Some MCC18 Examples

This small code archive has some code examples that are compatible with Microchip MCC18 compiler so that you could use the Microchip MCC18 compiler instead of the HI-TECH PICC18 compiler. The code is written in the HI-TECH compiler style in terms of bitnames (i.e. *TRISB0* instead of *TRISBbits.TRISB).* MPLAB project files are provided for both 18F242 and 18F4620 processors.

Why Use MCC18?

You may find it convenient use the MCC18 compiler if you want a compiler integrated with MPLAB on your personal PC instead of using the ECE Linux server. The MCC18 2.4 student compiler is a free download from Microchip. After 60 days, some compiler optimizations are disabled (resulting in slightly larger and slower code) but the compiler still works.

ZIP Archive Contents

In order for the MPLAB projects to work with MCC18, you must unzip this archive into the top level of your C: drive. This will create a directory called *C:\mcc18_examples*.

The directories are:

- *c:\mcc18_examples\common* - files common to all examples such as *config.h, serio.c*, etc.
- *c:\mcc18_examples\source* - C code source examples and MPLAB project files. The two examples provided are *pwm_example.c* and *stepmotor_test.c* (contains an example of an interrupt service routine).
- *c:\mcc18_examples\mcc18_startup* – linker scripts and C runtime code.

MPLAB Project Files

The project files are named:

- *sourcefile*_18F242_bootldr.mcp – project configured for 18F242 and bootloader
- *sourcefile*_18F4620_bootldr.mcp – project configured for 18F4620 and bootloader
- *sourcefile*_18F242.mcp – project configured for 18F242; no bootloader support.
- *sourcefile*_18F4620.mcp – project configured for 18F4620; no bootloader support.

The difference between the projects that produce hex files compatible with the Jolt/Colt bootloaders and the non-bootloader compatible projects are:

- The bootloader projects use the linker files *18f242i_bldr.lkr* or *18f4620i_bldr.lkr* while the non-bootloader projects uses the files *18f242i.lkr* or *18f4620i.lkr*. The bootloader linker files offset the code by the required amount (0x200 for the 18F242, 0x800 for the 18F4620). The bootloader linker files also link to *C* runtime files that are offset by the correct code amount as well.
- The project defines the compiler symbol HIGH_INTERRUPT=0x200 (for 18F242) or HIGH_INTERRUPT=0x800 (for 18F4620) to specify where the high interrupt vector lives.

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.

Differences between the HI-TECH PICC18 and Microchip MCC18 compilers

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 *regname*bits.*bitname*.

The file *config.h* (found in *mcc18_examples/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
```

These macros support the PIC18F242 (PIC18Fxx2) and PIC18F4620 processors. If you want to use a different processor, then add similar macros to support the new processor (or else just use MCC18 bit naming).

In-line Assembly Code

In-line assembly code such as *asm("sleep")* and *asm("clrwdt")* are represented by macros such as SLEEP() and CLRWDT() which are defined in *config.h* to be the correct in-line assembly for the MCC18.

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)  //HI-TECH compiler
{
```

to:

```
//MCC18 Compiler
#pragma interrupt pic_isr save=section(".tmpdata"),section("MATH_DATA"),PROD
void pic_isr(void)
{
```

The "save=section..." command causes the compiler to generate code to save some temporary data and registers used by foreground code operation so that the ISR does not corrupt these values when it is invoked. The above "save" command is a worst case – if your ISR does not touch these data sections, then you would not have to save this. Unfortunately, you would need to look at the MAP file to determine if your ISR used these or not, so the above "save" command may waste some CPU time, but at least your ISR will not produce strange results by corrupting data used by the foreground code.

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(HIGH_INTERRUPT)
#pragma code HighVector=HIGH_INTERRUPT
#else
```

```
#pragma code HighVector=0x0008
#endif
void HighVector (void)
{
    _asm goto pic_isr _endasm
}
#pragma code
```

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 (or HIGH_INTERRUPT=0x0800 for processors with a 2K boot sector like the 18F4620).

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*, *c018i_bldr_0x200.o* (this code is offset by 0x200 locations for bootloader), or *c018i_bldr_0x800.o* (this code is offset by 0x800 locations for bootloader). 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, BE CAREFUL!!!!
```

The variable 'n' above will have an unpredictable value at startup since it has not been initialized. This could be OK if you do not need an initial value; simply be aware of this.

Lack of *scanf()* in MCC18

The MCC18 library does not include *scanf()*; you must write a custom function to do something like:

```
 scanf("%d",&ivalue);   // read integer from string

char buf[20];
sscanf(buf,"%d %d", &i, &j);  //read two integers from string
```

The file *common/serio.c* has two functions named *console_getuint()* and *buffer_getuint()* that are useful for reading integers from the console or a string (see the *stepmotor.c* and *pwm_example.c* code for how these are used). The code fragments below show how to replace *scanf()* functionality with these function calls.

```
 //scanf("%d",&ivalue);
ivalue = console_getuint();   // will read integer from console
char buf[20], *s;

//sscanf(buf,"%d %d", &i, &j);  //read two integers from string
s = buffer_getuint(buf,&i);    // get first integer
s = buffer_getuint(s,&j);      // get second integer
```

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[] = {
```

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).

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.