <expression>

Valid address region is 0x0000 - 0x1FFF. Defines the starting address (and codepage) of the following code. The current active location on a codepage cannot be moved backwards, even if there is no code in that area. Origin cannot be changed inside a function.

Examples:

```
#pragma origin 4  // interrupt start address
#pragma origin 0x700 + 2
```

#pragma packedCdataStrings <0,1>

Strings will normally be packed into 2*7 bits when using cdata. This statement allows the packing of strings to be enabled and disabled for different parts of the source code. See Section *Storing EEPROM Data* on page 102 in Chapter 6.9 *The cdata Statement* for more details.

#pragma rambank [=] <0,1,2,3,..31, - >

14 bit core:

```
-   =>  mapped space: (chip specific)
0   =>  bank 0:    0 (0x000) -  127 (0x07F)
1   =>  bank 1:  128 (0x080) -  255 (0x0FF)
2   =>  bank 2:  256 (0x100) -  383 (0x17F)
3   =>  bank 3:  384 (0x180) -  511 (0x1FF)
```

12 bit core:

```
-   =>  mapped space:    8  -  15 (0x0F)
0   =>  bank 0:   16 (0x10) -  31 (0x1F)
1   =>  bank 1:   48 (0x30) -  63 (0x3F)
2   =>  bank 2:   80 (0x50) -  95 (0x5F)
3   =>  bank 3:  112 (0x70) - 127 (0x7F)
```

#pragma rambank defines the region(s) where the compiler will allocate variable space. The compiler gives an error message when all locations in the current bank are allocated.

RAM banks are only valid for some of the devices. Non-existing banks for the other devices are mapped into defined RAM space.

#pragma rambase [=] <n>

Defines the start address when declaring global variables. This statement is included for backward compatibility reasons only. The use of rambank and rambase are very similar. The address has to be within the RAM space of the chip used.

12 bit core note: The locations from address 0 to 31 are treated as a unit. Using start address 7 means that locations in the mapped register space and bank 0 are allocated. Using start address 32 implies that locations in the mapped register space are allocated.

NOTE: The start address is not valid for local variables, but *rambase* can be used to select a specific RAM-bank.

#pragma ramdef <ra> : <rb> [MAPPING]

This statement is normally used in PICmicro header files. Refer to file '**chip.txt**' for details.

#pragma resetVector <n>

Some chips have an unusual startup vector location (like the PIC16C508/9). The reset-vector then has to be specified. This statement is normally NOT required, because the compiler normally uses the default location, which is the first (14 bit core) or the last location (12 bit core).

```
#pragma resetVector 0x1FF // at last code location
#pragma resetVector 0     // at location 0
#pragma resetVector 10    // at location 10
#pragma resetVector -     // NO reset-vector at all
```

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

This allows multiple return statements to be inserted. This statement should be proceeded by the skip() statement. The compiler may otherwise remove most returns. The constant <n> is optional, but it allows the compiler to print a warning when the number of constants is not equal to <n>. Refer to Chapter *9.2 Computed Goto* on page 113 for more details. Note that 'const' data types should normally be used for constant data.

```
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

This pragma allows functions containing local variables and parameters to be shared between independent call trees (interrupt and the main program). However, when doing this there will be a risk of overwriting these shared variables unless special care is taken. Further description is found in Section *"Functions shared between independent call trees"* in Chapter *6.2 Subroutine Call Level Checking*.

#pragma stackLevels <n>

The number of call levels can be defined (normally not required). The 12 bit core uses 2 levels by default. The 14 bit core uses 8 levels by default, and the enhanced 14 bit core uses 16 levels by default.

```
#pragma stackLevels 4   // max 64
```

#pragma unlockISR

The interrupt routine normally has to reside at address 4. The following pragma statement will allow the interrupt routine to be placed anywhere. Note that the compiler will NOT generate the link from address 4 to the interrupt routine.

```
#pragma unlockISR
```

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

The main usage of #pragma updateBank is to allow the automatic updating of the bank selection register to be switched on and off locally. These statements can also be inserted outside functions, but they should surround a region as small as possible

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

Another use of #pragma updateBank is to instruct the bank update algorithm to do certain selections. These statements can only be used inside functions:

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

#pragma updateBank exit = 1
/* The 'exit' bank forces 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 for
   loops and labels when the algorithm gives up
   finding the optimal choice */
```

#pragma update_FSR [=] <0,1>

Allows the automatic updating of the bank selection bits FSR.5 and FSR.6 to be switched on and off locally. This can be useful in some cases when INDF is used directly in the user code. The statement works for core 12 devices with more than one RAM bank. It is ignored for the other devices.

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

These statements can be inserted anywhere, but they should surround a region as small as possible.

#pragma update_IRP [=] <0,1>

Allows the automatic updating of the indirect bank selection bit IRP to be switched on and off locally. The statement is ignored when using the 12 bit core and enhanced 14 bit core. The statements can be inserted anywhere.

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

#pragma update_PAGE [=] <0,1>

Allows the automatic updating of the page selection bits to be swiched on and off locally. This is not recommended except in special cases. The page bits are found in the STATUS register for core 12 devices, and in PCLATH for core 14.

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

#pragma update_RP [=] <0,1>

Allows the automatic updating of the bank selection bits RP0 and RP1 to be switched on and off locally. The statement is ignored when using the 12 bit core.

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

These statements can be inserted anywhere, but they should surround a region as small as possible.

#pragma user_ID_addr [=] <address>

The default start address of user ID can be changed. There is normally no need to do this. The following statement should be located in the chip header file:

```
#pragma user_ID_addr 0x440
```

#pragma versionFile [<file>]

Allows a version number at the end of the include file to be incremented for each compilation. The use of this statement is defined in Chapter 5.2 *Automatic incrementing version number in a file* on page 63.

#pragma wideConstData [<N> | p | r]

Enable storing of 14 bit data for 14 bit core devices. Details are found in Chapter *2.5 Const Data Support* on page 30.

4.2 PICmicro Configuration

PICmicro configuration information can be put in the generated hex and assembly file. ID locations can also be programmed. The configuration information is generated ONLY WHEN the #pragma config statement is used. The compiler supports both direct and symbolic setting of the device configuration. It is NOT allowed to combine direct and symbolic config settings.

SYMBOLIC CONFIG SETTING:

The config settings can be defined using standard symbols for the actual device. Example usage:

```
#pragma config <id> = <state> [, <id> = <state>]

#pragma config FOSC = ECL  // ECL, External Clock
#pragma config WDTE = SWDTEN
#pragma config PWRTE = ON, CP = ON, WRT = ALL
#pragma config BORV = 25
```

Option -VG or -Vg will list the available symbolic config settings at the end of the *.var file generated for the project. This list can be copied to a project C source file and modified to desired settings.

```
-VG : list default config settings and alternatives
-Vg : list config setting alternatives
```

The config symbols are found at at the end of the device header file. Example definitions:

```
#pragma config /<regNr> <value> <id> = <state>

#pragma config /1 0x3FF8 FOSC = LP
..
#pragma config /1 0x3FFD FOSC = ECL
#pragma config /1 0x3FE7 WDTE = OFF
#pragma config /1 0x3FEF WDTE = SWDTEN
..
```

It is possible to disable the symbolic config definitions found in the header files for backward compatibility with the fixed config symbols in compiler versions older than version 3.6.

```
#pragma config /<regNr> <value> <id> = <state>
```

DIRECT CONFIG SETTING:

```
#pragma config [<id>] = <state> [, <id> = <state>]

<id>    : <reg1, reg2, .. reg9>
<state> : <value> or <expression>

#pragma config = (0x3 | 0x10) & 0x1FF

#pragma config reg1 = 0x2  // equivalent to #pragma config = 0x2
#pragma config reg2 = 0x3 | 0x4
..
#pragma config reg9 = 0x0
```

For the first config register it is also possible use the '&=' and the '|=' operators. The default setting of the config bits are 0 when using these operators.

```
#pragma config |= 0x100  // set bit 8
#pragma config &= 0xFFFC // clear bit 0 and 1
#pragma config &= ~3     // clear bit 0 and 1
#pragma config |= 3      // set bit 0 and 1
```

Default configuration word address:

- 12 bit core: 0xFFF
- 14 bit core: 0x2007 and 0x2008
- 14 bit enhanced core: 0x8007 .. 0x800F

It is possible to use the STANDARD MPASM identifiers for defining to configuration bits. See example in file 'config.txt'.

Programming of ID-locations:

```
#pragma config ID=0x1234 // all 4 locations, 4*4 bit
#pragma config ID[0] = 0xF  // location 0
#pragma config ID[1] = 0x1  // location 1
#pragma config ID[2]=1, ID[3]=0x2
```

Default User ID addresses:

```
16C54/55:             0x200-0x203
16C56:                0x400-0x403
16C57/58:             0x800-0x803
14 bit core:          0x2000-0x2003
14 bit enhanced core: 0x8000-0x8003
```

5 COMMAND LINE OPTIONS

The compiler needs a C source file name to start compiling. Other arguments can be added if required. The syntax is:

```
CC5X [options] <src>.c [options]
```

-a[<asmfile>] : generate assembly file.
The default file name is <src>.asm

-A[scHDpftumiJRbeokgN+N+N] : assembly file options
s: symbolic arguments are replaced by numbers
c: no C source code is printed
H: hexadecimal numbers only
D: decimal numbers only
P: use '.' in front of decimal constants
f: no object format directive is printed
t: no tabulators, normal spaces only
u: no extra info at the end of the assembly file
m: single source line only
i: no source indentation, straight left margin
J: put source after instructions to achieve a compact assembly file.
R: detailed macro expansion
b: do not add rambank info to variables in the assembly file
e: do not add ',1' to instructions when result is written back to the register
o: do not replace OPTION with OPTION_REG
k: do not convert all hexadecimal numbers (11h -> 0x11)
g: do not use PROCESSOR instead of the list directive
N+N+N: label, mnemonic and argument spacing. Default is 8+6+10.

-b : do not update bank selection bits
12 bit core: FSR.5 and 6
14 bit core: STATUS.RP0 and RP1

-bu : non-optimized updating of the bank selection bits

-B[pims] : write output from preprocessor to *.cpr
p : partial preprocessing
i : no include files
m: modify symbols
s : modify strings

-cd : allow cdata outside program space (warning only)
-cfc : use old format on config and idlocs in generated assembly file
-chu : disable sorting of addresses in generated HEX file
-cif : search included file in directory containing current file(s)
-cl<N> : set code generator level, N=0:old, N=1:improved
-cu : use 32 bit evaluation of constant expressions
-cxc : do not search current directory for include files

-CF[<file>] : produce COFF (.cof) debugging file, C mode
-CC[<file>] : produce COD debugging file, C mode
-CA[<file>] : produce COD debugging file, ASM mode
-Ce : remove extra byte names (nnn_e<N>) from COD file

-dc : do not write compiler output file <src>.occ

-D<name>[<token>xxx] : define macro. Equivalent to #define name xxx

-e : single line error messages (no source lines are printed).
-ed : do not print error details
-ew : do not print warning details
-eL : list error and warning details at the end
-E<N> : stop after <N> errors (default is 4).

-f<hex-file-format> : i.e. INHX8M, INHX8S, INHX16, INHX32. Default is INHX32 for enhanced 14 bit devices, otherwise INHX8M. Note that INHX8S uses two output files: <file>.HXH and <file>.HXL

-F : generate error file *.err
-FM : MPLAB and MPLAB X compatible error format
-FSRm : enable constant indirect access FSRx[k] outside range -32,31

-g : do not replace call by goto
-gb : replace goto by branch always

-GW : dynamic selected skip() format, warning on long format (default)
-GD : dynamic selected skip() format
-GS : always short skip() format (error if boundary is crossed)
-GL : always long skip() format

-I<directory> : include files directory/folder. Up to 5 library directories can be supplied by using separate -I<dir> options. When using #include "test.h" the current directory is first searched. If the file is not found there, then the library directories are searched, in the same order as supplied in the command line option list (-I<dir>). The current directory is skipped when using #include <test.h>.

-j : do not update page selection bits
12 bit core: STATUS.PA0 and PA1
14 bit core: PCLATH

-li<ENVI> : include directory from environment variable (default CCINC)
-lh<ENVD> : load default directory from environment variable (default CCHOME)

-L[<col>,<lin>] : generate list file <src>.lst
The maximum number of columns per line <col> and lines per page <lin> can be changed. The default setting is -L80,60
-Ln : produce list file with no page formatting

-mc1 : default 'const' pointer size is 1 byte (8 bits)
-mc2 : default 'const' pointer size is 2 bytes (16 bits)
-mr1 : default RAM pointer size is 1 byte
-mr2 : default RAM pointer size is 2 bytes
-mm1 : default pointer size is 1 byte (all pointer types)
-mm2 : default pointer size is 2 bytes (all pointer types)

-Ma : truncate all automatic generated labels in the assembly/list files

-o<file> : write hex file to <file>

-O<folder> : output files folder. Files generated by the compiler are put on this folder, except when a full path name is supplied.

-p<device> : defines the chip type. The device has to be known internally: 16C54,55,56,57,58, 61,64,65, 71,73,74, 84 or supported by a header file (e.g., 16F877.H). Default device is 16C54.

-p- : clear any preceding option -p<chip> to allow chip redefinition

-q<N> : assume disabled interrupt at the <N> deepest call levels. For example, –q1 allows the main program to use all stack levels for function calls. Disabling interrupts at the deepest call level MUST then be properly ensured in the user application program.

-Q : generate call tree file (*.fcs).

-r : generate relocatable assembly (no hex file)
-r2[=][<filename.lkr>] : generate relocatable asm, use separate logical section for interrupt routine
-rb<N> : name on RAM bank 0 is BANK<N>, default BANK0
-ro<N> : add offset <N> when generating local variable block name
-rp<N> : name on codepage 0 is PROG<N>, default PROG1
-rx : make variables static by default

-S : silent operation of the compiler

-u : no optimizing

-V[rnuDGg] : generate variable file, <src>.var, sorted by address as default.
r: only variables which are referenced in the code
n: sort by name
u: unsorted
D: decimal numbers
G: list default config settings and alternatives
g: list config setting alternatives

-wB : warning when function is called from another code page
-wC : warning on upward compatibility issues
-we : disable warning when fixed point constants are rounded
-wf : disable warning for read-modify-write sequences on the same PORT
-wi : disable warning on multiple inline math integer operations
-wL : (12 bit core only) print warning on all GOTO links to functions residing on hidden half of a codepage.
-wm : disable warning on single call to math integer function
-wO : warning on operator library calls
-wP : warning when code page bits are not correct
-wr : disable warning on recursive calls
-wS : warning (no error) when constant expression loses significant bits
-wU : warning on uncalled functions
-wx : disable warning on "suspicious pointer conversion"
-wz : disable warning on "incompatible and nonportable pointer conversion"

-W : wait until key pressed after compilation

-x<file> : assembler executable: -x"C:\Program Files\Microchip\MPASM Suite\mpasmwin.exe"
-X<option> : assembler option: -X/q (all options must be separate)

Doublequotes " " allows spaces in the option :

```
-I"C:\Program Files\cc5x"
```

A path name can be written using '/' if this is supported by the file system, for example:

```
c:/compiler/lib/file.h
```

Default compiler settings:
- hex file output to file <name>.hex
- processor = 16C54
- optimizing on
- extended call level is allowed
- automatic update of bank and page selection bits

Permanent assigned settings:
- nested comments is allowed
- char is unsigned

5.1 Options in a file

Options can be put in a file. The syntax is:

```
cc5x [..] +<filename> [..]
```

Many option files can be included, and up to 5 levels of nested include files are allowed. Options in a file allow an unlimited number of options to be stated. Linefeed, space and TAB separates each option. Comments can be added in the option file using the syntax:

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

Spaces can be added to each option if a space is added after the '-' starting the option. This syntax disables using more than one option on each line. Examples:

```
- D MAC = 1 + OP
- p 16C54      // comment
-p 16C54       // this will not work
- p 15C64  -a  // not this either
```

Note that the file path is required if the file does not reside on the current directory.

String translation rules for options in a file:
1. Doublequotes " " allow spaces in the option; quotes are removed
2. Using \" means a single quote " in an option

```
-I"C:\Program Files\cc5x"      ==>   -IC:\Program Files\cc5x
-IC:"\Program Files"\cc5x      ==>   -IC:\Program Files\cc5x

-DMyString="\"Hello\n\""       ==>   -DMyString="Hello\n"
-DQuote='\\"'                  ==>   -DQuote='\"'
```

5.2 Automatic incrementing version number in a file

The compiler is able to automatically increment one or more version numbers for each compilation. Three different syntax alternatives are available.

```
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>
```

Note that the command line option is required to make this increment happen. It is the decimal number found at end of the included file that is incremented. The updated file is written back before the file is compiled. No special syntax is assumed in the version file. Suggestions:

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

If the decimal number is 99, then the new number will be 100 and the file length increases by 1. If the number is 099, then the file length remains the same. A version file should not be too large (up to 20k), otherwise an error is printed.

Formats 2 and 3 above allow more than one version file. It is recommended to use conditional compilation to manage several editions of the same program.

5.3 Environment Variables

Environment variables can be used to define include folders and primary folder:

The variable CCINC is an alternative to the -I<path> option. The compiler will only read this variable (or specified variable) when using the following command line option:

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

Variable CCHOME can be used to define the primary folder during compilation. The compiler will only read this variable (or specified variable) when using the following command line option:

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

6 PROGRAM CODE

6.1 Program Code Pages

Many of the PICmicro devices have more than one code page. A code page contains 512 words on the 12 bit core and 2048 words on the 14 bit core. Using more than one code page requires code page selections. All functions following a *#pragma codepage* statement are put on the page specified. Codepage 0 is used as default.

```
/* 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 */
```

When switching between codepages, the compiler will keep track on the next free location on each codepage. Use of codepages is just a matter of optimization, as long as the compiler accepts the selection. The optimal combination requires least code (or has a speed advantage). The optimizer removes unnecessary setting and clearing of the page selection bits.

Some of the PICmicro devices have 4 code pages. Note that calls which requires switching between page 0 and 3, or page 1 and 2 requires more instructions than the other combinations. The enhanced 14 bit core requires only one instruction for all code page switching.

The compiler produces an error message when page limits are exceeded. Invalid code pages are mapped to valid ones.

Another way of locating functions

The statement *#pragma location* is capable of locating prototypes on codepages as well as function definitions. The statement is useful when locating functions defined in library files, or when locating functions in large programs. Its normal use is in limited regions in header files. The rules when using *#pragma location* are:

1. A function prototype will locate the function on the desired codepage, even if the current active codepage is different when the function definition is compiled.
2. *#pragma location* has higher priority than *#pragma codepage*.
3. '*#pragma location* -' restores the active codepage defined by the last *#pragma codepage* (or *#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
```

Notes:

1. The location statements have to be compiled before the function definition
2. Functions not located are placed on the current active codepage
3. A warning is printed in case of conflicts

The *#pragma location* statement should only be used if required. An example is when functions inside a module (file) have to be placed on different codepages, or if much tuning is required to find the optimal combination. The *#pragma codepage* statement is normally sufficient.

The page type modifier

The page type modifiers page0 .. page15 can replace #pragma location/codepage.

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

The page type modifier defines the codepage to locate the function in, both for function prototypes and function definitions.

NOTE 1: The page type modifier has higher priority than both #pragma codepage and #pragma location.

NOTE 2: When the codepage have been defined using the page type modifier or #pragma location, then the location is fixed and cannot be changed in the function definition or by using a second prototype.

Invalid code pages are mapped to valid ones.

Page selection bits

The page selection bits are automatically updated by the compiler, and attempts to set or clear these bits in the source code are removed by the optimizer. This can be switched off by the -j command line option.

Core 12 note: assigning a value to the status register (f3) may cause the automatic updating to fail.

6.2 Subroutine Call Level Checking

Subroutine calls are limited to 2 levels for the 12 bit core and 8 levels for the 14 bit core. The compiler **automatically** checks that this limit is not exceeded.

The compiler can replace CALL by GOTO to seemingly achieve deeper call levels.

1. When a function is called once only, the CALL can be replaced by a GOTO. All corresponding returns are replaced by GOTO. Note that the call will only be replaced by GOTO when the call level must be reduced. Also, the CALL is NOT replaced by GOTO when:
   a) The program counter (PCL) is manipulated (computed goto) in a function of type *char*.
   b) The number of return literals exceeds 10
   c) The function is called from another codepage and the number of returns exceeds 10

2. Call followed by return is replaced by a single goto.

When subroutines are located in the second half of a codepage, it cannot be called directly when using 12 bit core devices. The compiler automatically inserts links to such "hidden" subroutines.

Stack level checking when using interrupt

CC5X will normally assume that an interrupt can occur anywhere in the main program, and also at the deepest call level. An error message is printed if stack overflow may occur. This is not always true, because the interrupt enable bits controls when interrupts are allowed. Sometimes the main program needs all 8 stack levels for making calls.

The -q<N> option forces CC5X to assume that an interrupt will NOT occur at the <N> deepest call levels of the main program.

The application writer must then ensure that interrupts will not occur when executing functions at the deepest <N> call levels, normally by using the global interrupt enable bit. CC5X will generate a warning for the critical functions. (The normal error message is always generated when the application contains more than 8 call levels.)

For example, the -q1 option generates a warning for functions calls that will put the return address at stack level 8 (no free stack entry for interrupt). Using -q2 means an additional warning at stack level 7 will be generated if the interrupt routine requires 2 levels, i.e. contains function calls.

It is NOT recommended to use the -q<N> as a default option.

Functions shared between independent call trees

An error message is normally printed when the compiler detects functions that are called both from main() and during interrupt processing if this function contains local variables or parameters. This also applies to math library calls and const access functions. The reason for the error is that local variables are allocated statically and may be overwritten if the routine is interrupted and then called during interrupt processing.

The error message will be changed to a warning by the following pragma statement. Note that this means that local variable and parameter overwriting must be avoided by careful code writing.

```
#pragma sharedAllocation
```

Recursive functions

Recursive functions are possible. Please note that the termination condition has to be defined in the application code, and therefore the call level checking cannot be done by the compiler. Also note that the compiler does not allow any local variables in recursive functions. Function parameters and local variables can be handled by writing code that emulates a stack.

A warning is printed when the compiler detects that a function calls itself directly or through another function. This warning can be switched off with the *-wr* command line option.

6.3 Interrupts

The 14 bit core allows interrupts. The structure of the interrupt service routine is as follows:

```
#include "int16CXX.h"
#pragma origin 4

interrupt serverX(void)
{
    /* W and STATUS are saved by the next macro. PCLATH is also
       saved if necessary. The code produced is CPU-dependent. */

    /* Note that the Enhanced 14 bit core has hardware register
       save and restore of W, STATUS, BSR, FSRx and PCLATH */

    int_save_registers    // W, STATUS (and PCLATH)
    //char sv_FSR = FSR;  // if required

    // handle the interrupt
```

```
    //FSR = sv_FSR;       // if required
    int_restore_registers // W, STATUS (and PCLATH)
}

/* IMPORTANT : GIE should normally NOT be set or cleared in
   the interrupt routine. GIE is AUTOMATICALLY cleared on
   interrupt entry by the CPU and set to 1 on exit (by
   RETFIE). Setting GIE to 1 inside the interrupt service
   routine will cause nested interrupts if an interrupt is
   pending. Too deep nesting may crash the program! */

/* IMPORTANT : The register save style (i.e. INT_xxx_style) is
   defined in the chip header file (i.e. 16F877.h) and should
   NOT be defined in the application. Wrong register save style
   may cause strange problems and is very difficult to trace. */
```

The keyword *interrupt* allows the routine to be terminated by a RETFIE instruction. It is possible to call a function from the interrupt routine (it has to be defined by a prototype function definition first).

The interrupt routine requires at least one free stack location because the return address is pushed on the stack. This is automatically checked by the compiler. Even function calls from the interrupt routine are checked. However, if the program contains recursive functions, then the call level cannot be checked by the compiler.

The interrupt vector is permanently set to address 4. The interrupt service routine can only be located at this address. The #pragma origin statement has to be used in order to skip unused program locations.

The following pragma statement will allow the interrupt routine to be placed anywhere. Note that the compiler will NOT generate the link from address 4 to the interrupt routine.

```
#pragma unlockISR
```

Vital registers such as STATUS and W should be saved and restored by the interrupt routine. However, registers that are not modified by the interrupt routine do not have to be saved. Saving and restoring registers is device dependent. The file *int16CXX.H* contains recommended program sequences for saving and restoring registers. The interrupt routine can also contain local variables. Storage for local variables is allocated separately because interrupts can occur anytime.

CC5X also supports CUSTOM save and restore sequences. If you want to use your own register save and restore during interrupt, please read the following Section *Custom interrupt save and restore*.

IMPORTANT: CC5X will AUTOMATICALLY check that the registers W, STATUS, PCLATH and FSR are saved and restored during interrupt.

Note that register save checking does not apply to the enhanced 14 bit core.

The compiler will detect if the FSR register is modified during interrupt processing without being saved and restored. The supplied macros for saving and restoring registers will not save FSR. This has to be done by user code when needed. If FSR is modified by a table or pointer access, or by direct writing, the compiler will check that FSR is saved and restored, also in nested function calls. Note that the FSR saving and restoring can be done in a local region surrounding the indexed access, and does not need to be done in the beginning and end of the interrupt routine.

A warning is printed if FSR is saved but not changed. The error and warning messages printed can be removed:

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

Note that the above pragma changes the checking done on all registers.

Custom interrupt save and restore

It is not required to use the above save and restore macros. CC5X also supports custom interrupt structures. This is not interesting for the enhanced 14 bit core because of its hardware register saving.

A) You might want to use your own save and restore sequence. This can be done by inline assembly. If CC5X does not accept your code, just insert (at your own risk):

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

B) No registers need to be saved when using the following instructions in the interrupt routine. The register save checking should NOT be disabled.

```
btss(bx1);      // BTFSS 0x70,bx1  ; unbanked RAM/SFR only
bx2 = 1;        // BSF   0x70,bx2  ; unbanked RAM/SFR only
bx1 = 0;        // BCF   0x70,bx1  ; unbanked RAM/SFR only
btsc(bx1);      // BTFSC 0x70,bx1  ; unbanked RAM/SFR only
sleep();        // SLEEP
vs = swap(vs);  // SWAPF vs,1      ; unbanked RAM/SFR only
vs = incsz(vs); // INCFSZ vs,1     ; unbanked RAM/SFR only
nop();          // NOP
vs = decsz(vs); // DECFSZ vs,1     ; unbanked RAM/SFR only
clrwdt();       // CLRWDT
```

C) It is possible to enable interrupt only in special regions (wait loops) in such a way that W, STATUS, PCLATH and FSR can be modified during interrupt without disturbing the main program. Note that interrupt must ONLY be enabled in codepage 0 when PCLATH is not saved. The register save can then be omitted and the save checking must be switched off to avoid error messages:

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

INTERRUPTS CAN BE VERY DIFFICULT. THE PITFALLS ARE MANY.

6.4 Startup and Termination Code

The startup code consists of a jump to *main()* which has to be located on page zero. No variables are initiated. All initialization has to be done by user code. This simplifies design when using the watchdog timer or MCLR pin for wakeup purposes.

The SLEEP instruction is executed when the processor exits *main().* This stops program execution and the chip enters low power mode. Program execution may be restarted by a watchdog timer timeout or a low state on the MCLR pin.

The 14 bit core also allows restart by interrupt. An extra GOTO is therefore inserted if main is allowed to terminate (SLEEP). This ensures repeated execution of the main program. No extra GOTO is added when a sleep() command is inserted anywhere else in the application program.

Clearing ALL RAM locations

The internal function *clearRAM()* will set all RAM locations to zero. The generated code uses the FSR (FSR0) register. The recommended usage is:

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

The code size and timing depends on the actual chip. The following table describes the basic types. Chip devices not found here maps to one of the described types.

INS	ICYC	TOTCYC	4MHz	RAM	START	LAST	BANKS	PICmicro
8	6	145	0.15ms	25	7	0x1F	-	16C54
9	4	202	0.20ms	41	7	0x3F	2	16C509
13	4	290	0.29ms	72	8	0x7F	4	16C57
8	7	254	0.25ms	36	12	0x2F	-	16C84
6	5	482	0.48ms	96	32	0x7F	1	16C620A
12	5	644	0.64ms	128	32	0xBF	2	12C671
9	4	770	0.77ms	176	32	0xFF	2	16C642
9	4	770	0.77ms	192	32	0xFF	2	16C74
10	4	771	0.77ms	192	32	0xFF	4	16C923
19	5	1110	1.11ms	224	32	0x14F	4	16C627
12	4	1058	1.06ms	256	32	0x17F	4	16C773
15	4	1807	1.81ms	368	32	0x1FF	4	16C77

INS: number of assembly instructions required
ICYC: cycles (4*clock) for each RAM location cleared
TOTCYC: total number of cycles (4*clock) required
4MHz: approx. time in milliseconds required at 4 MHz
RAM: total number of RAM locations
START: first RAM address
LAST: last RAM address
BANKS: number of RAM banks
PICmicro: chip type described

6.5 Library Support

The library support includes **standard math** and support for **user defined libraries**. The library files should be included in the beginning of the application, but after the interrupt routine for all libraries located on codepage 0.

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

CC5X will automatically delete unused library functions. This feature can also be used to delete unused application functions:

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

Math libraries

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

All libraries are optimized to get compact code. All variables (except for the floating point flags) are allocated on the generated stack to enable efficient RAM reuse with other local variables. A new concept of transparent sharing of parameters in a library is introduced to save code.

Note that fixed point requires manual worst case analysis to get correct results. This must include calculation of accumulated error and avoiding truncation and loss of significant bits. It is often straightforward to get correct results when using floating point. However, floating point functions require significantly more code. In general, floating point and fixed point are both slow to execute. Floating point is FASTER than fixed point on multiplication and division, but slower on most other operations.

Operations not found in the libraries are handled by the built in code generator. Also, the compiler will use inline code for operations that are most efficiently handled inline.

The following command line options are available:
-we : no warning when fixed point constants are rounded
-wO : warning on operator library calls
-wi : no warning on multiple inline math integer operations
-wm : no warning on single call to math integer function

Integer libraries

The math integer libraries allow selection between different optimizations, speed or size. The libraries contain operations for multiplication, division and division remainder.

```
math16.h  : basic library, up to 16 bit
math24.h  : basic library, up to 24 bit
math32.h  : basic library, up to 32 bit

math16m.h: speed, size, 8*8, 16*16
math24m.h: speed, size, 8*8, 16*16, and 24*8 multiply
math32m.h: speed, size, 8*8, 16*16, and 32*8 multiply

The math??m.h libraries can be used when execution speed is critical.
NOTE 1: they must be included first (before math??.h)
NOTE 2: math??.h contains similar functions (which are deleted)

The min and max timing cycles are approximate only. The enhanced 14
bit core will use fewer cycles and less code.


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
ABC  -   16 = 8 * 8       13       83    83    83
ABC  S   16 = 8 * 8       21       85    85    85
ABC  S/- 16 = 16 * 16     18      197   222   277
.B.  S   24 = 16 * 16     35      220   261   334
```

```
A..  S   32 = 16 * 16     42      223   253   313
A..  -   32 = 16 * 16     22      215   240   295
AB.  -   24 = 16 * 8      15      198   198   198
..C  -   16 = 16 * 8      16      179   179   179
.B.  -   24 = 24 * 8      16      247   247   247
A..  -   32 = 32 * 8      17      356   356   356
.B.  -   24 = 24 * 16     26      217   263   361
A..  -   32 = 32 * 16     31      239   310   447
.B.  -   24 = 24 * 24     25      337   410   553
A..  S/- 32 = 32 * 32     31      513   654   929

ABC  -   16 = 16 / 8      18      235   235   235
AB.  -   24 = 24 / 8      19      368   368   368
A..  -   32 = 32 / 8      20      517   517   517
ABC  -   16 = 16 / 16     25      287   291   335
.B.  -   24 = 24 / 16     31      481   512   633
A..  -   32 = 32 / 16     32      665   718   873
.B.  -   24 = 24 / 24     36      564   576   732
A..  -   32 = 32 / 32     47      943   966  1295

ABC  S   16 = 16 / 8      33      196   201   211
AB.  S   24 = 24 / 8      37      305   310   326
A..  S   32 = 32 / 8      41      430   436   457
ABC  S   16 = 16 / 16     49      296   309   361
.B.  S   24 = 24 / 16     53      450   473   543
A..  S   32 = 32 / 16     57      626   660   747
.B.  S   24 = 24 / 24     66      573   597   762
A..  S   32 = 32 / 32     83      952   990  1329

ABC  -    8 = 16 % 8      18      226   226   226
.B.  -    8 = 24 % 8      19      354   354   354
A..  -    8 = 32 % 8      20      502   502   502
ABC  -   16 = 16 % 16     23      280   283   312
.B.  -   16 = 24 % 16     29      463   497   599
A..  -   16 = 32 % 16     30      636   698   828
.B.  -   24 = 24 % 24     34      556   567   700
A..  -   32 = 32 % 32     45      934   955  1254

ABC  S    8 = 16 % 8      30      189   190   195
.B.  S    8 = 24 % 8      35      291   292   300
A..  S    8 = 32 % 8      39      413   415   425
ABC  S   16 = 16 % 16     46      290   297   332
.B.  S   16 = 24 % 16     50      442   455   501
A..  S   16 = 32 % 16     54      614   634   692
.B.  S   24 = 24 % 24     66      567   584   725
A..  S   32 = 32 % 32     86      944   974  1284


A:math32m.h
 B:math24m.h
  C:math16m.h            Code     min  aver   max
ABC  -   16 = 8 * 8       37       50    50    50
ABC  S/- 16 = 16 * 16    23+37     74   147   158
.B.  -   24 = 24 * 8     32+37    124   162   166
A..  -   32 = 32 * 8     43+37    178   212   222
```

Fixed point libraries

math16x.h : 16 bit fixed point, 8_8, signed and unsigned
math24x.h : 24 bit fixed point 8_16, 16_8, signed and unsigned
math32x.h : 32 bit fixed point 8_24, 16_16, 24_8, signed and unsigned

The libraries can be used separately or combined.

The timing stated is measured in instruction cycles (4*clock) and includes parameter transfer, call, return and assignment of the return value. The min and max timing cycles are approximate only. The enhanced 14 bit core will use fewer cycles and less code.

```
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      47      226   263   339
-   8_8 = 8_8 * 8_8      23      214   252   326
S   8_8 = 8_8 / 8_8      51      497   518   584
-   8_8 = 8_8 / 8_8      35      528   558   680

math24x.h:              Code     min  aver   max
S   16_8 = 16_8 * 16_8   60      376   450   577
-   16_8 = 16_8 * 16_8   27      364   437   580
S   16_8 = 16_8 / 16_8   68      850   893  1093
-   16_8 = 16_8 / 16_8   46      894   944  1222

S   8_16 = 8_16 * 8_16   60      354   428   555
-   8_16 = 8_16 * 8_16   28      342   415   558
S   8_16 = 8_16 / 8_16   68     1050  1116  1349
-   8_16 = 8_16 / 8_16   46     1104  1188  1520

math32x.h:              Code     min  aver   max
S   24_8 = 24_8 * 24_8   77      558   722   983
-   24_8 = 24_8 * 24_8   35      546   709  1026
S   24_8 = 24_8 / 24_8   85     1298  1366  1761
-   24_8 = 24_8 / 24_8   57     1361  1432  1929

S   16_16= 16_16*16_16   78      561   704   930
-   16_16= 16_16*16_16   36      549   690   965
S   16_16= 16_16/16_16   85     1546  1650  2097
-   16_16= 16_16/16_16   57     1617  1733  2305

S   8_24 = 8_24 * 8_24   77      529   672   896
-   8_24 = 8_24 * 8_24   35      517   658   933
S   8_24 = 8_24 / 8_24   85     1794  1936  2433
-   8_24 = 8_24 / 8_24   57     1872  2033  2680
```

Floating point libraries

```
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
```

NOTE: The timing values include parameter transfer, call and return and also assignment of the return value. The min and max timing cycles are approximate only. The enhanced 14 bit core will use fewer cycles and less code.

```
Basic 32 bit math:              Approx. CYCLES
                         Size   min  aver   max
a * b:  multiplication    91   380   468   553
a / b:  division         125   523   610   742
a + b:  addition         182    39   135   225
a - b:  subtraction    add+5    46   142   232
int32 -> float32          79    45    69   118
float32 -> int32          86    36    77   143

Basic 24 bit math:              Approx. CYCLES
                         Size   min  aver   max
a * b:  multiplication    77   226   261   294
a / b:  division         102   323   359   427
a + b:  addition         152    33   114   173
a - b:  subtraction    add+5    40   121   180
int24 -> float24          62    36    64   106
float24 -> int24          74    31    72   117

Basic 16 bit math:              Approx. CYCLES
                         Size   min  aver   max
a * b:  multiplication    62   104   107   114
a / b:  division          82   137   154   171
a + b:  addition         118    27    86   130
a - b:  subtraction    add+5    34    93   137
int16 -> float16          72    40    71   107
float16 -> int16          53    26    60    98
```

The following operations are handled by inline code: assignment, comparison with constants, multiplication and division by a multiple of 2 (e.g., a*0.5, b * 1024.0, c/4.0).

Floating point library functions

```
float24 sqrt(float24);   // square root
  Input range: positive numbers including zero
  Accuracy: ME:1, relative error: 1.5*10**-5 (*)
  Timing: min aver max   645   700   758  (**)
  Size: 62 words
  Minimum complete program example: 77 words

float32 sqrt(float32);   // square root
  Input range: positive numbers including zero
  Accuracy: ME:1, relative error: 6*10**-8 (*)
  Timing: min aver max  1174  1303  1415  (**)
  Size: 76 words
  Minimum complete program example: 97 words

float24 log(float24);   // natural log function
  Input range: positive numbers above zero
  Accuracy: ME:1, relative error: < 1.5*10**-5 (*)
  Timing: min aver max  2179  3075  3299  (**)
  Size: 214 words + basic 24 bit math library
  Minimum complete program example: 623 words
```

```
float32 log(float32);   // natural log function
  Input range: positive numbers above zero
  Accuracy: ME:1, relative error: < 6*10**-8 (*)
  Timing: min aver max  3493  4766  5145  (**)
  Size: 265 words + basic 32 bit math library
  Minimum complete program example: 762 words

float24 log10(float24);   // log10 function
  Input range: positive numbers above zero
  Accuracy: ME:1-2, relative error: < 3*10**-5 (*)
  Timing: min aver max  2435  3333  3569  (**)
  Size: 15 words + size of log()
  Minimum complete program example: 638 words

float32 log10(float32);   // log10 function
  Input range: positive numbers above zero
  Accuracy: ME:1-2, relative error: < 1.2*10**-7 (*)
  Timing: min aver max  3935  5229  5586  (**)
  Size: 17 words + size of log()
  Minimum complete program example: 779 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  1969  3026  3264  (**)
  Size: 251 words + 102(floor24) + basic 24 bit math
  Minimum complete program example: 673 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  4465  4741  5134  (**)
  Size: 322 words + 145(floor32) + basic 32 bit math
  Minimum complete program example: 847 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  1987  3005  3194  (**)
  Size: 256 words + 102(floor24) + basic 24 bit math
  Minimum complete program example: 678 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  3605  4716  5045  (**)
  Size: 326 words + 145(floor32) + basic 32 bit math
  Minimum complete program example: 851 words

float24 sin(float24);   // sine, input in radians
float24 cos(float24);   // cosine, 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   396  2492  2746  (**)
   Size: 215 words + basic 24 bit math library
   Minimum complete program example: 595 words

 float32 sin(float32);   // sine, input in radians
 float32 cos(float32);   // cosine, 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   543  5220  5855  (**)
   Size: 357 words + basic 32 bit math library
   Minimum complete program example: 818 words
```

(*) The accuracy of the math functions have been checked using many thousands of calculations. ME=1 means that the mantissa value can be wrong by +/- 1 (i.e. 1 bit). The relative error is then $1.5*10^{-5}$ for 24 bit floating point, and $6*10^{-8}$ for 32 bit floating point. Only a small fraction of the calculations may have the stated error.

(**) The min and max timing cycles are approximate only. The enhanced 14 bit core will use fewer cycles and less code. All timing is measured in instruction cycles. When using a 4 MHz oscillator, one instruction cycle is 1 microsecond.

Fast and compact inline operations

The compiler will use inline code for efficiency at some important operations:

Integer:
- converting to left and right shifts: a * 8, a / 2
- selecting high/low bytes/words: a / 256, a % 256, b % 0x10000
- replacing remainder by AND operation: a % 64, a % 0x80

Fixed Point:
- converting to left and right shifts: a * 8, a / 2
- all operations except multiplication and division are implemented inline

Floating point:
- add/sub (incr/decr) of exponent: a * 128.0, a / 2
- operations == and != : a == b, a != 0.0
- comparing with constants: a > 0, a <= 10.0
- inverting the sign bit: a = -a, b = -a

Combining inline integer math and library calls

It is possible to force the compiler to generate inline integer math code after a math library is included. This may be useful when speed is critical or in the interrupt service routine. Functions with parameters or local variables are not reentrant because local variables are mapped to global addresses, and therefore the compiler will not allow calls from both main and the interrupt service routine to the same function.

```
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 type modifier on math operations

It is possible to combine inline integer math and math library functions without making a special purpose math library. This is done by stating that the selected operations are inline BEFORE the standard math library is included. It is optimal to use inline code when there is only one operation of a certain type.

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

The math prototypes are found in the beginning of the standard math libraries. Just remember to remove the operator name before adding the inline type modifier.

A warning is printed when there is ONE call to a unsigned integer math library function. The warning can be disabled by the -wm command line option.

NOTE that the inline type modifier is currently IGNORED, except for the math operations.

Detection of multiple inline math integer operations

The compiler will print a warning when detecting more than one inline math integer operation of the same type. Including a math library will save code, but execute slightly slower. Note that assembly code inspection and type casts are sometimes needed to reduce the number of library functions inserted.

The warning can be disabled by the -wi command line option.

Using prototypes and multiple code pages

Using floating point on the 12 bit core where each codepage is 512 words will be challenging. It is normally not required to define prototypes, even when using many code pages. If you want to place the library functions to a certain codepage, this is easiest done by:

```
#pragma codepage 1
#include "math24f.h"
#pragma codepage 0
```

Prototypes can be used when functions are called before they are defined. Note that operator functions need a function name to be defined as prototypes.

Also note that the compiler use a linear search from the beginning of the operator table until a match for the current operation is found. The operator definition/prototype sequence may therefore influence the operator selected.

It is not recommended to modify the libraries by adding 'pragma codepage' between the functions. Instead, prototypes and 'pragma location' or the page type modifier makes function placement easier to set up and tune. For example, placing the division function on codepage 1 and the other on the default codepage can be done by:

```
..
#include "my_math.h"
..
#include "math24f.h"
..

//--------- beginning-of-file my_math.h --------------
float24 operator* _fmul24(sharedM float24 arg1, sharedM float24
arg2);
page1 float24 operator/ _fdiv24(sharedM float24 arg1, sharedM float24
arg2);
float24 operator+ _fadd24(sharedM float24 arg1, sharedM float24
arg2);
// .. etc.
//--------- end-of-file my_math.h --------------
```

Fixed point example

```
#pragma chip PIC16C73
#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
}
```

CODE: 274 instructions including library (130).

Floating point example

CODE: 635 instructions including library (470). The statements are identical to the above fixed point example to enable code size comparison.

```
#pragma chip PIC16C73
#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);
}
```

How to save code

Choices that influence code size:

1. What libraries to include (24/32 bit float or fixed point)

2. Rounding can be disabled permanently.

```
#define DISABLE_ROUNDING
#include "math32f.h"
```

3. Optimization, currently available on division only. Note that “optimize for speed” is default. Also note that the saving is only 5 - 7 instructions. Timing difference is up to 15-20 percent.

```
#define FP_OPTIM_SIZE   // optimize for SIZE
#define FP_OPTIM_SPEED  // optimize for SPEED: default
```

The recommended strategy is to select a main library for the demanding math operations. Different floating and fixed point operations should only be mixed if there is a good reason for it.

Mixing different data types is possible to save code and RAM space. For example by using a small type in an array and a larger type for the math operations.

So, first decide what math library to include. For floating point the main decision is between the 24 bit and the 32 bit library. If you use 32 bit operations, this can be combined with 24 (and 16) bit floating point types to save RAM.

Automatic type conversion:

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

In general, using the smallest possible data type will save code and RAM space. This must be balanced against the extra work to analyze the program to prevent overflow and too large accumulated errors. If there is plenty of code space in the device, and timing is no problem, then large types can be used. Otherwise analysis is required to get optimal selections.

It is recommended to keep the number of called library functions as low as possible. Although function selection is done automatically by the compiler, it is possible to use type casts or even make a custom library by copying the required functions from existing libraries. All libraries are written in C. CC5X can print a warning for each operator function that is called (option -wO).

6.6 Inline Assembly

The CC5X compiler supports inline assembly located inside a C function. There are some restrictions compared to general assembly. First, it is only possible to CALL other functions. Second, GOTO is

restricted to labels inside the function. If these restrictions make program design too difficult, consider using the linker support and link C and assembly modules using MPLINK.

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

Features:

- many assembly formats
- equ statements can be converted to variable definitions
- macro and conditional assembly capabilities
- call C functions and access C variables
- C style comments is possible
- optional optimization
- optional automatic bank and page updating

Note that the file *inline.h* is for emulating inline assembly, and should NOT be included when using real inline assembly. The compiler does not optimize inline assembly or update the bank or page bits unless it is instructed to do so.

Inline assembly is NOT C statements, but are executed in between the C statements. It is not recommended to write the code like this:

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

Inline assembly supports DW. This can be used to insert data or special instructions. CC5X will assume that the data inserted are instructions, but will not interpret or know the action performed. Bank selection bits are assumed to be undefined when finishing executing DW instructions. PCLATH bit 3 (and 4) must remain unchanged or restored to correct value if more than one code page is available on the device. Example use is found in file '**startup.txt**'.

```
#asm
 DW 0x3FFF   ; any data or instruction
 DW /*CALL*/ 0x2000 + ADDRESS
#endasm
```

Assembly instructions are not case sensitive. However, variables and symbols require the right lower or upper case on each letter.

```
clrw
Nop
NOP
```

The supported operand formats are:

```
k      EXPR
f      VAR + EXPR
f,d    VAR + EXPR, D
f,b    VAR + EXPR, EXPR
a      LABEL or FUNCTION_NAME

EXPR :=  [  EXPR OP EXPR  |  (EXPR)  |  -EXPR  ]
EXPR := a valid C constant expression, plus assembly extensions
```

Constant formats:

```
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
```

Formats when loading then result into the W register:

```
decf  ax,0
iorwf ax,w
iorwf ax,W
```

Formats when writing the result back to the RAM register:

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

Bit variables are accessed by the following formats:

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

Arrays, structures and variables larger than 1 byte can be accessed by using an offset.

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

Labels can start anywhere on the line:

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

Functions are called directly. A single unsigned 8 bit parameter can be transferred using the W register.

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

The ONLY way to transfer multiple parameters (and parameters different from 8 bit) is to end assembly mode, use C syntax and restart assembly mode again.

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

The enhanced 14 bit core allows more instructions and new formats:

```
ADDWFC   i,W
SUBWFB   i,1
LSLF     i,0
LSRF     i
ASRF     i,W
MOVLB    1
MOVLP    1

RESET

ADDFSR INDF0,31
ADDFSR INDF1,-1

MOVWI ++INDF0
MOVIW INDF0--

MOVIW 1[INDF0]
MOVIW -2[INDF1]
```

Some instructions are disabled, depending on core type:

```
option        ; 12 bits core only
tris PORTA    ; 12 bits core only
movwf OPTION  ; 14 bits core only
movwf TRISA   ; 14 bits core only
```

The EQU statement can be used for defining constants. Assembly blocks containing EQU's only can be put outside the functions. Note that Equ constants can only be accessed in assembly mode. Constants defined by #define can be used both in C and assembly mode.

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

Equ can also be used to define variable addresses. However, the compiler does not know the difference between an Equ address and an Equ constant until it is used by an instruction. When an Equ symbol is used as a variable, that location is disabled for use by other variables. The symbol then changes from an Equ symbol to a variable symbol and is made available in C mode also. There is a slight danger in this logic. DO NOT USE a series of Equ's to define an array. If one of the locations are not read or written directly, the compiler will not know that it is a part of an array and may use it for other purposes. Reading and writing through FSR and INDF is not used to transform equ definitions. Therefore, define arrays by using C syntax (or #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
```

The following address operations are possible when the variable (structure/array) is set to a fixed address:

```
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 &STATUS
#endasm
```

Comments types allowed in assembly mode:

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

Conditional assembly is allowed. However, the C style syntax has to be used.

```
#ifdef SYMBOLA
nop
#else
clrw
#endif
```

C style macros can contain assembly instructions, and also conditional statements. Note that the compiler does not check the contents of a macro when it is defined.

```
#define UUA(a,b)\
clrw\
movlw a \
#if a == 10 \
nop        \
#endif      \
clrf b

UUA(10,ax)
UUA(9,PORTA)
```

Note that labels inside a macro often need to be supplied as a parameter if the macro is used more than once. Also note that there should always be a backslash '\' after a #endasm in a macro to avoid

error messages when this macro is expanded in the C code. This applies to all preprocessor statements inside a macro.

```
#define waitX(uSec, LBM)  \
 #asm         \
   LBM:       \
       NOP    \
       NOP    \
       DECFSZ uSec,1 \
       GOTO LBM \
 #endasm      \

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

Most preprocessor statements can be used in assembly mode:

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

The compiler can optimize and perform bank and page updating in assembly mode. This does not happen automatically, but has to be switched on in the source code. It is normally safe to switch on optimization and bank/page updating. Instructions updating the bank and page register are removed before the compiler insert new instructions. If the assembly contains critical timing, then the settings should be left off, at least in local regions.

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

#asm             // using default local settings
#endasm

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

Interpretation:

```
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
p+ : current page bit updating is performed in assembly mode
p- : no page bit update in assembly mode
```

Note that *b+ o+ p+* means that updating is performed if the current setting in C mode is on. Updating is NOT performed if it is switched off in the C code when assembly mode starts. The command line options *-b, -u, -j* will switch updating off globally. The corresponding source code settings are then ignored.

Direct coded instructions

The file “hexcodes.h” contains C macros that allow direct coding of instructions.

Note that direct coded instructions are different from inline assembly seen from the compiler. The compiler will view the instruction codes as values only and not as instructions. All high level properties are lost. The compiler will reset optimization, bank updating, etc. after a DW statement.

Example usage:

```
#include "hexcodes.h"
..
```

```
// 1. In DW statements:
#asm
DW __SLEEP                  // Enter sleep mode
DW __CLRF(__INDF)           // Clear indirectly
DW __ANDLW(0x80)            // W = W & 0x80;
DW __DECF(__FSR,__F)        // Decrement FSR
DW __BCF(__STATUS,__Carry)  // Clear Carry bit
DW __GOTO(0)                // Goto address 0
#endasm
..
// 2. In cdata statements:
#pragma cdata[1] = __GOTO(0x3FF)
```

Generating single instructions using C statements

The file *INLINE.H* describes how to emulate inline assembly using C code. This allows single instructions to be generated. Intended usage is mainly for code with critical timing.

The compiler will normally generate single instructions if the C statements are simple. Remember to inspect the generated assembly file if the application algorithm depends upon a precisely defined instruction sequence. The following example shows how to generate single instructions from C code.

```
nop();          // NOP
f = W;          // MOVWF f
W = 0;          // CLRW
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);      // RRF f,W
f = rr(f);      // RRF f
W = rl(f);      // RLF f,W
f = rl(f);      // RLF 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
btsc(b);        // BTFSC f,b
btss(b);        // BTFSS f,b
```

```
OPTION = W;     // OPTION (MOVWF on core 14)
sleep();        // SLEEP
clrwdt();       // CLRWDT
TRISA = W;      // TRIS f (MOVWF on core 14)
return 5;       // RETLW 5
s1();           // CALL s1
goto X;         // GOTO X
nop2();         // GOTO next address (delay 2 cycles)
W = 45;         // MOVLW 45
W = W | 23;     // IORLW 23
W = W & 53;     // ANDLW 53
W = W ^ 12;     // XORLW 12
```

Additional for the 14 bit core:

```
W = 33 + W;     // ADDLW 33
W = 33 - W;     // SUBLW 33
return;         // RETURN
retint();       // RETFIE
```

Additional for the enhanced 14 bit core:

```
W = addWFC(i); // ADDWFC f,W
i = addWFC(i); // ADDWFC f
W = subWFB(i); // SUBWFB f,W
i = subWFB(i); // SUBWFB f
W = i << 1;    // LSLF   f,W
i = i << 1;    // LSLF   f
i = lsl(i);    // LSLF   f
W = i >> 1;    // LSRF   f,W
i = i >> 1;    // LSRF   f
i = lsr(i);    // LSRF   f
int x;
x = x >> 1;    // ASRF   f,W
x = x >> 1;    // ASRF   f
x = asr(i);    // ASRF   f
BSR = 3;       // MOVLB  k
PCLATH = 5;    // MOVLP  k
skip(W);       // BRW
softReset();   // RESET
W = *FSR0++;   // MOVIW
W = *--FSR1;   // MOVIW
*FSR0-- = W;   // MOVWI
*++FSR1 = W;   // MOVWI
FSR0 += 31;    // ADDFSR n,k
```

6.7 Optimizing the Code

The CC5X compiler contains an advanced code generator which is designed to generate compact code. For example when comparing a 32 bit unsigned variable with a 32 bit constant, this normally requires 16 (or 15) instructions. When comparing a 32 bit variable with 0, this count is reduced to 6 (or 5). The code generator detects and takes advantage of similar situations to enable compact code.

Most of the code is generated inline, even multiplication and division. However, if many similar and demanding math operations have to be performed, then it is recommended to include a math library.

Optimized syntax

Bit toggle uses the W register to get compact code:

```
bit b;
b = !b;   // MOVLW K, XORWF var
```

Testing multiple bits of 16 bit variables or greater:

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

Testing single bits using the '&' operator:

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

Peephole optimization

Peephole optimizing is done in a separate compiler pass which removes superfluous instructions or rewrite the code by using other instructions. This optimization can be switched off by the *-u* command line option. The optimization steps are:

1) redirect goto to goto.
2) remove superfluous gotos.
3) replace goto by skip instructions.
4) replace INCF and DECF by INCFSZ and DECFSZ.
5) remove instructions that affects the zero- flag only.
6) remove superfluous updating of the page selection bits.
7) remove other superfluous instructions.
8) remove superfluous loading of the W register.

NOTE: Optimization can also be switched on or off in a local region. Please refer to the *#pragma optimize* statement for more details.

```
                        ;    while (1)  {
                        ;        if (Carry == 0) {
m001    BTFSC status,Carry
        GOTO m004       ; REDIRECTED TO m001 (1)
                        ;            i++;
        INCF i          ; REPLACED BY INCFSZ (4)
                        ;            if (i != 0)
        MOVF i          ; REMOVED (5)
        BTFSS status,Zero_ ; REMOVED (4)
        GOTO m002       ; REMOVED (3)
                        ;                var++;
        INCF var
                        ;            test += 2;
m002    MOVLW .2
        ADDWF test
                        ;            if (test == 0)
        MOVF test       ; REMOVED (5)
        BTFSS status,Zero_ ; REPLACED BY BTFSC (3)
        GOTO m003       ; REMOVED (3)
                        ;                break;
        GOTO m005
```

```
                        ;            W = var;
m003    MOVF var,W
                        ;            if (W == 0)
        XORLW .0        ; REMOVED (5)
        BTFSS status,Zero_
        GOTO m004       ; REDIRECTED TO m001 (1)
                        ;                break;
        GOTO m005       ; REMOVED (7)
m004    GOTO m001       ; REMOVED (2)
                        ;       sub1();
m005    BSF status,PA0
        CALL sub1
        BCF status,PA0  ; REMOVED (6)
                        ;       sub2();
        BSF status,PA0  ; REMOVED (6)
        BSF status,PA1
        CALL sub2
        BCF status,PA0
        BCF status,PA1
```

6.8 Linker Support

CC5X supports the relocatable assembly format defined by Microchip. This means that MPLINK can be used to link code modules generated by CC5X, including MPASM assembly modules. There are many details to be aware of. It is therefore recommended to read this file carefully. The important issues are related to:

- external functions and variables
- ram bank updating
- page bit updating
- call level checking
- MPLINK script files

The command line option '-r' (or ‘-r2’) makes CC5X generate relocatable assembly. This file is then assembled by MPASM and linked together with other C and assembly modules by MPLINK. This can automated by using 'make' to build the whole application in several stages.

NOTE that if you need the application program to be as compact as possible, then it is recommended to use only ONE C module. Source code modularity is obtained by using many C files and include these in the main C module by using #include.

Command line options:

```
 -r : generate relocatable assembly (no hex file)
 -r2[=][<file.lkr>]: use separate section for interrupt
 -rx : make variables static by default
```

External assembler options:

```
 -x<file> : -x"C:\Program Files\Microchip\MPASM Suite\mpasmwin.exe"
 -X<option> : assembler option: -X/q (all options must be separate)
```

Assembly file options (normally not used):

```
 -rp<N> : name on codepage 0 is PROG<N>, default PROG1
 -rb<N> : name on RAM bank 0 is BANK<N>, default BANK0
 -ro<N> : add offset <N> on local variable block name
```

Using MPLINK or a single module

Currently it is best to use a single C module for several reasons. MPLINK support was mainly offered to enable asm modules to be added.

Limitations when using MPLINK:
1. Asm mode debugging only (C source code appear as comments)
2. Multiple C modules do not allow the static local variable stack to be calculated for the whole program, meaning that much more RAM space will be used for local variables.
3. Call level checking must be done manually
4. Computed goto will be slower because the compiler cannot check 256 byte address boundary crossing.
5. Inefficient RAM bank updating, meaning mode code.

Reasons for using multiple modules are often:
1. Faster build: However, CC5X is incredible fast.
2. Module separation: However, sufficient module separation can be achieved by using multiple C files.
3. Asm modules: Inline ASM is supported by CC5X.

C modules can be merged into a single module and still be viewed as single modules. Such C modules can be used in several projects without modification. The procedure is as follows:

1. Include the “separate modules” into the main module:

```
#include "module1.c"
#include "module2.c"
// ..
#include "moduleN.c"
// ..
void main( void) { .. }
```

2. Each merged “module” includes the required header files. This can be header files specific for the “module” or common header files:

```
#include "header1.h"
#include "header2.h"
// ..
#include "headerN.h"
// ..
// module functions
```

3. If the same header file is included in more than one “module”, it will be required to prevent compiling the same header file definitions more than once. This is done by using the following header file framing:

```
#ifndef  _HEADER_N_Symbol   // the first header file line
#define _HEADER_N_Symbol    // compile this line once only
// ..
// header definitions as required
// ..
#endif  // the last header file line
```

Variables and pointers

Variables defined in other module can be accessed. CC5X needs to know the type, and this is done by adding 'extern' in front of a variable definition.

```
extern char a;
```

All global variables that are not 'static' are made available for other modules automatically. CC5X inserts 'GLOBAL' statements in the generated assembly file.

CC5X will generate a 'MOVLW LOW (var_name+<offset>)' when using the address operators '&var_name'.

Global bit variables are a challenge. It is recommended to first define a char variable and then use 'bit bx @ ch.0;'. Otherwise CC5X will define a global char variable with random name. This name have the format '_Gbit<X><X>' where <X> is a (more or less) random selected letter. This variable is reserved by a RES statement and used in the assembly file when generating relocatable assembly.

```
bit b1;
b1 = 0;   // BCF _GbitQB+0,0
```

The variable file (*.var) is slightly modified when generating relocatable assembly. Note that most addresses stated in the variable file are reallocated by MPLINK.

Option -rx will make variables static by default. This means that variables will not be visible outside the module unless 'extern' is added in front of the type definition. Note that option -rx requires that an extern pointer definition need to be stated before the allocation of the pointer.

```
extern char *px;  // definition only, no allocation of space
char *px;         // space is allocated for the pointer
```

IMPORTANT: 'const' data cannot be 'extern' because MPLINK does not support the const access functions generated by CC5X. Identifiers with the 'const' modifier will not be made visible outside the module. This also applies to struct objects with const pointers.

IMPORTANT: Allocation of pointers is slightly different when using relocatable assembly. The main reason for this is that CC5X cannot trace how addresses are assigned to pointers between different modules. There is no change on local and static pointers. An extern visible pointer without a size modifier (size1/size2) will be:

a) 16 bit if RAM alone use more than 8 bit addresses regardless of the default memory model used.
b) 16 bit if special registers need more than 8 bit addresses when the default RAM memory model is 16 bit (option -mm2 or -mr2).
c) 8 bit otherwise.

An extern visible pointer with the size1 modifier will access addresses from 0 - 255. An error is printed if the pointer is assigned higher addresses. However, it is possible to force an extern 8 bit pointer to access addresses 256 - 511 by a pragma statement:

```
extern size1 char *px;
#pragma assume *px in rambank 2   // rambank 2 or 3
```

Note that 8 bit pointers in a struct can only access addresses from 0 - 255, even if the struct is static or local.

Enhanced core 14 and bank boundaries

Tables and structures that cross a bank boundary have to be located at predefined addresses (type table[] @ address;). This is required in order to calculate the address correct for mapped addressing. Tables that do not cross a bank boundary use standard address calculation and can be allocated by the linker.

Local variables

CC5X uses a different naming strategy on local variables when generating relocatable assembly. CC5X reserves a continuous block in each ram bank (or shared bank) and use this name when accessing local variables.

IMPORTANT RESTRICTION: The main() routine, interrupt service routines and all extern functions are defined as independent call trees or paths. A function called from two independent call paths cannot contain local variables or parameters because address sharing cannot be computed in advance. CC5X detects this and generates an error message.

The names of the local RAM blocks are _LcRA, _LcRB, etc. The last letter is related to the RAM bank and the second last to the module name. Adding option -ro1 will for example change name _LcAA to _LcBA. This can be used if there is a collision between local variable block defined in separate C modules. MPLINK detects such collisions.

```
-ro<N> : add offset <N> when generating  local variable block name
```

Local variables for external available functions are allocated separately, one block for each extern function. This often means inefficiently use of RAM. It is therefore recommended to use 'extern' only on those functions that have to be extern, and use few local variables in the extern functions. Also consider using global variables.

Header files

It is recommended to make common header files that contain global definitions that are included in all C modules. Such files can contain definitions (#define), IO variable names, etc.

Using RAM banks

RAM bank definitions only apply to devices with RAM located in more than one bank.

Note that the RAM bank of ALL variables has to be known (defined) during compilation. Otherwise the bank bit updating will not be correct. The bank is defined by using '#pragma rambank' between the variable definition statements, also for 'extern' variables. An alternative is to use the bank type modifier (bank0..bank3, shrBank).

```
#pragma rambank 0
char a,b;
#pragma rambank 1
extern char array1[10];
#pragma rambank -
extern char ex;   // shared/common RAM
```

Bank bit updating

CC5X use an advanced algorithm to update the bank selection bits. However, it is not possible to trace calls to external functions. Therefore, calling an external function or allowing incoming calls makes CC5X assume that the bank bits are undefined. This often means that more code compared to the optimal bank bit update strategy.

It is therefore recommended to only use 'extern' on those functions that have to be extern, and keep the number of calls between modules to a minimum.

Functions

Functions residing in other modules can be called. Functions defined can be called from other modules (also from assembly modules).

NOTE that ALL functions that are called from another module need an 'extern' first. This is an extra requirement that is optional in C. The reason is that the compiler needs to decide the strategy on bank bit updating and local variables allocation. It is most efficient to use FEW extern functions.

```
extern void func1(void);  // defined in another module

extern void fc2(void) { } // available to all modules
```

NOTE that extern functions can only have a single unsigned 8 bit parameter which is transferred in W. This is because local storage information is not shared between modules. The return value cannot be larger than 8 bit for the same reason (bit values are returned in Carry).

Supported extern function parameter types: char, uns8
Supported extern function return types: char, uns8, bit

CC5X inserts a 'GLOBAL <function>' in the generated assembly code for all external available functions. 'EXTERN <function>' is inserted for functions defined in other modules.

If the C module contains main(), then a 'goto main' is inserted in the STARTUP section.

Using code pages

Page bit updating only applies to functions with more than one code page.

The code page of all function calls have to be known (defined) during compilation. Otherwise the page bit updating will not be correct. The page is defined by using '#pragma location' or the page type modifier for functions defined in another module. For functions defined in the current module, '#pragma codepage' can also be used.

It is recommended to define the function heading (prototypes) for all extern functions in a header file including page information. This file should be included in all C modules.

IMPORTANT: When a module contains functions located on more than one codepage, all function belonging to the same page must be put in sequence in the source file. This because MPASM/MPLINK requires all object code sections to be continuous and CC5X is unable to change the definition order of the functions.

Interrupts

CC5X requires that the interrupt function is located at address 4. Writing the interrupt service routine in C using MPLINK will require some care. The main issue is to set up the linker script file as described later in this file. Two options are possible:

ALTERNATIVE 1: Use the linking sequence to locate the interrupt service routine. This is done by listing the module with the interrupt service routine FIRST in the module list used by MPLINK. This is the important point which makes MPLINK put the interrupt service routine in the beginning of the PROG/PROG1 logical code section (address 4). The list file generated by MPLINK should be inspected to ensure that the interrupt routine starts at address 4. Another important point is to remove the #pragma origin 4 when using MPLINK. This is the only difference in the C source compared to using the built in CC5X linker (single C module).

ALTERNATIVE 2: Set up a SEPARATE logical section in the linker script file for the interrupt service routine. This is a more robust solution. CC5X will generate a partial script file to avoid manual address calculation. The partial script file must be included in the main script file. The setup is described in Section *The MPLINK script file* on page 95.

It is also possible to design an assembly module containing the interrupt service routine. Information on how to do this should be found in the MPASM/MPLINK documentation.

Call level checking

CC5X will normally check that the call level is not exceeded. This is only partially possible when using MPLINK. CC5X can ONLY check the current module, NOT the whole linked application.

When calling an external function from the C code, CC5X will assume that the external call is one level deep. This checking is sometimes enough, especially if all C code is put in one module, and the assembly code modules are called from well known stack levels. Calling C function from assembly will require manual analysis.

Therefore, careful verification of the call structure is required to avoid program crash when overwriting a return value on the hardware stack (which is 2 or 8 levels deep). The compiler generated *.fcs files can provide information for this checking.

Calls to external functions is written in the *.fcs file. External function calls are marked [EXTERN].

Computed goto

14 bit core: CC5X will always use the long format when generating code for skip(). It is not possible to use the -GS option. The long format is 3 instructions longer than the short format.

12 bits core: All destination addresses must be located in the first 256 word half of the codepage. Unfortunately CC5X cannot check the layout done by MPLINK. It is therefore strongly recommended to apply some manual design rules that will prevent destination addresses to be moved into the invisible (high) code page half. This can be done by ensuring a linking sequence that will put all modules containing computed goto at the beginning (offset 0) of the codepage. Inspection of the generated list file is recommended.

Recommendations when using MPLINK

1. Use as few C modules as possible because of:
a) inefficient bank bit updating between modules
b) local variable space cannot be reused between modules
c) only a single unsigned 8 bit parameter in calls between modules
d) only 8 or 1 bit return values between modules

2. Use definition header files that are shared between modules. Include the shared definition in all C modules to enable consistency checking.

a) function headings (prototypes). Add page information when using more than one code page:

```
// module1.c
extern page0 void sub(char ax);
// module2.c
extern page1 void mpy(void);
// Do not add extern to functions that are not called
// from other modules.
char localFunctionA(void);  // local function
// Note that it is required to use extern in the
// function definition when an extern prototype
// is not defined.
```

b) variables: add bank information

```
// module1.c
extern shrBank char b;
#define ARRAY_SIZE 10
```

```
extern bank0 char array[ARRAY_SIZE];
// module3.asm
extern bank1 char mulcnd, mulplr, H_byte, L_byte;
```

c) constants, definitions, enumerations and type information

```
#define MyGlobalDef  1
enum { S1 = 10, S2, S3, S4 S5 };
// names assigned to port pins
#pragma bit in  @ PORTB.0
#pragma bit out @ PORTB.1
```

3. define bit variables to overlap with a char variable

```
/* extern */ char myBits;
bit b1 @ myBits.0;
bit b2 @ myBits.1;
// use 'extern char myBits;' for global bits and
// put the definitions in a shared header file.
// Move definition 'char myBits;' to one of the
// modules.
```

4. Make a linker script file according to the description stated later. Follow the guidelines when using interrupts.

5. Set up a 'makefile' to enable automatic (re)compilation and linking. Follow the guidelines when using MPLAB. Edit and use the option '+reloc.inc' when compiling C modules.

6. Do the final call level checking manually

7. Update conventions in assembly functions called from C modules:

a) The bank selection bits should be updated in the beginning of assembly functions that are called from C.
b) The page selection bits must be correct (set to the current module page) when returning.

MPASM

The linker script file must be made (or adapted) according to the description stated.

Note that MPASM will generate its own warnings and messages. These should normally be ignored. MPASM do not know about the automatic bank bit updating and will display messages about this. MPASM have generated the message if the asm file extension is used in the message.

Program execution tracing will always use the assembly file as source when using MPLINK. MPASM can generate object code from assembly modules. There are some restrictions and additions when using relocatable modules compared to using a single assembly module.

CC5X does not support the object code directly, but generates relocatable assembly that MPASM use to generate the object file. MPASM is started from within the CC5X so that no extra command is required (only the right command line options).

Case Sensitivity option in MPASM is by default On, and should remain On because C use case dependent identifiers.

Example options to start MPASM at the end of compilation:

```
-x"C:\Program Files (x86)\Microchip\MPLABX\v4.01\mpasmx\mpasmx.exe"
-x"C:\Program Files\Microchip\MPASM Suite\mpasmwin.exe"
```

Options starting with -X are forwarded to the assembler:

```
 -X/o   : generate relocatable object code
 -X/q   : assembler quiet mode
```

If the CC5X error file option (-F) is missing, CC5X will read the error file generated by MPASM and write the error and warnings found there to the screen and the output file (*.occ). The error file is then deleted.

If the CC5X error file option (-F) is present, CC5X will write error and warnings to the error file (*.err) and append the error and warnings generated by MPASM at the end of this file.

The MPLINK script file

MICROCHIP supplies sample linker script files for each device with the file extension '.lkr' (look in the MPLAB directory). When making a linker script file for a specific project, this file can be copied and edited to suit the needs of CC5X.

The sample MPLINK script files must be changed slightly if the interrupt function is written in C. The reason is that the interrupt function must start at address 4 when using CC5X. It could be possible to use a vector at address 4, but this slows down interrupt response. Anyway, using a goto vector directly is not possible when the device contains more than 2048 words of code. This is because PCLATH needs to be saved before it can be updated.

CHANGE 1: Interrupt routine in C WITH a separate logical section. CC5X generates a partial script file when using the -r2 (or -r2[=]<file.lkr>) command line option. This file is written if (and only if) CC5X compiles a module with an interrupt service routine. The generated script file may look like:

```
 CODEPAGE  NAME=intserv  START=0x4  END=0x1C
 CODEPAGE  NAME=page0    START=0x1D END=0x7FF
```

Example change in the main script file:

```
 //CODEPAGE   NAME=page0    START=0x4      END=0x7FF
 INCLUDE  module1.lkr   // or other script file name
 SECTION  NAME=ISERVER  ROM=intserv // Interrupt
```

CHANGE 2: Interrupt routine in C WITHOUT a separate logical section. Example change:

```
 CODEPAGE  NAME=vectors  START=0x0  END=0x3  PROTECTED
 // NEW VALUE:                          ^------

 CODEPAGE   NAME=page0    START=0x4  END=0x7FF
 // NEW VALUE:                 ^------
```

CHANGE 3: If INTERRUPTS are not used, then the first code page can start at address 1. Example change:

```
 CODEPAGE NAME=vectors  START=0x0 END=0x0 PROTECTED
 // NEW VALUE:                        ^------

 CODEPAGE   NAME=page0    START=0x1  END=0x7FF
 // NEW VALUE:                 ^------
```

CHANGE 4: The 12 bit core devices need a logic section for the startup vector. Example change for the 16C57:

```
 CODEPAGE NAME=page3    START=0x600 END=0x7FE
```

```
// NEW VALUE:                          ^------

CODEPAGE NAME=vectors  START=0x7FF END=0x7FF   // NEW
SECTION  NAME=STARTUP  ROM=vectors             // NEW
```

CHANGE 5: Certain devices require a special interrupt save sequence that needs to use certain RAM locations (0x20 and 0xA0). These addresses must be made unavailable for allocation in the linker script file. This applies to 14000, 16C63, 16C63A, 16C65, 16C65A, 16C65B, 16C73, 16C73A, 16C73B, 16C74, 16C74A, 16C74B, 16F873, 16F874 and similar devices. CC5X generates a warning when variables are assigned to fixed addresses. Example change:

```
DATABANK   NAME=gpr0     START=0x21     END=0x7F
// NEW VALUE:                ^------

DATABANK   NAME=gpr1     START=0xA1     END=0xFF
// NEW VALUE:                ^------
```

CHANGE 6: LOGICAL RAM sections must be added, one for each DATABANK that contains RAM locations (not special function registers). Note that if a logical RAM section is missing, then the variables that belong to this section will be put in the "default" section. MPLINK gives no error on missing logical sections in the script file and the program will fail.

```
SECTION    NAME=BANK0    RAM=gpr0  // RAM bank 0
SECTION    NAME=BANK1    RAM=gpr1  // RAM bank 1
SECTION    NAME=BANK2    RAM=gpr2  // RAM bank 2
SECTION    NAME=BANK3    RAM=gpr3  // RAM bank 3
SECTION    NAME=SHRAM    RAM=gprnobnk // shared RAM
SECTION    NAME=GPRAM    RAM=gprs  // no RAM banks
```

Logical code blocks:

```
STARTUP   startvector
ISERVER   optional section for interrupt routine
PROG      for devices with one codepage only
PROG1     first codepage
PROG2
PROG3
PROG4
CONFIG    config word
IDLOCS    id-locations
```

Logical RAM blocks:

```
GPRAM     devices without RAM banks
BANK0     bank 0
BANK1     bank 1
BANK2     bank 2
BANK3     bank 3
SHRAM     shared/common RAM (if available on device)
```

Command line options:

Page naming:

```
-rp0   : PROG0 is the name of the first codepage
-rp1   : PROG1 is the name of the first codepage
```

Bank naming:

```
-rb0   : BANK0 is the name of the first RAM bank
-rb1   : BANK1 is the name of the first RAM bank
```

Separate interrupt logical section (named ISERVER):

```
 -r2               : use name of current module (.lkr)
 -r2[=]<file.lkr>  : use defined file name
```

Example with 3 modules

This example demonstrates the syntax only.

```
//************************************************
// MODULE1.C

#include "globdef1.h"
#include "int16CXX.H"

#pragma origin 4

interrupt int_server(void)
{
    int_save_registers    // W, STATUS (and PCLATH)
    if (T0IF)  {
        /* TMR0 overflow interrupt */
        TMR0 = -45;
        T0IF = 0;  /* reset flag */
    }
    if (INTF)  {
        /* INT interrupt */
        INTF = 0;  /* reset flag */
    }
    int_restore_registers // W, STATUS (and PCLATH)
}

char a;
bit b1, b2;

void main(void)
{
    PORTA = 0b0010;
    TRISA = 0b0001;

    if (TO == 1 && PD == 1 /* power up */)  {
        clearRAM();  // set all RAM to 0
        a = 5;
        b1 = 1;
    }
    mulcnd = 10;
    mulplr = 20;
    mpy();  // assembly routine (demo)

    b2 = !b1;
    do  {
        if (in == 1)
             break;
        sub(a&3);
    } while (a < 200);
}
```

```
//*************************************************
// MODULE2.C

#include "globdef1.h"

void sub(bank1 char ax)
{
    bank1 char i;      /* a local variable */

    /* generate pulses */
    for (i = 0; i <= ax+1; i++)  {
        out = 1;
        nop2();
        out = 0;
        a ++;   // increment global variable
    }
}


;*************************************************
; GLOBDEF1.H
// names assigned to port pins
#pragma bit in  @ PORTA.0
#pragma bit out @ PORTA.1

// module1.c
extern bank0 char a;

// module3.asm
extern bank1 char mulcnd, mulplr, H_byte, L_byte;

// module2.c
extern page0 void sub(char ax);

// module3.asm
extern page0 void mpy(void);


; *************************************************
; MODULE3.ASM
        #INCLUDE "P16F877.INC"
BANK1   UDATA
mulcnd  RES 1 ; 8 bit multiplicand
mulplr  RES 1 ; 8 bit multiplier
H_byte  RES 1 ; High byte of the 16 bit result
L_byte  RES 1 ; Low byte of the 16 bit result
count   RES 1 ; loop counter

        GLOBAL mulcnd, mulplr, H_byte, L_byte

PROG1   CODE    ; page0
mpy
        GLOBAL  mpy
        bsf STATUS, RP0  ; access bank 1
        clrf H_byte
```

```
        movlw 8
        movwf count
        movf mulcnd, W
loop    rrf mulplr, F
        btfsc STATUS, C
        addwf H_byte, F
        rrf H_byte, F
        rrf L_byte, F
        decfsz  count, F
        goto  loop
        retlw 0

        END


//************************************************
// File: 16f877.lkr
LIBPATH  .

CODEPAGE   NAME=vectors  START=0x0      END=0x3      PROTECTED
//CODEPAGE   NAME=page0    START=0x4      END=0x7FF
INCLUDE  module1.lkr
CODEPAGE   NAME=page1    START=0x800    END=0xFFF
CODEPAGE   NAME=page2    START=0x1000   END=0x17FF
CODEPAGE   NAME=page3    START=0x1800   END=0x1FFF
CODEPAGE   NAME=.idlocs  START=0x2000   END=0x2003   PROTECTED
CODEPAGE   NAME=.config  START=0x2007   END=0x2007   PROTECTED
CODEPAGE   NAME=eedata   START=0x2100   END=0x21FF   PROTECTED

DATABANK   NAME=sfr0     START=0x0      END=0x1F     PROTECTED
DATABANK   NAME=sfr1     START=0x80     END=0x9F     PROTECTED
DATABANK   NAME=sfr2     START=0x100    END=0x10F    PROTECTED
DATABANK   NAME=sfr3     START=0x180    END=0x18F    PROTECTED

DATABANK   NAME=gpr0     START=0x20     END=0x6F
DATABANK   NAME=gpr1     START=0xA0     END=0xEF
DATABANK   NAME=gpr2     START=0x110    END=0x16F
DATABANK   NAME=gpr3     START=0x190    END=0x1EF

SHAREBANK  NAME=gprnobnk START=0x70     END=0x7F
SHAREBANK  NAME=gprnobnk START=0xF0     END=0xFF
SHAREBANK  NAME=gprnobnk START=0x170    END=0x17F
SHAREBANK  NAME=gprnobnk START=0x1F0    END=0x1FF

SECTION    NAME=STARTUP  ROM=vectors    // Reset vector
SECTION    NAME=ISERVER  ROM=intserv    // Interrupt routine
SECTION    NAME=PROG1    ROM=page0      // ROM code space - page0
SECTION    NAME=PROG2    ROM=page1      // ROM code space - page1
SECTION    NAME=PROG3    ROM=page2      // ROM code space - page2
SECTION    NAME=PROG4    ROM=page3      // ROM code space - page3
SECTION    NAME=IDLOCS   ROM=.idlocs    // ID locations
SECTION    NAME=CONFIG   ROM=.config    // Configuration bits
SECTION    NAME=DEEPROM  ROM=eedata     // Data EEPROM

SECTION    NAME=SHRAM    RAM=gprnobnk   // unbanked locations
SECTION    NAME=BANK0    RAM=gpr0       // RAM bank 0
SECTION    NAME=BANK1    RAM=gpr1       // RAM bank 1
```

```
SECTION    NAME=BANK2    RAM=gpr2       // RAM bank 2
SECTION    NAME=BANK3    RAM=gpr3       // RAM bank 3


//************************************************
// File: module1.lkr : generated by CC5X when using the -r2 option
//  Note that -r2 must be used instead of -r (file 'reloc.inc')

CODEPAGE  NAME=intserv  START=0x4  END=0x1C
CODEPAGE  NAME=page0    START=0x1D END=0x7FF
```

6.9 The cdata Statement

The cdata statement stores 14 bit data in program memory.

NOTE 1: cdata[] can currently **not** be used with relocatable assembly. When using MPLINK, such data statements can be put in an assembly module.

NOTE 2: Constant data should normally be stored using the 'const' type modifier. However, cdata[] is useful for storing EEPROM data, or data and instructions at fixed addresses.

NOTE 3: There is no check on validity of the inserted data or address. However, it is NOT possible to overwrite program code and other cdata sections. The data is added at the end of the assembly and hex file in the same order as it is defined.

NOTE 4: cdata outside legal program and EEPROM space is disabled. The error message can be changed to a warning by using the -cd command line option. EEPROM address range is 0x2100 - 0x21FF for legacy 14 bit core devices and 0xF000 - 0xF0FF for enhanced 14 bit core devices.

SYNTAX:

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

ADDRESS: 0 - 0x1FFF, 0x2100 - 0x21FF
VXS : < VALUE | EXPRESSION | STRING>
VALUE: 0 .. 0x3FFF
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"
```

Strings are stored as 7 bit ASCII characters (14 bit core devices). The least significant 7 bits of each code word are filled first. Strings are aligned on word addresses for each <VXS>. However, alignment does not occur when writing "abc" "def".

IDENTIFIER: any undefined identifier. It is converted to a macro identifier and set to the current cdata word address. The purpose is to provide an automatic way to find the address of stored items.

Empty cdata statements can be used to set or read the current cdata address.

```
#pragma cdata[ADDRESS]    // set current cdata address
#pragma cdata.IDENTIFIER  // "get" current cdata address
```

Only cdata within the valid code space is counted when calculating the total number of code words.

Using the cdata statement

1. Defining special startup sequences:

```
#include "hexcodes.h"
#pragma cdata[0] = __NOP
#pragma resetVector 1    // goto main at address 1
```

2. Restoring calibration values:

```
#include "hexcodes.h"
#define ResetAddress  0x3FF   // 16C509(A)
#pragma cdata[ResetAddress]= __MOVLW(CalValue)
```

3. Storing packed strings and other data in flash devices (16F87X)

The cdata definitions should be put in a separate file and included in the beginning of the program. This enables identifiers to be used in the program and checking to be performed.

```
#define CDATA_START  0x80
#pragma cdata[CDATA_START]  // start of cdata block
#pragma cdata[] = 0x3FFF, 0x2000, 0x1000
#pragma cdata[] = 0x100, (10<<4) + 3456,\
                  10, 456, 10000

#define D7(l,h)  l + h*128
#define D28(x)  x%0x4000, x/0x4000
#pragma cdata[] = D7(10,20), D28(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);
```

All cdata start addresses have to be decided manually. The setup could be as follows:

```
.. 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
```

The #pragma origin CDATA_START is not required, because data overlapping is detected automatically. However, the compiler tells how many instructions are skipped for each origin statement. The cdata words are not counted at this printout.

Statement #pragma origin CDATA_END allows functions to be stored right after the cdata area. This origin statement is not required if all cdata are located at the end of the code space.

Preprocessor statements can be used for checking size during compilation:

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

Storing EEPROM data

EEPROM data can be put into the HEX file at addresses 0x2100 - 0x21FF (Enhanced 14 bit core use 0xF000 - 0xF0FF) for transfer to the internal EEPROM during programming of a device. Note that only the lower 8 bit of the HEX value is used for each EEPROM location. The compiler does not know how much EEPROM space a device has.

```
#if __EnhancedCore14__
 #define EEPROM_START  0xF000     // Enhanced 14 bit core devices
#else
 #define EEPROM_START  0x2100     // Legacy 14 bit core devices
#endif
#pragma cdata[EEPROM_START]       // start of cdata block
#pragma cdata[] = 0x3F, 10, 'a'   // 3 bytes EEPROM data
```

Note that strings will normally be packed into 2*7 bits when using cdata. This will not work for the EEPROM area. It is possible to add '\0' in the strings ("a\0b\0c\0"), but it is better to use a pragma to specify unpacked strings:

```
#pragma packedCdataStrings 0
// Store following strings unpacked
#pragma cdata[] = "Hello world!\0"

#pragma packedCdataStrings 1
// Store remaining strings packed
```

LINKER NOTE: EEPROM data must be put in an assembly module when using MPLINK.

7 DEBUGGING

Removing compilation errors is a simple task. The real challenge is to reveal the many application bugs. ALWAYS remember to check the assembly file if the application program does not behave as expected. Using a compiler does not remove the need for understanding assembly code.

Debugging methods

There are several ways of debugging the program:

1. Test (parts of) the program on a *simulator*. This allows full control of the input signals and thus exact repetition of program execution. It is also possible to speed up testing to inspect long term behavior and check out rare situations. How to do this is application dependent.
2. Use a hardware *emulator*. An emulator allows inspection and tracing of the internal program state during execution in the normal application environment, including digital and analog electronics.
3. Insert application specific *test-code* and run the program on a prototype board. Then gradually remove the extra code from the verified program parts. The key is to take small steps and restore the program to a working state before doing the next change. The extra test code can consist of:
   1) Code that produces patterns (square waves) on the output pins. This can be checked by an oscilloscope.
   2) Repetition of output sequences.
   3) Extra delays or extra code to handle special situations.

The different debugging methods have their advantages and disadvantages. It can be efficient to switch between several methods.

Compiler bugs

Compiler bugs are hard to detect, because they are not checked out until most other tests have failed. (Silicon bugs can be even harder). Compiler bugs can often be removed by rewriting the code slightly, or, depending on the type of bug, try:

1) #pragma optimize
2) #pragma update_FSR
3) #pragma update_RP
4) command line option: -u
5) command line option: -bu
6) command line option: -b

ALWAYS remember to report instances of compiler bugs to B Knudsen Data.

7.1 Compilation Errors

The compiler prints error messages when errors are detected. The error message is preceded by 2 lines of source code and a marker line indicating where the compiler has located the error. The printing of source and marker lines can be switched off by the *-e* command line option. The maximum number of errors printed can also be altered. Setting the maximum to 12 lines is done by the command line option *-E12*.

The format of the error messages is:

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

Some errors are fatal, and cause the compiler to stop immediately. Otherwise the compiling process continues, but no output files are produced.

If there is a syntax error in a defined macro, then it may be difficult to decide what the problem actually is. This is improved by printing extra error messages which points to the macro definition, and doing this recursively when expanding nested macros.

NOTE: When an error is detected, the compiler deletes existing hex and assembly files produced by the last successful compilation of the same source file.

Error and warning details

The compiler prints a short description of the error message to the output screen and to the *.occ file, but not to the *.err file. Note that the description will not be visible when enabling the error file in MPLAB. The *.occ file can then be opened and inspected.

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

Some common compilation problems

- not enough variable space

Solution: Some redesign is required. The scope of local variables can be made more narrow. A better overlapping strategy for global variables can be tried.

- the compiler is unable to generate code

Solution: Some of the C statements have to be rewritten, possibly using simpler statements.

- too much code generated

Solution: rewrite parts of the code. By checking the assembly file it may be possible to detect inefficient code fragments. Rewriting by using the W register directly may sometimes reduce the code size. Experience has shown that around 10% of the hex code can be removed by hand-optimizing the C code. Optimal usage of the code pages and RAM banks is important. Note that the code reduction estimate is compared to the initial code written.

- codepage limits are exceeded

Solution: move functions to another codepage by using the pragma codepage or location statements. It is sometimes necessary to split a function into two separate functions.

- too deep call level

Solution: rewrite the code. The compiler will automatically reduce the call level when functions are called once only. If there is a return array at the deepest call level, this code can be moved to the calling function:

```
void sel(char i)  {
    Carry = 0;
    W = rl(i); /* multiply by 2 */
    skip(W);
   #pragma computedGoto 1
    W = '0'; goto ENDS;
    W = '1'; goto ENDS;
    W = '4';
   #pragma computedGoto 0
  ENDS:
   /* processing continues here */
}
```

7.2 MPLAB Debugging Support

The CC5X compiler can be used inside the MPLAB environment (both MPLAB X and older MPLAB). The COFF and COD file format for debugging purposes are supported. Two modes of source file debugging are available:

a) Using the C source files (COFF and COD).

b) Using the generated assembly file as the source file (COD only). COFF file debugging in this mode can be supported by generating an assembly file and send it to MPASM in order to generate the COFF debugging file. The format of the assembly file can be changed in order to suit the debugging tool. Take a look at the assembly file options. Some suggestions:

```
-A1+6+10 -AmiJ    : simulator I
-A1+6+6  -AmiJs   : simulator II
-A6+8+12Jt        : compact I
-Am6+8+12Jt       : compact II
```

Enabling the COD-file is done by a command line option:

-CF<filename>: generate COFF debugging file using C source file(s). <*filename*> is optional. The asm file option is also switched on.
-CC<filename>: generate COD debugging file using C source file(s). <*filename*> is optional. The asm file option is also switched on.
-CA<filename>: generate COD debugging file using generated assembly file as source. <*filename*> is optional. The asm file option is also switched on.

Arrays:

COD FILE PROBLEM ONLY: Arrays and structures represent a slight challenge, because all variables passed in the COD file are currently either char or bit types.

This is solved by adding new variables which appears during debugging:

```
char table[3];    -->  table,      /* offset 0 */
                       table_e1,   /* offset 1 */
                       table_e2    /* offset 2 */
struct {
   char a;
   char b;
} st;      -->  st,    /* offset 0 (element 'a') */
                st_e1  /* offset 1 (element 'b') */
```

This means that the name of a structure element is not visible when inspecting variables in a debugger.

ICD and ICD2 debugging

ICD and ICD2 debugging requires defining a symbol before the header file is compiled to avoid that the application use reserved resources:

a) By a command line option:
```
-DICD_DEBUG   or   -DICD2_DEBUG
```

b) By using #define in combination with #pragma chip or #include:
```
#define ICD_DEBUG   // or ICD2_DEBUG
..
#pragma chip PIC16F877  // or #include "16F877.H"
```

7.3 Assert Statements

Assert statements allows messages to be passed to the simulator, emulator, etc.

```
Syntax:   #pragma assert [/] <type> <text field>

[/] : optional character

<type> : a = user defined assert
         e = user defined emulator command
         f = user defined printf
         l = user defined log command

<text field>: undefined syntax, valid to the end of
       the line. The line can be extended by a '\'
       character like other preprocessor statements.

#pragma assert /e text passed to the debugger
#pragma assert e text passed to the debugger

#pragma assert ; this assert command is ignored
```

NOTE 1: comments in the <text field> will not be removed, but passed to the debugger.

NOTE 2: Only ASCII characters are allowed in the assert text field. However, a backslash allows some translation:

```
\0 => 0, \1 => 1, \2 => 2, \3 => 3, \4 => 4
\5 => 5, \6 => 6, \7 => 7, \a => 7, \b => 8
\t => 9, \n => 10, \v => 11, \f => 12, \r => 13
```

USE OF MACROS: Macros can be used inside assert statements with some limitations. The macro should cover the whole text field AND the <type> identifier (or none of them). Macros limited to a part of the text field are not translated. Macros can be used to switch on and off a group of assert statements or to define similar assert statements.

```
#define COMMON_ASSERT a text field
#define AA /
..
#pragma assert COMMON_ASSERT
#pragma assert AA a text field
```

Macro AA can also disable a group of assert statements if writing:

```
#define AA ;

#define XX /a  /* this will NOT work */
#pragma assert XX causes an error message
```

7.4 Debugging in Another Environment

Testing a program larger than 500-1000 instructions can be difficult. It is possible to debug parts of the program in the Windows/MSDOS environment. Another C compiler has to be used for this purpose. Using another environment has many advantages, like faster debugging, additional test code, use of printf(), use of powerful debuggers, etc. The disadvantage is that some program rewriting is required.

All low level activity, like IO read and write, have to be handled different. Conditional compilation is recommended. This also allows additional test code to be easily included.

```
#ifdef SIM
  // simulated sequence
  // or test code (printf statements, etc.)
```

```
#else
  // low-level PICmicro code
#endif
```

The following can be compiled and debugged without modifications:
1. General purpose RAM access
2. Bit operations (overlapping variables requires care)
3. Use of FSR and INDF (with some precautions)
4. Use of rl(), rr(), swap(), nop() and nop2(). Carry can be used together with rl() and rr(). Direct use of Zero_ should be avoided.
5. Use of the W register

The recommended sequence is to:
1. Write the program for the actual PICmicro device.
2. Continue working until it can be compiled successfully.
3. Debug low-level modules separately by writing small test programs (i.e. for keyboard handling, displays, IIC-bus IO, RT-clocks).
4. Add the necessary SIM code and definitions to the code. Debug (parts of) the program in another environment. Writing alternative code for the low-level modules is possible.
5. Return to the PICmicro environment and compile with SIM switched off and continue debugging using the actual chip.

8 FILES PRODUCED

The compiler generates a hex file that can be used for programming the PICmicro devices directly. The hex file normally contains code, data and optionally device configuration information. However, it is possible to successfully compile a source file that contains data only. In this case the source code typically will contain #pragma cdata statements with FLASH or EEPROM data.

The hex file is produced only there are no errors during compilation. The compiler may also produce other files by setting some command line options:

- assembly, variable, list, function outline, debugging, preprocessor output and error files

8.1 Hex File

The default hex file format is INHX8M. The format is changed by the *-f* command line option. The INHX8M, INHX8S and INHX32 formats are:

```
: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.
```

The 16 bit format used by INHX16 is defined by:

```
: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.
```

The records in the HEX file are sorted according to the address. Option -chu will disable this sorting for backward compatibility with older compiler versions (version 3.2R and earlier).

8.2 Assembly Output File

The compiler produces a complete assembly file. This file can be used as input to an assembler. Text from the source file is merged into the assembly file. This improves readability. Variable names are used throughout. A hex format directive is put into the assembly file. This can be switched off if needed. Local variables may have the same name. The compiler will add an extension to ensure that all variable names are unique.

The compiler will use __config and __idlocs in the generated assembly file when #pragma config is used in the source. The old assembly format is still available by using the command line option -cfc.

Command line option -Ma will truncate all automatic generated labels in the assembly and list files. This option is sometimes useful when comparing assembly files generated by different compiler versions.

There are many command line options which change the assembly file produced. Please note the difference between the *-a* and the *-A* options. The *-a* option is needed to produce an assembly file, while the *-A* option changes the contents of the assembly and list files.

The general format is *-A[scHDpftmiJRbeokgN+N+N]*.

s: symbolic arguments are replaced by numbers
c: no C source code is printed
H: hexadecimal numbers only
D: decimal numbers only
P: use '.' in front of decimal constants
f: no object format directive is printed
t: no tabulators, normal spaces only
m: single source line only
i: no source indentation, straight left margin
J: put source after instructions to achieve a compact assembly file.
R: detailed macro expansion
b: do not add rambank info to variables in the assembly file
e: do not add ',1' to instructions when result is written back to the register
o: do not replace OPTION with OPTION_REG
k: do not convert all hexadecimal numbers (11h -> 0x11)
g: do not use PROCESSOR instead of the list directive
N+N+N: label, mnemonic and argument spacing. Default is 8+6+10.

Note that the options are CASE sensitive.

Some examples:

```
 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++;
```

8.3 Variable File

The variable list file contains information on the variables declared. Variables are sorted by address by default, but this can be changed. The compiler needs the command line option *-V* to produce this file. The file name is *<src>.var*.

The general format is *-V[rnuDGg]*. The additional letters allows the file contents to be adjusted:

r: only variables which are referenced in the code
n: sort variables by name
u: keep the variables unsorted
D: use decimal numbers
G: list default config settings and alternatives
g: list config setting alternatives

Variable file contents:

```
 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 ->    -  : mapped RAM (available in all banks)
        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
```

Examples:

```
 X  B  Address  Size  #AC   Name
 P [-] 0x000      1  :  0:  INDF
 R [-] 0x006.0    0  :  1:  in
 R [-] 0x00B      1  : 10:  alfa
 P [-] 0x00B      1  : 12:  fixc
 L [-] 0x00D      1  :  1:  lok
 L [0] 0x012.0    0  :  6:  b1
 G [0] 0x012.1    0  : 16:  bx
 G [0] 0x015      1  : 23:  b
```

When a function is not called (unused), all its parameters and local variables are truncated to the same location. Example:

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

Options -*VG* and -*Vg* will list the available device configuration bit symbols (config as found in the device header file) at the end of the variable list file. Note that option -*VG* will list the default settings enabled and not the actual settings for the project. The intension of the list is to provide an easy way to copy-and-paste the config symbols into a C source file where the actual settings can be decided by simple editing of the list. Example listing for option -*VG*:

```
 //#pragma config PWRTE = ON  // PWRT enabled
 #pragma config PWRTE = OFF  // PWRT disabled
```

Example listing for option -*Vg*:

```
 //#pragma config PWRTE = ON  // PWRT enabled
 //#pragma config PWRTE = OFF  // PWRT disabled
```

8.4 List File

The compiler can also produce a list file. The command line option is –L or -L[<col>,<lin>]. The maximum number of columns per line <col> and lines per page <lin> can be altered. The default setting is -L200,60. The contents of the list file can be changed by using the -A option.

8.5 Function Call Structure

The function call structure can be written to file <src>.fcs. This is useful for codepage optimization and function restructuring in case of call level problems. Note that two different formats are produced; the first is a list of functions, the second is a recursive expansion of the function call structure. The command line option is -*Q* for both formats.

Format sample:

```
 F: function1              :#1  : p0 <- p1
    func2                 : #5  : p0 -> p3 **
    delay                 : #2  : p0 -> p2 *
    func3                 : #3  : p0 -> p0
```

The meaning of the symbols is:
1. func2, delay and func3 are called from function1
2. #1 : function1 is called once
3. #3 : func3 is called 3 times (once from function1)
4. p0 <- p1 : function1 resides on page 0
5. p0 <- p1 : function1 is called from page 1
6. p0 -> p3 : call to func2 (resides on page 3)
7. * : one pagebit have to be updated before call
8. ** : both pagebits have to be updated

The call structure is expanded recursively. The indentation show the nesting of the function calls in the source. The true call level is printed at the beginning of the line. The true call level is different from the indentation level when CALL's have been replaced by GOTO's. A mark is then printed at the end of the line in such cases. The interrupt call level is handled automatically and checked. There is a separate expansion for the interrupt service routine.

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

Explanation of symbols used:
- L1 : stack level 1 (max 2 or 8 levels). This is the REAL stack level, compensated when CALL's have been replaced by GOTO.
- .. : only the first call is fully expanded if more that one call to the same function occur inside the same function body.
- [CALL->GOTO] : CALL replaced by GOTO in order to get more call levels
- [T-GOTO] : CALL+RETURN is replaced by GOTO to save a call level.
- [RECURSIVE] : recursive function call

8.6 Preprocessor Output File

The compiler will write the output from the preprocessor to a file (<src>.cpr) when using the -B command line option. Preprocessor directives are either removed or simplified. Macro identifiers are replaced by the macro contents. This file can be useful to check out macro expansion, for example when the compiler produce an error message when nested macros are used.

The option format is *-B[pims]* where the additional letters allow some alternatives:

p : partial preprocessing
i : no include files
m: modify symbols
s : modify strings

Compilation will stop after preprocessing when using any of the additional letters.

9 APPLICATION NOTES

9.1 Delays

Delays are frequently used. There are various methods of generating them:

1. Instruction cycle counting
2. Use of the TMR0 timer
3. Watchdog timeout for low power consumption
4. Use of variables achieves longer intervals

```
void delay(char millisec)
/* delays a multiple of 1 millisecond at 4 MHz */
{
    OPTION = 2;  /* prescaler divide by 8 */
    do  {
        TMR0 = 0;
        clrwdt();  /* only if necessary */
        while (TMR0 < 125)   /* 125 * 8 = 1000 */
            ;
    } while (-- millisec > 0);
}

void delay10(char n)
/*Delays a multiple of 10 millisec.
  Clock : 4 MHz   => period T = 0.25 microsec.
  DEFINITION: 1 is = 1 instruction cycle
  error: 0.16 percent
*/
{
    char i;
    OPTION = 7;
    do  {
        clrwdt();      /* only if necessary */
        i = TMR0 + 39; /* 256 us * 39 = 10 ms */
        while (i != TMR0)
            ;
    } while (--n > 0);
}

void _delay10(char x)
/*
  Delays a multiple of 10 millisec.
  Clock : 32768 Hz   => period T = 30.518 microsec.
  DEFINITION: 1 is = 1 instruction cycle
                   = 4 * T = 122 microsec
  10 ms = 82 is (81.92) =>  error: 0.1 percent
*/
{
    char i;
    do  {
        i = 26;             /* 2 is */
        do
            i = i - 1;
        while (i > 0);    /* 26 * 3 - 1 = 77 is */
```

```
    } while (--x > 0);     /* 3 is */
}

char counter;

void main(void)
{
    if (TO == 1)  {
        /* power up or MCLR */
        PORTA = 0;   /* write output latch first */
        TRISA = 0;    /* all outputs */
        TRISB = 0xFF; /* all inputs */
    }
    else  {
        /* watchdog wakeup */
        counter -= 1;
        if (counter > 0)  {
            OPTION = 0x0B; /* WDT divide by 16 */
            sleep();  /* waiting 16 * 18 ms =
                         288 ms = 0.288 seconds */
        }
    }
    delay(100);  /* 100 millisec */
    /* .. */
    delay10(100);  /* 1 second */
    /* .. */
    counter = 7;  /* 7*0.288ms = 2000 ms */
    OPTION = 0x0B; /* 0 1011 : WDT divide by 16 */
    /* sleep(); waiting 16*18 ms = 0.288 seconds */
}   /* total of 2 seconds, low power consumption */
```

9.2 Computed Goto

Computed goto is a compact and elegant way of implementing a multi-selection. It can also be used for storing a table of constants. However, the ‘const’ type modifier is normally the best way to store constant data in program memory.

WARNING: Designing computed goto's of types not described in this section may fail. The generated assembly file will then have to be studied carefully because optimization and updating of the bank selection bits can be wrong.

The 12 bit core requires that all destinations of the computed goto are within the first half code page. The 14 bit core requires that PCLATH is correctly updated before loading PCL. The compiler can do ALL updating and checking automatically. Study the following code samples.

```
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 on the 12 bit core and for
    the first 256 addresses of the 14 bit core */
}
```

Built in skip() function for computed goto

The skip() function also allow a 16 bit parameter. When using an 8 bit parameter, carry is automatically generated (3 code words extra) if the table cross a 256 word address boundary. Carry is always inserted when generating relocatable assembly. Options available:

-GW : dynamic selected skip() format, warning on long format (default)
-GD : dynamic selected skip() format
-GS : always short skip() format (error if boundary is crossed)
-GL : always long skip() format

When using the -GS option, CC5X will generate an error if the table cross a 256 word code address boundary. The short format enables most compact code, but requires manually moving the table in the source code if the error is produced.

Origin alignment

It is possible to use #pragma origin to ensure that a computed goto inside a function does not cross a 256 word address boundary. However, this may require many changes during program development. An alternative is to use #pragma alignLsbOrigin to automatically align the least significant byte of the origin address. Note that this alignment is not possible when using relocatable assembly, and also that it does not apply to the 12 bit core.

Example: A function contains a computed goto. After inspecting the generated list file, there are 16 instructions between the function start and the first destination address (offset 0) right after the ADDWF PCL,0 instruction that perform the computed goto. The last destination address (offset 10) resides 10 instructions after the first destination. A fast a compact computed goto requires that the first and last destination resides on the same “byte page” (i.e. (address & 0xFF00) are identical for the two addresses). This is achieved with the statement:

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

The alignment pragma statement is not critical. The compiler will generate an error (option -GS) or a warning (-GW) if the computed goto cross a boundary because of a wrong alignment. An easier approach is to align the LSB to a certain value (as long as program size is not critical).

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

Computed goto regions

The compiler enters a goto region when skip() is detected. In this region optimization is slightly changed, and some address checks are made. The goto region normally ends where the function ends.

A goto region can also be started by a pragma statement:

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

A goto region can also be stopped by a pragma statement:

```
   #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). */
```

Computed Goto Regions affects:

1. Optimization
2. Register bank bit updating (RP0/1, FSR5/6)
3. 256 word page checks

Examples

```
char sub01(char W)
{
 /* The computed goto region can be constructed
    just as in assembly language. However, '#pragma
    computedGoto' should be inserted around such a
    region. Otherwise unexpected results may
    occur. */

   #pragma computedGoto 1
    PCLATH = 0;  // 14 bit core only
    PCL += W;
   /* 14 bit core: REMEMBER to make sure that the
      function is located within the first 256
      addresses. (There is no warning on this when
      'skip(W)' is NOT used) */
    return 'H';
    return 'e';
    return 'l';
    return 'l';
    return 'o';
   #pragma computedGoto 0
}

/* A VERY LARGE TABLE (with more than 256 byte)
   can also be constructed (14 bit core): */
char L_dst, H_dst;

char sub02(void)
{
   /* H_dst,L_dst : index to the desired element,
                    starting from 0 */
   #define CGSTART 0x100
    PCLATH = CGSTART/256 + H_dst;   // MSB offset
    PCL = L_dst;  // GLOBAL JUMP AT THIS POINT
    return W;    // dummy return, never executed

   /* IMPORTANT : THIS FUNCTION AND THE DESTINATION
      ADDRESSES HAVE TO BE LOCATED IN THE SAME 2048
```

```
      WORD CODEPAGE. OTHERWISE PCLATH WILL NOT BE
      CORRECT ON RETURN */
}

#pragma  origin  CGSTART // the starting point
  /* The origin statement is the best way to
     set the starting point of the large return
     table. The address should be defined by a
     '#define' statement, because it then can
     be safely changed without multiple updating. */

char sub02r(void)
{
   #pragma computedGoto 2   // start of large table
    #pragma return[] = "ALFA"
    #pragma return[] = 0x10 0x11
    ..
}
#pragma origin 0x0320
   /* using an origin statement after a large return
      table is useful to check the number of return
      instructions generated. In this case, there
      should be 0x320-0x100=0x250=544 instructions.
      If not, any differences will be reported by
      the compiler, either as an error, or as a
      message. */

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;
    Carry = 0;
    s = rl(s);  /* multiply by 2 */
    s = rl(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:

  /*
    NOTE: "goto END" is necessary for ALL cases if the function is
    called from another codepage. NOTE: '#pragma optimize ..' can
    be useful in this situation. If the call level is too deep, note
    that the compiler can only replace CALL by GOTO if there are few
    'return constant' inside the function.
  */
}
```

9.3 The switch statement

```
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 */
}
```

The compiler performs a sequence of *XORLW <const>*. These constants are NOT the same as the constants written in the C code. However, the produced code is correct! If more compact code is required, then consider rewriting the switch statement as a computed goto. This is very efficient if the cases are close to each other (i.e. 2, 3, 4, 5, ..).

APPENDIX

A1 Using Interrupts

```
#pragma bit pin1 @ PORTA.1
#pragma bit pin2 @ PORTA.2

#include "int16CXX.H"

#pragma origin 4

interrupt int_server(void)
{
    int_save_registers    // W, STATUS (and PCLATH)
    /* It is recommended to use int_save_registers and
       int_restore_registers on all devices for compatibility */
    /* Note 1: The Enhanced 14 bit core has hardware register
       save and restore of W, STATUS, BSR, FSRx and PCLATH */
    /* Note 2: The Enhanced 12 bit core has hardware register
       save and restore of W, STATUS, BSR and FSR */

    if (T0IF)  {
        /* TMR0 overflow interrupt */
        TMR0 = -45;
        if (pin1 == 1)
            pin1 = 0;
        else
            pin1 = 1;
        T0IF = 0;  /* reset flag */
    }

    if (INTF)  {
        /* INT interrupt */
        INTF = 0;  /* reset flag */
    }

    if (RBIF)  {
        /* RB port change interrupt */
        W = PORTB; /* clear mismatch */
        RBIF = 0;  /* reset flag */
    }

   /*
     NOTE: GIE is AUTOMATICALLY cleared on interrupt entry and set
           to 1 on exit (by RETFIE). Setting GIE to 1 inside the
           interrupt service routine will cause nested interrupts
           if an interrupt is pending. Too deep nesting may crash
           the program !
   */
    int_restore_registers // W, STATUS (and PCLATH)
}

void main(void)
{
```

```
  #ifdef _16C71
    ADCON1 = bin(11); /* port A = digital */
  #endif
  #if defined _16F873 || defined _16F874 || defined _16F876 || \
      defined _16F877
    ADCON1 = 0b0110; // PORT A is digital
  #endif
    PORTA = 0; /* 76543210 */
    TRISA =     0b11111001;

    OPTION = 0; /* prescaler divide by 2 */
    TMR0 = -45;  /* 45 * 2 = 90 periods */
    T0IE = 1;   /* enable TMR0 interrupt */
    GIE = 1;    /* interrupts allowed */


    while (1)  {  /* infinite loop */
        pin2 = 0;
        nop2();  // 2 Instruction cycles
        nop();   // 1 Instruction cycle
        pin2 = 1;
    }
}
```

A2 Predefined Register Names

Core 12:

```
char W;
char INDF, TMR0, PCL, STATUS, FSR, PORTA, PORTB;
char INDF0, RTCC, PC; // alternative names
char OPTION, TRISA, TRISB;
char PORTC, TRISC;
bit Carry, DC, Zero_, PD, TO, PA0, PA1, PA2;
bit FSR_5, FSR_6;
```

Enhanced Core 12:

```
char W;
char INDF, TMR0, PCL, STATUS, FSR, PORTA, PORTB;
char OPTION, TRISA, TRISB, BSR;
char PORTC, TRISC;
bit Carry, DC, Zero_, PD, TO, PA0, PA1, PA2;
```

Core 14:

```
char W;
char INDF, TMR0, PCL, STATUS, FSR, PORTA, PORTB;
char INDF0, RTCC, PC, OPTION; // alternative names
char OPTION_REG, TRISA, TRISB;
char PCLATH, INTCON;
bit PS0, PS1, PS2, PSA, T0SE, T0CS, INTEDG, RBPU_;
bit RTE, RTS; // alternative names
bit Carry, DC, Zero_, PD, TO, RP0, RP1, IRP;
bit RBIF, INTF, T0IF, RBIE, INTE, T0IE, GIE;
bit RTIF, RTIE; // alternative names
bit PA0, PA1;  // PCLATH
```

Enhanced Core 14:

```
char *FSR0, *FSR1;
char INDF0, INDF1;
char FSR0L, FSR0H, FSR1L, FSR1H;
char W, WREG;
char PCL, PCLATH, STATUS, INTCON;
bit Carry, DC, Zero_, PD, TO;
```

A3 Assembly Instructions

```
Assembly:  Status:  Function:
NOP    -      -     No operation
MOVWF  f      -     f = W;       Move W to f
CLRW   -      Z     W = 0;       Clear W
CLRF   f      Z     f = 0;       Clear f
SUBWF  f,d  C,DC,Z  d = f - W;   Subtract W from f
DECF   f,d    Z     d = f - 1;   Decrement f
IORWF  f,d    Z     d = f | W;   Inclusive OR W and f
ANDWF  f,d    Z     d = f & W;   AND W and f
XORWF  f,d    Z     d = f ^ W;   Exclusive OR W and f
ADDWF  f,d  C,DC,Z  d = f + W;   Add W and f
MOVF   f,d    Z     d = f;       Move f
COMF   f,d    Z     d = f ^ 255; Complement f
INCF   f,d    Z     d = f + 1;   Increment f
DECFSZ f,d    -     Decrement f, skip if zero
RRF    f,d    C     Rotate right f through carry bit
RLF    f,d    C     Rotate left f through carry bit
SWAPF  f,d    -     Swap halves f
INCFSZ f,d    -     Increment f, skip if zero

BCF    f,b    -     f.b = 0;   Bit clear f
BSF    f,b    -     f.b = 1;   Bit set f
BTFSC  f,b    -     Bit test f, skip if clear
BTFSS  f,b    -     Bit test f, skip if set

OPTION -      -     OPTION = W;   Load OPTION register
SLEEP  -    TO,PD   Go into standby mode,  WDT = 0
CLRWDT -    TO,PD   WDT = 0;      Clear watchdog timer
TRIS   f      -     Tristate port f  (f5,f6,f7)
RETLW  k      -     Return, put literal in W
CALL   k      -     Call subroutine
GOTO   k      -     Go to address
MOVLW  k      -     W = k;      Move literal to W
IORLW  k      Z     W = W | k;  Incl. OR literal and W
ANDLW  k      Z     W = W & k;  AND literal and W
XORLW  k      Z     W = W ^ k;  Excl. OR literal and W
```

Additional for the 14 bit core

```
ADDLW  k    C,DC,Z  W = k + W;  Add literal to W
SUBLW  k    C,DC,Z  W = k - W; Subtract W from literal
RETURN -      -     Return from subroutine
RETFIE -      -     Return from interrupt
```

Additional for the Enhanced 14 bit core

```
ADDWFC f,d  C,DC,Z  d = f + W + C;  Add W and f and Carry
SUBWFB f,d  C,DC,Z  d = f - W - ~C;  Subtract W from f with borrow
```

```
LSLF   f,d    C,Z   Logical shift left f
LSRF   f,d    C,Z   Logical shift right f
ASRF   f,d    C,Z   Arithmetic shift right f
MOVLB  k      -     Move literal to BSR
MOVLP  k      -     Move literal to PCLATH
BRA    k      -     Branch always
BRW    -      -     Branch forward with WREG (PC = PC + 1 + WREG)
CALLW  -      -     Subroutine call with WREG (PC = PCLATH:WREG)
RESET  -      -     Software reset
MOVIW  mm n   Z     Move INDFn to W, with pre inc/dec
       n mm   Z     Move INDFn to W, with post inc/dec
       k[n]   Z     Move INDFn to W, Indexed Indirect
MOVWI  mm n   -     Move W to INDFn, with pre inc/dec
       n mm   -     Move W to INDFn, with post inc/dec
       k[n]   -     Move W to INDFn, Indexed Indirect
ADDFSR n,k    -     Add Literal to FSRn
```

Additional for the Enhanced 12 bit core

```
MOVLB  k      -     Move literal to BSR
RETURN -      -     Return from subroutine
RETFIE -      -     Return from interrupt
```

Note:

```
d = 1  : destination f:  DECF   f    :  f = f - 1
d = 0  : destination W:  DECF   f,W  :  W = f - 1
f      : file register 0 – 31, 0 – 127
mm     : increment ++, or decrement --
Z      : Zero bit  : Z = 1 if result is 0
C      : Carry bit  :
          ADDWF  :  C = 1 indicates overflow
          SUBWF  :  C = 0 indicates overflow
          RRF    :  C = bit 0 of file register f
          RLF    :  C = bit 7 of file register f
DC     : Digit Carry bit  :
          ADDWF  :  DC = 1 indicates digit overflow
          SUBWF  :  DC = 0 indicates digit overflow
TO     : Timeout bit
PD     : Power down bit
```

Instruction execution time

Most instructions execute in 4 clock cycles. The exceptions are instructions that modify the program counter. These execute in 8 clock cycles:

- GOTO and CALL
- skip instructions when next instruction is skipped
- instructions that modify the program counter, i.e: ADDWF PCL