Elements of a C Program

Simplest C Program (does nothing).
Note that line numbers are for reference only. They MUST NOT be typed in as part of a program.

```
1    #include <ez8.h>
2    int main(void)
3    {
4            return 0;
5    }//end main
```

Line 1 – Preprocessor directive that specifies that the file *ez8.h* is included in the Z8 project. This file contains many definitions of addresses for the different hardware components of the Z8 processor. It also defines **symbolic constants** that allow us to use easy-to-remember names for registers instead of their addresses. For example, the address of the Port A Control Register is 0xFD1. (See Figure 1 on page 25 of your course package to verify this.) The *ez8.h* file allows us to write PACTL instead of 0xFD1, which makes our programs easier to understand.

Line 2 – The *main()* function accepts nothing as input (*void*), but returns an integer value (*int*) when it is finished . Every program requires a *main()* function.

Lines 3 and 5 – {} Braces or curly brackets. These enclose the contents of a function or a block of code. If a C statement causes several statements (a block of code) to be executed, the braces will enclose those statements.

Line 4 – The *return* statement indicates that the *main()* function is finished. A value of 0 is assigned to the function. Notice that the *return* statement ends with a semicolon (;). Most program statements in C end this way. One exception is preprocessor directives, which start with # and do not have ending punctuation.

Line 5 - // Is one way to indicate comments. Text after // is not part of the program. It is used to explain program operation. Placing // after a closing brace helps to show the structure of the program. This type of comment can only be used on one line, and is thus called an "inline comment". Another commenting style (not shown above) is to use /* to open a comment and */ to close it. This type of comment, called a "block comment" can continue onto more than one line.

e.g.
```
// This is an inline comment

/* These lines
   make up
   a block comment */

/*********************************************
 * These lines also make up a block comment,*                        *
 * but with a more "graphic design" look    *
 *********************************************/
```

Simple C Program (outputs data to LEDs on PORT A[2..0]).
Note that line numbers are for reference only. They MUST NOT be typed in as part of a program.

```
/*      This program configures the last three bits of Port A as outputs
           and writes a value of 0x03 to the LEDs on PORT A.
        The result is the active-LOW LED on PORT A, bit 2 turns on.
*/
1       #include <ez8.h>
2       int main(void)
3       {
4               char led_value = 0x03;
5
6               PAADDR = 0x01; // Data Direction
7               PACTL = 0xF8;   // PA[7..3] = input, PA[2..0] = output
8
9               PAADDR = 0x02; // Alternate Function
10              PACTL = 0x00;   // Disable
11
12              PAADDR = 0x03; // Output Control
13              PACTL = 0x00;   // Normal (not open drain)
14
15              PAADDR = 0x00;        // Prevent accidental reconfiguration
16
17              PAOUT = led_value;   // LED output
18
19              return 0;
20      }//end main
```

Line 4 – Declare a variable called *led_value* of type *char* (i.e, a character variable, which always has 8 bits). Assign the value 0x03 or 00000011_2 to *led_value*.

Lines 6-15 – Set up PORT A for the proper kind of operation, using the Port A Address Register (PAADDR) and the Port A Control Register (PACTL).

PAADDR	**PACTL**	**Comment**
0x01	0xF8	Select the Data Direction sub register for Port A. By writing 0xF8 to PACTL, the upper 5 bits of Port A are set as inputs and the lower 3 bits are set as outputs. This is because we write 0xF8, which is 11111000 in binary. A data direction bit with a 1 is an input. A data direction bit with a 0 is an output. (This convention is specific to the Z8 CPU. Other processors may have the opposite convention.)
0x02	0x00	Select the Alternate Function sub register for Port A. Disable the Alternate Function by writing 0x00 to PACTL, so that Port A acts like a normal input/output port.
0x03	0x00	Select the Output Control sub register for Port A. Set the Port A outputs to "normal" outputs by writing 0x00 to PACTL. The alternative is to set the outputs as "open drain", which has only logic LOW and open-circuit states. Open drain outputs have no logic HIGH states. A bit is set to open drain by writing a 1 to its bit position in PACTL when the Output Control sub register is selected.
0x00	-	Disable the PACTL register by selecting no Port A sub register. This prevents accidental reconfiguration of the Port A control settings.

Line 17 – Send the contents of the variable *led_value* to the Port A output register.

Comments: Good and Bad

A comment should ideally shed light on why a statement is written. For example the comments for the following two lines of code both answer the question, "Why am I writing this line?"

PAADDR = 0x01; // Select Data Direction Register
PACTL = 0xF8; // Configure outputs: PA[7..3] = input, PA[2..0] = output

Think of having a dialog between yourself and someone who doesn't know the program:
You: "PAADDR = 0x01;"
Someone else: "Why?"
You: "I want to configure the Data Direction Register of Port A. This is how I select it."

You: "PACTL=0xF8;"
Someone else: "Why?"
You: "Now that I have selected the DDR for Port A, I need to write a value to it that sets bits 7 to 3 of the port as inputs and bits 2 to 0 of the port as outputs. 0xF8 does this."

The same two lines can also be commented really badly by essentially restating the code in English, as follows. Do not do this. These comments do not actually explain what the code does.

PAADDR = 0x01; // Set the Port A Address Register to 0x01
PACTL = 0xF8; // Set the Port A Control Register to 0xF8

Preferred indentation style: one tab stop for every pair of braces {}

```
#include<ez8.h>

int main(void)
{
    int x = 0;
    int y;
    while(x<5)
    {
        if(x==0)
        {
            y=0;
        }//end if
        else
        {
            y=y+x;
        }//end else
        x++;
    }//end while
    return 0;
}//end main
```

OK, but not great (no indentation):

```
#include <ez8.h>
int main(void)
{
return 0;
}//end main
```

Works, but bad (hard to read):

```
#include <ez8.h> int main(void){return 0;}
```

OK, but not great:

```
#include <ez8.h>
int main(void)
{
char led_value = 0x03;
PAADDR = 0x02; // Alternate Function
PACTL = 0x00; // Disable

PAADDR = 0x01; // Data Direction
PACTL = 0xF8; // PA[7..3] = input, PA[2..0] = output

PAADDR = 0x03; // Output Control
PACTL = 0x00; // Normal (not open drain)

PAADDR = 0x00;     // Prevent accidental reconfiguration

PAOUT = led_value;      // LED output

return 0;
}//end main
```

Works, but really bad (no line breaks or indents):

```
#include <ez8.h> int main(void) {char led_value = 0x03;PAADDR = 0x02;PACTL = 0x00;PAADDR = 0x01;
PACTL = 0xF8; PAADDR = 0x03; PACTL = 0x00; PAADDR = 0x00; PAOUT = led_value; return 0;}
```

C Variables and Constants

Variable
A value stored in memory that can change when a program executes.

Constant
A value stored in memory that cannot change when a program executes.

Type
The **type** of a variable or constant specifies the values that it can hold. Mostly, C has integer, floating point, and literal types.

Integers
Integers are whole numbers. The bit size of an integer type depends on the computer for which it is defined. The main integer types for the Z8 are:

Type	Size	Range
char	8 bits (1 byte)	-128 to +127
unsigned char	8 bits	0 to 255
int	16 bits (2 bytes)	-32768 to +32767
unsigned int	16 bits	0 to 65535
long int	32 bits (4 bytes)	-2^{31} to $+2^{31}-1$
unsigned long int	32 bits	0 to $2^{32}-1$

Floating Point
Floating point numbers are positive or negative real numbers. That is, they have a decimal point. They are indicated by the C keyword **float**. For example:

```
float pi;
pi = 3.1415926;
```

Literals
Numbers and letters can be represented by the 7- or 8-bit codes of the American Standard Code for Information Interchange (ASCII). ASCII characters can be assigned to integer variables as literals, or quoted values. For example:

```
char x,y;
x = 'A';
y = 'a';
```

Preprocessor Directives

A **preprocessor directive** is text, not part of the actual C program, that is used to make the C program easier to read.

#include

The #include directive is used to include program code from files other than the one being written. For example, *#include <ez8.h>* is used to include code that defines addresses for various processor hardware. Other includes might include code that initializes the Z8 system. This takes away the necessity for the programmer to focus on a lot on complicated initialization code every time a program is written.

#define

The #define directive is used to substitute text in a program. It can be used to define constants to make them easy to write in a program. For example, to calculate 128 points of the function **x=sin(2nπf/f$_s$)**:

```
#include <ez8.h>
#include<math.h>

#define PI      3.1415926
#define F       1000
#define FS      8000

int main(void)
{
        float x[128];    // declare an array of 128 points
        int n;           // declare a variable to track which point is being used

        for(n=0; n<128; n++)
        {
                x[n] = sin(2*n*PI*F/FS); // Calculate x[n] for each value of n
        } //end for
        return 0;
}//end main
```

Possible Typo Errors in C

The first place to look for errors in a C program is in common typographical errors. Some of the most common are:

- missing semicolon (;) at the end of a line
- missing /* */ or // for comments
- unbalanced () or {}
- case error (recall that myvar and MyVar are not the same)
- spelling error (e.g., **int main(vois)** instead of **int main(void)**)

Tip: If the compiler returns an error message for code on line *x*, try looking on an earlier line. For example, if an error is indicated on line 5, but line 5 looks correct, look for an error on line 4 or earlier. Sometimes the compiler will try to make sense of an error by continuing to read code on later lines. If this does not work, the program will crash, but maybe not exactly where the error occurred.