ECE331 Lab #2: -- Double Click Detector in C

By Jianjian Song and Keith Hoover
Department of Electrical and Computer Engineering
Rose-Hulman Institute of Technology
Updated September 8, 2010 (KEH)

Table of Contents

1 Objectives

(1) Understand C program development procedure for the 9S12C128 microprocessor using CodeWarrior Development Studio for S12(X) V5.0.

(2) Learn embedded C programming by studying the C-language versions of the assembly-language programming examples from Lab 1, the "LED Flashlight" and the "Coin Tosser".

(3) Understand how the C compiler works by observing how each line of C code is turned into one or more lines of assembly code.

(4) Write your own embedded C program for implementing a double/single click detector.

2 Deliverables

Turn in a written memo-style lab report that focuses on your design of the double/single click detector. Be sure to describe how you tested your double click detector, and the results of your tests. Your memo must refer to several attachments, which include (1) ORCAD drafted schematic diagram of interface circuitry (2) Flow diagram of the double/single click detector program. (3) Adequately commented C source code file. (4) Resulting C code listing file

produced by the CodeWarrior C compiler and linker, which shows the underlying assembly instructions behind each of the lines of C code. This file is created by selecting the **main.c** program in the project view window, and then clicking on **Project – Disassemble**.

3 Reference materials and required equipment

3.1 Reference materials (These materials are in the 9S12C128 documents AFS class folder.)

1. CSMB12C128 Module User's Guide (SLKS12UG.pdf)
2. CSMB12C128 Module Schematic Diagram (CSMB12_SCH_A1_0..pdf)
3. Getting Started with the Microcontroller Student Learning Kit Project Board (S12QSG.pdf)
4. Project Board Schematic Diagram. (PBMCUSCHEMSLKREVB.pdf)
5. MC9S12C Family Device User Guide (***9S12C128DGV1.pdf***)
6. HCS12 Microcontrollers CPU Reference Manual (***S12CPUV2.pdf***)
7. HC12 CPU Awareness and True Time Simulator Manual (***Manual True Time Simulator HC12.pdf***)

3.2 Equipment

Same equipment as for Lab 1.

3.3 Source Code

Source code for the C-language LED flashing program (flash.c) and for coin-tossing program (cointoss.c) are available from the ECE331 Angel class folder.

4 First C-language programming example: flashlight program revisited

Study the C-language source code for the flashlight program shown in Figure 2, assuming the hardware configuration of Figure 1. You may find the summary of looping and branching constructs in the C language to be helpful, which is included as **Appendix A**. For more complete information on the C language, see Chapter 5 of our textbook.

You may download this example code from the Lab 2 class folder, or enter it yourself.

Note the use of the ***#define*** "preprocessor" directives to assign meaningful I/O address names (PTAD) and DDRAD) to their corresponding numerical values (0x270 and 0x272). These numerical address values have been "typecast" as pointers to unsigned character (8-bit integer) values, as shown below:

```
#define PTAD  (*(unsigned char *) 0x270)  /* Define PORT AD data register  address*/
#define DDRAD  (*(unsigned char *) 0x272)  /*PORT AD data direction register address */
```

4.1 Steps to compile and run flashlight program

Working as you did in the previous lab, your first task is to create a project for the program. Do this by following the steps presented below:

Figure 1. Flashlight Interface to 9S12C128 module

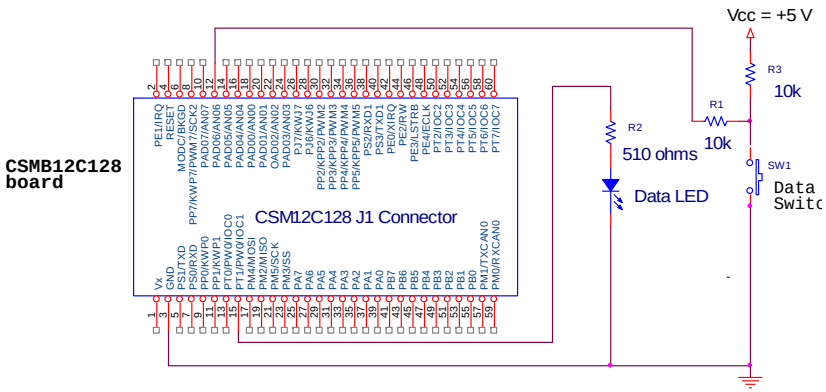

Figure 2. C-language version of flashlight program

```
//***********************************************************************
// ECE331 Lab 2 Flashlight.c demo program
// C-language implementation of flashlight program (KEH, August 2009)
// Hardware interface: SW on input pin PAD6 (HIGH level when not pressed)
//                     LED on output pin PT1 (Turn on with HIGH level)
//***********************************************************************
#define PTAD  (*(unsigned char *) 0x270)          /* Define the relevant register addresses */
#define DDRAD  (*(unsigned char *) 0x272)
#define PTT  (*(unsigned char *) 0x240)
#define DDRT  (*(unsigned char *) 0x242)
#define ATDDIEN  (*(unsigned char *) 0x8D)
#pragma LINK_INFO DERIVATIVE "mc9s12C128"   /* Tell linker to allocate the program into the
                                               RAM and (Flash) ROM address spaces
                                               that conform to those available on the
                                               9S12C128 microcontroller. */
void main(void)
{
 DDRT =    0b00000010;                      // LED on PT1 */
```



```
 ATDDIEN = 0b01000000;                     // Make PAD6 a digital I/O pin
 DDRAD =   0b00000000;                     // Make PAD6 an input (A switch is connected to this pin) */
 for(;;)
 {
  PTT = PTT & 0b11111101;                  // Turn OFF LED
  while ((PTAD & 0b01000000) == 0b01000000); // Hang here while SW is not pressed (HIGH).
  PTT = PTT | 0b00000010;                  // Turn on LED .
  while ((PTAD & 0b01000000) == 0);        // Hang here while SW is pressed (LOW)
 }
}
```

4.1.1 Launch Code Warrior

Click on ***Start – Freescale CodeWarrior – CodeWarrior Development Studio for S12(X) V5.0 – CodeWarrior IDE*** in order to launch CodeWarrior.

4.1.2 Create New Project

Create a new project by clicking on the "***Create New Project***" button. Or else click on ***File – New Project.*** The ***Device and Connection*** window appears. Expand the ***HCS12*** derivative category, and then expand the ***HCS12C*** Family category. Finally single left click on ***MC9S12C128***. Select the ***P&E Multilink/Cyclone Pro*** connection. Click ***Next***.

The ***Project Parameters*** window appears. Leave the "***C***" language box checked. Type in a project name (Lab2_Flashlight), click the "***Set***" button and browse to the desired folder location (no embedded spaces in the path are allowed.) You may make a new folder to put the project in at this time using the file-shaped "Create New Folder" icon which appears at the top of the "Choose Project Location" window. By the way, I strongly recommend that you create a *separate folder* for each new project you create, and that you name your project file the same as the folder name it is in, since there are many files in each project. Click ***Next.***

The ***Add additional files*** opens. You may simply click ***Next***.

The ***Processor Expert*** window appears. Make sure "***None***" is checked. Click ***Next***.

The ***C/C++ Options*** window appears. Make sure ***ANSI startup*** code is checked. Check the ***Small*** memory model. Select "***None***" for floating point format. Click ***Next.***

The ***PC-Lint*** window opens. Make sure ***No*** is checked. Click ***Finish***.

The project now opens.

To resume work on this project at a later time, you would click ***File – Open – LAB2_FLASHLIGHT.mcp***. This will reopen this project for further work.

4.1.3 Entering the flashlight C-language source program

Note the project file name appears in the upper left corner of the screen. Just below this is the "***project view***" that lists the files in the project. Expand the "***Sources***" category (which contains the source code files) by single-left clicking on the [+], and then double-left click on the only source code file, "***main.c***". This should open up a source code editing window labelled ***main.c***. Expand this window so it fills the screen. The initial "***main.c***" program serves as a starting template (or "***project stationary***", using the Freescale terminology) for the creation of

a "C-language" program. The initial source code template file "***main.c***" is shown in Figure 3.

Figure 3. Initial template file for main.c generated by CodeWarrior

```
#include <hidef.h>      /* common defines and macros */
#include "derivative.h"      /* derivative-specific definitions */
void main(void) {
  /* put your own code here */
      EnableInterrupts;
  for(;;) {
    _FEED_COP( ); /* feeds the dog */
  } /* loop forever */
  /* please make sure that you never leave main */
}
```

Replace the contents of this main.c template file with the flashlight program shown in Figure 2.

4.1.4 Building the project and the main.c.o.lst file

The project can be built as in the first lab, by clicking on the ***Make*** button, which will cause the C compiler to compile each line of C code into one or more assembly language instructions, and then the assembler will assemble these instructions into machine language, and finally the linker allocates the code and data portions of the program into the correct areas of memory. Once the project has been successfully built, you can see how the various lines of C code have been interpreted by the compiler into underlying assembly (machine) language instructions that the microcontroller hardware can understand. Do this by ***highlighting*** the "***main.c***" program in the project view (click on this program if it is not highlighted), and then click on ***Project – Disassemble***. The ***main.c.o.lst*** file that is shown in Figure 4 should appear. This file shows each line (C-language instruction) of C source code in the ***main.c*** program of Figure 2 followed by the underlying assembly language instructions that the C compiler has generated in order to carry out that C-language instruction!

Note that this list file represents the compiler output ***before*** the linker has done its job, so it has been assembled as if it were to be run at the fictitious program memory location 0. That is why the first instruction in the program "CLI" begins at location 0 in Figure 4. However, the 9S12C128 microcontroller has no program (Flash ROM) memory at location 0, and furthermore locations 0 - $3FF are reserved for I/O registers on this microcontroller.

Also note that this project file contains another standard C program file called ***Startup.c*** (you can see it by expanding the Startup Code category in the project view.) Startup.c is a standard file that accompanies the C compiler. It contains some standard initialization code that depends upon the target microcontroller that must be executed on the microcontroller to initialize it before it begins executing the actual user C program. This file has been separately compiled into machine language, and so it also has been assembled to begin running at the fictitious starting location 0. To see that this is true, highlight ***Startup.c*** and then click on ***Project – Disassemble***.

The linker has the job of relocating the compiled machine language program code from both the ***main.c*** and ***Startup.c*** programs so that the two programs are placed immediately adjacent to each other, one

after the other (linked together), in the user Flash ROM program memory, with the starting address of the Startup.c program shifted from address 0 up to a convenient starting location in Flash ROM.

Figure 4. Portion of main.c.o.lst file corresponding to the flashlight program of Figure 2. Note how each line of C code is translated into one or more machine (assembly) language instructions.

```
 17:    DDRT =    0b00000010;   // LED on PT1 */
0000 cc4002       [2]     LDD   #16386
0003 7b0242       [3]     STAB  578
 18:    ATDDIEN = 0b01000000;   // Make PAD6 a digital I/O pin
0006 5a8d         [2]     STAA  141
 19:    DDRAD =   0b00000000;   // Make PAD6 an input (Switch on this pin) */
0008 790272       [3]     CLR   626
 20:    for(;;)
21:    {
 22:      PTT = PTT & 0b11111101;                  // Turn OFF LED
000b 1d024002     [4]     BCLR  576,#2
 23:      while ((PTAD & 0b01000000) == 0b01000000); // Hang here while SW not pressed
(HIGH).
000f 1e027040fb   [5]     BRSET 624,#64,*+0 ;abs = 000f
 24:      PTT = PTT | 0b00000010;                  // Turn on LED .
0014 1c024002     [4]     BSET  576,#2
 25:      while ((PTAD & 0b01000000) == 0);          // Hang here while SW pressed (LOW)
0018 1f027040fb   [5]     BRCLR 624,#64,*+0 ;abs = 0018
001d 20ec         [3]     BRA   *-18 ;abs = 000b
 26:    }
```

Note, for example, how the "***DDRT = 0b00000010***;" instruction at Line #17 in Figure 4 was assembled by the C compiler into two assembly-language (machine) instructions: "LDD #$4002" followed by "STAB 578", where 578 is the decimal equivalent of 0x242, which is the correct address of the DDRT register.

Next note that the "***while ((PTAD & 0b01000000) = = 0b01000000)***;" instruction at Line #23 in Figure 4 was assembled by the C compiler into just one assembly-language (machine) instruction: "BRSET 624,#64,*+0". As you can see by consulting the HCS12 Microcontrollers Reference Manual (***S12CPUV2.pdf***), the BRSET (branch on set) instruction has three arguments separated by two commas. This instruction reads the byte at the address indicated in the first argument, which in this case is address 624 = 0x0270 = PTAD. Next it checks the data it read at the bit position(s) that are set to "1" in the "immediate mask pattern" found in the second argument, which in this case is #64 = #0b01000000. So in this case, Bit #6 of PTAD is checked. If Bit #6 is a 1, then the branch is taken to the location indicated in the 3rd argument, which in this case is indicated by "*+0". The asterisk "*" is called the "***assembly location counter***", and so it represents the address of the first byte of the current instruction being assembled. Thus this instruction will branch back to itself if Bit #6 is a 1! This is certainly the tightest of all possible loops! If Bit #6 is not a 1, then the branch is not taken, and so the next instruction is executed.

Finally note how the infinite loop "***for(;;) {…….}***" was implemented by the single assembly language instruction at the end of the program "BRA *-18". This "branch always" instruction will always (unconditionally) branch back 18 bytes from the beginning of the current instruction. If start at the BRA instruction OP CODE address and count backwards 18 instruction bytes, you will see that , when BRA *-18 instruction is executed (at the bottom of the infinite loop), it will cause the program execution to branch back to the top of the main(;;) infinite loop, at the "BCLR 576" instruction.

Please look through the rest of Figure 4, making sure that you understand how each C language instruction has been translated by the C compiler into one or more underlying machine language instructions.

4.1.5 Running the Flashlight Program

Now connect an LED to PT1 and a pushbutton switch to PAD6. Be sure that the "***P&E Multilink Cyclone Pro***" is selected, hit the ***RESET*** button on the 9S12C128 module, and click on the green "***Debug***" button. Next click on the ***Run*** button in the debugger, and the flashlight program should begin running. Note as you operate the pushbutton switch, the LED changes accordingly. Try setting a breakpoint in the program, say at the "***PTT = PTT | 0b00000010;***" instruction, then reset the program and hit the ***Run*** button. The program should run at speed until the breakpoint is hit. Note that you will have to depress the pushbutton before the breakpoint will be hit.

5 Second C-language programming example: Coin-tosser program revisited

The coin-tosser program has been rewritten in C, and it is presented in Figure 6. Its flowchart, which is very similar to the coin-tosser flowchart in Lab 1, appears in Figure 5. Note in Figure 6 that this program uses the preprocessor directive ***"#include <mc9s12C128.h>***" to include the 9S12C128 microcontroller's register definitions, so the I/O registers need not be separately defined as they were in the flashlight program of Fig. 2. Make sure that you understand how the flowchart of Figure 5 has been implemented in the C-language program of Figure 6.

Figure 5. Coin-Tosser Program Flowchart

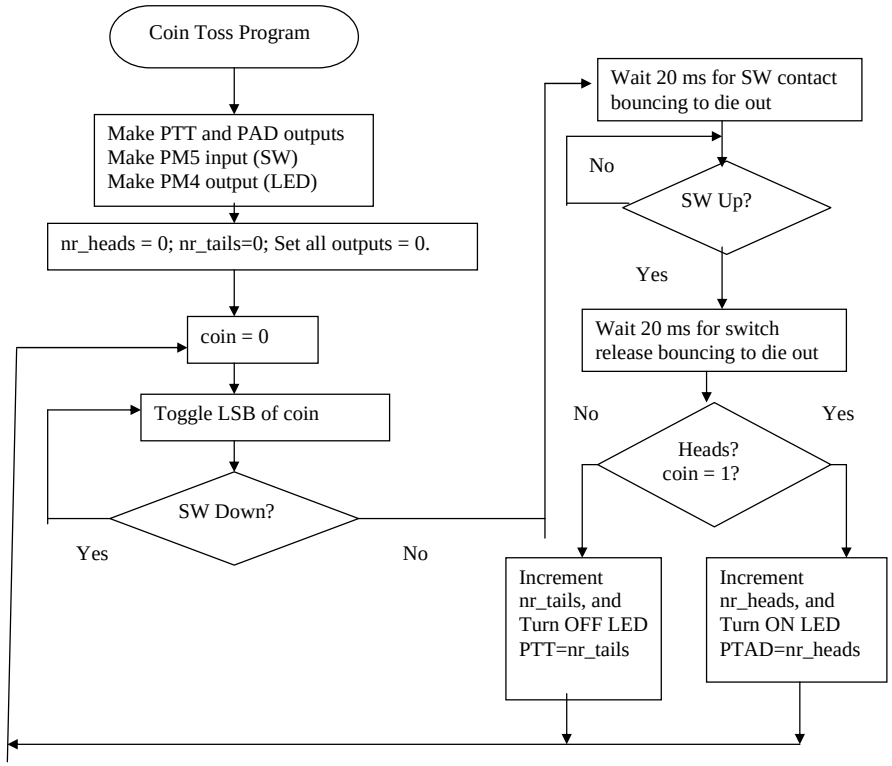

Figure 6. C-language version of coin tosser program corresponding to the flowchart of Fig. 5.

```
/* C-language version of COIN TOSSER example
   Hardware interface:      PM5 = Pushbutton SW (Click to toss coin)
                            PM4 = Head/Tail Outcome LED
                            PTT = 8 LEDs displaying # tails
                            PTAD= 8 LEDs displaying # heads */
#include <mc9s12c128.h>     /* derivative information */
#pragma LINK_INFO DERIVATIVE "mc9s12c128"
void delay20ms(void);     /* This "function prototype statement" lets the compiler know
                             that a function called "delay20ms( )" which accepts no arguments
                             and returns now arguments will follow the main program. */
void main(void)
{
  unsigned char coin,nr_heads, nr_tails;
  DDRT = 0xff;              //Make PTT and PTAD all outputs
  ATDDIEN = 0xff;           //Enable all pins of PORTAD for digital I/O
  DDRAD = 0xff;             //Make PORTAD all digital outputs
  DDRM  = 0x10;             //Make PM4 = output (LED)
                            //and make PM5 = input (SWITCH).
  PERM = 0x00;              //Disable internal pullup on SW input PM5.
  nr_heads = 0;             //Initialize the RAM locations (variables) nr_heads and nr_tails to zero.
  nr_tails = 0;
  PTT = 0;                  //Set all output pins to zero.
  PTAD = 0;
  PTM = 0;
  for(;;)
  {
    coin = 0;               //Variable "coin" indicates outcome
    do
     {
       coin = coin ^ 1;                       //Toggle LSB of coin variable
     } while ((PTM & 0x20) == 0x20);   /      /until SW (on PM5) is pressed.
    delay20ms();                              //Delay 20 ms to debounce SW press.
    while ((PTM & 0x20) == 0);        /       /Wait here until SW released
    delay20ms();                              //Delay 20 ms to debounce SW release.
    if ((coin & 1) == 1)
     {
      nr_heads++;                             //If heads outcome,
      PTM = PTM | 0b00010000;                 //increment nr_heads, PM4 = 1 (LED on)
      PTAD = nr_heads;                        //and update nr_heads on PTT output port.
     }
```



```
    else
     {
      nr_tails++;                             //If tails outcome,
      PTM = PTM & 0b11101111;                 //increment nr_tails, PM4 = 0 (LED off)
      PTT = nr_tails;                         //and update nr_tails on PTAD output port.
     }
  }

}
void delay20ms(void)                          //This routine simply wastes time
{                                             //providing switch debouncing delay.
  unsigned int i;
  for(i=0;i<40000;i++);
}
```

Make a new project and use this file for the main.c function. Verify that the program works by running the program using the in-circuit debugger.

Note that if you choose to single step through this program, you will have to take about 14 initial steps through the startup.c code before you begin stepping through the main.c program. Note that, before you press the pushbutton, the program will soon be hung in the coin toggling inner loop. If you are watching the program variables in the debugger that appear to the left in Data:2 window, you may be surprised to see that the value of the "coin" variable is NOT toggling as it should be. What is going on? The answer can be found if you look at Accumulator B in the register window as you step around this first inner loop. You will see that the LSB of register B is indeed as we expect the coin variable to be doing. The C compiler has "optimized" our code by letting the coin variable be (temporarily) in Accumulator B. This accumulator can be accessed faster than an external memory location (coin). If you set a breakpoint further down in the program, such as at the "***if ((coin & 1) == 1)***" statement, and then reset and run the program down to this breakpoint, you will see that the value of the coin variable has been updated, since Accumulator B is eventually needed for another function. Likewise, you will see that if you step into the delay20ms( ) function, that the local counting integer "i" that has been defined within the delay routine does not appear to be incrementing as expected as you step around the delay loop, but once again, if you look at the register window, you will find that Register X is being used as this integer counting variable.

The lesson to be learned here is to check the register windows (A, B, X or Y) when a variable (memory location) is not changing as expected during single-step debugging.

6 Write a Double-Click/Single-Click Detecting Program.

Once you have thoroughly gone through the previous sections and checked out and fully understood the flashlight and the coin-toss programs, you are now ready to write your own C program.

Your assignment is to design a "***double click***" detecting system that may be used to allow a single pushbutton switch to perform two different control functions, such as on a "mouse" computer peripheral pointing device. You may find the summary of looping and branching constructs in the C language to be helpful, which is included as **Appendix A**.

First interface a second buffered LED to the 9S12C128 module (on PM3), so now your hardware interface consists of a pushbutton switch (on PM5) and two buffered LEDs (on PM4 and PM3).

You must write a C program that operates in the following way: Both LEDs should be initially turned off, with the program waiting for the button to be pushed. If the pushbutton is clicked (by "click", I mean it is pressed *down*, and then *released* in rapid succession, just as you would click a mouse button) and then, within (about) a one-second time interval, it is again rapidly clicked a second time, LED2 (the double-click LED) should be made to light for 1 second and then turn off. If the pushbutton is rapidly clicked (turned ON and then OFF) and this process is NOT repeated within 1 second after the initial click, then LED1 (the single-click LED) should flash on for 1 second and then turn off.

You will have to experiment with the number of delay loop iterations needed to approximate a 1-second delay time. If you declare a counting (index) variable as an unsigned 16-bit integer data type: "***unsigned int***", the variable can range between 0 and 2^{16}-1 = 65535. Counting a variable over this range in a simple for loop, for(i=0; i<65535; i++); delays about 50 ms, when the 9S12C128 is clocked at 2 MHz, as it is on our modules. So counting an ***unsigned int*** variable will not give you enough delay time. If you declare a counting variable as an unsigned 32-bit integer data type: "***unsigned long int***", it will be able to range from 0 up to a value of 2^{32}-1 = 4,294,967,295. Counting a variable up to this highest possible 32-bit integer value results in a delay of approximately one-half hour, so the use of this 32-bit integer data type should be more than sufficient for realizing the 1-second delay needed in this double-click embedded design application.

Demonstrate your working double-click detector system to your lab instructor and obtain his signature on your program listing that you turn in as part of your lab report. Your report (described in the "deliverables" at the beginning of this document) and your demonstration are due at the beginning of next week's lab period.

Appendix A. Summary of Flow Control Statements in a C-Language Program

In the C language, program flow is controlled by means of loops (***while*** loop, ***for*** loop, and ***do-while*** loop), and branching (***if-else*** statement and the ***switch*** statement). The syntax of each of these flow control mechanisms is described in detail below, along with simple illustrative examples of how they are used. In the general syntax descriptions below, the "statement;" could either be a simple single C instruction, or it could be a compound statement that consists of any number of C instructions bracketed by a pair of curly braces.

A. Loops

1. ***while*** Loop

Syntax:

```
        while (expression)
statement;
```

Example 1: Example of while loop with a simple statement

```
while (i++ < 20)
        q = 2 * q;
```

Example 2: Example of a while loop with a compound statement (enclosed by curly braces)

```
while ((PTM & 4)  = =  4)       // Repeatedly execute all of the instructions below
                                // while the switch on input pin PT2 = 1.
    {
        count_var++;
        PTM = PTM & 0b11110111;   //Set output pin PT3 = 0;
        delay20ms( );
    }
```

2. ***for*** Loop

Syntax:

```
        for (initialize; test; update)
                statement;
```

Example:

```
        DDRT = 0;                       /* Make Port T all outputs */
        for (n = 0; n < 10; n++)
```

```
        {
                PTT = n;          /* Make Port T values count in binary from 0 to 9 */
                delay_ms(100);  /* with a 100 ms delay in between counts. */

        }
```

3. ***do - while*** Loop

Syntax:

```
do
        statement;
while (expression);
```

Example (Calculate10 factorial = 10*9*8*7*6*5*4*3*2*1)

```
j = 1;  i = 1;
do
        j = j * i;
while (i++  <=  10);
```

B. Branching

1. ***if - else*** Statement

Syntax of "*if* statement" without "else clause"):

```
if (expression)
        statement;
```

Syntax of "*if* statement" with "else clause":

```
if (expression)
        statement;
else
        statement;
```

Example:

```
if (amt > 400)
        rate = 56;
else
        rate = 62;
```

2. ***switch*** Statement

Syntax:


```
switch (expression)
{
        case label1 : statement(s);
        case label2 : statement(s);
. . . . . .
        default      : statement(s);
}
```

The *expression* must evaluate to an *integer*, and the labels should be integer constants (including *char*) or integer constant expressions.

Example:

```
switch(choice)
        {
                case 'a' : act++;   /*Execute "act++;" if choice = ASCII
                                      code of letter a.*/
                break;              /* this statement "breaks out" of the
                                       present "switch( ) control structure. */
                case 'b' : bct++;
                break;
                case 'c' : cct++;
                default : otherct++;
                break;
        }
```

Note that if we leave out the "break" statements, all of the cases that appear underneath the one that was specified will be executed, which is often not desired.