C Programming Primer

Introduction to “C”

The C programming language was developed by Dennis Ritchie and Brian Kerninghan for Bell Labs between 1969 and 1972. It is a classically imperative programming language – each command does something when it is executed. The language has only 32 key words and, with the exception of “pointers,” is easy to follow and understand. It is well suited for embedded applications where memory (flash and ram) and the CPU are limited, as it compiles very compactly into a limited footprint. In short, C programs run fast and don’t use memory they do not really need.

A basic understanding of C programming is required in order to use even a minimally configured PSoC as it controls the central CPU of the PSoC.

The C programming toolchain (C-Preprocessor, Compiler, and Linker) is a set of programs (integrated into PSoC Creator) that interpret the C code you write and compile it into a hex file that the PSoC CPU can execute. A program is just a collection of instructions and data that get downloaded into the PSoC device.

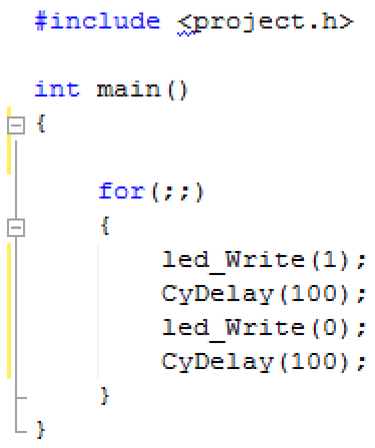
A C program (for a PSoC) has the following steps:

- Define your data
- Start the main program
- Initialize your variables
- Start your components
- Loop infinitely
  - Act on the inputs
  - Control the outputs

The seminal first C program is called “Hello World” which starts up, prints “Hello World” and then finishes. The program looks like this if you are writing it for a computer system:

```
#include <stdio.h>
int main(int argc, char **argv)
{
        printf(“Hello World\n”);
}
```

The equivalent of this program in the embedded world is the “blink the LED” and looks like this on a PSoC:



```
#include <project.h>

int main()
{

    for(;;)
    {
        led_Write(1);
        CyDelay(100);
        led_Write(0);
        CyDelay(100);
    }
}
```

It is important to note that white space (spaces, tabs, and blank lines) do not alter or affect the program or its execution in any way. White space is one of the first elements of the ASCII file that is stripped by the compilation process. However, white space is very important for readability of the program and proper practices should be followed (more details to follow).

Variables

Defining Data

Data is created by defining a variable. In PSoC Creator, there are a few types of data that are pre-defined in “cytypes.h”. It is highly recommended that only these types be used since they are architecture and compiler independent (that is, they will always mean the same thing on any PSoC device). These types are:

int8, int16, int32	(signed 1-byte, 2-byte and 4-byte numbers)
uint8, uint16, uint32	(unsigned 1-byte, 2-byte and 4-byte numbers)
float32	(32 bit floating point number)

The variables in the first group are signed, meaning that they can hold either positive or negative whole numbers (int stands for “integer”). An int8 is 8-bits and can therefore hold values from -128 to +127. The variables in the second group are unsigned (uint stands for unsigned integer). A uint8 can hold values from 0 to 255.

In order to create a variable, specify the type and then provide a name like this:

```
uint8 myvar;
```

This C language statement creates a variable that can hold an 8-bit value from 0 to 255 and lets you refer to it as “myvar”. Note that the statement is completed by a semicolon. This character tells the compiler that you have finished the statement. You can use any name you like but the first character should always be a letter (i.e. don’t create a variable like “5times”). Note that case matters so that MyVar and myvar are two different things.

You can create a variable and give it a value at the same time with the “=” operator.

```
uint8 myvar = 99;
```

The equals is a command (assignment), not a comparison (like it is in mathematics). The above statement says “allocate space for an unsigned integer, call it myvar, and set it to 99”. It is not asking whether myvar has the value 99. To ask that question, use a double-equals operator (myvar == 99).

Arrays

These are consecutive groups of variables (e.g. lookup tables or blocks of data). Use square brackets to create and use arrays.

```
uint8 buffer[4];            // Create an array of 4 integers
```

This creates a 4-element array of integers (16 bytes in total) in SRAM. Array indices always start at zero and so buffer[0] is the first element in the above array and buffer[3] is the last. In order to assign the third element in the array a value of 22, you would do the following:

```
buffer[2] = 22;                // Set the third element to 22 (numbering starts from 0)
```

You can initialize array values when you define the array like this:

```
uint8 buffer[4] = {10,20,30,40};
```

If you only specify some of the values, the rest are set to 0. If you do specify all initial values, then the size of the array can be left out – it will be automatically determined by the compiler:

```
uint8 buffer = {10,20,30,40};
```

Programs

Creating Programs - Instructions

Programs are sequences of instructions that manipulate data. The obvious way is to assign values to variables.

```
x = 7;          // Set x to the value 7
y = 4;          // Set y to the value 4
x = y + 4;      // Add 4 to y and assign the result to x
x += y;         // Add y to x and assign the result to x (shorthand for x = x + y)
x++;            // Add one (increment) to x (shorthand for x = x + 1)
x--;            // Subtract one (decrement) from x. (shorthand for x = x -1)
x = 5 * y;      // set x to the value of y multiplied by 5
x = y / 5;      // set x to the value of y divided by 5
```

Conditional Execution - “if” and “else”

The “if” keyword lets you choose whether to run code. The format is “if(*condition*)”. The following statement is only executed when the expression in parentheses is true.

```
if( x == 10 )
{
        x = 0;          // if x is equal to 10, reset it to zero
}
```

The compiler handles this by making a sequence of instructions in flash. The instructions read the value of x and compare it to 10. If the comparison is true it executes the instructions between the braces (curly brackets). If it is not true then it jumps over the instructions that are within the braces.

Strictly speaking, the braces are not required if there is only one statement associated with the “if”. However, it is always good practice to include these even for a single statement, since it makes the code easier to understand. Also note the indenting - this is not required but is highly recommended for any blocks of code within braces.

The “else” keyword is used in conjunction with “if”. It allows you to have an alternative action for when the expression is false.

```
if( x == 10 )
{
        x = 0;          // if x is equal to 10, reset it to zero
}
else
{
        x++;            // if x is NOT equal to 10, increment it
}
```

You can make quite complex programs with “if” and “else.” For example, you can use “else if”!

```
if( x == 10 )           // if x is equal to 10, reset it to zero
{
        x = 0;
}
else if( y == 10 )
{
        y = 0;          // if x is NOT equal to 10 but y is, reset y
}
else
{
        x++;            // if neither x nor y are equal to 10, increment x
}
```

Blocks

In the examples shown above, we have only one statement within the braces. If we want multiple statements associated with one condition, we can just add additional statements within the braces. The statements within the braces are called a “block.” For example, the following “if” statement executes two instructions when the condition is true.

```
if( level == 100 )
{
        // reset level and increment the count
        level = 0;
        reset_count++;
}
```

Loops – “while”

Loops let you repeat an instruction, or block, while a condition is true. The format is “while(*condition*)”.

```
uint8 cnt = 0;
uint16 total = 0;
while( cnt < 100 )              // Loop100 times
{
        cnt++;
        total += cnt;
}
```

Loops – “for”

The “for” loop is a short-hand form of “while”. In the above example there is an initializer (cnt = 0) and an increment (cnt++). This is a very common type of loop and so the “for” loop lets you put all of that in one line – “for(*initializer* ; *condition* ; *increment*)”.

```
uint8 cnt;
uint16 total = 0;
for( cnt = 0; cnt < 100; cnt++ )                // Loop100 times
{
            total += cnt;
}
```

Recommendation: use “while” in loops of indeterminate length, e.g. Repeat until a condition is no longer true.

Use “for” in loops with known number of iterations, e.g. Repeat 100 times.

Running Programs

Your code needs a place to start and the compiler does this for you with the “main” function. A function is a block of code with a name. This main function is the start point for all programs. When you turn on a PSoC, or reset the board, it boots up and starts running the code in main. The syntax is like the “if” and “while” keywords with parentheses after the name of the function.

```
uint8 count = 0;
uint8 main()
{
            while( count < 100 )            // this simple program just increments a variable 100 times
            {
                        count++;
            }
}
```

The Super-Loop or Infinite Loop

Embedded Systems rarely stop. If power is applied, they run forever. So you often see a super-loop, which is a never-terminating loop. PSoC Creator gives you a template main function with a super-loop. The for loop has a special syntax for these kinds of loops – “for(; ;)”

You can do the same thing with while – “while(1)”.

Functions

Main need not be the only function in a real program. Good programs put frequently-used code into a function so that it can be used repeatedly without using up flash space. A function is defined once (and the compiler allocates flash code for it) but can be used many times (without using more flash).

Functions are “called” from other functions. In this example, main calls a (fictitious) function “print”, which runs, then returns to main.

```
int main()
{
        uint16 x = 0;
        for( ; ; )
        {
                print( x );             // Call the print function
                x++;                    // Increment x once the print has completed
        }
}
```

Creating Functions - Arguments

Functions let you pass data into them in the form of parameters. In the example above, print is said to “accept” an unsigned 16-bit integer. The parameter can be a number, like 5, or a variable, like x. The print function does not discern the difference – it just prints ‘5’ or whatever the value of x is. The parameters you “pass” into the function are treated like new variables within the function. This method of parameter passing is called pass-by-value.

```
void print( uint16 print_value )            // My function is called “print”. It accepts a 16 bit unsigned integer
{
        LCD_Position(0,0);                  // Set the LCD cursor to row 0, column 0
        LCD_PrintNumber(print_value);       // Display the value on the LCD
}
```

From main(), or any other function, you can then call your function as often as you like.

```
        print( 0 );
        print( 123 );
        print( x );
```

Note that if you modify the value of a parameter in the function its new value is NOT available in the main program. That is because the variables inside a function are local variables. If you need a value in a function to be available to the main program, you can have the function return a value (discussed later), you can use global variables, or you can use a method called pass-by-reference (also more to come later).

Global variables are declared at the start of the program (outside of the main function) and are accessible to any functions in that file. Global variables can be accessed and modified by any function in the program. However, this method is discouraged for large projects especially when multiple people are working on it as it can be difficult to debug and maintain.

Functions that take no arguments should be defined with the “void” keyword within parenthesis:

```
void print( void )                  // This function takes no argument
{…}
```

Creating Functions – Multiple Arguments

Functions can have multiple arguments. Use commas to separate them when you create and call your function.

```
void bigfunc( uint8 first_run, int8 use_me, uint32 number )
{ … }
```

The bigfunc() accepts three parameters and you would call it like this:

```
bigfunc( 1, ‘a’, 55 );
```

The notation ‘a’ in C replaces the single character with its ASCII equivalent value.

Functions can accept parameters of any type, including arrays. To pass an array to a function, just provide the name of the array as shown here:

```
uint8 array[10];                    // An array of ten integers
init_buffer( array );               // init_buffer can now access all elements of the array
```

Unlike regular variables, any changes to elements of an array inside a function will be reflected in the array within main program. This is because the name of an array is actually a pointer rather than a variable. This method of passing a value is called pass-by-reference and can also be applied to regular variables. To do this, we need to provide the location of the variable in memory instead of its value. This is called a “pointer”. We can provide a pointer by putting the “&” in front of the variable name.

```
uint8 myvar;                        // An integer
set_buffer( &myvar );               // set_buffer can now read and write to myvar from the calling function
```

When you want to read or modify the value of the variable from inside the function, you need to dereference it. The “*” is placed in front of a pointer in order to dereference it as shown here:

```
void set_buffer( uint8 *val )       // declare function which accepts a pointer to a unit 8
{
        (*val)++;                   // increment the value stored in val
        …
}
```

If the pointer was not dereferenced, the program would be incrementing the memory location of the myvar variable rather than its actual value. Fortunately, the compiler will tell you that you have type mismatches if you try to do this.

Creating Functions – Return Values

Functions can return data to the calling function by using the “return” keyword. You use the equals operator in the main program to get the returned value, e.g. `x = Am_I_Frisky(1);`

```
uint8 Am_I_Frisky( int frisky )
{
        if( frisky == 1)
        {
                return 1;
        }
        else
        {
                return 0;
        }
}
```

Functions that do not return a value use the “void” keyword before the function name. Functions that return void cannot be used with the equals operator.

```
void return_nothing( void );
```

Cypress Component Functions

When you use a Cypress component in PSoC Creator, a set of functions will be generated by the badass software. These are often referred to as “APIs” - application program interface - but they are just regular functions. These APIs allow the program you write to interact with the component such as to read the state of a pin or write a string to the LCD.

Example projects, shipped with Creator, show how to use these functions. They always start with the instance name of the component and an underscore.

```
PWM_1_WriteCompare(1000);
InputPin_Read();
OutputPin_Write(1);
```

The _Start() API is particularly important because it turns the component on. Almost all components have an _Start() API so make sure you call it or your program will not work. For example:

```
PWM_Start();
ADC_Start();
```

Building a Real Program

```
void print_num( uint16 num)                             /* My function that prints numbers on the LCD */
{
        LCD_Position( 10, 0 );                          /* set the LCD position */
        LCD_PrintNumber( num );                         /* Print the decimal number */
}

int main( void )
{
        uint16 number = 0;
        LCD_Start();                                    /* Start the LCD or it will do nothing! */
        LCD_PrintString( “number =“ );                  /* Just print this once, it says on-screen */

        for(;;)
        {
                if( number < 100 )                      /* Loop 100 times */
                {
                        print_num( number );            /* Update the LCD */
                        number++;                       /* Increment the number */
                        CyDelay(500);                   /* Wait 500msec (0.5 second) */
                }
                else
                {
                        number = 0;
                }
        }                                               // End of the super-loop
}                                                       // End of main
```

Macros

Before the compiler runs, a “preprocessor” is called that reads the code and makes macro substitutions. Make your programs easier to read by using macros to avoid “magic numbers”. In the following example the meaning is clearer which makes for easier maintenance, and you can change things like the array size with just one edit which reduces silly mistakes.

```
#define NUMBER_ROW_POS (0)
#define NUMBER_COL_POS (10)
#define LOOP_COUNT (100)
/*
print_num - prints the integer argument to the LCD.
*/
void print_num( int num)
{
        LCD_Position(NUMBER_COL_POS, NUMBER_ROW_POS );
        LCD_PrintNumber( num );
}
/*
main - sets up the LCD and prints an incrementing number in a loop.
*/
void main( void )
{
        int number = 0;

        /* Start the LCD and print some unchanging text */
        LCD_Start();
        LCD_PrintString( “number =“ );

        /* Repeatedly update the LCD with the incremented number */
        while( number < LOOP_COUNT )
        {
                print_num( number );
                number++;
        }
}
```

Typically the names of macros are in all uppercase letters to distinguish them from variables within the program. It is also a good practice to enclose the substituted expression in parenthesis. This can prevent unintended mistakes when substitution is done. It is also recommended NOT to use comments on the same line as a macro since some compilers may actually substitute in the comment which can cause program errors.

Comments

Note how the above code uses a lot of white space and comments to explain what is going on. The next person to read your code may not be as smart as you!

There are multi-line comments like this…

```
/*
Function: void FindFreddy()
*/
```

And there are end-of-line comments, like this…

```
dessert += split( banana );                    // Make dessert just a little better!!!!
```

Quick Reference

Program Flow Control

if, else if, else	Execute based on one or more conditions
switch, case	Select which case to execute based on an integer switch value
for	Execute code a specified number of times
while	Execute code while a condition is true
do-while	Execute code while a condition is true, always execute at least once

Operators

	Operation	Example use
+	Addition	var1 = var2 + var3;
-	Subtraction	var1 = var2 – var3;
*	Multiplication	var1 = var2 * var3;
/	Division	var1 = var2 / var3;
<<	left shift	var1 = var2 << 2; /* Shift left by 2. Same as multiply by 4 */
>>	right shift	var1 = var2 >>1; /* Shift right by 1. Same as divide by 2 */
&&	logical AND	if(var1 == var2 && var3 == var4) /* Both arguments must be true */
||	logical OR	if(var1 == var2 || var3 == var4) /* Either argument must be true */
!	logical NOT	if(!var1) /* Same as if(var1 == 0) */
&	bitwise AND	var1 = var2 & 0x0F; /* Clear the first 4 bits of var2 and set result as var1 */
|	bitwise OR	var1 = var2 | 0x0F; /* Set the first 4 bits of var2 and set result as var1 */
^	bitwise XOR	var1 = var2 ^ var3; /* XOR bits of var2 with var3 and set result as var1 */
~	bitwise NOT (ones compliment)	var1 = ~var2; /* Invert all bits of var2 and set result as var1 */
=	assignment	var1 = 10;
+=	addition assignment	var1 += 10; /* Add 10 to var1. Same as var1 = var1 + 10;*/
-=	subtraction assignment	var1 -= 10; /* Subtract 10 from var1. */
*=	multiplication assignment	var1 *= 10; /* Multiply var1 by 10. */
/=	division assignment	var1 /= 10; /* Divide var1 by 10. */
<<=	left shift assignment	var1 <<= 3; /* Shift var1 left by 3 (i.e. multiply by 8) */
>>=	right shift assignment	var1 >>=4; /* Shift var1 right by 4 (i.e. divide by 16) */
&=	bitwise AND assignment	var1 &= ~0x01; /* Clear first bit of var1 */
|=	bitwise OR assignment	var1 |= 0x01; /* Set first bit of var1 */
++	increment	var1++ /* Same as var1 = var1 + 1; or var1 += 1; */
--	decrement	var1-- /* Same as var1 = var1 – 1; or var1 -= 1; */
==	comparison - equal	if(var1 == var2)
!=	comparison – not equal	if(var1 != var2)
>	comparison – greater than	if(var1 > var2)
<	comparison – less than	if(var1 < var2)
>=	comparison – greater than or equal	if(var1 >= var2)
<=	comparison – less than or equal	if(var1 <= var2)
? :	conditional expression	expr1 ? expr2 : expr3 /* If expr1 1 is non-zero (true), do expr2. Otherwise, do expr3 */

Note that operators have an order of precedence so expressions are not always evaluated from left to right. You should always use parenthesis to make sure operations happen the way you expect. For example, 10+2*3 will return 16 since * has a higher precedence than + while (10+2)*3 will return 36.

A Complete Example PSoC Program

```
#include <project.h>              /* Include source code generated by Creator */

#define FALSE    0                /* Compiler directives. The compiler will replace FALSE with 0 */
#define TRUE     1                /* and TRUE with 1 everywhere in the code from this point on */

uint8 flag = FALSE;               /* Variables declared outside any functions are global */

int main()
{
        /* Initialization - stuff that is only done once (e.g. start components, initialize variables) */
        /* Variables must be declared before they are used */
        uint8 var1 = 0;                 /* 8 bit unsigned integer (0 to 255) */
        uint16 var2 = 0;                /* 16 bit unsigned integer (0 to 65535) */
        int8 var3[5] = {0,1,2,3,4};     /* 8 bit signed integer array (2's compliment: -128 to 127) */
        uint8 i;                        /* Loop counter */

        /* Super Loop - stuff that repeats forever (e.g. read inputs, drive outputs) */
        for(;;)
        {
                if(flag == FALSE)               /* To test a value, use ==, <, >, <=, >= */
                {
                        var1 = 1;               /* Assign a value with "=" */
                }
                else if(flag == TRUE)
                {
                        var2 = 10;
                }
                else                            /* This executes if flag is not equal to 1 or 0 */
                {
                        var3[0] = 100;
                }

                switch(flag)
                {
                        case 1:                 Executed if flag is 1 */
                                var1 = 5;
                                break;          /* If break is not included, execution will fall through */
                        case 2:
                        case 3:
                                var1=15;        Executed for flag values of 2 or 3 */
                                break;
                        default:
                                var1=25;        Executed for flag values that are not 1, 2, or 3 */
                }

                for(i = 0; i < sizeof var3 ; i++) /* sizeof will give us the size of the array var3 */
                {
                        var3[i] = var3[i] + 10;   /* Add 10 to each element of var3 array */
                }

                while(i < 10)                   /* Execute until i is 10 */
                {
                        var1++;
                        i++;
                }

                do
                {
                        var2++;
                        i++;
                } while(i < 10);                /* Execute until i is 10. Always execute at least
                                                   once regardless of the starting value of i. */
                                                /* Note the ending ";" */
        }                                       /* End of Super Loop */
```