## General Framework of a C program for MSP430

```
/* Compiler directives (includes & defines) come first */
/* Code you write in this class will need this include */
#include  "msp430.h"       // CCS header for MSP430 family
#include  <stdlib.h>       // C standard library

/* defines are mostly used to declare constants */
#define   MAX_ITER   10    // max number of iterations

/* Function prototypes */
int someFunction(char opt, unsigned int scale);


/* Global variables are visible everywhere in code */
/* Usually try to avoid local variables as much as possible. */
int   exitFlag = 0;
const float   pi = 3.14156;


/* Function implementations */
int someFunction(char opt, unsigned int scale)
{
     int a, b, ret_val;  // Local variables (only visible in function)
     //  . . .
     scale = scale * opt + a;
     // . . .
     ret_val = scale * b
     // . . .
      return ret_val;    // returns an integer
}

/* Execution of program always starts in main () */
/* There can be only one main() function! */
void main(void)
{
    char   choice;
    int   test_num, stuff;
    // ...

    stuff = someFunction('a', test_num);
    // ...
}
```

## Basic Types of Programming Instructions

1) Variable declarations – Defining what your data is

```
int        num1, sum, myArray[10];

char       initial, name[16];

float      myReal, flt_arr[5];
```

2) Assignment statements – Most are straight forward and "calculator like"

```
int    a, b, c;
float  d, e, f;
char   byte1, byte2;

c = a + b;

e = c * d;

f = cos(e);           // would require including math.h

byte2 = byte1 & 15;   // we'll use a lot of bitwise ops
```

3) Decision and Control

```
tempOK = 1;
degF = get_degF();
while (tempOK > 0)
{
   degF = degF - 10;
   degC = 5 * (degF-32)/9;
   degK = degC + 273.15;

   if  (degK <  0) {
      // IMPORTANT: printf() is a standard IO function in C but
      // our MSP430F5529 board doesn't support it!!
      // We use it here as an example of how to show output.
      printf("ERROR - Temp < 0 K\n");
      printf("Breaking Multiple Laws of Physics!\n");
      tempOK = 0;
   }
   else {
      printf("%10.4f F = %10.4f C = %10.4f K \n",
                                       degF, degC, degK);
   }
}
```

## Basic Instructions and Syntax

After writing a C program you need to compile and link it to create an ***Executable*** file
The ***COMPILER*** defines the size of the different data types and how they should be stored in the computer's memory

Most general purpose compilers like *gcc* or Microsoft Visual Studio assume default data word size (usually of 32 bits or 64 bits depending on your computer). BUT, the Code Composer Studio Compiler is designed for the MSP430 and must use its own set of type definitions specific to the MSP430 architecture.

***→ The MSP430 is a 16-bit processor! Default data sizes are based on 16 bits!***

>> Here are some data types and sizes (as they are defined in CCS!)

```
int             a;  // 16 bit signed integer (2's comp)
float           b;  // 32 bit IEEE floating point
char            c;  // 8 bits (unsigned)
unsigned int    d;  // 16 bit unsigned integer
long            e;  //  32 bit signed integer
double          f;  //  64 bit Floating Point
```

>> First task in programming is creating variables of suitable size and type for the data in your problem!

→ For example, a is a 16 bit integer and can hold integer values between -32768 and 32767. If you need to represent a value outside of that range you cannot do store in in variable a!

>> Arrays … In C arrays are indexed starting at 0

```
char   initial, name[5];    // declare array of 5 char

. . .

name = "Susan";             // fill the name array

initial = name[0];          // initial = 'S'
                            // name[1] = 'u'  and
                            // name[4] = 'n'
```

## Standard C Operators

**Arithmetic Operators**: `+ - * / %` (modulo)

```
a = 2*c + 1;

b = ((float)e) / b;  // "cast" e as float before division

c = e % a;  // Modulo: c = remainder of e divided by a
```

**Unary and Assignment Operators**: `++ -- += -= *= /=`

These operators provide a shorthand notation for performing certain arithmetic operations.

```
a++;        // a = a + 1;

c--;        // c = c - 1;

c *= 2;     // c = 2*c;
```

Note: You do not need to use these operators ***but*** you should to be able to read code written by others who do use them.

**Relational Operators**: `> >= < <= == !=`

```
if (x > y) {
   // this code is executed when x > y
   z = z - x;
} else {
   // otherwise this code is executed
   z = z - y;
}


while (counter != marker)
{
    // while counter not equal marker,
    // do something
    //    . . .

    counter--;   // update counter
}

if (A == test_val)
{  // this code is executed when A equals test_val
   . . .
}
```

**Logical Operators**: && (AND) || (OR) == (EQUIVALENCE)

```
if ((A == 0) || (g > 100))
{  // if A equals 0 OR if g greater than 1000

        . . .

}
```

***WARNING*: Do not use single = to compare values!! Single = is an assignment operator!!**

```
while ((j < k) && (run_flag))
{  // while j less than k AND run_flag not 0)

        . . .

}
```

**Bitwise Operators**: & (AND) | (OR) ^ (XOR) ~ (NOT)
>> (Right shift) << (Left shift)

All variables are encoded in binary. What is stored in memory is just 1's and 0's
Bitwise operators perform AND , OR, NOT and XOR operations on each bit in a word:

```
unsigned int i, j, k, m, n;

i = 0x0085;      // i = 0085h =  0000000010000101 b
j = 0xF0F0;      // j = F0F0h =  1111000011110000 b

// each bit in i is AND'ed with corresponding bit in j
k = i & j;       // k = 0080h =  0000000010000000 binary

// >> and << are right and left shifts
// m = bit pattern in k shifted to right by 2 places

// Equal to dividing k by 4 = 2^2
m = k >> 2;      // m = 0020h =  0000000000100000 b

// n = bit pattern in k shifted to left by 4 places
// Equal to multiplying k by 16 = 2^4
n = k << 4;      // n = 0800h =  0000100000000000 b
```

***Note: Bit-shifting can be faster than multiplication!***

## Control Flow: statements for making decisions or looping...

### IF-ELSE statements:

```
if  ( kk > 100 ) {
   kk = 0;
} else {
   kk++;
   y = y+kk;    // can also write as y =+ kk
   ...
}
```

### SWITCH statements:

```
switch (choice)
{
     case 1: /* do something */
          ...
          break;
     case 2: /* do something else */
          ...
          break;
     case 4: /* do something else again*/
          ...
          break;
     default: /* for all other values */
         break
}
```

### WHILE loops

```
i = strt;  // initialize loop counter

while (i < end_pt)
{
    /* Body of loop  */
      ...

    i++;   // increment loop count

}
```

***FOR loops –*** Really just a different syntax for a while loop

```
for (i = strt; i < end_pt; i++)
{
    /* Body of loop  */
      ...

}   // end for i
```

>> This for loop accomplishes exactly the same thing as the while loop above

***The "Forever Loop" -- Never Ends***

>> Endless loops are not desirable (errors!) in most "general purpose" PC-style programs but are *used all the time* in the main( ) of embedded code

```
int main()
{
     /* Declare and initialize variables */

      while (1)
      {

         /* update variable values,

            call functions and do

            everything my device has to do */

            ....

      }  // end while (1)
}
```

***C is a Structured or Modular Programming Language***

>> A C `main()` calls a series of smaller *functions* to complete a task

--> These may be functions declared and implemented in the same file or in a different file

```
#define MAX_BUTTONS   4    // This systems has 4 buttons

// Function prototypes
int   checkButton(char buttonNum);
void  lighLED(char ledNum);
void  wait_awile( );

/* Later in file implement checkButton */
int  checkButton(char buttonNum)
{
     /* define local variables */
     int buttonState = 0;

     /* Body of function goes here */

     /* Return  */
     return buttonState;
}

/*  Need to implement other functions too! */
// < ... Code omitted ... >


/* Now we get to main() */
void  main( )
{   /* define local variables */
     int    i, b_stat;
     char   button;

     while (1)        /* loop forever */
     {
        for (i = 0; i < MAX_BUTTONS ;  i++)
        {
          button = i + 1;
          b_stat = checkButton(button);
          if  (b_stat == 1) {
             lightLED(button);
           }
          wait_awhile();
        }
        // . . . Do other stuff . . .
    }
}
```