

- Compiling, Loading and Running your C code

Compile your .c file from the DOS prompt. Type

```
c452 <filename.c>
```

or

```
c2320 <filename.c>
```

to compile for the 18F452 or the 18F2320 respectively. This will generate the corresponding files such as .HEX, .LST, etc.

Load and run your code using the same procedure as before. Use Teraterm to send the .HEX file as before.

- Function Prototypes

Every new function requires a "heading" at the top of your code (except for "main")

- Some data types are as follows:

```
long   : a 32-bit (signed) number (Rarely used)
int    : a 16-bit (signed) number
char   : an 8-bit (signed) number (Most often used)
```

NOTE: You can define unsigned data types as shown below. For example, you need to #define the following lines near the top portion of your code before you can use these types.

```
#define byte  unsigned char
#define uint  unsigned int
#define ulong unsigned long
```

NOTE: There isn't much of a distinction between using signed vs. unsigned numbers, except in situations of arithmetic operations for example.

- Defining your own bits and other data types

Structures can be used to define individual bits and/or nibbles. For example:

```
struct {
    unsigned bit0:1;
    unsigned :6;
    unsigned bit7:1;
} FLAGS;
```

This allows us to access the 0th and 7th bit of FLAGS with FLAGS.bit0 and FLAGS.bit7 respectively.

```
struct {
    unsigned NibbleL:4;
    unsigned NibbleH:4;
} myByte;
```

This allows us to access individual nibbles using myByte.NibbleL or myByte.NibbleH

- Using unions to combine structures

In some cases it is useful to be able to read/modify a variable both bit by bit or as an entire byte (i.e. clearing the entire FLAGS register in initial). For these cases we can use unions to combine structures. For example:

```
union { struct {          // Now we can individually set/clear
    unsigned bit0:1;      // bits through the same way using
    ...                  // FLAGS.bit0 for the 0th bit, etc...
    unsigned bit7:1;      //
};                        //
    struct {              // However we can also treat it as an
    unsigned char reg;    // 8 bit char and clear all bits by
};                        // using FLAGS.reg = 0;
} FLAGS;                  //
```

- Array Definition

Examples:

```
// Constant Arrays
const char array1[] = {0x80,'T','o','g',' ','T','i','m','e',0};
const char array1[] = "/x80Tog Time";
```

```
// Variable Arrays
// Two ways to create an array of 10 elements
char array2[] = {0,0,0,0,0,0,0,0,0,0};
char array3[10];
```

Array indexing starts with 0 !!!

NOTE: Something like DisplayC("/x80Tog Time"); will work fine.

- The CARRY and ZERO bits in STATUS are referenced as STATUSbits.C and STATUSbits.Z. Borrow bit is !STATUSbits.C (NOT CARRY). The W register is the same, WREG.

- Access of Reserved Names

To access flags or bits directly in C the general format is <register>.<bit>

Examples:

```
PIR1.TMR2IF = 0;          // Clears TMR2IF
PIE1.CCP1IE = 1;         // Enables CCP1 interrupts
```

- Port names are now accessed with PORTBbits.RB1, TRISAbits.TRISA1, etc...

Examples:

```
PORTAbits.RA0 = 1;       // Set PORTA bit 0
PORTCbits.RC1 = 0;       // Clear PORTC bit 1
PORTBbits.RB4 = !PORTBbits.RB4; // toggle RB4
```

- The following are examples of "macro" definitions in C

```
#define bitset(var,bitno) ((var) |= 1 << (bitno) )
#define bitclr(var,bitno) ((var) &= ~(1 << (bitno)))
#define bittest(var,bitno) ((var) & 1 << (bitno) )
```

Example usage:

```
bitset(TEMP,4);          // Set bit 4 of the variable TEMP
if(bittest(TEMP,5)) {   // true if bit 5 of variable TEMP is 1
    // NOTE: use !bittest(TEMP,5) to test for 0
```

The bit ordering in the example above uses the same convention as before. Bit 7 is the Most Significant bit and bit 0 is the Least Significant bit.

- IF_ELSE_ constructs are as follows -> you can also perform IF_ELSE_IF_ as

```

if (condition1)
{
    <task1>
}
else
{
    <task2>
}

if (condition1)
{
    <task1>
}
else if (condition2)
{
    <task2>
}
else
{
    <task3>
}

```

- Special Syntax Examples

Note the difference between a bit-wise comparison versus a logical comparison

Examples:

```

// Example of a logical AND (&&)
if (Condition1 && Condition2) // If BOTH Condition1 and Condition2 are
{                               // true, then task1 is performed
    <task1>
}
// Example of a bit-wise AND (&)
if (Condition3 & 4)           // The bits in Condition3 are bit-wise
{                               // ANDed with 4 (i.e. 4 = 00000100 binary)
    <task1>                     // so if bit 2 of variable Condition3 is
}                               // a 1, then task2 is performed

while ((1==Value) || (3==Value)) {} // Logical OR
temp = (temp | 1);                // Bit-wise OR

while (PIE1.CCP1IE && FLAGS.bit0) ; // Logical AND
while (ADCON0 & 0b00000100) ;      // Bit-wise AND

PORTBbits.RB1 = !PORTBbits.RB1; // Good method for bit toggling
PORTBbits.RB1 = ~PORTBbits.RB1; // Another way of bit toggling
PORTBbits.RB1 = PORTBbits.RB1^1; // Bit-wised XOR (inefficient way of toggling)

TEMP += 32;                        // TEMP = TEMP + 32;
TEMP -= 10;                         // TEMP = TEMP - 10;

loopCounter--;                      // loopCounter = loopCounter - 1 (decrement by 1)
--loopCounter;                      // loopCounter = loopCounter - 1 (another way to decrement)
// To get a little more technical
if (--loopCounter) {}              // loopCounter is FIRST decremented, then
// tested in the if-statement
if (loopCounter--) {}             // Test loopCounter THEN decrement it

aCounter++;                         // aCounter = aCounter + 1 (i.e. increment by 1)
++aCounter;                         // Same as above in this usage but a "pre-increment"

RATE >>= 2;                          // Right shift the bits in RATE by 2 (RATE = RATE >> 2)
BITCOUNT <<= 1;                      // LEFT shift the bits in BITCOUNT by 1
// (BITCOUNT = BITCOUNT << 1) *** (Zeroes are shifted in)

ADCON0 |= 0b00000100;              // ADCON0 = ADCON0 bit-wised "ORed" with 0b00000100;

```

- Comments versus Code

```
// a double slash is used for comments on a single line

/*
A slash followed by an asterisk begins a block of comments.
All text in between is considered comments.
An asterisk followed by a slash signals the end of the block of comments
*/
```

- A "return" statement at the end of a function is NOT necessary if it returns void
- Local variables can be created within a procedure
- Static variables are variables that retain their values between calls to the procedure.

Example:

```
void ToggleStep()          // Function called ToggleStep
{
    static char time1 = 1; // Create a local static variable called time1
                          // and initialize to 1
    time1++;              // Increment time1
}                          // End of function
```

When this function is called the very first time, it creates a local variable called time1 and is assigned a value of 1. Because the variable is created as 'static', time1 will retain its value at the exit of this function. For this example, time1 will initially be set to 1 and then increment to 2. The next time this procedure is called, time1 will have the value of 2 and will be incremented to 3 before exiting. (static variables in functions act the same way as global variables with the exception that only this function can use it)

- Type Casting and Two-Byte Manipulations

* It is good practice to "type cast" prior to assigning a variable value to another variable of different type.

Example:

```
char Value_1;
char Value_2;
unsigned int Big_value;

Big_value = (unsigned int) (Value_1 * Value_2);
           // Multiplies Value_1 by Value_2, casts result into an unsigned
           // integer then stores final result into Big_value
```

* When incrementing or decrementing a 16-bit variable with C, the CARRY and BORROW bits are automatically handled by the compiler if you use an int or unsigned int type definition (or greater such as a long type).

Example:

```
int Var_A;          // 16-bit variable created
Var_A = 255;        // Assigned a value of 255 which uses all lower 8 bits
Var_A = Var_A + 1; // Var_A is now 256 which requires 9 bits to represent
                  // but since Var_A is an int, adding 1 to it will
                  // now result in Var_A = 256 without any additional
                  // code to handle the "Carry"
```

- Interrupt Vectors

For High/Low priority interrupt subroutines add the following section of code after the "Function prototypes" section:

```

/*****
 * Interrupt Vectors
 *****/

// For high priority interrupts:      | // For low priority interrupts:
|
void high_isr(void);                  | void low_isr(void);
#pragma code high_vector=0x08         | #pragma code low_vector=0x18
void interrupt_at_high_vector(void) | void interrupt_at_low_vector(void)
{                                     | {
  _asm GOTO high_isr _endasm         |   _asm GOTO low_isr _endasm
}                                     | }
#pragma code                          | #pragma code
#pragma interrupt high_isr            | #pragma interruptlow low_isr

```

From this point high_isr() and/or low_isr() subroutines are the interrupt subroutines for you to define.

NOTE: The compiler handles the retfie instruction and shadow registers automatically, so you do not need to worry about them.