Arduino is an Italian word that means “valuable friend”.

Arduino is a single-board microcontroller designed to make the process of using electronics in multi-disciplinary projects more accessible. The hardware consists of a simple open source hardware board designed around an 8-bit Atmel AVR microcontroller, though a new model has been designed around a 32-bit Atmel ARM. The software consists of a standard programming language compiler and a boot loader that executes on the microcontroller.

Arduino boards can be purchased pre-assembled or as do-it-yourself kits. Hardware design information is available for those who would like to assemble an Arduino by hand. It was estimated in mid-2011 that over 300,000 official Arduinos had been commercially produced at that point.

The foregoing 3 paragraphs were derived from doing a GOOGLE SEARCH. The second two paragraphs were from Wikipedia. As was mentioned, there is a hardware aspect to Arduino that is based on 8 and 32-bit Atmel processors. More interesting is the SOFTWARE aspect of ARDUINO. There is an extensive library of functions to deal with many complex devices.

On the DE1 board, devices such as the KEYPAD, and LCD are complicated devices that may be somewhat difficult to program for someone taking an introductory MicroController course. To allow people to use these devices, I have adopted the Arduino philosophy of making a functions based library. But, unlike Arduino which uses a specialized SKETCH environment, this library works with codewarrior.

Under the LAB VIDEO section of BlackBoard, is a video called: ARDUINO LIKE FUNCTION LIBRARY. This video shows you how to download and install the library_de1.lib file into Codewarrior as well as how to implement the library_de1.h header file. The video also takes you through several examples of how to use functions such as: lcd_setcursor(), lcd_print(), lcd_write(), hex2dec(), get_int(), delay_milli(), delay_micro(), etc.

The header file library_de1.h is printable and provides description and function prototypes for each of these functions. It is recommended that you print out this header file to see which of these functions you might need, as well as how to properly implement them.

Suppose we want to count continuously on the GREEN LEDS from HEX 00 to FF. However, we want to the count to take only 15 seconds to go from 00 - FF. Since we will be counting through 256 values in 15 seconds, we need to calculate how much time should there be between each count.

I think that you can see that simply putting in for loop to waste time will not work in this instance, since we wish to waste a specific amount of time. The actual time between each count can be calculated by dividing:

$$15 / 256 = 0.05859375 \text{ seconds} \quad \text{or approximately} = 59 \text{ milliseconds}$$

We can use the function delay_milli(59) to waste this amount of time.
Chapter 3 - Using Arduino-Like Functions

Consider the following program that does this:

```c
#include <hidef.h>            /* for EnableInterrupts macro */
#include "derivative.h"       /* include peripheral declarations */
#include "C:\library_del1.h"  // for arduino-like function library

void main(void)
{
    volatile unsigned char count;
    LCD_ROWS=4;
    LCD_COLS=16;
    SOPT_COP = 0;           /* disable COP */
    devices_init();         // initialize all devices
    for(count=0; count <=255; count++)
    {
        LEDG=count;
        delay_milli(59);
    }
}
```

If you run this code you may find that it seems to take a little longer than 15 seconds to count from 00-FF on the Green leds. You might think that executing the for loop as well as assigning count to LEDG takes a little time. The delay_milli() function is based on a hardware timer, and therefore should be reasonably accurate. However other instructions executed between successive calls to this function might also add to the overall delay.

Don’t forget that starting and stopping a STOP watch to determine the accuracy of this time adds human reaction time as well. The 1.5 second splash screen that is displayed on the LCD also makes it hard to determine the accuracy. The devices_init() function initializes all devices then puts up the splash screen. So between hitting the GREEN RUN button and having your code execute, there is an indeterminate amount of time wasted.

To improve your measurement on a STOP watch of the time wasted you can enter this code after the devices_init() function:

```c
while(SW_KEY3 !=0);      // wait for KEY3 pushbutton to be pressed
```

When you test now, you should find that you have very close to 15 seconds being wasted.
Consider the following, where the LCD is also used to count as well as the LEDs.

```c
/*
 Title: Program to illustrate FUNCTION LIBRARY
 Author: Dave Ross 123-456-789 MCO455J
 Date: June 7th, 2013
 Description: This program will illustrate how to use the delay_milli() function.
 */

#include <hidef.h>               /* for EnableInterrupts macro */
#include "derivative.h"          /* include peripheral declarations */
#include "C:\library_del1.h"       // for arduino-like function library

void main(void)
{
    volatile unsigned char count;
    LCD_ROWS=4;
    LCD_COLS=16;
    SOPT_COPE = 0;           /* disable COP */
    devices_init();           // initialize all devices
    lcd_setcursor(0,4);   // LCD CURSOR to (0,4)
    lcd_print("Count");   // print "Count"
    while(SW_KEY3 !=0);       // wait for KEY3 to be pressed
    for(count=0; count <=255; count++)
    {
        LEDG=count;
        lcd_setcursor(1,5);  // LCD CURSOR to (1,5)
        lcd_write(count);  // send count to LCD
        delay_milli(59);
    }
}
```

In this case, only `lcd_setcursor(1,5)` and `lcd_write(count)` have been added inside the loop. However, this has added 3 seconds to the total time wasted. Setting the cursor and writing values to the LCD does take time. If you were to change the time delay to `delay_milli(234)` for a 1 minute duration, the extra time over 1 minute is only 4 seconds.

What this means is that the LCD is not the best device to show things that change rapidly, since it can’t keep up because of internal processing. Even with slower changing events, it will add extra time. However for slower changing events, you can adjust the time delay to take this into account.

For example it takes 64 seconds to display 60 seconds with time delay `delay_milli(234)`. To adjust this:

\[(60/64)* 234 = 219\], therefore `delay_milli(219)` should waste very close to 1 minute when using the LCD.
Chapter 3 - Using Arduino-Like Functions

Consider the following program that is an enhanced version of the one on the previous page. The following program will allow user input to set the timing used to count on the Green Leds. By pressing the KEY3 pushbutton, it will break out of the loop and allow for a different time delay value to be entered. This program also uses the `hex2dec()` function to convert both the count and timing values to BCD for display on the LCD.

```c
#include <hidef.h>                  /* for EnableInterrupts macro */
#include "derivative.h"             /* include peripheral declarations */
#include "C:\library_de1.h"         // for arduino-like function library
// header on root directory of C drive

void main(void)
{
    volatile unsigned char count;
    volatile unsigned int timing,d_timing,d_count;
    LCDROWS=4;        // set LCD ROWS for 4 line display
    LCDCOLS=16;       // set LCD COLS for 16 column display
    SOPT_COPE = 0;                /* disable COP */
    devices_init();                 // initialize all devices
    for(;;)         // OUTER LOOP for INPUT OF TIMING
    {
        lcd_setcursor(0,0);             // set cursor top left of lcd
        lcd_print("Time(ms)?");         // prompt for time in ms
        timing=get_int(0,9,1);          // get time at (0,9) and display value
        if(timing > 500) timing =500;    // accept only values <=500 ms
        lcd_setcursor(1,0);             // set cursor to start of 2nd line
        lcd_print("KEY3 to RESET");     // print RESET message
        lcd_setcursor(2,1);             // set cursor to (2,1)
        lcd_print("Count   DELAY");     // print TITLE
        for(count=0; count <=255; count++)  // INNER LOOP FOR DISPLAY of COUNT
        {
            LEDG=255;                 // send binary count to Green Leds
            d_timing=hex2dec(timing);  // convert timing to BCD
            d_count=hex2dec(count);    // convert count to BCD
            lcd_setcursor(3,1);        // cursor to (3,1)
            lcd_write(d_count>>8);     // display upper digits of BCD count
            lcd_write(d_count&0x00ff); // display lower digits of BCD count
            lcd_setcursor(3,10);       // cursor to (3,10)
            lcd_write(d_timing >> 8);  // display upper digits of BCD timing
            lcd_write(d_timing &0x00ff); // display lower digits of BCD timing
            delay_milli(timing);       // delay for number of ms input
            if(SW_KEY3 ==0) break;     // break if KEY3 pressed - enter new value
        }
    }
}
```

This program can be used to determine the ADJUSTED TIME DELAY values that will make the LEDS and LCD count from 00-FF within 15, 20, 25, 30 ... seconds. To make these adjustments you will need a STOP WATCH or a digital watch. You can adjust the time delay by following the procedure on the previous page.
The LCD is not always the best device for displaying changing values. The reason for this is that the LCD takes time to display individual characters and even more time to reposition the cursor. One thing the LCD is good at is to display menus of choices. Consider the following program that uses a menu to select various built-in functions to execute:

```
#include <hdef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#include "c:\library_de1.h"

void main(void)
{
    volatile unsigned char bob;  // variable for menu choice
    SOPT_COPE = 0;  /* disable COP */
    LCD_ROWS=4;     // 4 line LCD
    LCD_COLS=16;    // 16 character LCD
    devices_init(); // initialize all devices
    for(;;)      // loop forever
    {
        do      // display menu
        {
            lcd_setcursor(0,0);
            lcd_print("A - Diagnostics ");
            lcd_setcursor(1,0);
            lcd_print("B - Int Convert ");
            lcd_setcursor(2,0);
            lcd_print("C- Varying Count");
            lcd_setcursor(3,0);
            lcd_print("Your choice? ");
            bob=get_char(3,13);    // bob gets char from keypad
        }
        while((bob < 'A') || (bob > 'C')); // if not A,B, or C keep looping
        switch(bob)       // switch based on bob
        {
        case 'A':       diag();   // if A then diag()
                            break;
        case 'B':       int_conver();  // if B then conver()
                    break;
        default:       variable_count(); // if C then variable_count
        }
    }
}
```

The `diag()` function does a complete diagnostic on the DE1 board. The `int_conver()` function, converts numbers entered on the keypad from 0 - 65535 to HEX, OCTAL, and Binary. Both of these functions can be terminated by pressing the # key on the keypad. The `variable_count()` function allows you to count on the LEDs and 7 Segment displays. You can decrease the counting rate by pushing KEY3, increase the counting rate by pushing KEY2, and exit the function by pressing KEY1.
As you start to develop more and more complicated control code in C, you will need a way to STRUCTURE your code. In the past, people have used FLOW CHARTS. However the problem with Flow Charts is that they are not suitable to a structured programming language such as C.

The best way to organize logic for C programs is using STRUCTURED DIAGRAMS. Structured Diagrams are a simplified version of NASSI SCHNEIDERMAN Diagrams. Below is the STRUCTURED DIAGRAM for the program of the previous page:

Consult the following pages for the meaning of these symbols.
A **STRUCTURED DIAGRAM** can be thought of as a **compressed form** of **FLOW CHART**. It uses **symbols** somewhat **similar** to those found in a **FLOW CHART**. **STRUCTURED DIAGRAMS** are the best way to illustrate **PROGRAM flow** for **STRUCTURED** Programming Languages like C, C++, C#, Java, Pascal, etc.

```c
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

main()
{
    int A, B, C;
    
    A = 5;
    B = 7;
    C = A + B;

    printf("\nA=%3i, B=%3i, C=%3i", A, B, C);

}
```

In the **STRUCTURED DIAGRAM** version, A gets 5, B gets 7, C gets A+B, then A, B, and C are output to the screen. As you will note, statements in the above **STRUCTURED DIAGRAM** get done in order (Sequentially) until the program **ENDS** when it hits the bottom of the last **RECTANGULAR BOX**.

**#includes**, Function Prototypes/Definitions and Variable Declarations are details that are **NOT** found in **STRUCTURED DIAGRAM**.

**Variable assignments** and other **simple statements** like **OUTPUT STATEMENTS** (printf’s) show up as simple **RECTANGULAR BOXES** in a **STRUCTURED DIAGRAM**. Consider the following program and equivalent **STRUCTURED DIAGRAM**.

For most statements, Structured Diagrams use **RECTANGLE shapes**. The **UPPER LEFT** edge of the **RECTANGLE shape** is equivalent to the **START** of a block of code - indicated by the **{**. The **BOTTOM LEFT** corner of the **RECTANGLE** indicates the **END** of the **BLOCK** of code - indicated by the **}**.

© Copyright David Elmo Ross 2013
Consider the following program:

```c
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

main()
{
    int a,b;

    printf("Number to square ==> ");
    scanf("%d", &b);
    a = b * b;
    printf("a=%d, b=%d", a, b);
    getch();
}
```

The parallelogram symbol is used to denote input from the keyboard. In this case, variable \( b \) is defined by what is typed in from the keyboard.

Then the value typed in is squared and both variables are output to the screen.

At this point, Structured Diagrams might seem to be a waste of time, and if you were only going to do sequential processing of instructions you would be right.

Besides sequential processing of instructions, there are two other types of structures:

- **Selection** - where you select a piece of code to be executed only if a certain condition(s) is met

- **Looping** - where you repeatedly execute a block of code while a condition exists

In a structured diagram, a half diamond shaped symbol is used to check to see if a condition exists that will cause non-sequential program flow.

Program flow enters the top of the box, where there are but two choices:

- **If** the condition is **true**, then follow the Yes.

- **If** the condition is **not true**, then follow the No.

There are a number of mathematical conditions that can be checked for in a program. These are listed on the next page.
<table>
<thead>
<tr>
<th>Mathematical Condition</th>
<th>Description</th>
<th>C SYMBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>is less than</td>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>is greater than</td>
<td>&gt;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>is less than or equal to</td>
<td>&lt;=</td>
</tr>
<tr>
<td>&gt;=</td>
<td>is greater than or equal to</td>
<td>&gt;=</td>
</tr>
<tr>
<td>==</td>
<td>is equal to</td>
<td>==</td>
</tr>
<tr>
<td>not ==</td>
<td>is NOT equal to</td>
<td>!=</td>
</tr>
</tbody>
</table>

Of all the **C SYMBOLS** that describe these Mathematical Conditions, the one that is the most confusing to deal with is the `==` or **EQUAL TO** condition.

Consider the following two statements and their meaning in a C Program.

```c
a = 4;    // variable a is **ASSIGNED** the value 4
a == 4;   // checks to see if variable a **is EQUAL TO** 4
```

Mathematical Conditions in C, like `a == 4` will yield a result of:

- 1 (if the condition is **true**)
- 0 (if the condition is **false**)

If the **CONDITION** is **TRUE**, the **BLOCK** of Code in the **YES BLOCK** is executed. If the **CONDITION** is **NOT TRUE**, the BLOCK of Code in the **NO BLOCK** is executed.
First let’s consider the SELECTION STRUCTURE as illustrated in the following program and STRUCTURED DIAGRAM.

```c
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

int a, b;
system("cls");
a = 5;
b = 7;

if (a < b)
{
    printf("\n a < b\n");
}
if (a == 7)
{
    printf("\n a == 7\n");
}
getch();
```

In the above example, there are 2 choices for program flow illustrated by the two half diamond shapes in the Structured Diagram. For the first decision, I think that you can see that \( a < b \) is a TRUE statement, therefore the \( \text{printf("a < b")} \); statement will be executed. In other words it will follow the YES path in the Structured Diagram. Control will then pass to the NEXT decision \( a == 7 \).

In the second decision (half DIAMOND shape), the variable \( a \) is NOT equal to 7, and therefore the program FLOW follows the NO path, goes to End of the BLOCK of CODE, and does NOT print out anything else on the screen.

Actually the \text{getch()}\ instruction is executed before the program terminates. But as you will recall, the \text{getch()}\ instruction is there just to hold the screen open so we can see the output. Because it is not essential to understanding program FLOW, it doesn’t need to be included in the Structured Diagram.

Variable declarations, \#includes, \#defines, and other such declarations are NOT included in STRUCTURED DIAGRAMS since they do no affect program flow. We have also not included the \text{system("cls")}; instruction either.
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

main()
{
    int a,b;

    system("cls");
    a = 5 ;
    b = 7 ;

    if ( a < b )
    {
        printf("\na < \ b\n");
    }

    else
    {
        if ( a > b )
            { 
                printf("\na > \ b\n");
            }
    
        else
            { 
                printf("\na = = \ b \n");
            }

    getch();
}

In the above example, there are ONLY three cases: a < b , a > b , or a = = b.

Only two of these cases need to be tested, the third case becomes the DEFAULT case. In other words, if a < b is NOT true, and a > b is NOT true, then a = = b must be true.

Notice that when using SELECTION ( if , or if..else ) there is ONLY 1 PATH that will be followed. In the above case, ONLY the a < b message will be output to the screen.

If we were to get the values for variable a , and variable b , from the keyboard, then I think you can see that the output would depend on the values entered.
Example of while and for .. loops

```c
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

main()
{
    int a;
    system("cls");
    a = 4;
    while (a<=7)
    {
        printf("a=%4i\n",a);
        a++;
    }
    getch();
}
```

```c
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

main()
{
    int a;
    system("cls");
    for (a=4;a<=7;a++)
    {
        printf("a=%4i\n",a);
    }
    getch();
}
```

In the top program, we have a while loop. While the condition a <= 7 is true, the statements within the braces get executed. The STRUCTURED DIAGRAM shows the LOGIC FLOW of the while loop. The statements in the BODY of code (contained in braces), will be done WHILE the condition is TRUE.

When the LOOP CONDITION is NOT true, the LOOP terminates, and the statement after the loop is executed, in this case the getch(). You will note that the LOOP CONDITION is tested at the TOP of the LOOP.

The for loop (in the second program above) is a compressed form of the while loop, and does exactly what the while loop does in the first example. However, it is not as obvious in the for loop, that the loop parameter a is incremented at the bottom of the for loop.
Example of do..while loop

```c
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

main()
{
    int a;
    system("cls");
    a = 4;
    do
    {
        printf("a=%4i
",a);
        a++;
    } while (a<=7);
    getch();
}
```

For the do..while loop, you test the LOOP CONDITION at the bottom of the loop. What this means is that you will always execute the BLOCK of code at least once, regardless of the condition you are checking for.

The while and for loops will not execute any of the instructions in the BLOCK of code if the condition is NOT true.

With the while, for, and do..while loops, it is possible to have a LOOP whose LOOP CONDITION is ALWAYS TRUE. This is called an INFINITE LOOP, because there is no way to end the loop. The block of code in this case is executed over and over for all time. The only way to terminate such an INFINITE LOOP is to hold down the <CTRL> key and hit C. Another option is to CLOSE the DOS output window.

The only remaining thing to discuss is FUNCTIONS. When you call up a function, it is a rectangle that has an extra vertical line at each end of the RECTANGLE. The only other difference is the return() statement. Functions can contain if, if..else, while, for, and do..while loops.

C Program statements such as scanf(), printf(), and system(), only apply to MicroComputers (Home Computers) not MicroControllers. When we cover the SCI (Serial Communications Interface), we will use our own functions (NOT printf, scanf) to deal with the KEYBOARD and SCREEN.
Consider the following program that \textbf{SCROLLS} a message \textbf{across the four 7 Segment displays:}

```
#include <hidel.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#include "c:\library_de1.h"

void main(void)
{
  volatile unsigned char x;
  unsigned char mess[] =
  {
    0xff,0xff,0xff,0xff,   // first 4 values clear the four 7 segment displays
    0xe0,0xf4,0x88,0xff,   // trY
    0x10,0x40,0x10,0x9e,0x12,  // AGAIN
    0xff,0xff,0xff,0xff   // blanks four 7 segment displays
  };  // message “tryAGAIN”

  SOPT_COPE = 0;   /* disable COP */
  LCD_ROWS=4;   // 4 line by 16 character LCD display
  LCD_COLS=16;
  devices_init();    // initialize all devices
  for(;;)     // forever
  {
    for(x=0;x<=13;++x)   // element 0 to element 13 of array mess
    {
      HEX3=mess[x];   // four 7 segments get first 4 values
      HEX2=mess[x+1];   // or blanks
      HEX1=mess[x+2];   // each time through the loop, it gets the next
      HEX0=mess[x+3];   // 4 values till it displays the last 4 blanks
      delay_milli(1000 );   // wait 1000 ms or 1 second
    }
  }
}
```

As you can see, since this program uses the “\texttt{c:\library_de1.h}” include, you must also have the library file \texttt{library_del.lib} added to the \texttt{LIBS} section of Codewarrior as shown in the video (\textbf{ARDUINO LIKE FUNCTIONS}).

\textbf{Scrolling messages} like this can be found on some \textbf{POP MACHINES} or \textbf{PUBLIC PHONES}. 
Here is a list of the most commonly used Arduino-like functions and brief descriptions. For a more detailed description, look at the header file library_de1.h:

- **devices_init()** - initializes all devices (timers, lcd, keypad, A/D, D/A, etc. then puts message on LCD for 1.5 seconds then clears display.

- **delay_milli()** - takes a *long* as input. For example: `delay_milli(1000);` will waste 1 second.

- **delay_micro()** - takes an *int* as input. For example: `delay_micro(10000);` will waste 10 ms.

- **gled_gauge()** - takes a *char* as input and will act as a fuel gauge on the Green Leds.

- **rled_gauge()** - takes a *char* as input and will act as a fuel gauge on the 8 right most Red Leds.

- **analog_read()** - takes a 0 or 1 as its input to either select TP7 (0-2.5V input) or TP8 (-5 to +5V input). It returns a *char*. For example:
  - `data=analog_read(0);` will convert the voltage on TP7 (0-2.5V) into a value from 00-FF and put it into *char* variable data.
  - `data=analog_read(1);` will convert the voltage on TP8 (-5 to +5V) into a value from 00-FF and put it into *char* variable data.

- **analog_read_int()** - takes a 0 or 1 as its input to either select TP7 (0-2.5V input) or TP8 (-5 to +5V input). It returns a 12-bit *int*. (12-bit A/D conversion 0x0000-0x0FFF)

- **analog_write()** - takes a 0 or 1 as its input to either select TP9 (0-2.5V output) or TP10 (-5 to +5V output). It also takes a *char* value to be converted to the appropriate voltage:
  - For example:
    - `analog_write(0, data);` will take the *char* value in data, convert it to a voltage from 0V to 2.5V and send it out on TP9.
    - `analog_write(1, data);` will take the *char* value in data, convert it to a voltage from -5V to +5V and send it out on TP10.

- **analog_write_int()** - takes a 0 or 1 as its input to either select TP9 (0-2.5V output) or TP10 (-5 to +5V output). It also takes a *int* value(0-0xFFFF) to be converted to the appropriate voltage:

- **scaler()** - scales an 8 bit A/D value (0-0xFF), into a meaningful range (e.g. temperature, voltage etc) and displays it on the LCD. It returns a *long*. For example:
  - `bob = scaler(analog_read_int(0) >> 4, 10000, 4, 3, 1, 5);` will take a 12 bit reading from TP7, shift it 4 times to the right to get a *char* value as the first parameter of the function, uses *int* 10000 to scale to 100 degrees as second parameter, uses 4 digits accuracy as third parameter, places decimal point after the digit 3 as fourth parameter, and last two parameters are where the value is placed on the LCD.
  - `bob = scaler(analog_read_int(0) >> 4, 25000, 4, 1, 2, 6);` to display voltage from 0-2.5V on LCD at (2,6) with 1 digit before the decimal point and 4 digit accuracy.
**lcd_setcursor()** - used to position the cursor on the LCD to either display messages or data.

For example:

```cpp
lcd_setcursor(0, 5);  // will position the cursor on row 0 of 4 rows (0-3) and on column 5 of 16 columns (0-15).
```

**lcd_print()** - used to send text messages to the LCD display.

For example:

```cpp
lcd_print("Hi there");  // will print the message Hi there on the LCD
```

**lcd_write()** - used to send char data values from 00-FF to LCD display.

For example:

```cpp
char abc=0x4A;
lcd_write(abc);  // sends 4A to LCD
```

**lcd_writech()** - used to send ASCII characters to LCD.

For example:

```cpp
calc_writech(0xdf);  // sends degree symbol to LCD - refer to OPTREX lcd manual for special characters such as degree symbol

calc_writech('C');  // sends character C to LCD
```

**segdis()** - takes an int input and displays this int on the four 7 segment displays. If the 2 MSB HEX digits are 00, the left two 7 segment displays will be blank.

For example:

```cpp
segdis(65535);  // will display FFFF on the 7 segment displays.
```

**clrseg()** - will clear all four 7 segment displays. It has no input or output parameters.

**get_char()** - will wait until a key on the keypad is pressed then it returns it as a char.

**get_dec()** - will only accept digits 0-9 as input. It will accept 0-255 only. The # key will terminate input if there are less than 3 digits typed in. You need to specify where on the LCD you will prompt for this input. You can either display what is being typed or simply echo an * for each digit entered.

For example:

```cpp
sam=get_dec(2,3,1);  // Positions the cursor at row 2 col 3 of the LCD and waits for you to enter any value from 0-255. If it is greater than 255 it returns a 0.

The last parameter 1, indicates that it will display each digit as you type it. A 0 instead would echo a * for each digit typed.
```

**get_int()** - similar to get_dec(), except it accepts inputs from 0-65535, and returns an int.

For example:

```cpp
bob=get_int(2,3,1);
```

**hex2dec()** - takes an int as input and returns a long. It converts values from 0-FFFF to BCD values 0-65535.

```cpp
sam=hex2dec(0xFFFF);  // would return 0x65535
```
The following functions deal with the **PC KEYBOARD** and **SCREEN**. This connection is made by using an **RS-232C cable** connected to the **DE1** interface board. **HYPERTERMINAL** is a terminal emulation program that provides a **24 row by 80 column** screen to all user input and output.

If you use the **default baud rate** of **9600** in hyperterminal, and you set the **EMULATION** to **ANSI**, the following functions will work properly:

- **sci_init();** - initializes the **SCI** to **9600 baud**, and enables the **Transmitter** and **Receiver**.
- **scr_clear();** - clears the **HYPERTERMINAL** screen.
- **scr_setcursor();** - assuming you have **ANSI** emulation set up in Hyperterminal, this function will position the cursor.
  For example:
  ```c
  scr_setcursor(12,40); // will position the cursor in the middle of the screen
  ```
- **scr_print();** - will print a text message on the screen at the current cursor position.
  For example:
  ```c
  scr_print(“Hi there World!”);
  ```
- **scr_write();** - will print the equivalent hex value as two ascii characters on the screen.
  For example:
  ```c
  scr_write(SWL); // will print the current switch position on the screen
  ```
- **kb_getchar();** - if no key was pressed when this function was called it returns a 0x00 or NULL.
  For example:
  ```c
  while((bob=kb_getchar())== ‘\0’); // when this loop exits, bob has a character.
  ```

Last but not least there are **3 demonstration functions** that take no input and generate no output. These are:

- **diag();** - this function check out practically everything except A/D and D/A.
  Hitting the # key exits this demo.
- **int_conver();** - this function takes **BCD input** from **0-65535** and converts it to **HEX, OCTAL** and **Binary**.
  Hitting the # key exits this demo.
- **variable_count();** - this function will **count up** on the **LEDS** and **7 Segment displays**. The **LCD** shows the **current delay** in **milliseconds**. Pushing **KEY3** will **decrease** the **time delay**, while pushing **KEY2** will **increase** the **time delay**.
  Pressing the **KEY1** pushbutton will **exit** this demo.