CC8E

C Compiler for the
PIC18 Devices

Version 1.3

User's Manual

B Knudsen Data
Trondheim - Norway

This manual covers CC8E version 1.3 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: **D**

CONTENTS

1 INTRODUCTION

Welcome to the CC8E C compiler for the Microchip PIC18 family of microcontrollers. The CC8E 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 CC8E 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 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.

CC8E 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 bank selection bits
- Enhanced and compact support of bit operations, including bit functions
- Floating and fixed point math up to 32 bit
- Math libraries including functions like sin(), log(), exp(), sqrt(), etc.
- Supports standard C constant data and strings in program memory (const)
- 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
- Extended call level by using GOTO instead of CALL when possible
- Access to most assembly instructions through corresponding C statements
- Inline assembly
- Integrated interrupt support
- Device configuration information in source code

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

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

1.1 Supported devices

16 bit PIC18 core:

- up to 16 RAM banks of 256 byte, plus access bank

1.2 Installation and System Requirements

The CC8E compiler uses 32 bit processing (console application) and runs on PC compatible machines using Microsoft Windows.

Installing CC8E is normally done by running the installation program for the latest version. Multiple versions can be installed.

CC8E 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 same installation program can be used to un-install the compiler. Alternatively can the CC8E files be deleted without any un-installation procedure.

Support for long file names

CC8E 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\cc8e"
-IC:\progra~1\cc8e
```

Equivalent include file formats:

```
#include "C:\Program Files\cc8e\C file"
#include "C:\progra~1\cc8e\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 CC8E compiler is a command-line program. It requires a list of command line options to compile a C source file and produce the required files.

Starting CC8E from Windows can be done from the Start->Run menu. Then type the full path name including cc8e.exe (or use Browse). The list of compiler command line options is then printed. The normal way of using CC8E is to use it as a tool from an integrate environment like MPLAB.

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

```
cc8e -a demo.c <enter>
```

1.3 MPLAB Support

CC8E can be selected as a tool in 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 the COD file format used by MPLAB for program debugging. CC8E offers two modes of source file debugging is available: C or assembly mode. Thus, tracing programs in MPLAB can be done using assembly instructions or C statements. MPLAB is free, and can be downloaded from the Microchip Internet site.

Please refer to the supplied file '**install.txt**' for a description on how to install and use CC8E in the MPLAB environment.

1.4 Summary of Delivered Files

```
CC8E.EXE    : compiler

INSTALL.TXT : installation guide and MPLAB setup
INLINE.TXT  : information on inline assembly syntax
CHIP.TXT    : how to make new chip definitions
CDATA.TXT   : info on the #pragma cdata statement
CONFIG.TXT  : the chip configuration bits
LINKER.TXT  : using MPLINK to link several modules (C or asm)
MATH.TXT    : math library support
ERRATA.TXT  : silicon errata issues

INT18XXX.H  : interrupt header file

HEXCODES.H  : direct coded instructions

CC8E.MTC    : MPLAB tool configuration file
TLCC8E.INI  : MPLAB tool configuration file

OP.INC      : command line options in a file
RELOC.INC   : options for generating object modules for linking

DEMO.C      : syntax demo file
DEMO-VAR.C  : defining RAM variables
DEMO-MAT.C  : integer math
DEMO-FPM.C  : floating point math
DEMO-FXM.C  : fixed point math
DEMO-ROM.C  : const data and DW
DEMO-PTR.C  : tables and pointers
DEMO-INS.C  : generating single instructions

MATH16.H    : 8-16 bit math library
MATH24.H    : 8-24 bit math library
MATH32.H    : 8-32 bit math library
MATH16X.H   : 16 bit fixed point library
MATH24X.H   : 24 bit fixed point library
MATH32X.H   : 32 bit fixed point library
MATH16F.H   : 16 bit floating point library
MATH24F.H   : 24 bit floating point library
MATH32F.H   : 32 bit floating point library
MATH24LB.H  : 24 bit floating point functions
                (log,sqrt,cos,..)
MATH32LB.H  : 32 bit floating point functions
                (log,sqrt,cos,..)

18C242.H .. 18F452.H   : PICmicro header files

NEWS.TXT      : recent added features
README.TXT
```

1.5 Short Program Example

```
/* global variables */
char a;
```

```
bit b1, b2;

/* assign names to port pins */
#pragma bit in  @ PORTB.0
#pragma bit out @ PORTB.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 */
}
```

1.6 Defining the PICmicro Device

CC8E offers 3 ways to select the PICmicro device in an application:

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

```
-p18F242
```

2) By a pragma statement in the source code. Note that the command line option will override the selection done by #pragma chip.

```
#pragma chip PIC18F242
```

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 "18F242.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 proceed the *#include/#pragma* statement.

NOTE 2: CC8E will use **automatic** include of the right header file when using the –p<device> or #pragma chip statement.

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: New header files can be defined according to file '**chip.txt**'.

NOTE 5: 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:

```
-DICD2_DEBUG
```

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

```
#define ICD2_DEBUG
..
#pragma chip PIC18F452    // or  #include "18F452.H"
```

1.7 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

Typical steps when developing programs is 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 CC8E 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?*

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:

```
CC8E Version 1.2, Copyright (c) B Knudsen Data, Norway 2001-2004
 --> EXTENDED edition, 8-32 bit int, 16-32 bit float, 32k code words
18\demo.c:
 Chip = 18C242
 RAM : ======== ======== ======== =======. .6****** ******** ******** ********
  40h: ******** ******** ******** ******** ******** ******** ******** ********
  80h: ..****** ******** ******** ******** ******** ******** ******** ********
  C0h: ******** ******** ******** ******** ******** ******** ******** ********
 Bank  0:220  1:254  bytes free
RAM usage: 38 bytes (31 local), 474 bytes free
 File 'demo.fcs'
 Optimizing - removed 27 code words (-16 %)
 File 'demo.var'
 File 'demo.asm'
 File 'demo.lst'
 File 'demo.cod'
 File 'demo.occ'
 File 'demo.hex'
Total of 175 code words (2 %)
```

2.1 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

The compiler prints information on RAM allocation. A full map is printed for the access bank and bank 0, which is useful to check out which RAM locations are still free. Detailed information on memory allocation is written to file <src>.var when using the –V command line option.

Symbols:

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

The compiler first allocates space for all variables no larger than 256 byte. Then will the remaining (large) variables be allocated. Variables are allocated from the start of each bank.

Using of bank type modifiers on variables (tables, structures) larger than 256 bytes tells the compiler NOT to use a lower bank, but the actual bank can be higher depending on free space. The search is done from bank stated (or address 0) and towards higher addresses. Manual allocation (fixed address) should be considered for such large items. The compiler will then allocate the remaining variables at the free space. It is normally only large variables that may need manual allocation, but also smaller variables that need to cross a bank boundary (256 byte).

2.2 Defining Variables

CC8E supports integer, fixed and floating point variables. The variable sizes are 1, 8, 16, 24 and 32 bit. The default *int* size is 8 bit, and *long* is 16 bit. Char variables are unsigned by default and thus range from 0 to 255. Note that 32 bit integer variables are not supported by all CC8E editions.

Math libraries may have to be included for math operations (Chapter *6.4 Library Support* on page 59).

CC8E 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 in a specific RAM bank or in the access bank. Tables and structures greater than 256 byte are allowed.

Note that variables are assigned to the access bank by default. See Chapter *2.3 Using RAM Banks* on page 21 on how to use RAM banks.

Special purpose registers are either predefined or defined in chip-specific header files. This applies to WREG, INDF0, PCL, STATUS, FSR0, Carry, 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, semafor;

int i;          //  8 bit signed
signed char sc; //  8 bit signed
long i16;       // 16 bit signed

uns8  u8;  //  8 bit unsigned
uns16 u16; // 16 bit unsigned
uns24 u24; // 24 bit unsigned
uns32 u32; // 32 bit unsigned

int8  s8;  //  8 bit signed
int16 s16; // 16 bit signed
int24 s24; // 24 bit signed
int32 s32; // 32 bit signed
```

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 32 bit floating point can be converted to and from IEEE754 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 is found in '**math.txt'**. Information on floating point libraries is found in Chapter *6.4 Library Support* on page 59.

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 use formats where the decimal point is permanently set at byte boundaries. For example, fixed8_8 use one byte for the integer part and one byte for the decimal part. Fixed point operations maps nicely to integer operations except for multiplication and division which are supported by library functions. Information on fixed point libraries is found in Chapter *6.4 Library Support* on page 59.

```
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 decimals, 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 decimal point
2. All types ending on _16 have 4 correct digits after decimal point
3. All types ending on _24 have 7 correct digits after decimal point
4. All types ending on _32 have 9 correct digits after 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 CC8E 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
```

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 allow 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.

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 CC8E use the
address to automatically decide that most of the special purpose
registers are volatile */

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

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

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

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

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 in 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 normally reside in a single block not crossing any bank boundaries, but it is possible to define a large stack that may cross bank boundaries. The compiler will not move local variables from the access bank to bank 0. Such moving is allowed for global variables, although with a warning.

The stack for local variables, parameters and temporary variables is normally allocated separately in each bank and the access bank. 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.

Using a large stack

It is possible to use a single main stack for all local variables. The main stack is not an additional stack, but tells the compiler where the main stack is located (which bank). The main stack can be larger than a single bank, and is defined by the following pragma statement:

```
#pragma mainStack 3 @ 0x110  // 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 0x110. 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 #pragma rambank is ignored for variables stored in the main stack. Addresses ranging from 0x100 to 0x1FF are equivalent to the bank1 type modifier, although the actual bank will be different after stack allocation for some variables if the main stack crosses a bank boundary.

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

```
#pragma minorStack 2 @ 0x10
```

In this case, local variables, parameters and temporary variables up to 2 bytes will be put in the access bank from address 0x10 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 increase 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.

```
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, also 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;
```

Bitfields

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

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

The CC8E 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

The RAM bank definitions are:

```
access bank: 0x000 – 0x07F and 0xF80 – 0xFFF
bank 0:      0x080 – 0x0FF
bank 1:      0x100 – 0x1FF
bank 2:      0x200 – 0x2FF
..
bank 15:     0xF00 – 0xF7F
```

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

```
/* variables proceeding the first rambank statement are placed in the
access bank. This is also valid for local variables and parameters */

#pragma rambank 1

char a,b,c; /* a,b and c are located in bank 1 */
```

```
/* parameters and local variables in functions placed here are also
located in bank 1 ! */

#pragma rambank 0
char d;     /* located in bank 0 */
```

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 the access bank. The access bank 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..bank15, shrBank/accessBank : can replace #pragma rambank
// shrBank and accessBank is the access bank

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 least code to update the bank selection bits. Some advise when locating variables:

1. All local variables and function parameters should preferably be put in the access bank.
2. The most frequently used variables (except arrays) should be placed in the access bank.
3. It is efficient to put the most frequent used arrays in bank 0
4. Try to locate variables which are close related to each other in the same bank.
5. Try to locate all variables accessed in the same function in the same bank.

RAM bank selection

RAM and special purpose registers can be located in up to 16 banks. A special bank instruction is used to select the right bank.

The bank selection bits are automatically checked and updated by the compiler, and attempts to update the bank in the source code may be 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.

NOTE: The compiler REMOVE attempts to use the bank instruction (MOVLB) in user source code. However, it is possible to switch to manual updating by 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 updateBank 0   /* OFF */
#pragma updateBank 1   /* ON */
```

These statements can be inserted anywhere, but they should surround a 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 48 in Chapter 4.1 *The pragma Statement* 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 are traced to enable optimal updating when the local update region ends.

2.4 Pointers

Single level pointers are implemented.

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

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

The compiler allows using a 8 bit RAM pointer when all accesses using this pointer is limited to the same bank. The bank is automatically detected and used. In some cases it may be needed to define this bank directly. For example when using a 8 bit pointer from several modules through relocatable assembly. An error message is printed if a restricted pointer is loaded with an address from the wrong RAM bank.

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

Pointer models

Using 8 bit pointers when possible saves both code and RAM space. CC8E 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.

That 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) Automatic 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;
```

The supported pointer types are:
a) 8 bit pointer to RAM. The compiler will automatically update the MSB bits (FSR0H).
b) 16 bit pointer to RAM. This format is required only when the same pointer have to access locations in different 256 byte RAM segments.
c) 8 bit pointer to program memory. This pointer can access up to 256 byte data.
d) 16 bit pointer to program memory. This pointer can access more than 256 byte data.
e) 16 bit pointer to RAM or program memory. Bit 15 is used to detect RAM or program memory access.

2.5 Const Data Support

CC8E 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 float ftx[] = { 1.0, 33.34, 1.3e-10 };
..
t = *ps;
ps = "";
fx = ftx[i];
```

The compiler will normally insert 'const' data at the end of the user code (high address). The following pragma statement will allow the 'const' data to be inserted between two user functions, or at a specific address (if using #pragma origin first):

```
#pragma insertConst
```

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
- 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 bit or more in structures with more than 256 byte data must be aligned

Data of size 16 bit or more

The compiler allows access of 8, 16, 24 and 32 bits data, including fixed and floating point formats. When using arrays or structures with more than 256 byte data, single data items have to be aligned. Alignment means that there should not be any remainder when dividing the offset with 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
```

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 resides 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”;
```

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 bit. 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 is 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 use typically 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 a *L* means conversion to long (16 bit).

The command line option -*cu* force 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 look 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. Signed variables and variables larger than 8 bits also use temporary variables on the computed stack.

A function can return any value type. The W register is used for 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 are 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);   - branch (2 cycles)
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 (single word)
skipL(i);     // void skipL(char);  - computed goto (double word)
skipM(i);     // void skipM(char);  - computed goto (mixed size)
f = swap(f);  // char swap(char);   - SWAPF f,d
W = addWFC(f);// char addWFC(char); - ADDWFC f,d
f = subFWB(f);// char subFWB(char); - SUBFWB f,d
f = subWFB(f);// char subWFB(char); - SUBWFB f,d
W = rlnc(f);  // char rlnc(char);   - RLNCF f,d
f = rrnc(f);  // char rrnc(char);   - RRNCF f,d
f = decsnz(f);// char decsnz(char); - DCFSNZ f,d
W = incsnz(f);// char incsnz(char); - INFSNZ f,d
f = negate(f);// char negate(char); - NEGF f
W = decadj(W);// char decadj(char); - DAW
multiply(f);  // void multiply(char); - MULWF f
multiply(50); // void multiply(char); - MULLW literal
skipIfEQ(f);  // void skipIfEQ(char);   - CPFSEQ f
skipIfLT(f);  // void skipIfLT(char);   - CPFSLT f
skipIfGT(f);  // void skipIfGT(char);   - CPFSGT f
skipIfZero(f);// void skipIfZero(char); - TSTFSZ f

pushStack();  // void pushStack(void); - PUSH
popStack();   // void popStack(void);  - POP
softReset();  // void softReset(void); - RESET

tableRead();       // void tableRead(void);        - TBLRD *
tableReadInc();    // void tableReadInc(void);     - TBLRD *+
tableReadDec();    // void tableReadDec(void);     - TBLRD *-
tableReadPreInc(); // void tableReadPreInc(void);  - TBLRD +*
tableWrite();      // void tableWrite(void);       - TBLWT *
tableWriteInc();   // void tableWriteInc(void);    - TBLWT *+
tableWriteDec();   // void tableWriteDec(void);    - TBLWT *-
tableWritePreInc();// void tableWritePreInc(void); - TBLWT +*
```

The internal rotate functions (rl, rr) are also available for the larger variable sizes:

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

Note that skip(i) requires that all instructions in the table are single word (single word increments). Similarly, skipL(i) requires that all instructions in the table are double words (GOTO,CALL,etc.), and the skipL argument skips double words on each increment. The compiler will adapt the optimization to this

need, and print an error message if this is not possible. The skipM(i) allows both double and single word instructions in the table, but note that the skipM use single word increments when calculating the offset.

The inline function nop2() is implemented by a BRANCH 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 CC8E compiler use 8 bit int size and contains significantly 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 use 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.
- 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 parenthesis just isolate 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 required 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

CC8E adds some extensions to the standard C syntax:

1. The *bit* variable type

2. The *interrupt* function type

3. C++ style comments are allowed:
   // a comment, valid to the end of the line

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

5. 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)
   ```

6. 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           \
}
```

7. Several of the type modifiers are not standard in C (bank0..bank15, accessBank, 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):

```
TOSU, TOSH, TOSL,
STKPTR,
PCLATU, PCLATH, PCL,
TBLPTRU, TBLPTRH, TBLPTRL, TBLPTR, TABLAT,
PRODH, PRODL,
INTCON, INTCON2, INTCON3,
INDF0, POSTINC0, POSTDEC0, PREINC0, PLUSW0, FSR0H, FSR0L, FSR0,
W, WREG,
INDF1, POSTINC1, POSTDEC1, PREINC1, PLUSW1, FSR1H, FSR1L, FSR1,
BSR, BSRL,
INDF2, POSTINC2, POSTDEC2, PREINC2, PLUSW2, FSR2H, FSR2L, FSR2,
STATUS,
Carry, DC, Zero_, Overflow, Negative
```

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, skipL, skipM, swap, decsnz, incsnz, addWFC, subWFB,
subFWB, rlnc, rrnc, negate, decadj, multiply, skipIfEQ, skipIfLT,
skipIfGT, skipIfZero, pushStack, popStack, softReset, tableRead,
tableReadInc, tableReadDec, tableReadPreInc, tableWrite,
tableWriteInc, tableWriteDec, tableWritePreInc
```

Extensions to the standard C keywords

```
bank0, .. bank15, bit, fixed8_8, .. fixed24_8, float16, float24,
float32, int8, int16, int24, int32, interrupt, accessBank, 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 CC8E compiler, and can be used in preprocessor macros:

```
__CC8E__  := Integer version number: 1000 means version 1.0
             1102 means version 1.1B
             * first 2 digits : main version
             * last 2 digits : minor release (01='A', 02='B', etc.)

__CoreSet__ := 1800 always for all PIC18 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 timing critical parts (depends 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.

4 PREPROCESSOR DIRECTIVES

The preprocessor recognizes the following keywords:

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

A preprocessor line can be extended by putting a '\' at the end of the line. This requires that there are no space characters behind 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 splitted, but not strings using <>
2. Only the first partial string can be a macro
3. Nested macros is 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 accessGPR <n>

The number of RAM bytes in the access bank can be defined. The default setting is 128 (0x80). The start address of the SFR regeisters in the access bank at the end of the data space will also change accordingly. This statement is normally found in the chip header files.

```
#pragma accessGPR 0x60  // access RAM from 0 to 0x5F
```

#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 -254 to 254, and should be even numbers.

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

Such alignment is useful to make sure that a computed goto does not cross a 256 byte address boundary. More details are found in Section *Origin alignment* on page 99 in Chapter 9.1 Computed Goto.

#pragma asm2var 1

Enable equ to variable transformation. This is defined in Chapter *6.5 Inline Assembly* on page 67.

#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 93 for details.

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

The *#pragma assume* statement tells the compiler that a 8 bit RAM pointer operates in a limited address range. Refer to Chapter *2.4 Pointers* on page 23 for details.

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

#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 rx    @ FSR0H.1
#pragma bit C_bit @ Carry
```

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 16 bit data in program memory at fixed addresses. Refer to Chapter *6.8 The cdata Statement* on page 88 for details.

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

ADDRESS: 24 bit byte address
VXS : < VALUE | EXPRESSION | STRING>
VALUE: 0 .. 0xFFFF
EXPRESSION: any valid C constant expression,
            i.e. 0x1000 | (3*1234)
STRING: "Valid C String\r\n\0\x24\x8\xe\xFF\xff\\\""
```

#pragma char <name> @ <constant or variable>

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 PIC18C242
```

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

The supported devices are defined in a PICmicro header file (i.e. ‘18C242.h’). It is also possible to make new header files. Refer to file ‘**chip.txt**’ for details.

#pragma computedGoto [=] <0,1>

This statement can be used when constructing complicated computed goto's. Refer to Chapter *9.1 Computed Goto* on page 99 for details.

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

#pragma config [<offset>] = <expression>

This statement allow PICmicro configuration information to be put in the generated hex and assembly file. ID locations can also be programmed.

```
#pragma config[<offset>] = <expression>
#pragma config[<offset>] |= <expression>
#pragma config[<offset>] &= <expression>

#pragma config ID[<offset>] = <expression>
```

Examples:

```
#pragma config[0] = 0b.1000.0101      // byte at address 0x300000
#pragma config[1] |= 3                // set bit 0,1 (addr 0x300001)
#pragma config[2] &= ~0xF0, |= 0x80   // clear bit 4-7, set bit 7
#pragma config[2+1] = 0xF | 0x80

#pragma config ID[0] = 0x99           // byte at address 0x200000
#pragma config ID[1] = 0x88           // byte at address 0x200001
#pragma config ID[2] = 0x03
```

The CONFIG and ID are specified in BYTES. Refer to Chapter *4.2 PICmicro Configuration* on page 49 for more details.

#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 end of the user code (high address). The following pragma statement will allow the 'const' data to be inserted between two user functions, or at a specific address (if using #pragma origin first):

```
#pragma insertConst
```

#pragma interruptSaveCheck <n,w,e>

The compiler will automatically check that vital registers are saved and restored during interrupt. Please refer to Chapter *6.2 Interrupts* on page 56 for details (or file ‘int18xxx.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>

CC8E 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 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). The main stack can cross bank boundaries if necessary. 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 @ 0x110
```

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 0x110. 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 #pragma rambank is ignored for variables stored in the main stack. Addresses ranging from 0x100 to 0x1FF are equivalent to the bank1 type modifier, although the actual bank will be different after stack allocation for some variables if the main stack crosses a bank boundary.

#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 the access bank 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 @ 0x10
```

In this case, local variables, parameters and temporary variables up to 2 bytes will be put in the access bank from address 0x10 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.

#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.

1. redirect goto to goto
2. remove superfluous gotos
3. replace goto by skip instructions
4. remove instructions that affects the zero-flag only.
5. replace INCF and DECF by INCFSZ and DECFSZ
6. remove superfluous updating of PA0 and PA1
7. remove other superfluous instructions
8. remove superfluous loading of W
9. to be defined
10. inserts TSTFSZ, CPFSEQ
11. inserts branch

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 byte address region : 0x0000 - <upper device byte code address>

Defines the byte address of the following code. The current active location cannot be moved backwards, even if there is no code in that area. Origin cannot be changed inside a function.

```
#pragma origin 8  // high priority interrupt start address
#pragma origin 0x700 + 2
#pragma origin SECTION(APPSEC)  // relocatable asm (option –rsc)
```

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

```
-   =>  access bank: 0x000 – 0x07F
0   =>  bank 0:      0x080 – 0x0FF
1   =>  bank 1:      0x100 – 0x1FF
2   =>  bank 2:      0x200 – 0x2FF
..
15  =>  bank 15:     0xF00 – 0xF7F
```

#pragma rambank defines the region 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 bank 0.

#pragma rambase [=] <n>

Defines the start address when declaring global variables. The use of rambank and rambase are very similar. The address has to be within the RAM space of the chip used. NOTE that the start address is not valid for local variables, but *rambase* can be used to select a specific RAM-bank.

#pragma resetVector <n>

Some chips have an unusual startup vector location. The reset-vector then have to be specified. This statement is normally NOT required, because the compiler normally use the default location, which is the first location. It is possible to locate main() in any codepage when not using a reset-vector.

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

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

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 *9.1*

Computed Goto on page 99 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 sectionDef <name> [:<id> <start> <end> [PROTECTED]]

#pragma sectionDef allows code sections to be defined and used in the application when generating relocatable assembly (option *-rsc*) . Predefined code sections are STARTUP, ISERVER8, ISERVER18 and PROG. These definitions will also automatically appear in the script file.

```
#pragma sectionDef IDLOC:idlocs 0x200000 - 0x200007  PROTECTED
#pragma sectionDef CONFIGS:config 0x300000 - 0x30000D  PROTECTED
#pragma sectionDef EEPROM:eedata 0xF00000 - 0xF000FF  PROTECTED
#pragma sectionDef APPSEC:appdef1  0x1000 - 0x102F
#pragma sectionDef PROG
```

Further details are found in Section *Using code sections* on page 79 in Chapter 6.7 *Linker Support*.

#pragma sharedAllocation

This pragma allow 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 6.1 Subroutine Call Level Checking.

#pragma stackLevels <n>

The number of call levels can be defined (normally not required). PIC18 uses by default 31 levels.

```
#pragma stackLevels 30   // max 64
```

#pragma unlockISR

The interrupt routines normally have to reside on address 0x8 and 0x18. The following pragma statement will allow the interrupt routine to be placed anywhere. Note that the compiler will NOT generate the link from address 0x8/0x18 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 the 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 the functions:

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

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

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

#pragma versionFile [<file>]

Allows a version number at the and 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 53.

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 IF AND ONLY IF the *#pragma config* statement is included. Note that some PICmicro programming devices may reject this information.

Syntax:

```
#pragma config[<offset>] = <expression>
#pragma config[<offset>] |= <expression>
#pragma config[<offset>] &= <expression>

#pragma config ID[<offset>] = <expression>
```

Examples:

```
#pragma config[0] = 0b.1.000.0101     // byte at address 0x300000
#pragma config[1] |= 3                // set bit 0,1 (addr 0x300001)
#pragma config[2] &= ~0xF0, |= 0x80   // clear bit 4-7, set bit 7
#pragma config[2+1] = 0xF | 0x80

#pragma config ID[0] = 0x99           // byte at address 0x200000
#pragma config ID[1] = 0x88           // byte at address 0x200001
#pragma config ID[2] = 0x03
```

The CONFIG and ID are specified in BYTES. CC8E will join 2 bytes into a 16 bit WORD in the generated hex, asm and list files. If only one byte is defined for a word, then the default value is used for the undefined byte. The default setting of config attributes are 1. ID locations have 0 as default value.

It is also possible to define the CONFIG and ID locations by using #pragma cdata statements. Then 16 bits words are defined directly.

CONFIG word start address:
PIC18 : 0x300000

ID word start address:
PIC18 : 0x200000

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:

```
CC8E [options] <src>.c [options]
```

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

-A[scHDftumiJRN+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
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
N+N+N: label, mnemonic and argument spacing. Default is 8+6+10.

-b : do not update bank selection bits (BSR register)
-bu : non-optimized updating of the bank selection bits
-ban : do not update ADSHR (sfr selection bit)

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

-cae : do not move global variables from the access bank to bank 0 when the access bank is full
-cd : allow cdata outside program space (warning only)
-cfc : use old format on config and idlocs in generated assembly file
-cif : search included file in directory containing current file(s)
-cu : use 32 bit evaluation of constant expressions
-cxc : do not search current directory for include files

-CC[<file>] : produce COD file, C mode
-CA[<file>] : produce COD 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. Note that INHX8S use output files: <file>.HXH and <file>.HXL

-F : produce error file <src>.err
-FM : MPLAB compatible error format

-g : do not replace call by goto

-GW : dynamic selected skip() format, warning on long format
-GD : dynamic selected skip() format (default)
-GS : always short skip() format (error if 256 byte 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>.

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

-L[<col>,<lin>] : produce list file <src>.lst
The maximun 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

-LFSR- : do not use the LFSR instruction
-LFSR+ : use the LFSR instruction

-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<name> : write hex file to name

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

-p<device> : defines the chip type (i.e. –pPIC18C242 or –p18C242). The device have to supported by a header file (i.e. 18C242.H). No default device is available.

-p- : clear any preceding -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 interrupt at the deepest call level MUST then be properly ensured in the user application program.

-Q : write the call tree to <src>.fcs.

-r : generate relocatable assembly (no hex file)
-rsc[=][<filename.lkr>] : generate relocatable asm, and update the linker script 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
-rx : make variables static by default

-S : silent operation of the compiler

-u : no optimizing

-V[rnuD] : 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

-wC : warning on upward compatibility issues
-we : no warning when fixed point constants are rounded
-wi : no warning on multiple inline math integer operations
-wm : no warning on single call to math integer function
-wO : warning on operator library calls
-wr : no warning on recursive calls
-wS : warning (no error) when constant expression loses significant bits
-wU : warning on uncalled functions

-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)

-zZ : optimize (inline multiplication) for size
-zD : optimize (inline multiplication) for speed (default)

Doublequotes " " allows spaces in the option:

```
-I"C:\Program Files\cc8e"
```

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

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

Default compiler settings:

- hex file output to file <name>.hex
- optimizing on
- extended call level is allowed
- update bank 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:

```
cc8e [..] +<filename> [..]
```

Many option files can be included, and up to 5 levels of nested include files are allowed. Options in a file allows 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 behind the '-' starting the option. This syntax disables using more than one option on each line. Examples:

```
- D MAC = 1 + OP
- p 18C242      // comment
-p 18C242       // this will not work
- p 18C242  -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 " " allows spaces in the option, quotes are removed
2. Using \" means a single quote " in an option

```
-I"C:\Program Files\cc8e"     ==>   -IC:\Program Files\cc8e
-IC:"\Program Files"\cc8e     ==>   -IC:\Program Files\cc8e

-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 allows 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:

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 Subroutine Call Level Checking

Subroutine calls are limited to 31 levels for the PIC18 devices. 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 in the user code (computed goto) in a function of type *char*.
   b) The number of return literal exceeds 10

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

Stack level checking when using interrupt

CC8E will normally assume that an interrupt can occur anywhere in the main program, 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 need all 31 stack levels for making calls.

The -q<N> option force CC8E to assume that interrupt will NOT occur at the <N> deepest call levels of the main program.

The application writer must then ensure that interrupt will not occur when executing functions at the deepest <N> call levels, normally by using the global interrupt enable bit. CC8E will generate a warning for the critical functions. (The normal error message is always generated when the application contains more than 31 call levels.)

For example, the -q1 option generates a warning for functions calls that will put the return address at stack level 31 (no free stack entry for interrupt). Using -q2 means an additional warning at stack level 30 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 have 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 call itself directly or through another function. This warning can be switched off with the *-wr* command line option.

6.2 Interrupts

The PIC18 devices allows both low priority and high priority interrupts.

The structure of the interrupt service routine is as follows:

```
#include "int18XXX.h"

void _highPriorityInt(void);

#pragma origin 0x8
interrupt highPriorityIntServer(void)
{
    // W, STATUS and BSR are saved to shadow registers

    // handle the interrupt
    // 8 code words available including call and RETFIE
    _highPriorityInt();

    // restore W, STATUS and BSR from shadow registers
    #pragma fastMode
}

#pragma origin 0x18
interrupt lowPriorityIntServer(void)
{
    // W, STATUS and BSR are saved by the next macro.
    int_save_registers

    /* NOTE : shadow registers are updated, but will be
       overwritten in case of a high-priority interrupt.
       Therefore #pragma fastMode should not be used on
       low-priority interrupts. */

    // save remaining registers on demand (error/warning)
    //uns16 sv_FSR0 = FSR0;
    //uns16 sv_FSR1 = FSR1;
    //uns16 sv_FSR2 = FSR2;
    //uns8 sv_PCLATH = PCLATH;
    //uns8 sv_PCLATU = PCLATU;
    //uns8 sv_PRODL = PRODL;
    //uns8 sv_PRODH = PRODH;
    //uns24 sv_TBLPTR = TBLPTR;
    //uns8 sv_TABLAT = TABLAT;

    // handle the interrupt
    // ..
    // restore registers that are saved
    //FSR0 = sv_FSR0;
    //FSR1 = sv_FSR1;
    //FSR2 = sv_FSR2;
```

```
    //PCLATH = sv_PCLATH;
    //PCLATU = sv_PCLATU;
    //PRODL = sv_PRODL;
    //PRODH = sv_PRODH;
    //TBLPTR = sv_TBLPTR;
    //TABLAT = sv_TABLAT;

    int_restore_registers // W, STATUS and BSR
}

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

void _highPriorityInt(void)
{
    // save registers on demand

    // restore registers on demand
}
```

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 have 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. 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 0x8 and 0x18. The interrupt service routines can only be located at these addresses. The #pragma origin statement have 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 0x8/0x18 to the interrupt routine.

```
#pragma unlockISR
```

Vital registers such as STATUS, BSR 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. The file '**int18xxx.h**' contains recommended program sequences for saving and restoring registers. Other registers must be saved manually. The interrupt routine can also contain local variables. Storage for local variables is allocated separately because interrupts can occur anytime.

IMPORTANT: CC8E will AUTOMATICALLY check that vital registers are saved and restored during interrupt. This applies to:

```
Group 1:  W/WREG, STATUS, BSR  : most frequent used
Group 2:  FSR0, FSR1, FSR2     : indirect access
Group 3:  TBLPTR, TABLAT       : reading 'const' data
Group 4:  PRODL, PRODH         : multiplication instructions
Group 5:  PCLATH, PCLATU       : computed goto
```

NOTE that it is not required to save registers before starting to service the interrupt. Section *Custom interrupt save and restore* on page 58, shows a list of instructions that that will not disturb the main registers.

It is possible to limit the save and restore of a specific register to a small region inside the interrupt service routine, if this register is modified only inside this region.

CC8E 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*.

The compiler will detect if the initially mentioned registers are modified during interrupt processing without being saved and restored. The supplied macros for saving and restoring registers will only save W, STATUS and BSR. The other registers have to be saved and restored by user code when needed.

For example, if FSR0 is modified by a table or pointer access, or by direct writing, the compiler will check that FSR0 is saved and restored, also in nested function calls. Note that the FSR0 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 the Group 2 - 5 registers mentioned above are 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. CC8E also supports custom interrupt structures.

A) You may want to use your own save and restore sequence. This can be done by inline assembly. If CC8E does not accept your code, just insert (on 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  ; access RAM/SFR only
bx2 = 1;        // BSF   0x70,bx2  ; access RAM/SFR only
bx1 = 0;        // BCF   0x70,bx1  ; access RAM/SFR only
bx3 = !bx3;     // BTG   0x70,bx3  ; access RAM/SFR only
btsc(bx1);      // BTFSC 0x70,bx1  ; access RAM/SFR only
vs = swap(vs);  // SWAPF vs,1      ; access RAM/SFR only
vs = incsz(vs); // INCFSZ vs,1     ; access RAM/SFR only
nop();          // NOP
vs = decsz(vs); // DECFSZ vs,1     ; access RAM/SFR only
clrwdt();       // CLRWDT
a = b;          // MOVFF a,b       ; all RAM/SFR
..etc.          // CALL, GOTO, BRA, RCALL,..
```

C) It is possible to enable interrupt only in special regions (wait loops) in such a way that main registers can be modified during interrupt without disturbing the main program. The register save can then be omitted and the save checking must be switched off to avoid the error messages:

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

INTERRUPTS CAN BE VERY DIFFICULT. THE PITFALLS ARE MANY.

6.3 Startup and Termination Code

The startup code consists of a jump to *main()*. 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.

It is possible to locate main() in any codepage if the reset vector is omitted. This is done by the following pragma statement. Proper startup code must be inserted manually when removing the automatic reset vector, for example by cdata[] statements (file '**cdata.txt**').

```
#pragma resetVector -
```

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

PIC18 also allows restart by interrupt. A RESET instruction is therefore inserted if main is allowed to terminate (SLEEP). This ensures repeated execution of the main program.

Clearing ALL RAM locations

The internal function *clearRAM()* will set all RAM locations to zero. The generated code uses the 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. Typically 4 or 5 instruction cycles is required for each RAM location. At 4 MHz, each instruction cycle is 1 microsecond. The PIC18C452 device contains 1536 RAM locations which means 1536*5 = 7680 instruction cycles or 7.68 milliseconds at 4 MHz.

6.4 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 routines.

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

CC8E will automatically delete unused library functions. This feature can also be used to delete unused application functions:

```
#pragma library 1
// library functions defined here 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.

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 straight forward to get correct results when using floating point. However, floating point functions requires 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 efficient 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

The min and max timing cycles are approximate only.

Sign: -: unsigned, S: signed

Sign Res=arg1 op arg2   Program   Approx. CYCLES

A:math32.h
 B:math24.h
  C:math16.h              Code     min  aver   max
.B.  S   24 = 16 * 16      31       45    45    47
A..  S   32 = 16 * 16      41       53    54    59
A..  -   32 = 16 * 16      29       47    47    47
.B.  -   24 = 24 * 24      31       51    51    51
A..  S   32 = 32 * 16      49       68    68    71
A..  -   32 = 32 * 16      43       65    65    65
A..  S/- 32 = 32 * 32      57       83    83    83

ABC  -   16 = 16 / 8       19      231   231   231
AB.  -   24 = 24 / 8       20      364   364   364
```

```
A..  -   32 = 32 / 8      21      513   513   513
ABC  -   16 = 16 / 16     22      234   237   282
.B.  -   24 = 24 / 16     25      391   407   487
A..  -   32 = 32 / 16     26      548   576   676
.B.  -   24 = 24 / 24     29      442   448   562
A..  -   32 = 32 / 32     36      714   724   938

ABC  S   16 = 16 / 8      33      192   196   205
AB.  S   24 = 24 / 8      35      301   305   316
A..  S   32 = 32 / 8      37      426   431   443
ABC  S   16 = 16 / 16     43      243   251   304
.B.  S   24 = 24 / 16     45      376   395   463
A..  S   32 = 32 / 16     47      525   555   638
.B.  S   24 = 24 / 24     52      451   463   586
A..  S   32 = 32 / 32     61      723   740   964

ABC  -    8 = 16 % 8      18      222   222   222
.B.  -    8 = 24 % 8      19      353   353   353
A..  -    8 = 32 % 8      20      500   500   500
ABC  -   16 = 16 % 16     20      227   229   259
.B.  -   16 = 24 % 16     23      374   393   454
A..  -   16 = 32 % 16     24      521   557   633
.B.  -   24 = 24 % 24     27      434   439   530
A..  -   32 = 32 % 32     34      705   713   897

ABC  S    8 = 16 % 8      26      186   186   189
.B.  S    8 = 24 % 8      28      294   295   298
A..  S    8 = 32 % 8      30      417   418   422
ABC  S   16 = 16 % 16     40      238   242   277
.B.  S   16 = 24 % 16     42      371   382   426
A..  S   16 = 32 % 16     44      519   537   591
.B.  S   24 = 24 % 24     50      445   453   549
A..  S   32 = 32 % 32     60      716   728   918
```

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.

```
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      44       51    53    57
-   8_8 = 8_8 * 8_8      23       39    39    39
S   8_8 = 8_8 / 8_8      46      423   441   508
-   8_8 = 8_8 / 8_8      29      438   454   534

math24x.h:              Code     min  aver   max
S   16_8 = 16_8 * 16_8   74       84    88    92
-   16_8 = 16_8 * 16_8   50       72    72    72
```

```
S   16_8 = 16_8 / 16_8   55      687   719   862
-   16_8 = 16_8 / 16_8   36      710   735   902

S   8_16 = 8_16 * 8_16   89       99   103   107
-   8_16 = 8_16 * 8_16   65       87    87    87
S   8_16 = 8_16 / 8_16   55      847   896  1062
-   8_16 = 8_16 / 8_16   36      878   919  1118

math32x.h:              Code     min  aver   max
S   24_8 = 24_8 * 24_8  150      125   129   135
-   24_8 = 24_8 * 24_8  123      113   113   113
S   24_8 = 24_8 / 24_8   64     1015  1055  1312
-   24_8 = 24_8 / 24_8   43     1046  1080  1366

S   16_16= 16_16*16_16  135      148   152   158
-   16_16= 16_16*16_16  108      136   136   136
S   16_16= 16_16/16_16   64     1207  1273  1560
-   16_16= 16_16/16_16   43     1246  1305  1630

S   8_24 = 8_24 * 8_24  112      163   167   173
-   8_24 = 8_24 * 8_24   85      151   151   151
S   8_24 = 8_24 / 8_24   64     1399  1491  1808
-   8_24 = 8_24 / 8_24   43     1446  1529  1894
```

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.

```
Basic 32 bit math:              Approx. CYCLES
                         Size   min  aver   max
a * b:  multiplication    131   128   132   147
a / b:  division          104   441   495   580
a + b:  addition          154    38   123   207
a - b:  subtraction     add+5    45   130   200
int32 -> float32           70    42    69   115
float32 -> int32           75    36    81   138

Basic 24 bit math:              Approx. CYCLES
                         Size   min  aver   max
a * b:  multiplication     82    78    81    93
a / b:  division           87   270   297   345
a + b:  addition          132    32   104   161
a - b:  subtraction     add+5    39   111   168
int24 -> float24           56    34    63   105
float24 -> int24           65    31    68   112

Basic 16 bit math:              Approx. CYCLES
```

```
                         Size   min  aver   max
a * b:  multiplication     51    46    49    57
a / b:  division           73   121   136   151
a + b:  addition          104    26    80   123
a - b:  subtraction   add+5    33    87   130
int16 -> float16           63    37    67   103
float16 -> int16           48    26    57    96
```

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

Floating point library functions

```
float24 sqrt(float24);   // square root
  Input range: positive number including zero
  Accuracy: ME:1, relative error: 1.5*10**-5 (*)
  Timing: min aver max   532   572   614  (**)
  Size: 53 words
  Minimum complete program example: 68 words

float32 sqrt(float32);   // square root
  Input range: positive number including zero
  Accuracy: ME:1, relative error: 6*10**-8 (*)
  Timing: min aver max   955  1038  1111  (**)
  Size: 63 words
  Minimum complete program example: 84 words

float24 log(float24);   // natural log function
  Input range: positive number above zero
  Accuracy: ME:1, relative error: < 1.5*10**-5 (*)
  Timing: min aver max  1210  1714  1985  (**)
  Size: 210 words + basic 24 bit math library
  Minimum complete program example: 584 words

float32 log(float32);   // natural log function
  Input range: positive number above zero
  Accuracy: ME:1, relative error: < 6*10**-8 (*)
  Timing: min aver max  1713  2377  2699  (**)
  Size: 264 words + basic 32 bit math library
  Minimum complete program example: 743 words

float24 log10(float24);   // log10 function
  Input range: positive number above zero
  Accuracy: ME:1-2, relative error: < 3*10**-5 (*)
  Timing: min aver max  1293  1794  2067  (**)
  Size: 15 words + size of log()
  Minimum complete program example: 599 words

float32 log10(float32);   // log10 function
  Input range: positive number above zero
  Accuracy: ME:1-2, relative error: < 1.2*10**-7 (*)
  Timing: min aver max  1840  2502  2793  (**)
  Size: 17 words + size of log()
  Minimum complete program example: 760 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  903  1539  1725  (**)
  Size: 247 words + 76(floor24) + basic 24 bit math
  Minimum complete program example: 641 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  1920  2073  2301  (**)
  Size: 317 words + 115(floor32) + basic 32 bit math
  Minimum complete program example: 834 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  917  1610  1739  (**)
  Size: 254 words + 76(floor24) + basic 24 bit math
  Minimum complete program example: 648 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  1565  2086  2335  (**)
  Size: 323 words + 115(floor32) + basic 32 bit math
  Minimum complete program example: 840 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   374  1195  1396  (**)
  Size: 211 words + basic 24 bit math library
  Minimum complete program example: 563 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  1789  2193  2529  (**)
  Size: 352 words + basic 32 bit math library
  Minimum complete program example: 807 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. 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.

Fixed point example

```
#pragma chip PIC18C242
#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: 266 code words including library (129)

Floating point example

CODE: 596 code words including library (424). The statements are identical to the above fixed point example to enable code size comparison.

```
#pragma chip PIC18C242
#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. Note that “optimize for speed” is default. Also note that the code saving is small.
```
#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. CC8E can print a warning for each operator function that is called (option -wO).

6.5 Inline Assembly

The CC8E 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.

```
#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 updating

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. CC8E 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.

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

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

```
clrwdt
Nop
NOP
```

The supported operand formats are:

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

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

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
```

Note that the specification of access bank (,0) or banked access (,1) is OPTIONAL, and automatically decided by the variable accessed. If this information is added, then the compiler checks that it is equal to the access required by the variable. Variables residing at address 0-0x7F and 0xF80-0xFFF must use the access bank. All other accesses must be banked.

```
decf  ax,W,0 // load result into W, ax in access bank
decf  ax,W,1 // load result into W, ax in bank 0 .. 15
decf  ax,W   // load result into W, implicit banked/accessbank
```

Formats when loading then result into the W register (implicit banked/accessbank):

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

Formats when writing the result back to the RAM register (implicit banked/accessbank):

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

Bit variables are accessed by the following formats (implicit banked/accessbank):

```
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);
rcall 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 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 are:

```
NOP    ; a comment
NOP    // C style comments are also valid
/*
CLRWDT   ;
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
clrwdt
#endif
```

Most preprocessor statements can be used in assembly mode:

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

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)\
clrwdt\
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);
```

The compiler can optimize and perform bank 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 updating. Instructions updating the bank 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-
#pragma asm default b+ o+   // change default settings

#asm             // using default local settings
#endasm

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

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
```

Note that *b+ o+* 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* 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 __MOVWF(__INDF0,0)         // Store indirectly
DW __ANDLW(0x80)              // W = W & 0x80;
DW __DECF(__FSR0L,__F,0)      // Decrement FSR0L (access bank)
DW __CLRF(0xFF,1)             // Clear ram (banked access)
DW __BCF(__STATUS,__Carry,0) // Clear Carry bit
DW __BRA(3)                   // Branch 3 instruction words forward
DW __BRA(0)                   // Branch 0 (= no operation)
DW __BRA(-1)                  // Branch -1 backward (infinite loop)
DW __BC(-2)                   // Branch on Carry 2 words backwards
DW __MOVFF(__INDF0, __INDF1) // Move byte indirectly
DW __LFSR(0, 0x130)           // Load 12 bit constant into FSR0
DW __GOTO(0)                  // Goto byte address 0
#endasm
..
// 2. In cdata statements:
#pragma cdata[1] = __GOTO(0x3FF)
```

Generating single instructions using C statements

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

```
f = subWFB(f); // SUBWFB f
W = subFWB(f); // SUBFWB f,W
f = subFWB(f); // SUBFWB f
W = rrnc(f);   // RRNCF f,W
f = rrnc(f);   // RRNCF f
W = rlnc(f);   // RLNCF f,W
f = rlnc(f);   // RLNCF f
W = decsnz(i); // DCFSNZ f,W
f = decsnz(i); // DCFSNZ f
W = incsnz(i); // INFSNZ f,W
f = incsnz(i); // INFSNZ f
f = negate(f); // NEGF f
W = decadj(W); // DAW
multiply(f);   // MULWF f
multiply(50);  // MULLW 50
skipIfEQ(f);   // CPFSEQ f
skipIfLT(f);   // CPFSLT f
skipIfGT(f);   // CPFSGT f
skipIfZero(f); // TSTFSZ f
pushStack();   // PUSH
popStack();    // POP
softReset();   // RESET
tableRead();   // TBLRD *
tableReadInc();// TBLRD *+
tableReadDec();// TBLRD *-
tableReadPreInc(); // TBLRD +*
tableWrite();   // TBLWT *
tableWriteInc();// TBLWT *+
tableWriteDec();// TBLWT *-
tableWritePreInc(); // TBLWT +*
```

6.6 Optimizing the Code

The CC8E 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 12 instructions. When comparing a 32 bit variable with 0, this count is reduced to 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

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 PA0 and PA1
7) remove other superfluous instructions
8) remove superfluous loading of the W register
9) to be defined
10) inserts TSTFSZ, CPFSEQ
11) inserts branch

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

6.7 Linker Support

CC8E supports the relocatable assembly format defined by Microchip. This means that MPLINK can be used to link code modules generated by CC8E, 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
- call level checking
- MPLINK script files
- MPLAB integration

The command line option '-rsc' (or '-r2' or '-r') makes CC8E 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:

```
-rsc[=<file.lkr>] : generate relocatable assembly and use separate
logical sections for the interrupt routines. Also generate or update
the complete linker script file. If the linker script file name is
not specified, then the C module name will be used with the extension
'.lkr'. The script file will only be generated when compiling the
module containing main().

-r2[=][<file.lkr>] : generate relocatable assembly with separate
logical sections for the interrupt routines. A partial linker script
file (containing dynamic definitions of page and intserv8 or
intserv18) is generated and should be included in the main linker
script file. If the partial linker script file name is not specified,
then the C module name will be used with the extension '.lkr'.
```

```
-r : generate relocatable assembly without separate logical sections
for the interrupt routines. NO linker script file is generated.

-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):

```
-rb<N> : name on RAM bank 0 is BANK<N>, default BANK0
-ro<N> : add offset <N> when generating 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, CC8E 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 CC8E.

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
```

Restrictions on the demo edition

There are some restrictions when using relocatable assembly on the CC8E DEMO:

1. A single demo C module is possible, plus many assembly modules.
2. The demo C module can call EXTERN functions and use extern variables defined in assembly modules.
3. The demo C module is NOT allowed to export extern functions or variables.
4. main() must be defined in the demo C module. The interrupt routines should preferably also be defined in the demo C module.
5. The demo C module can maximum contain 1024 instructions

Variables and pointers

Variables defined in other module can be accessed. CC8E 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. CC8E inserts 'GLOBAL' statements in the generated assembly file.

CC8E 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 CC8E will define a global char variable with random name. This name have the format '_Gbit<X><X>' where <X> is (more or less) random selected letters. 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 CC8E. 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 CC8E 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 16 bit wide.

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 a specific bank by a pragma statement:

```
extern size1 char *px;
#pragma assume *px in rambank 2
```

Note that 8 bit pointers in a struct can only access addresses from 0 - 255, even if the struct is static or local.

Local variables

CC8E uses a different naming strategy on local variables when generating relocatable assembly. CC8E reserves a continuous block in each ram bank (or access 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. CC8E 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;   // access RAM
```

Bank bit updating

CC8E 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 CC8E 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 func2(void) { .. }  // can be called from another module
```

NOTE that extern functions can only have a single unsigned 8 bit parameter which is transferred in W (not const/pointer/bit). 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

CC8E 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 sections

It is possible to use #pragma origin and #pragma cdata when the compiler knows the section the following code should be placed in. The compiler automatically knows the STARTUP, ISERVER8, ISERVER18 and PROG sections. In addition it is possible to define sections manually in the C file. These definitions will also automatically appear in the script file when using the -rsc option.

```
#pragma sectionDef IDLOC:idlocs 0x200000 - 0x200007  PROTECTED
#pragma sectionDef CONFIGS:config 0x300000 - 0x30000D  PROTECTED
#pragma sectionDef EEPROM:eedata 0xF00000 - 0xF000FF  PROTECTED
#pragma sectionDef APPSEC:appdef1  0x1000 - 0x102F
#pragma sectionDef PROG
```

Note the difference between using:

```
#pragma origin 0x1000          => generates:  APPSEC CODE 0x1000
```

and

```
#pragma origin SECTION(APPSEC) => generates:  APPSEC CODE
```

The first origin says "from the start of the APPSEC section", while the last origin says "somewhere" in the APPSEC section". Locating code from a specific address is sometimes useful, but note that MPLINK does not allow the above origin statements to be mixed for code belonging to the same section.

If the sections starting at address 0x200000 and 0x300000 are not defined, then the compiler will automatically use section names IDLOCS and CONFIG. However, it is recommended to use the above definitions in the C file, especially when using the -rsc option.

The compiler estimate the current hex address, and use this to detect when to insert the ISERVER8 and ISERVER18 sections when using the '-rsc' or '-r2' options. Sometimes these sections should not be inserted. This problem will occur infrequently and is easily detected because MPLINK will report the conflict. This problem can be solved by inserting the following statements at the right place:

```
#pragma sectionDef PROG
#pragma origin SECTION(PROG)
```

NOTE: It is recommended to use code sections of maximum 64k byte, without crossing a 64k boundary ((START & 0xFF0000) should be equal (END & 0xFF0000)).

Interrupts

CC8E requires that the interrupt functions are located at address 8 and 0x18. Writing the interrupt service routine in C using MPLINK will require some care.

The best method is to use SEPARATE logical sections in the linker script file for the interrupt service routines. This is a robust solution. CC8E will generate a full (or partial) script file to avoid manual address calculation. Node that #pragma origin 0x8 and 0x18 must be used for the interrupt routines.

It is also possible to design an assembly module containing the interrupt service routines. Information on how to do this should be found in the MPASM/MPLINK documentation.

Call level checking

CC8E will normally check that the call level is not exceeded. This is only partially possible when using MPLINK. CC8E can ONLY check the current module, NOT the whole linked application.

When calling an external function from the C code, CC8E 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 of the call level.

Therefore, careful verification of the call structure is required to avoid program crash when using too deep calls (max 31 levels). 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

CC8E will always use the long format when generating code for skip(). It is not possible to use the -GS option in combination with relocatable assembly.

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) 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;
```

b) 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. It is recommended to use the -rsc option to enable the compiler to AUTOMATICALLY generate and later update the linker script.

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: The bank selection bits should be updated in the beginning of assembly functions that are called from C.

MPLAB and MPASM support

Please refer to file '**linker.txt**' for details on how to set up a project with several modules in MPLAB.

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.

CC8E 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 CC8E 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.

Options to start MPASM and generate relocatable object code:

```
-x"C:\Program Files\Microchip\MPASM Suite\mpasmwin.exe" -X/o -X/q
```

Options when assembling and linking a single file:

```
 -x"C:\Program Files\Microchip\MPASM Suite\mpasmwin.exe" -X/q
```

If the CC8E error file option (-F) is missing, CC8E 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 CC8E error file option (-F) is present, CC8E 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.

Note that MPLAB will automatically show the contents of the error file if it is non-empty. Otherwise the screen output is displayed in 'Build Results' window.

The MPLINK script file

The compiler is able to generate the FULL linker script file. This is done when using command line option -rsc[=<file.lkr>]. The compiler will automatically do the following when the C module contains interrupt routine(s):

a) Generate or update a complete script. If the file exists initially, then it should preferably contain an empty line. The script file will only be generated when compiling the module containing main().

b) Add, remove and adjust definitions that are maintained by the compiler.

c) Forward code section definitions to the script file, and allow the definitions to be used in #pragma origin statements etc.:

```
 #pragma sectionDef <secID>[:<secDef> <start> - <last> [<PROTECTED>]]

 #pragma sectionDef IDLOC:idlocs 0x200000 - 0x200007  PROTECTED
 #pragma sectionDef CONFIGS:config 0x300000 - 0x30000D  PROTECTED
 #pragma sectionDef EEPROM:eedata 0xF00000 - 0xF000FF  PROTECTED
 #pragma sectionDef APPSEC:appdef1  0x1000 - 0x102F
 #pragma sectionDef PROG
```

d) Code definition will be adjusted automatically to ensure that there are no overlapping.

e) When a variable is located at a fixed address, the compiler will automatically EXCLUDE this address from the default bank definition in the script file. The largest free region in the bank will be defined, and made available for allocation by MPLINK. It is important that the main C module knows all fixed RAM definitions. Otherwise it is required to make a manual definition in the script file to exclude "unknown" locations from normal allocation.

It is also possible to make the linker script file manually, although this should normally not be preferred. 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 CC8E.

The sample MPLINK script files must be changed slightly if the interrupt function is written in C. The reason is that the interrupt functions must start at addresses 8 and 0x18 when using CC8E. It could be possible to use a vector at address 8 / 0x18, but this slows down interrupt response.

CHANGE 1: Interrupt routine in C WITH a separate logical section. CC8E generates a partial script file when using the -r2 command line option (or -r2[=]<file.lkr>). This file is written if (and only if) CC8E compiles a module with an interrupt service routine. The generated script file may look like:

```
CODEPAGE  NAME=intserv18  START=0x18 END=0x31
CODEPAGE  NAME=page       START=0x32 END=0x7FFF
```

The required change in the main script file is then:

```
INCLUDE  module1.lkr   // change to right module/script file name
```

CHANGE 2: Interrupt routine in C WITHOUT a separate logical section. Example change:

```
CODEPAGE   NAME=vectors  START=0x0  END=0x7   PROTECTED
// NEW VALUE                            ^------

CODEPAGE   NAME=page     START=0x8  END=0x7FFF
// NEW VALUE                    ^------
```

CHANGE 3: If INTERRUPTS are not used, then the first code page can start at address 4. Example change:

```
CODEPAGE NAME=vectors    START=0x0  END=0x3    PROTECTED
// NEW VALUE                            ^------

CODEPAGE   NAME=page     START=0x4  END=0x7FFF
// NEW VALUE                    ^------
```

CHANGE 4: LOGICAL sections must be added. Note that if a logical RAM section is missing, then the variables that belongs 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=STARTUP   ROM=vectors    // Reset vector
SECTION    NAME=ISERVER8  ROM=intserv8   // High priority interrupt
SECTION    NAME=ISERVER18 ROM=intserv18  // Low priority interrupt
SECTION    NAME=PROG      ROM=page       // code space
SECTION    NAME=IDLOCS    ROM=idlocs     // ID locations
SECTION    NAME=CONFIG    ROM=config     // Configuration bits
location
SECTION    NAME=EEDATA    ROM=eedata     // EEPROM data

SECTION    NAME=ACSRAM    RAM=accessram  // ACCESS RAM
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=BANK4     RAM=gpr4       // RAM bank 4
SECTION    NAME=BANK5     RAM=gpr5       // RAM bank 5
```

CHANGE 5: modifications when using ICD2:

```
CODEPAGE    NAME=page       START=..          END=0x7DBF
CODEPAGE    NAME=debug      START=0x7DC0      END=0X7FFF     PROTECTED

DATABANK    NAME=gpr5       START=0x500       END=0x5F3
DATABANK    NAME=dbgspr     START=0x5F4       END=0x5FF      PROTECTED
```

Logical code blocks:

```
STARTUP   startvector
ISERVER8  logical section for the high priority interrupt
ISERVER18 logical section for the low priority interrupt
```

```
PROG      code space
CONFIG    config word
IDLOCS    id-locations
EEDATA    EEPROM data
```

Logical RAM blocks:

```
ASCRAM    Access RAM
BANK0     bank 0
BANK1     bank 1
BANK2     bank 2
BANK3     bank 3
BANK4     bank 4
BANK5     bank 5
..
BANK15    bank 15
```

Command line options:

Bank naming:

```
-rb0   : BANK0 is the name of the frist RAM bank (default)
-rb1   : BANK1 is the name of the frist RAM bank
```

Separate interrupt logical section (named ISERVER8/ISERVER18)

```
-r2            : use name of current module (<module>.lkr)
-r2[=]<file.lkr>  : use defined file name
```

Example with 2 modules

This example demonstrates the syntax only.

```
// ******************************************************
// MODULE1.C
#pragma chip PIC18F452
#include "globdef1.h"
#include "int18xxx.H"

void _highPriorityInt(void);

#pragma origin 0x8
interrupt highPriorityIntServer(void)
{
    // W, STATUS and BSR are saved to shadow registers

    // handle the interrupt
    // 8 code words available including call and RETFIE
    _highPriorityInt();

    // restore W, STATUS and BSR from shadow registers
    #pragma fastMode
}

#pragma origin 0x18
interrupt lowPriorityIntServer(void)
{
    // W, STATUS and BSR are saved by the next macro.
    int_save_registers

    /* NOTE : shadow registers are updated, but will be
```

```
       overwritten in case of a high-priority interrupt.
       Therefore #pragma fastMode should not be used on
       low-priority interrupts. */

    // save remaining registers on demand (error/warning)
    //uns16 sv_FSR0 = FSR0;
    //uns16 sv_FSR1 = FSR1;
    //uns16 sv_FSR2 = FSR2;
    //uns8 sv_PCLATH = PCLATH;
    //uns8 sv_PCLATU = PCLATU;
    //uns8 sv_PRODL = PRODL;
    //uns8 sv_PRODH = PRODH;
    //uns24 sv_TBLPTR = TBLPTR;
    //uns8 sv_TABLAT = TABLAT;

    // handle the interrupt
    // ..

    // restore registers that are saved
    //FSR0 = sv_FSR0;
    //FSR1 = sv_FSR1;
    //FSR2 = sv_FSR2;
    //PCLATH = sv_PCLATH;
    //PCLATU = sv_PCLATU;
    //PRODL = sv_PRODL;
    //PRODH = sv_PRODH;
    //TBLPTR = sv_TBLPTR;
    //TABLAT = sv_TABLAT;

    int_restore_registers // W, STATUS and BSR
}

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

void _highPriorityInt(void)
{
    // save registers on demand

    // restore registers on demand
}

bank0 char a;
bit b1, b2;

static char *ppm;

shrBank char sr;

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
    }
}

void main( void)
{
    PORTA = 0b0010;
    TRISA = 0b0001;

    if (TO == 1 && PD == 1 /* power up */)  {
        clearRAM();  // set all RAM to 0
        a = 5;
        b1 = 1;
    }
    ppm = 0;
    sr ++;
    a = reverse(sr);  // call assembly routine

    b2 = !b1;
    do  {
        if (in == 1)
             break;
        sub(a&3);
    } while (a < 200);
}


// ******************************************************
// File: globdef1.h
// GLOBAL DEFINITIONS TO BE INCLUDED IN ALL C MODULES

// names assigned to port pins
#pragma bit in  @ PORTA.0
#pragma bit out @ PORTA.1

// module1.c
extern bank0 char a;

// module2.asm
extern bank1 char result;
extern char reverse( char W);


; ******************************************************
; MODULE2.ASM
        #INCLUDE "P18F452.INC"

BANK1   UDATA
result  RES 1 ; result holder
tmp     RES 1 ; temporary location
count   RES 1 ; loop counter
```

```
        GLOBAL result

PROG    CODE
reverse
        GLOBAL  reverse
        movlb 1
        movwf tmp
        movlw 8
        movwf count
loop    rrcf tmp, F, 1
        rlcf result, F, 1
        decfsz count, F, 1
        goto loop
        movf result, W, 1
        return

        END


// ******************************************************
// File: 18F452.LKR

// This example linker script file is for use with the -r2 option.

// However, it is recommended to instead use the -rsc option to let
// the compiler automatically generate and update the whole script!

LIBPATH  .

CODEPAGE    NAME=vectors    START=0x0        END=0x7        PROTECTED
CODEPAGE    NAME=intserv8   START=0x8        END=0x17
INCLUDE     module1.lkr
// *** File 'module1.lkr' is generated by the compiler when
// *** using option -r2, and defines 'intserv18' and 'page'.
CODEPAGE    NAME=idlocs     START=0x200000   END=0x200007   PROTECTED
CODEPAGE    NAME=config     START=0x300000   END=0x30000D   PROTECTED
CODEPAGE    NAME=devid      START=0x3FFFFE   END=0x3FFFFF   PROTECTED
CODEPAGE    NAME=eedata     START=0xF00000   END=0xF000FF   PROTECTED

ACCESSBANK  NAME=accessram  START=0x0        END=0x7F
DATABANK    NAME=gpr0       START=0x80       END=0xFF
DATABANK    NAME=gpr1       START=0x100      END=0x1FF
DATABANK    NAME=gpr2       START=0x200      END=0x2FF
DATABANK    NAME=gpr3       START=0x300      END=0x3FF
DATABANK    NAME=gpr4       START=0x400      END=0x4FF
DATABANK    NAME=gpr5       START=0x500      END=0x5FF
ACCESSBANK  NAME=accesssfr  START=0xF80      END=0xFFF      PROTECTED

SECTION    NAME=STARTUP   ROM=vectors    // Reset vector
SECTION    NAME=ISERVER8  ROM=intserv8   // High priority interrupt
SECTION    NAME=ISERVER18 ROM=intserv18  // Low priority interrupt
SECTION    NAME=PROG      ROM=page       // code space
SECTION    NAME=IDLOCS    ROM=idlocs     // ID locations
SECTION    NAME=CONFIG    ROM=config     // Configuration bits loc.
SECTION    NAME=EEDATA    ROM=eedata     // EEPROM data
```

```
SECTION    NAME=ACSRAM    RAM=accessram  // ACCESS RAM
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=BANK4     RAM=gpr4       // RAM bank 4
SECTION    NAME=BANK5     RAM=gpr5       // RAM bank 5
```

6.8 The cdata Statement

The cdata statement stores 16 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 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 (except config, ID and EEPROM data). The data is added at the end of the assembly and hex file in the same order as it is defined.

SYNTAX:

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

ADDRESS: 24 bit byte address
VXS : < VALUE | EXPRESSION | STRING>
VALUE: 0 .. 0xFFFF
EXPRESSION: C constant expr. (i.e. 0x1000+(3*1234))
STRING: "Valid C String\r\n\0\x24\x8\xe\xFF\xff\\\""

String translation: \xHH or \xH  : hexadecimal number
  \0 => 0   \1 => 1    \2 => 2    \3 => 3    \4 => 4
  \5 => 5   \6 => 6    \7 => 7    \a => 7    \b => 8
  \t => 9   \n => 10   \f => 12   \v => 11   \r => 13
  \\ => the backslash character itself (0x5C)
  \" => '"' (0x22)
  \xHH  or  \xH  : hexadecimal number
  "\x1Conflict"  is better written as  "\x1" "Conflict"
```

Strings are stored as 8 bit ASCII characters. The least significant 8 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 2    // goto main at byte address 2
```

2. Storing packed strings and other data

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[] = 0xFFFF, 0x2000, 0x1000
#pragma cdata[] = 0x100, (10<<4) + 3456,\
                  10, 456, 10000

#define D8(l,h)  (((l)&0xFF) + ((h)&0xFF)*256)
#define D32(x)   x%0x10000, x/0x10000
#pragma cdata[] = D8(10,20), D32(10234543)

#pragma cdata.ID0 = 0x10, 200+3000
#pragma cdata.ID1 = “Hello world\0"
#pragma cdata.ID2 = “Another string\r\n" “merged”

#pragma cdata.ID_TABLE = ID0, ID1, ID2  // store addresses

#pragma cdata.CDATA_END    // end of cdata block
..
#pragma origin CDATA_END  // program code follow here

void write(uns16 strID);
..

write(ID1);
write(ID2);
```

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 0xF00000 and forward for transfer to the internal EEPROM during programming of a device. Note that each cdata item is 16 bit wide, and will thus define 2 EEPROM locations. The low 8 bit will be stored first (even addresses) then the high 8 bit (odd addresses). The compiler does not know how much EEPROM space a device has.

```
#define EEPROM_START  0xF00000
#pragma cdata[EEPROM_START]      // start of cdata block
#pragma cdata[] = 0xFFFF, 1000   // 4 bytes EEPROM data

#define D8(l,h)  (((l)&0xFF) + ((h)&0xFF)*256))
#pragma cdata[] = D8( 10, 20), D8( 0, 2+5)  // 4 bytes
```

Strings will be stored as a number of 2*8 bits when using cdata.

```
#pragma cdata[] = "Hello world!\0"
```

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 updateBank
3) command line option: -u
4) command line option: -bu
5) 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 RAM banks is important.

- too deep call level

Solution: rewrite the code. Remember that the compiler handles most cases where functions are called once only.

7.2 MPLAB Debugging Support

The CC8E compiler can be used inside the MPLAB environment. The COD file format for debugging purposes is supported. Two modes of source file debugging are available:

a) Using the C source file(s).

b) Using the generated assembly file as the source 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:

-CC<filename>: generate debug file using C source file(s). *<filename>* is optional. The asm file option is also switched on.
-CA<filename>: generate debug file using generated assembly file as source. *<filename>* is optional. The asm file option is also switched on.

Arrays:

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.

ICD2 debugging

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:

```
-DICD2_DEBUG
```

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

```
#define ICD2_DEBUG
..
#pragma chip PIC18F452    // or  #include "18F452.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 FSRx and INDFx (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 produces a compiler output file and a hex file that may be used for programming the PICmicro chips directly. 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, COD, preprocessor output and error files

8.1 Hex File

The default hex file format is INHX32. 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.
```

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[scHDftumiJN+N+N]*.

s: symbolic arguments are replaced by numbers
c: no C source code is printed
H: hexadecimal numbers only
D: decimal numbers only
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
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[rnuD]*. 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

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 ->  -  : access RAM
         0  : bank 0
         1  : bank 1
         .. etc.

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

   Size -> size in bytes (0 for bit)

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

Examples:

```
 X  B  Address  Size  #AC   Name
 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_
```

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 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
    func2                 : #5
    delay                 : #2
    func3                 : #3
```

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)

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 31 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 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.

Note that PCLATU and PCLATH in most cases have to be updated before writing to PCL. The compiler can do ALL updating and checking automatically.

Built in skip(), skipL(), skipM() functions for computed goto

The different skip functions allows both single and double word instructions to be used in the table.

1. skip(i): all instructions in the table must be single word, using single word increments.
2. skipL(i): all instructions in the table must be double words, using double word increments.
3. skipM(i): both double and single word instructions in the table, using single word increments.

Note that the compiler will check the table contents when using skip() and skipL(), and generate en error message if the single/double word conditions are not met. Also note that ‘goto LABEL;’ statements inside the table will not be changed to a branch if skipL() is used. This makes it easy to change a skip() table to a skipL() table if a branch is out of reach.

The skip functions use 8 bit parameters only. Note however that the range of skipL() is 128 double word instructions. Carry is automatically generated if the table cross a 256 byte address boundary. Options available:

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

When using the -GS option, CC8E will generate an error if the table cross a 256 byte 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 byte 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. Relocatable assembly requires another approach to fix the address. This is found in Section *Using code sections* on page 79 in Chapter 6.7 *Linker Support*.

Example: A function contains a computed goto. After inspecting the generated list file, there are 15 instructions words between the function start and the address latch update instruction (MOVF PCL,W,0 updates PCLATH and PCLATU). The last computed goto destination address (offset 10) resides further 2 + 10 instructions words below the address latch update instruction. A fast a compact computed goto requires that these addresses resides on the same “byte page” (i.e. (address & 0xFFFF00) are identical for the two addresses). This is achieved with the statement:

```
#pragma alignLsbOrigin -(15+1)*2 to 254 – (2+10)*2 – (15+1)*2
```

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

16 bit computed goto

The skip inline function can use a 16 bit argument. This requires normally 11 instructions (14 instructions when a 64k byte boundary is crossed). When using relocatable assembly 14 instructions are used on devices with more than 64k byte memory.

```
uns16 s16;
..
skip(s16);
// skip to any instruction in a 65536 instruction word table
// offset 0 is the first instruction in the table
// offset 1 is the next instruction word in the table

/* skipM(s16) must be used when the table contains double
word instructions. The second word of a double word
instruction executes a NOP if jumped to. */

/* The following pragma is recommended if the function does
not end immediate after the table */
#pragma computedGoto 0
/* Optional C statements after the table */
} /* end of function */
```

Computed goto regions

The compiler enters a goto region when skip, skipL or skipM 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
3. 256 byte boundary checks

Examples

```
char sub0(char i)
{
    skip(i); // jumps 'i' code words 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;
}

void sub3(char s)
{
    /* the next statements could also be written as
       a switch statement, but this solution is
       fastest and most compact. */

    if (s >= 3)
        goto Default;
    skip(s);

    goto Case0;
    goto Case1;
    goto LastCase;
  #pragma computedGoto 0  // end of c-goto region

  Case0:
    /* user statements */
    return;

  Case1:
  LastCase:
    /* user statements */
    return;

  Default:
    /* user statements */
    return;
}

void sub4(char s)
{
    /* this solution can be used if very fast
       execution is important and a fixed number of
       instructions (2/4/8/..) is executed at each
       selection. Please note that extra statements
       have to be inserted to fill up empty space
       between each case. */
```

```
    if (s >= 10)
        goto END;
    s = rlnc(s);  /* multiply by 2 */
    s = rlnc(s);  /* multiply by 2 */
    skip(s);

    // execute 4 instructions at each selection
  Case0:  nop(); nop(); nop(); return;
  Case1:  nop(); nop(); nop(); return;
  Case2:  nop(); nop(); nop(); return;
  Case3:  nop(); nop(); nop(); return;
  Case4:  nop(); nop(); nop(); return;
  Case5:  nop(); nop(); nop(); goto END;
  Case6:  nop(); nop(); nop(); goto END;
  Case7:  nop(); nop(); nop(); goto END;
  Case8:  nop(); nop(); nop(); goto END;
  Case9:  nop(); nop(); nop(); goto END;
  #pragma computedGoto 0  /* end of region */
  END:
  ; // More statements
}
```

9.2 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 Predefined Register Names

All register names, including the predefined ones, are found in the header files. The predefined register names are:

```
char TOSU, TOSH, TOSL;
char STKPTR;

char *TBLPTR;  // 24 bit address
char TBLPTRU, TBLPTRH, TBLPTRL, TABLAT;

char PCLATU, PCLATH, PCL;
char PRODH, PRODL;
char INTCON, INTCON2, INTCON3;
char W, WREG;
char BSR, BSRL;

size2 char *FSR0, *FSR1, *FSR2;
char INDF0, POSTINC0, POSTDEC0, PREINC0, PLUSW0, FSR0H, FSR0L;
char INDF1, POSTINC1, POSTDEC1, PREINC1, PLUSW1, FSR1H, FSR1L;
char INDF2, POSTINC2, POSTDEC2, PREINC2, PLUSW2, FSR2H, FSR2L;

char STATUS;
bit Carry, DC, Zero_, Overflow, Negative;
```

A2 Assembly Instructions

```
Assembly:      Status:      Operation:
ADDLW  k       C,DC,Z,N,OV  W = k + W;   Add literal and W
ADDWF  f,d,a   C,DC,Z,N,OV  d = f + W;   Add W and f
ADDWFC f,d,a   C,DC,Z,N,OV  d = f + W + C;  Add with Carry
ANDLW  k            Z,N     W = W & k;   AND literal and W
ANDWF  f,d,a        Z,N     d = f & W;   AND W and f
BC     l            -       Branch if Carry
BN     l            -       Branch if Negative
BNC    l            -       Branch if No Carry
BNN    l            -       Branch if Not Negative
BNOV   l            -       Branch if Not Overflow
BNZ    l            -       Branch if Not Zero
BOV    l            -       Branch if Overflow
BRA    l            -       Branch always
BZ     l            -       Branch if Zero
BCF    f,b,a        -       f.b = 0;   Bit clear f
BSF    f,b,a        -       f.b = 1;   Bit set f
BTG    f,b,a        -       f.b = !f.b;  Bit toggle
BTFSC  f,b,a        -       Bit test f, skip if clear
BTFSS  f,b,a        -       Bit test f, skip if set
CALL   k            -       Call subroutine
CLRF   f,a          Z       f = 0;     Clear f
CLRWDT -           TO,PD    WDT = 0;  Clear watchdog timer
COMF   f,d,a        Z,N     d = f ^ 255;   Complement f
CPFSEQ f,a          -       Skip if f=W
CPFSGT f,a          -       Skip if f>W
```

```
CPFSLT f,a          -       Skip if f<W
DAW            C            Decimal adjust W
DCFSNZ f,d,a        -       Decrement f, skip if not zero
DECFSZ f,d,a        -       Decrement f, skip if zero
DECF   f,d,a   C,DC,Z,N,OV  d = f - 1;   Decrement f
GOTO   k            -       Go to address
INCF   f,d,a   C,DC,Z,N,OV  d = f + 1;   Increment f
INCFSZ f,d,a        -       Increment f, skip if zero
INFSNZ f,d,a        -       Increment f, skip if not zero
IORLW  k           Z,N      W = W | k;  Inclusive OR literal
IORWF  f,d,a       Z,N      d = f | W;  Inclusive OR W and f
LFSR   f,k          -       FSRx = 12 bits literal
MOVF   f,d,a       Z,N      d = f;      Move f
MOVFF  f,f          -       Move f to f
MOVLB  k            -       BSR = k
MOVLW  k            -       W = k;      Move literal to W
MOVWF  f,a          -       f = W;      Move W to f
MULLW  k            -       PRODH,PRODL = W * k;  Multiply
MULWF  f,a          -       PRODH,PRODL = W * f;  Multiply
NEGF   f,a     C,DC,Z,N,OV  f = ~f+1;  Negate
NOP    -            -       No operation
POP                 -       Pop value from stack
PUSH                -       Push PC on stack
RESET               *       Reset device
RETLW  k            -       Return, put literal in W
RETURN -            -       Return from subroutine
RETFIE -            -       Return from interrupt
RLCF   f,d,a      C,Z,N     Rotate left f through carry bit
RLNCF  f,d,a        Z,N     Rotate left f
RRCF   f,d,a      C,Z,N     Rotate right f through carry bit
RRNCF  f,d,a        Z,N     Rotate right f
RCALL  l            -       Relative call
SETF   f,a          -       f = 0xFF;  Set f
SLEEP  -          TO,PD     Go into standby mode,  WDT = 0
SUBLW  k       C,DC,Z,N,OV  W = k - W;   Subtract W from literal
SUBWF  f,d,a   C,DC,Z,N,OV  d = f - W;   Subtract W from f
SUBFWB f,d,a   C,DC,Z,N,OV  d = W – f - ~C; Subtract with borrow
SUBWFB f,d,a   C,DC,Z,N,OV  d = f – W - ~C; Subtract with borrow
SWAPF  f,d,a        -       Swap halves f
TBLRD  *            -       TABLAT = *TBLPTR;
TBLRD  *+           -       TABLAT = *TBLPTR++;
TBLRD  *-           -       TABLAT = *TBLPTR--;
TBLRD  +*           -       TABLAT = *++TBLPTR;
TBLWT  *            -       *TBLPTR = TABLAT;
TBLWT  *+           -       *TBLPTR++ = TABLAT;
TBLWT  *-           -       *TBLPTR++ = TABLAT;
TBLWT  +*           -       *++TBLPTR = TABLAT;
TSTFSZ f,a          -       Skip if f=0
XORLW  k           Z,N      W = W ^ k;  Exclusive OR literal
XORWF  f,d,a       Z,N      d = f ^ W;  Exclusive OR W and f
```

Notes:

```
d = 1  : destination f
d = 0  : destination W
a = 1  : using BSR to select bank (0 – 15)
a = 0  : using access bank (0x0 – 0x7F, 0xF80 - 0xFFF)
```

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

Instruction execution time

Most instructions execute in 1 instruction cycle (4 clock cycles), except:

- branch instructions when PC is modified (BC, BRA, RCALL)
- skip instructions when next instruction is skipped
- double word instructions (MOVFF, LFSR, GOTO, CALL)
- instructions that modify the program counter, i.e: ADDWF PCL