

Last Update (August 2006) MCC18 V3.02, HI-TECH V9.50

This section contains a summary of the updates to the *reese микро.zip* archive made in August 2006.

- The *reese микро.zip* archive and this document have been updated to be compatible with the Microchip MCC18 Compiler, V3.02 (The MCC18 runtime startup files changed from Version V3.0 to V3.02).
- The *config.h* file has been updated to be compatible with the PIC18Fxx20 family, which is a pin-compatible update for the PIC18Fxx2 family. The project *code/chap8/mcc18_F_8_5_ledflash_noboot_2420.mcp* is an example MCC18 project that produces a hex file that does not assume the use of the Colt/Jolt bootloader; the project *code/chap8/mcc18_F_8_5_ledflash_w_bldrt_2420.mcp* produces a hex file compatible with the Colt/Jolt bootloader. Use these projects as templates for other projects using the PIC18Fxx20. The primary difference between the PIC18Fxx20 and PIC18Fxx2 is that the PIC18Fxx20 family uses the new PIC18 core that has the extended instruction set, has more configuration bit options, and has a larger boot block (0x800 instead of 0x200). The demo compiler with the book does support the PIC18Fxx20 family, so there are no PIC18 demo project files that support the PIC18Fxx20 (if you are using the commercial version of the HI-TECH compiler, all you would have to do in these project files is to select the PIC18F2420 instead of the PIC18F242, and set the configuration bits to your desired settings.)
- If you are using a Linux version of the HI-TECH PICC18 compiler, version 9.5 and later, the command line options have changed from what is documented in the book. The following is an example command line executed in the *code/chap8* directory to compile the led flash example for the PIC18F2420 using the HI-TECH PICC18 compiler”

```
picc18 --chip=18F2420 --CODEOFFSET=800 -I./common F_8_5_ledflash.c
```

The `--CODEOFFSET` option is used to produce a hex file compatible with the Colt/Jolt bootloader; the value 800 is used because the PIC18F2420 has a larger boot block than the PIC18F242 (the value 200 would be used with the PIC18F242). The `--chip` option is used to specify the device. If you are not using the Colt/Jolt bootloader, then the `--CODEOFFSET` option should not be used. For a complete listing of the PICC18 compiler options, execute: `picc18 -help` .

Using this book's C Examples with the Microchip MCC18 Compiler

This supplement discusses the use of this book's examples with the Microchip MCC18 C compiler. The accompanying zip archive contains all of the book's C code examples from

Chapter 8 onwards; these examples have been modified to be compatible with **both** the HI-TECH PICC18 and Microchip MCC18 C compilers. Also, MPLAB[®] project files for all examples are included. The ZIP archive (*reesemicro.zip*) assumes the MCC18 Compiler 3.02 version or later (there is an older ZIP archive on the site named *reesemicro_old.zip* that is compatible with MCC18 Compiler version 3.0). There are continual updates to the MCC18 compiler; this document assumes that the reader is familiar with the MCC18 compiler operation. Some of the comments in this document about particular MCC18 behavior have been made obsolete by new compiler releases.

The reesemicro.zip Archive

This archive will unzip into a directory called *reesemicro/*, this should be placed at *C:/reesemicro* if you intend to use the MPLAB[®] project files (*.mcp* extension) without modification. The *reesemicro/code* directory contains the C code examples divided into chapters. The *reesemicro/code/lab* directory contains the C code examples for the lab exercises of Appendix E.

MPLAB[®] Project Files

Any project files that start with *mcc18* use the Microchip MCC18 C compiler and assume that the compiler has been installed in the default location of *C:\mcc18*. Any project files that begin with *picc18* use the HI-TECH PICC18 compiler and assume that the compiler has been installed in the default location of *C:\HTSOFT\PICC18*.

All of the MCC18 project files use the *auto* storage class, which allocates function parameters and local parameters on the stack. The other two storage classes are *overlay* and *static*. Overlay uses static allocation for local variables, and shares memory locations used for local variables between functions that are not active at the same time. This is the allocation strategy also used by the HI-TECH PICC18 compiler, and results in smaller and faster code, at the cost of not being able to do recursion. Overlay mode also uses static allocation for function parameters. The static storage class uses static allocation as its name implies, but does not allow overlaying of memory locations. I tested all of the files with the auto storage class, but they should also work with the overlay storage class as well. However, the *_user_putc* (*auto char c*) function that is used by *serial.c* and in the LCD examples for single character output via the *printf()* function must remain with its parameter list explicitly specified as *auto* storage class. This is because *printf()* is compiled under the auto storage class in the MCC library and it is *printf()* that calls this function.

Also, the MCC18 project file for any example with a *printf()* function was configured to use the *large code model*; this uses 24 bit pointers for any pointers to program memory space. The MCC18 library is compiled with the large code model, and if the project is compiled with the small code model (16-bit pointers to code space) a warning “*type qualifier mismatch in assignment*” is generated for each *printf()*. The code still works, but these warnings can mask other important warnings, so I used the large memory model for these project files to remove the warnings.

MPLAB[®] Project Files and the Jolt/Colt Serial Bootloaders

Any project files that end in *_nboot* are configured to produce a hex file that is not compatible with the Jolt/Colt serial bootloaders and thus must be programmed with either the PICSTART[®] PLUS or ICD2 programmers; the majority of the project files are configured to produce hex files that are compatible with the Jolt/Colt serial bootloaders.

For the HITECH PICC18 compiler, the only change required to a project file to make it produce Jolt/Colt compatible hex files is to add the “-a200” flag to the project linker options.

The Microchip MCC18 compiler requires more work to produce Jolt/Colt compatible hex files. The linker script used in the project file must be *reesemicro/code/common/18f242_bldr.lkr* instead of the default *18f242.lkr* script found in the *C:\mcc18\lkr* directory; this linker script specifies that the code sector begins at location 0x0200. The linker script also specifies that the runtime object file *c018i_bldr.o* be used; which is found in *reesemicro/code/mcc18_startup*. The only difference between *c018i_bldr.o* and the standard *c018i.o* found in the *C:\mcc18\lib* directory is that this object’s code section begins at location 0x0200 in program memory as required by Jolt/Colt (see the next section about variable initialization and runtime code). Finally, any project with source code that uses interrupts has the compiler flag -
DHIGH_INTERRUPT=0x0208 defined in the project options to redefine the location of the high priority interrupt vector from location 0x008 to location 0x208. None of the book’s examples uses low priority interrupts so the low priority interrupt location did not have to be changed.

Differences between book examples and these files; PICC18 vs MCC18

Making the book examples compatible with both the MCC18 and PICC18 compilers required a few changes to the source code but not so many changes as to make the resulting code unrecognizable or unwieldy.

The new code examples compile without warnings for both the MCC18 and PICC18 compilers. All of the examples were retested on the PIC18F242 reference board and other prototype boards to ensure their functionality.

As part of the changes, various code cleanups were also done such as ensuring that any function with an empty parameter list is declared as *function_name(void)*, and if a function returns no value its return value is declared as *void*.

Special Function Registers and Named Bits, *config.h*

For the PICC18 compiler, a named bit is simply accessed via its name, i.e, as *TRISB0*, since each named bit is declared as a *bit* type by the PICC18 header files.

However, the MCC18 compiler uses *C* unions to represent special function registers and named bits, so *TRISB0* would be accessed as *TRISBbits.TRISB0*. In general, a named bit in the MCC18 compiler is represented by the union *regnamebits.bitname*.

The file *config.h* (found in *reesemicro/code/common*) is used to define equivalences between these two naming conventions, so the following *#define* is conditionally declared in *config.h* if the MCC18 compiler is being used:

```
#define TRISB0    TRISBbits.TRISB0
```

All of the named bits used in the book's examples are included in the *config.h* file. In general, the *config.h* file is used to isolate as many compiler differences as possible between the PICC18 and MCC18 compilers.

The *#include <pic18.h>* statement was removed from all files since this is PICC18 compiler dependent, and instead the first line of all *C* files is now *#include "config.h"*. This file uses *#if defined(HI_TECH_C)* and *#if defined(__18CXX)* blocks to define compiler-dependent statements such as header file inclusion and configuration bit settings, in addition to named bit equivalences and macros for common in-line assembly language statements. The symbols *HI_TECH_C* and *__18CXX* are defined by their respective compilers.

In-line Assembly Code

All uses of in-line assembly code such as *asm("sleep")* and *asm("clrwdt")* have been removed and replaced by macros such as *SLEEP()* and *CLRWDT()* which are defined in *config.h* to be the correct in-line assembly for the appropriate compiler. The PICC18 compiler already has these macros defined and mappings were made to similar macros defined by the MCC18 compiler.

Interrupt Service Routine Declarations

The MCC18 and PICC18 compilers declare interrupt service routines differently, so this required changing an ISR declaration like the following:

```
void interrupt pic_isr(void)
{
```

to:

```
#if defined(HI_TECH_C)
void interrupt pic_isr(void)
#endif
#if defined(__18CXX)
#pragma interrupt pic_isr
void pic_isr(void)
#endif
```

Also, the MCC18 compiler requires the user to explicitly specify code to be placed at the interrupt vector location, so all interrupt examples has the following code included (I placed this at the of the `main()` code for less clutter):

```
//for MCC18, place the interrupt vector goto
#if defined(__18CXX)
#if defined(HIGH_INTERRUPT)
#pragma code HighVector=HIGH_INTERRUPT
#else
#pragma code HighVector=0x0008
#endif
void HighVector (void)
{
    _asm goto pic_isr _endasm
}
#pragma code
#endif
```

The value of the `HIGH_INTERRUPT` macro in the above code must be passed in by the compiler; if this code is to be used with the Jolt/Colt serial bootloaders then the compiler flag `-DHIGH_INTERRUPT=0x0208` should be used since the code is offset by `0x0200` locations. The MCC18 project files in the zip archive for the interrupt examples are all configured to produce Jolt/Colt compatible hex files and already have the compiler flag `-DHIGH_INTERRUPT=0x0208` defined.

WARNING: This code was based on version 2.4 of the MCC18 compiler; this version requires the user to manually specify extra registers and compiler resources to be saved for interrupts if certain conditions are met. This was not necessary for the examples in this book, but could be necessary for more complex examples. You should read the section in the MCC18 compiler user guide on compiler-managed resources and their interaction with interrupts before writing any ISRs of your own if you use the MCC18 compiler.

Global variable initialization

The runtime code for the PICC18 compiler gives all global variables their explicit initialization value, or a value of '0' if they are not explicitly initialized. The PICC18 compiler also defines the *persistent* qualifier for variables that are to be left un-initialized by the C startup code. The `reset_cnt` variable in the `chap8/F_8_11_reset_cnt.c` code is one example that makes use of this capability. Thus, in the PICC18 compiler, the global variables below of *k*, *j*, *n* are initialized as described in the comments.

```
char k;                // in PICC18, is initialized to 0
char j = 5;           // in PICC18 is initialized to 5
persistent char n;    // value is not initialized.
```

In the MCC18 compiler, the initialization of global variables is controlled by the runtime code that is linked in. In *mcc18\lib* there are three object files to choose from: *c018.o*, *c018i.o*, and *c018iz.o*. The *c018.o* code does no global variable initialization; the *c018i.o* code only initializes those global variables with explicit initial values, while the *c018iz.o* code first zeros all memory, then initializes those global variables with explicit initial values. The *persistent* qualifier is not supported in the MCC18 compiler.

The MCC18 project files all either link to *c018i.o* or *c018i_bldr.o* (this code is offset by 0x200 locations to be compatible with Colt/Jolt bootloaders). Using this run time code, the global variables below of *k, j, n* are initialized as described in the comments.

```
// assuming c018i.o code is used
char k = 0;           // initialized to 0
char j = 5;          //initialized to 5
char n ;             // value is not initialized.
```

The original code examples in the book relied on the PICC18 compiler giving all global variables a value of '0' if not explicitly initialized; all examples have been modified to explicitly initialize variables if initialization is required for proper operation.

Variables placed in ROM (program memory)

The PICC18 compiler used the *const* qualifier for any constant variables that should be placed in program memory. The MCC18 compiler requires the additional *rom* qualifier as shown below (see *reesemicro/code/labs/sinegen.h*):

```
#if defined(__18CXX)
rom
#endif
const unsigned char sine64tab[] = {
```

Lack of *scanf()* in MCC18

The MCC18 library does not include *scanf()* so book examples that use *scanf()* have been modified for the MCC18 compiler to read a string from the console into a buffer and then use the *atoi()* function to parse ASCII decimal data as shown below:

```
#if defined(HI_TECH_C)
    scanf("%d",&ivalue);
#endif
#if defined(__18CXX)
    { // MCC18 does not have scanf, just get string from console, use
      atoi
        char buf[30];
        my_gets(buf); // get a string from console
        ivalue = atoi(buf); // convert string to integer
    }
#endif
```

The *my_gets()* string function is defined in the book examples that use *scanf()*; the *my_gets()* function is shown below and returns a string from the console, terminated by a carriage return:

```
#if defined(__18CXX)
void my_gets (char *s);

//return a string from the console
void my_gets (char *s){
    char c;

    do {
        c = getche();
        *s = c;
        s++;
    } while (c != '\r');
}
#endif
```

USART I/O (Console Input/Output)

The *scanf()/printf()* library functions for the PICC18 compiler use the single character functions *getch()/putch()*; the book examples defined these for PIC18 USART I/O for both non-interrupt driven and interrupt driven I/O. The *putch()* function is also used for single character LCD output in the LCD examples so that *printf()* can be used for LCD output.

The MCC18 library does not have any formatted input functions, so the *getch()* code is unchanged. The MCC18 formatted output functions can either be redirected to the USART by setting STDOUT to the predefined output stream `_H_USART` or to a user-defined function single-character output function named `_user_putc` by using the output stream `_H_USER`. The declaration for this function must be `_user_putc (auto char c)` where the parameter list is explicitly declared as *auto* storage class, since *printf()* is compiled under the *auto* storage class.

These modified examples set STDOUT to `_H_USER` and then just maps the `_user_putc` function to the *putch()* function used in the book examples:

```
#if defined(__18CXX)
void putch(char c);

void _user_putc (auto char c) {
    putch (c);
}
#endif
```

The assignment of STDOUT to `_H_USER` is either done in the *serial_init()* function used for USART I/O (see *common/serial.c*) or in *lcd_init()* for the LCD output examples.

Integer promotion

ANSI C states that computation must be done at *int* precision at the minimum. In the code below, the value of the computation ‘`k << 8`’ is done with 16-bit precision assuming that an *int* is 16 bits;

```
int value;  
char k;  
  
value = value | (k << 8);
```

The above code works fine with the PICC18 compiler. However, for the MCC18 compiler, by default a computation is done at the size of the largest operand. Thus the computation “`k << 8`” is done with 8-bit precision before being applied to the rest of the computation, resulting in incorrect results. If the code is changed to:

```
int value;  
char k;  
  
value = value | (k * 256);
```

then the correct result is obtained because the value 256 requires 16-bit precision. You can also change the code to:

```
int value;  
char k;  
  
value = value | ((int) k << 8);
```

where *k* is explicitly cast to an *int* in the expression.

You can also explicitly enable integer promotion in the MCC18 compiler by using the “-Oi” flag; this has been done in all MCC project files for the Chapter 12 examples because the code examples have “`ADRESH << 8`” in them. In cases in other chapters, an expression like “`a_var << 8`” has been replaced by “`a_var * 256`”.

If you want to be safe, you can always enable integer promotions via the “-Oi” compiler flag at the cost of extra code space (this is not an issue in the HI-TECH compiler as it seems to be a bit smarter on when integer promotions are needed or not needed).

Miscellaneous Changes

The book code examples use *bittst/bitset/bitclr* macros for setting, clearing or testing a bit in a variable; a typical definition of one of these macros is:

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

In the modified examples, any code line that used these macros to affect the LSB, such as `bitset(a_var, 0)` was changed to use the equivalent logical operation (`a_var = a_var | 0x01`). This was done to remove a compiler warning issued by the MCC18 compiler in this case; the warning is “shift amount is zero”. While this warning is superfluous and could be ignored, I wanted all of the code examples to compile with no warnings so I changed the code examples in order to remove these MCC18 compiler-generated warnings.

Things to look out for

The following contains some code examples of coding mistakes to watch for when using these compilers.

Pointers to Program Memory Space

In both compilers, a “`char *`” is a pointer to *data memory* space.

In the PICC18 compiler, a “`const char *`” is a pointer to program memory space.

In the MCC18 compiler, a “`rom far const char *`” or “`rom near const *`” is a pointer to program memory space, depending on if the large or small code model is used.

Be careful not to pass a program memory pointer to a function expecting a data memory pointer. In both compilers, if you do something like:

```
foo("Hello There!")
```

the string “Hello There!” is placed in program memory space. If the function `foo()` is declared as:

```
foo(char *s){
```

then the function `foo()` will not operate correctly as it is expecting a pointer to data memory, but it is being passed a pointer to program memory space.

Mixed INT, LONG arithmetic

For *both* compilers, if you execute the code below (example taken from Microchip user forum):

```
long sum;           //longs are 4 bytes
unsigned int a,b;   //ints are 2 bytes

a = 40000;
b = 30000;
sum = a + b;
```

the result is 4464 (0x1170) instead of 70000 (0x11170) because the addition is done as 16-bit addition before being assigned to the *long* variable. Having integer promotions enabled within Microchip does not help this case.

You need to put a typecast in front of one of the operands in order to achieve the correct result:

```
sum = (long) a + b;
```

If the calculation is something like:

```
sum = (long) a + (b << 1);
```

then you need to put typecasts in front of both variables:

```
sum = (long) a + ( (long) b << 1);
```

Which Compiler Is Best?

I prefer the HI-TECH compiler because there is a Linux version; each person has their own reasons as to why they might prefer one compiler to another. The examples in this ZIP archive allow you try both compilers and decide for yourself.

Microchip Trademark Acknowledgement

MPLAB[®] is a registered trademark of Microchip Technology Inc., in the U.S.A. and other countries.

PICSTART[®] is a registered trademark of Microchip Technology Inc., in the U.S.A. and other countries.