

# EEE174 –CPE185 INTRODUCTION TO MICROPROCESSORS

## LAB 2 - PROPELLOR

**Lab Session: Wednesday 6:30PM - 9:10PM**

**Section 32385**

**Lab Instructor: Sean Kennedy**

**Student Name: Andrew Robertson**

## TABLE OF CONTENTS

Part 1 .....	3
Overview.....	3
Lab Discussion .....	4
Work Performed / Solution: .....	4
Part 2 .....	22
Overview.....	22
Lab Discussion .....	23
Work Performed / Solution: .....	23
Listing Files(s):.....	<b>Error! Bookmark not defined.</b>
Part 3 .....	37
Overview.....	37
Lab Discussion .....	38
Work Performed / Solution: .....	38
Part 4 .....	46
Overview.....	46
Lab Discussion .....	47
Work Performed / Solution: .....	47
Listing Files(s):.....	48

# PART 1

## OVERVIEW

Part 1 is a giant C programming refresher with some circuit basics tied in near the end. This section mainly takes concepts that are already learned and shows you their extremely C-like representations. Since my first language coming in to programming was C++, this was very intuitive to me.

## LAB DISCUSSION

### WORK PERFORMED / SOLUTION:

After installing SimpleIDE I then restarted my computer as per installation instructions and updated my leard folder as recommended through a zipside file since I have never seen that extension before and was interested. Now time to make my way through the 13 introduction pages.

#### Simple Hello Message-

Below is the code as provided, it's function is to display Hello on the console

```
#include "simpletools.h" // Include simpletools header

int main() // main function
{
    print("Hello!!!"); // Display a message
}
```

After uploading the project to the propeller board with the terminal window option I see this:

```
Hello!!!|
```

This is very simple but confirms the base code works, the board can do some simple processing, and the serial communication works. After this I was then instructed to modify the program to display a second line. Adding another print() command with an immediate string yields a new result:

```
Hello!!!
Hello again!!!
```

As a test, the walkthrough now prompts us to guess what the “\n” character does at the end of these strings. I already knew this was the newline character but just for completeness I removed it from the first print() function and saw both immediate strings output to the same line:

```
Hello!!!Hello again!!!
```

This is the code post modification:

```
#include "simpletools.h" // Include simpletools header

int main() // main function
{
    /*
    block comment
    */

    //single line comment
    |
    print("Hello!!!"); // Display a message
    print("Hello again!!!\n");
}
```

## Variables and Math-

Below is the base code provided for the variables and calculations exercise:

```
#include "simpletools.h" // Include simpletools

int main() // main function
{
    int a = 25; // Initialize a variable to 25
    int b = 17; // Initialize b variable to 17
    int c = a + b; // Initialize c variable to a + b
    print("c = %d ", c); // Display decimal value of c
}
```

In the walkthrough we are asked to examine the code and attempt to predict what its outcome will be. I believe this will display "c = 42", and I think this is the case because print() looks to be formatted like C's printf() function. Here is the console result as expected:

```
c = 42 |
```

After this, we are instructed to modify the previous code to add two new lines of output representing a different mathematical operation. Copying and pasting the two print() lines, changing their contents and performing a new operation on the c variable gave:

```
a = 25, b = 17
a + b = 42
a = 25, b = 17
a - b = 8
```

Last, we were then instructed to expand this idea even further to cover the basic mathematic operations. This only required more copying, pasting and modifying of the print contents as well as the operation done to the c variable. The resulting code and output are shown below:

```
int main() // main function
{
    int a = 25; // Initialize a variable to 25
    int b = 17; // Initialize b variable to 17
    int c = a + b; // Initialize c variable to a + b
    print("a = %d, b = %d\n", a, b);
    print("a + b = %d\n", c);
    c = a - b;
    print("a = %d, b = %d\n", a, b);
    print("a - b = %d\n", c);
    c = a / b;
    print("a = %d, b = %d\n", a, b);
    print("a / b = %d\n", c);
    c = a * b;
    print("a = %d, b = %d\n", a, b);
    print("a * b = %d\n", c);
    c = a % b;
    print("a = %d, b = %d\n", a, b);
    print("a modulus b = %d\n", c);
}
```

SimpleIDE Terminal

```
a = 25, b = 17
a + b = 42
a = 25, b = 17
a - b = 8
a = 25, b = 17
a / b = 1
a = 25, b = 17
a * b = 425
a = 25, b = 17
a modulus b = 8
```

## Floating Point Math-

The provided base code for this example is as follows:

```
#include "simpletools.h"           // Include simpletools

int main()                       // main function
{
    float r = 1.0;               // Set radius to 1.0
    float c = 2.0 * PI * r;      // Calculate circumference
    print("circumference = %f\n", c); // Display circumference
}
```

I first performed the calculation  $2 * \pi * 1$  on my calculator and then ran the program to compare results:

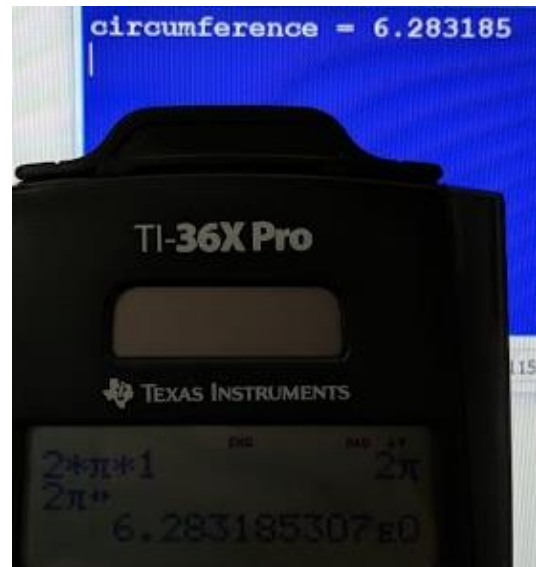
This was as expected, however with a lesser number of significant digits. I believe there is another flag that can be passed to the print function to change the number of floating point numbers or significant digits to be displayed but we did not go in to that here. Instead we were tasked with modifying the program to also display the area of a circle using the defined radius. To accomplish this I created a new variable a to contain the area, computed and stored the result, then displayed it:

We were then tasked with changing the radius to 3 to see the new results, I decided to make this program also display the radius so it's known to the user and verifiable.

Last, we repeat this process to calculate the volume. Since I thought it was a bit redundant to type  $R * R * R$  for radius cubed I wanted to see if the math library was the same as C and included the math.h library for access to the pow() function.

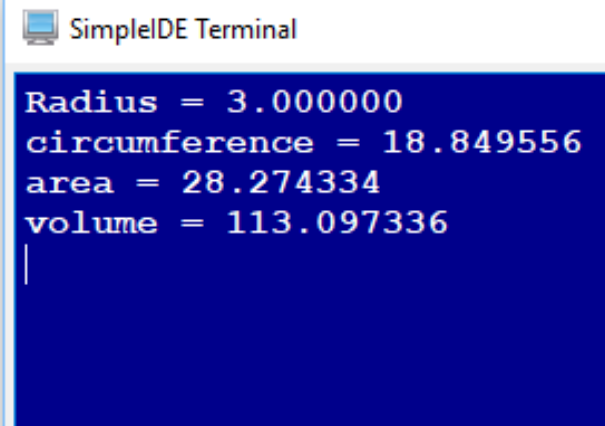
```
#include "simpletools.h"           // Include simpletools
#include "math.h"

int main()
{
    float r = 3.0;
    print("Radius = %f\n", r);
    float c = 2.0 * PI * r;
    print("circumference = %f\n", c);
    float a = PI * r * r;
    print("area = %f\n", a);
    float v = (4.0/3.0) * PI * pow(r, 3);
    print("volume = %f\n", v);
}
```



```
circumference = 6.283185
area = 3.141593
```

```
Radius = 3.000000
circumference = 18.849556
area = 28.274334
```



## Array Variables-

Below is the code we are given to start and to the right, its output:

```
/*
Array Variables.c

Declare and initialize an array and display a couple of its elements.
*/

#include "simpletools.h"          // Include simpletools

int main()                      // main function
{
    int p[] = {1, 2, 3, 5, 7, 11}; // Initialize the array
    print("p[0] = %d\n", p[0]);    // Display what p[0] stores
    print("p[3] = %d\n", p[3]);    // Display what p[3] stores
}
```

```
p[0] = 1
p[3] = 5
```

This exercise is to teach the basics of arrays.

Above, we can see that an array of integers has been created without a size specification, this leaves the compiler to determine the size based upon the initialization list. This first bit was just to show that individual components of the array can be displayed by using indexing. The next part is to prove individual content can be both displayed and modified through indexing:

```
#include "simpletools.h"

int main()
{
    int p[] = {1, 2, 3, 5, 7, 11};
    print("p[0] = %d\n", p[0]);
    print("p[3] = %d\n", p[3]);
    p[3] = 101;
    print("p[3] = %d\n", p[3]);
}
```

SimpleIDE Terminal

```
p[0] = 1
p[3] = 5
p[3] = 101
|
```

Last, for a bit of fun, we modify all the indexed values and display their contents:

```
int main()                      // main fu
{
    int p[] = {1, 2, 3, 5, 7, 11}; // Initial
    print("p[0] = %d\n", p[0]);
    print("p[3] = %d\n", p[3]);
    p[3] = 101;
    print("p[3] = %d\n", p[3]);
    p[0] = 11;
    p[1] = 7;
    p[2] = 5;
    p[3] = 3;
    p[4] = 2;
    p[5] = 1;
    print("p[0] = %d\n", p[0]);
    print("p[1] = %d\n", p[1]);
    print("p[2] = %d\n", p[2]);
    print("p[3] = %d\n", p[3]);
    print("p[4] = %d\n", p[4]);
    print("p[5] = %d\n", p[5]);
}
```

SimpleIDE Terminal

```
p[0] = 1
p[3] = 5
p[3] = 101
p[0] = 11
p[1] = 7
p[2] = 5
p[3] = 3
p[4] = 2
p[5] = 1
|
```

Clear Options

## Make a Decision-

Below is the code we are given to start and to the right, its output:

```
/*
Make a Decision.c

If a condition is true, display a second message.
*/

#include "simpletools.h"          // Include simpletools

int main()                      // main function
{
    int a = 25;                  // Initialize a variable to 25
    int b = 17;                  // Initialize b variable to 17
    print("a = %d, b = %d\n", a, b); // Print all
    if(a > b)                    // If a > b condition is true
    {
        print("a is larger \n"); // ...then print this message
    }
}
```

```
a = 25, b = 17
a is larger
|
```

The conditional statement above is satisfied when a is greater than b. When b and a have reversed values then the message is not displayed, as can be seen to the right:

```
a = 17, b = 25
|
```

Checking this condition once again with negative values yields the expected result, since -17 is greater than -25 the message is again not displayed.

```
a = -25, b = -17
```

The final task of this section is to expand the code to cover the bases of equal to, not equal to, greater than, greater than or equal to, less than, and less than or equal to. The new code and result are as follows:

```
int main()                      // main function
{
    int a = -25;                 // Initialize a variable to 25
    int b = -17;                 // Initialize b variable to 17
    print("a = %d, b = %d\n", a, b); // Print all
    if(a > b)                    // If a > b condition is true
    {
        print("a is larger than b \n");
    }
    if(a == b)                  // If a == b condition is true
    {
        print("a and b are equal \n");
    }
    if(a != b)                  // If a != b condition is true
    {
        print("a and b are not equal \n");
    }
    if(a >= b)                   // If a >= b condition is true
    {
        print("a is greater than or equal to b \n");
    }
    if(a < b)                    // If a < b condition is true
    {
        print("a is less than b \n");
    }
    if(a <= b)                   // If a <= b condition is true
    {
        print("a is less than or equal to b \n");
    }
}
```

SimpleIDE Terminal

```
a = -25, b = -17
a and b are not equal
a is less than b
a is less than or equal to b
|
```

Clear

Options

Disable

115200 ▼

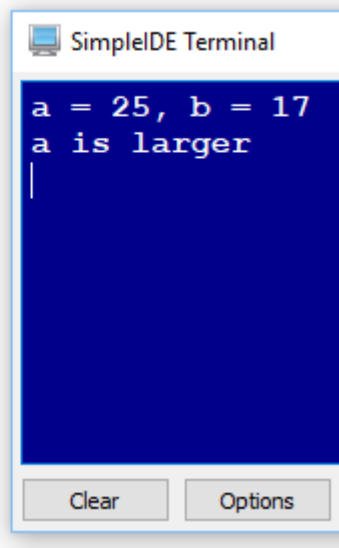
COM



## Make Several Decisions-

Below is the code we are given to start and to the right, its output:

```
// main func
int main()
{
    int a = 25;
    int b = 17;
    print("a = %d, b = %d\n", a, b);
    if(a > b)
    {
        print("a is larger \n");
    }
    else if (a < b)
    {
        print("b is larger \n");
    }
    else
    {
        print("a equals b");
    }
}
```



We now compare the same two values as the last segment, but through the if, if else, and else statement we can now cover multiple bases with less comparisons. If a number is neither greater nor less than another, they must be equal. To test all the current statement, I altered the values of a and b to meet their requirements. First, A = B for the else statement, then A < B for the else if statement.

```
a = 25, b = 25
a equals b
```

```
a = 10, b = 17
b is larger
```

We were then instructed to add another condition, to cover a special case when b == 1000. The placement of this case is important as no other cases after the one satisfied is examined, this is the short circuit feature. To prove this I used the same statement 2 times. First, with the special case before testing if A is less than B – and second, after the comparison.

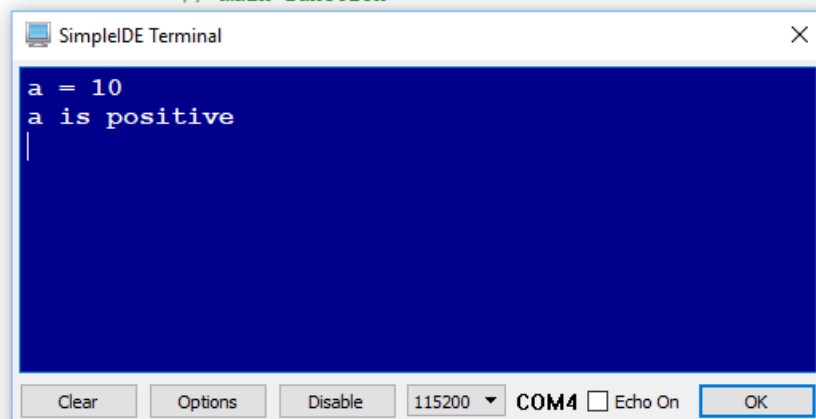
```
a = 10, b = 1000
WARNING, b is 1000
```

```
a = 10, b = 1000
b is larger
```

Last, was to modify the code to show when A is positive, negative, or zero. I simply made B zero and changed the strings that were output to the console window.

```
#include "simpletools.h" // Include simpletools

// main function
int main()
{
    int a = 10;
    int b = 0;
    print("a = %d\n", a, b);
    if(a > b)
    {
        print("a is positive \n");
    }
    else if (a < b)
    {
        print("a is negative \n");
    }
    else
    {
        print("a is zero");
    }
}
```

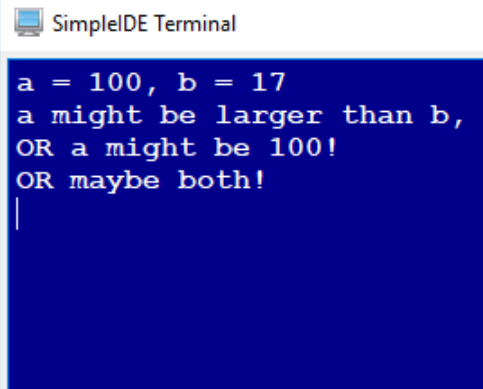


## Make Complicated Decisions-

Below is the code we are given to start and to the right, its output:

```
#include "simpletools.h"

int main()
{
    int a = 100;
    int b = 17;
    print("a = %d, b = %d\n", a, b);
    if((a > b) || (a == 100))
    {
        print("a might be larger than b,\n");
        print("OR a might be 100!\n");
        print("OR maybe both!\n");
    }
}
```



```
SimpleIDE Terminal

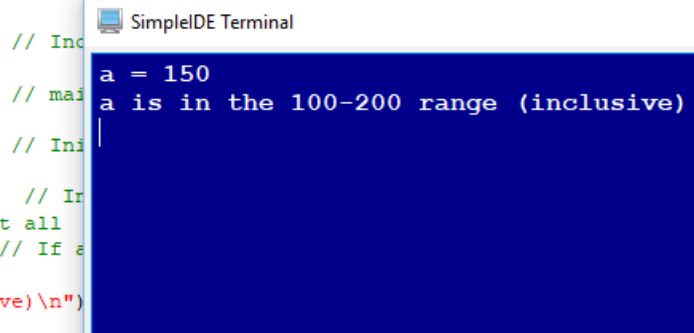
a = 100, b = 17
a might be larger than b,
OR a might be 100!
OR maybe both!
|
```

We are now shown that if statements do not need to be single comparisons. The and (&&) and or (||) logical operators plus some parenthesis for forced order of operations can make way for some complex cases to be tested. In the first example, the walkthrough uses the logical or operator to evaluate if a is greater than b, or if a equals 100, to execute the braced statements.

Our job is to use this new information to display a message when a is in the range of 100-200. Since they did not state inclusiveness I decide to make it inclusive. I created a new variable c so that variables b and c together define the range we want to check for. If a is both greater than or equal to b and less than or equal to c, then we have  $b \leq a \leq c$ . Since b was 100 and c was 200 this means a is in the 100-200 range, inclusively.

```
*/
#include "simpletools.h"

int main()
{
    int a = 150;
    int b = 100;
    int c = 200;
    print("a = %d\n", a, b);
    if((a >= b) && (a <= c))
    {
        print("a is in the 100-200 range (inclusive)\n");
    }
}
```



```
SimpleIDE Terminal

a = 150
a is in the 100-200 range (inclusive)
|
```

## Code That Repeats-

Below is the code we are given to start and to the right, its output:

```
#include "simpletools.h"

int main()
{
    int n = 0;
    while(n < 200)
    {
        print("n = %d\n", n);
        n = n + 5;
        pause(500);
    }
    print("All done!");
}
```

```
n = 155
n = 160
n = 165
n = 170
n = 175
n = 180
n = 185
n = 190
n = 195
All done!
```

An introduction to loops. The while loop executes the braced statements *while* the condition in its argument list is true. Here, we simply count by fives until  $n = 205$  and breaks out of the loop without executing the statements for that case. Showing us one way we can have an infinite loop we are instructed to evaluate `while(1)` as can be seen to the right. Another infinite loop case can occur if a variable is checked for a condition which is initially true but never updated to another value within the braces.

```
1 int main()
2 {
3     int n = 0;
4     while(1)
5     {
6         print("n = %d\n", n);
7         n = n + 5;
8         pause(500);
9     }
10    print("All done!");
11 }
```

```
n = 180
n = 185
n = 190
n = 195
n = 200
n = 205
n = 210
n = 215
n = 220
```

Last, we are shown the `break` statement, which I have been told is bad practice but nevertheless we must still be familiar with it in case it's ever seen. When the `break` is executed, we exit the current level of control so in the case below, we return to the `main()` function and print all done.

```
#include "simpletools.h"

int main()
{
    int n = 0;
    while(1)
    {
        print("n = %d\n", n);
        n = n + 5;
        pause(500);
        if (n == 50) break;
    }
    print("All done!");
}
```

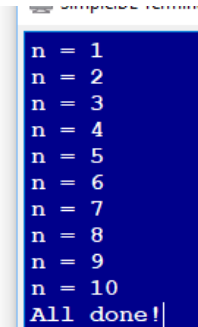
```
n = 5
n = 10
n = 15
n = 20
n = 25
n = 30
n = 35
n = 40
n = 45
All done!
```

## Counting Loops-

Below is the code we are given to start and to the right, its output:

```
#include "simpletools.h"

int main()
{
    for(int n = 1; n <= 10; n++)
    {
        print("n = %d\n", n);
        pause(500);
    }
    print("All done!");
}
```

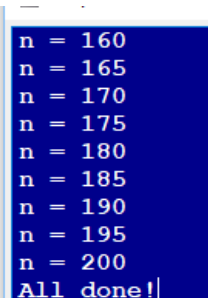


We are now shown a more specific loop, the for loop. Although anything that needs to be done in a for loop can be accomplished using a while loop, the format for a task so widely used as counting or iterating needed to be more compact. The for loop has three main sections in its argument list. The first is the scope variable, this can be assigned and instantiated here as well. The second is the condition to be tested exactly like a while loop. The third, is the task to be completed that modified the scope variable after every loop completion.

We are told to modify the count from 1-10 to 0-200 counting by 5 instead of 1. N's initial value is changed, the condition of the loops execution changed and the incrementor to be changed. I was curious to see if the extended operators worked here so I tried `n+=5` to take the place of `n = n+5` and it worked fine.

```
#include "simpletools.h"

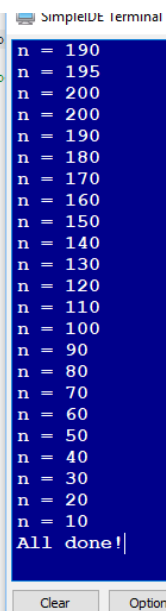
int main()
{
    for(int n = 0; n <= 200; n+=5)
    {
        print("n = %d\n", n);
        pause(500);
    }
    print("All done!");
}
```



Last, we are instructed to create a for loop of our own to count back down to zero by tens. At this point I became somewhat impatient with the long delays between evaluations so I changed the delay time from 500 milliseconds to 50.

```
Version 0.94 for use with Simp
http://learn.parallax.com/prop
*/
#include "simpletools.h"

int main()
{
    int n = 0;
    for(n; n <= 200; n+=5)
    {
        print("n = %d\n", n);
        pause(50);
    }
    n-=5;
    for(n; n > 0; n-=10)
    {
        print("n = %d\n", n);
        pause(50);
    }
    print("All done!");
}
```



## Index Array Variables-

Below is the code we are given to start and to the right, its output:

```
#include "simpletools.h"

int main()
{
    int p[] = {1, 2, 3, 5, 7, 11};
    for(int i = 0; i < 6; i++)
    {
        print("p[%d] = %d\n", i, p[i]);
        pause(50);
    }
}
```

SimpleIDE Termi

```
p[0] = 1
p[1] = 2
p[2] = 3
p[3] = 5
p[4] = 7
p[5] = 11
```

Here, we combine two previous sections to show the true power of arrays and loops. Indexing of an array can be done with a variable that holds the desired value so pairing the scope variable of a for loop for use as an indexing variable makes executing statement to many elements in an array very compact. After discovering how much faster these tasks go by when the delay is down to 50 milliseconds I decided to continue using this delay.

A nifty trick I completely forgot for when an arrays number of elements is unknown is to take the size in bytes of the array, and divide it by the size in bytes of a single item of its contents type – all provided by the function `sizeof()`. This results in the number of elements in the array and can be used in a for loop as can be seen to the right.

```
http://learn.parallax.com/propeller-c-start-sim
*/
#include "simpletools.h"

int main()
{
    int p[] = {1, 2, 3, 5, 7, 11, 13, 17, 19, 23};
    for(int i = 0; i < sizeof(p)/sizeof(int); i++)
    {
        print("p[%d] = %d\n", i, p[i]);
        pause(50);
    }
}
```

SimpleIDE Termi

```
p[0] = 1
p[1] = 2
p[2] = 3
p[3] = 5
p[4] = 7
p[5] = 11
p[6] = 13
p[7] = 17
p[8] = 19
p[9] = 23
```

Last, we perform a multiplication operation on all elements in an array then display the arrays contents using two for loops.

```
#include "simpletools.h"

int main()
{
    int p[] = {1, 2, 3, 5, 7, 11, 13, 17, 19, 23};
    for(int i = 0; i < sizeof(p)/sizeof(int); i++)
    {
        p[i]*=100;
        pause(50);
    }
    for(int i = 0; i < sizeof(p)/sizeof(int); i++)
    {
        print("p[%d] = %d\n", i, p[i]);
        pause(50);
    }
}
```

SimpleIDE Terminal

```
p[0] = 100
p[1] = 200
p[2] = 300
p[3] = 500
p[4] = 700
p[5] = 1100
p[6] = 1300
p[7] = 1700
p[8] = 1900
p[9] = 2300
```

## Variable Values and Addresses-

Below is the code we are given to start and to the right, its output:

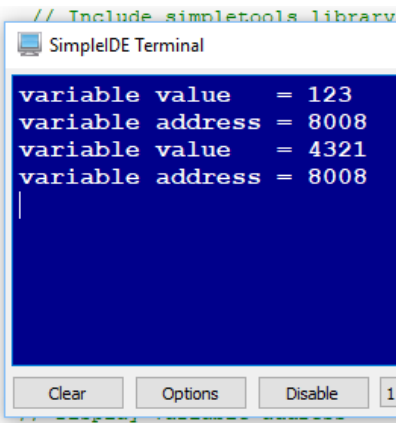
```
#include "simpletools.h"

int variable = 123;

int main()
{
    print("variable value  = %d \n",  variable);
    print("variable address = %d \n", &variable);

    variable = 4321;

    print("variable value  = %d \n",  variable);
    print("variable address = %d \n", &variable);
}
```



The SimpleIDE Terminal window shows the following output:

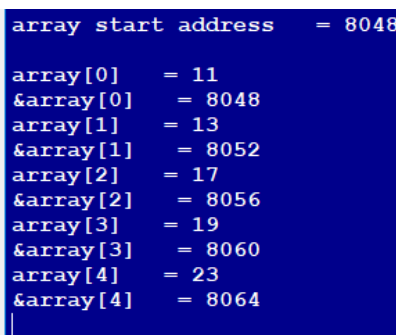
```
variable value  = 123
variable address = 8008
variable value  = 4321
variable address = 8008
```

We begin to explore some of the inner workings of how variables work. A variable is simply a name, or label for a specific location and size of memory. The size is determined by the type of data to be stored there and, in the case of arrays, how many iterations of that type are to be stored. This example starts by showing us that we can see the value of the location in memory our variable is stored through the dereferencing operator(&). It also shows that the location the variable is stored does not change when its value does.

```
//
#include "simpletools.h"

int array[5] = {11,13,17,19,23};

int main()
{
    print("array start address  = %d \n\n",  array);
    for(int i = 0; i<5; i++){
        print("array[%d]    = %d \n",i, array[i]); // Disp
        print("&array[%d]   = %d \n",i, &array[i]); // Di
    }
```



The SimpleIDE Terminal window shows the following output:

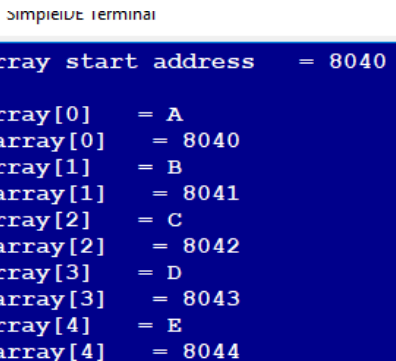
```
array start address  = 8048
array[0]    = 11
&array[0]   = 8048
array[1]    = 13
&array[1]   = 8052
array[2]    = 17
&array[2]   = 8056
array[3]    = 19
&array[3]   = 8060
array[4]    = 23
&array[4]   = 8064
```

The second example shows us the size of an integer. Since array elements occupy contiguous space, its easy to see each integer here takes 4 bytes of space. The last example, below reinforces this idea by using an array characters instead – showing character elements only take a single byte each.

```
#include "simpletools.h"

char array[5] = {'A','B','C','D','E'};

int main()
{
    print("array start address  = %d \n\n",  array);
    for(int i = 0; i<5; i++){
        print("array[%d]    = %c \n",i, array[i]); // Disp
        print("&array[%d]   = %d \n",i, &array[i]); // Di
    }
```



The SimpleIDE Terminal window shows the following output:

```
array start address  = 8040
array[0]    = A
&array[0]   = 8040
array[1]    = B
&array[1]   = 8041
array[2]    = C
&array[2]   = 8042
array[3]    = D
&array[3]   = 8043
array[4]    = E
&array[4]   = 8044
```

### Get Values From Terminal-

Below is the code we are given to start and to the right, its output:

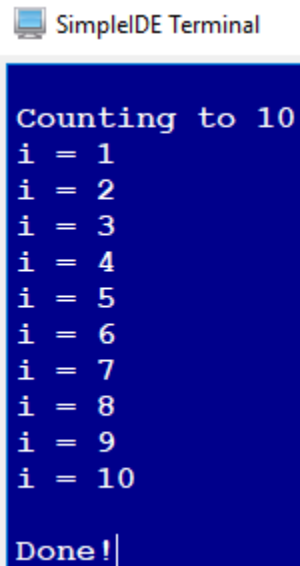
```
#include "simpletools.h"

int reps;

int main()
{
    print("Enter reps: ");
    scan("%d\n", &reps);
    print("\nCounting to %d\n", reps);

    for(int n = 1; n <= reps; n++)
    {
        print("i = %d\n", n);
    }

    print("\nDone!");
}
```



```
SimpleIDE Terminal

Counting to 10
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
Done!
```

This example has us make many small changes to the program and note what changed in between, making this portion of my report a bit graphic heavy unfortunately. We are now introduced to the scan function, one formatted much like print, but is meant to receive data from the user via console input as can be seen above. In the example below, however, we now ask the user for two numbers, one as a number to begin counting from, and the other to tell us when to stop counting.

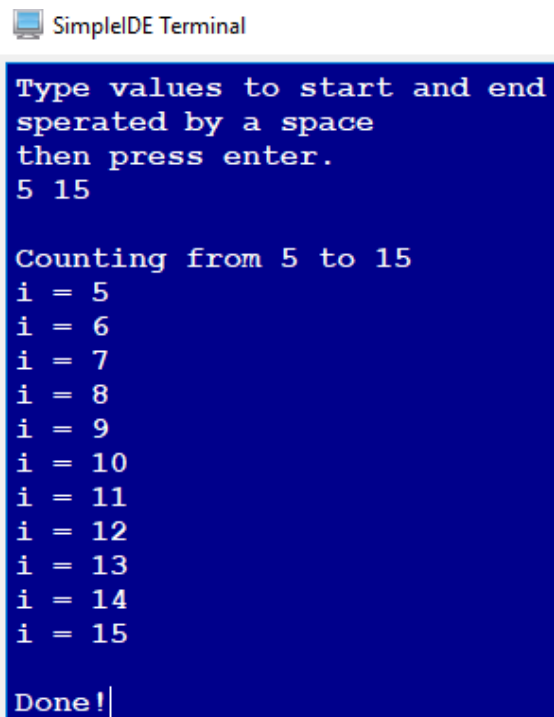
```
#include "simpletools.h"

int startVal;
int endVal;

int main()
{
    print("Type values to start and end\n");
    print("seperated by a space\n");
    print("then press enter.\n");
    scan("%d%d\n", &startVal, &endVal);
    print("\nCounting from %d to %d\n", startVal, endVal);

    for(int n = startVal; n <= endVal; n++)
    {
        print("i = %d\n", n);
    }

    print("\nDone!");
}
```



```
SimpleIDE Terminal

Type values to start and end
seperated by a space
then press enter.
5 15

Counting from 5 to 15
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
i = 11
i = 12
i = 13
i = 14
i = 15
Done!
```

In the next step, we begin modifying the program by exploring a different way to input data from the console, a function called `getDec()` for receiving a decimal number.

```
int startVal;
int endVal;

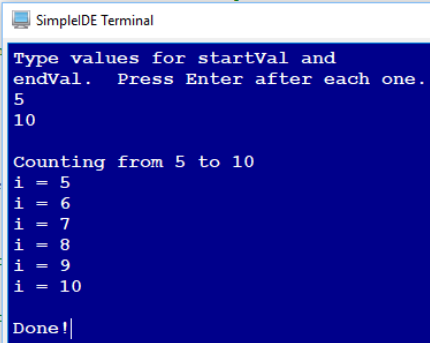
// Declare variable named reps

int main()
{
    // Main
    print("Type values for startVal and \n");
    print("endVal. Press Enter after each one. \n");

    startVal = getDec();
    endVal = getDec();
    print("\nCounting from %d to %d\n", startVal, endVal);

    for(int n = startVal; n <= endVal; n++)
    {
        print("i = %d\n", n);
    }

    print("\nDone!");
}
```



This example is expanded by then showing us there is also a function like this for outputting data too, the “put” type functions.

```
int startVal;
int endVal;

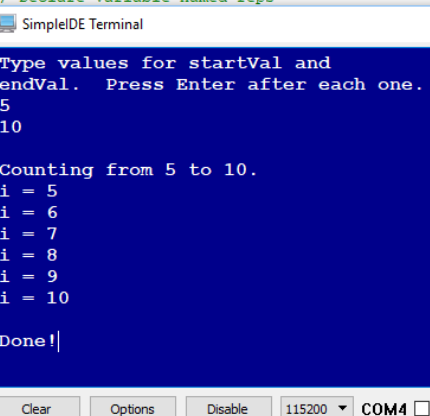
// Declare variable named reps

int main()
{
    // Main
    print("Type values for startVal and \n");
    print("endVal. Press Enter after each one. \n");

    startVal = getDec();
    endVal = getDec();
    putStr("\nCounting from ");
    putDec(startVal);
    putStr(" to ");
    putDec(endVal);
    putStr(".\n");

    for(int n = startVal; n <= endVal; n++)
    {
        print("i = %d\n", n);
    }

    print("\nDone!");
}
```



Last, we are tasked with comparing the original program to the new one in terms of storage allocated, the put and get type functions proved to be much more memory efficient taking up only about half as much space. 9392 bytes vs 4176 respectively

```
int startVal;
int endVal;

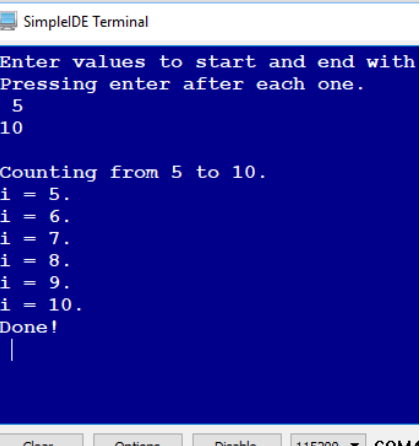
// Declare variable named reps

int main()
{
    // Main
    putStr("Enter values to start and end with \n");
    putStr("Pressing enter after each one. \n ");

    startVal = getDec();
    endVal = getDec();
    putStr("\nCounting from ");
    putDec(startVal);
    putStr(" to ");
    putDec(endVal);
    putStr(".\n");

    for(int n = startVal; n <= endVal; n++)
    {
        putStr("i = ");
        putDec(n);
        putStr(".\n");
    }

    putStr("Done!\n ");
}
```





## Get Text from Terminal-

Below is the code we are given to start and to the right, its output:

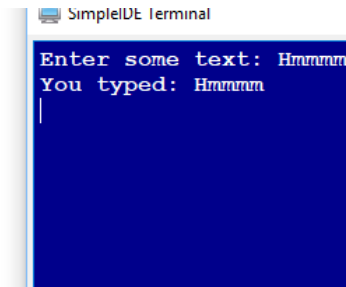
```
#include "simpletools.h"

char text[16];

int main()
{
    print("Enter some text: ");

    getStr(text, 15);

    print("You typed: %s \n", text);
}
```



Now we explore another get type function, `getStr()`, which receives a string from the console that is terminated generally by hitting enter or reaching the end of the allowed number of characters. In the above example we can see that the array of characters `text` (a string) allows for 16 characters. This only allows for 15 characters worth of input as the last space is reserved for the zero-terminating character to mark the end of a string.

```
#include "simpletools.h"

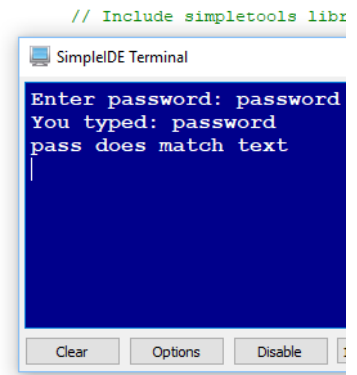
char text[16];
char pass[9] = {"password"};

int main()
{
    print("Enter password: ");

    getStr(text, 15);

    print("You typed: %s \n", text);

    if(!strcmp(pass, text))
    {
        print("pass does match text\n");
    }
    else
    {
        print("pass does NOT match text\n");
    }
}
```



In the above example yet another tool is added to our belt in the ability to compare strings through the function `strcmp()` or string compare. This function returns the character difference between the two strings post mathematical operation. If strings are equal there is no difference and it returns zero. Any number other than zero evaluates to true so that is why we check to see if `strcmp()` is not true.

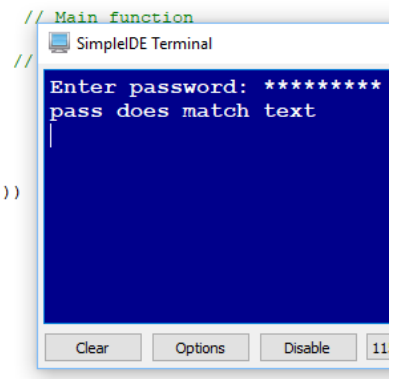
```
#include "simpletools.h" // Include simpletools library

char text[16]; // Declare character array
char pass[9] = {"password"};

int main()
{
    print("Enter password: ");

    for(int i = 0; i < 16; i++)
    {
        text[i] = getChar();
        putChar('*');
        if((text[i] == '\r') || (text[i] == '\n'))
        {
            putChar('\n');
            text[i] = 0;
            break;
        }
    }

    if(!strcmp(pass, text))
    {
        print("pass does match text\n");
    }
    else
    {
        print("pass does NOT match text\n");
    }
}
```



The last image displays the option of accepting input one character at a time then confirming input via asterisk instead of the characters being typed. Much like a password mask we would see for banking or social media logins.

## Blink a light-

We are finally able to move outside the IDE environment and interact with some physical elements. In this case we begin in the same place as always for a new platform such as this, a blinking light. Below is the code as given:

```

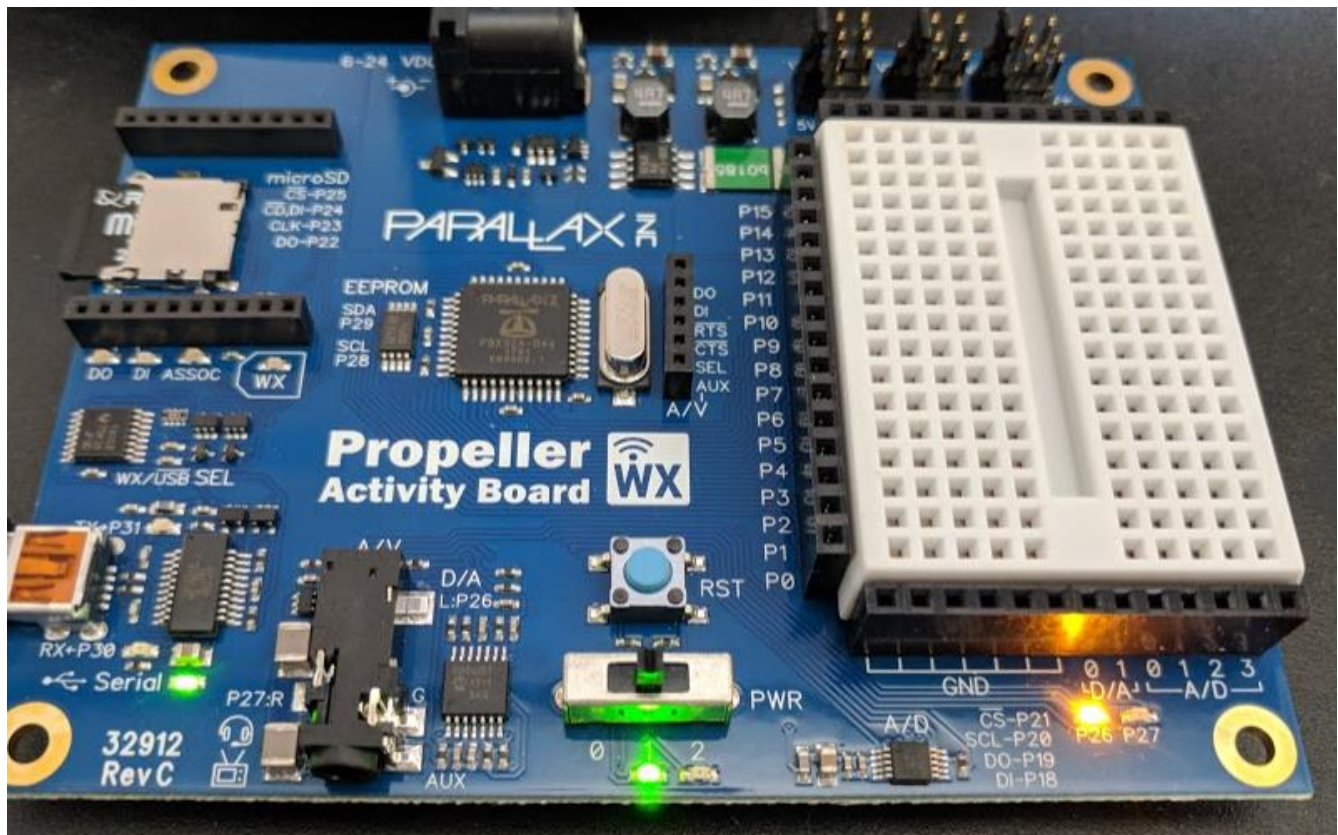
/*
Blink Light.c
Blink light circuit connected to P26.
http://learn.parallax.com/propeller-c-simple-circuits/blink-light
*/

#include "simpletools.h" // Include simpletools

int main() // main function
{
while(1) // Endless loop
{
high(26); // Set P26 I/O pin high
pause(100); // Wait 1/10 second
low(26); // Set P26 I/O pin low
pause(100); // Wait another 1/10 second
}
}

```

This code functions much like the corresponding Arduino sketch. Pause = Delay, High = DigitalWrite(Pin#, HIGH), and low = DigitalWrite(Pin#, LOW). I like how compact this form is much more than the Arduino code already, even while(1) is so obvious in the way it works vs loop() which at first made me wonder "loop until when?" Below is a picture of pin 26 illuminated by the high(26) command.



Our next task was to decrease the delay of time between changing states from HIGH to LOW resulting in a faster blink:

```
int main() // main function
{
while(1) // Endless loop
{
high(26); // Set P26 I/O pin high
pause(50); // Wait 1/10 second
low(26); // Set P26 I/O pin low
pause(50); // Wait another 1/10 second
}
}
```

Next we are asked to use pin 27 in a similar manner and have them blink at virtually the same time, this was achieved by sending both a high signal before delay then a low signal before the next delay.

```
int main() // main function
{
while(1) // Endless loop
{
high(26); // Set P26 I/O pin high
high(27);
pause(100); // Wait 1/10 second
low(26); // Set P26 I/O pin low
low(27);
pause(100); // Wait another 1/10 second
}
}
```

And last, we are instructed to have them alternate, this was achieved by sending a low signal to one pin while the other was high and vice versa.

```
int main() // main function
{
while(1) // Endless loop
{
high(26); // Set P26 I/O pin high
low(27);
pause(200); // Wait 1/10 second
low(26); // Set P26 I/O pin low
high(27);
pause(200); // Wait another 1/10 second
}
}
```

## Check Pushbuttons-

The final example of this lab's part 1. We now test components both off the board and on the board by using its built-in breadboard as well as our own externally. Below is the code as provided:

```
/*
  Button Display.c

  Displays the state of the P3 button in the SimpleIDE Terminal.
  1 -> button pressed, 0 -> button not pressed.
*/

#include "simpletools.h"          // Include simpletools

int main()                      // main function
{
  while(1)                      // Endless loop
  {
    int button = input(3);       // P3 input -> button variable
    print("button = %d\n", button); // Display button state
    pause(100);                 // Wait 0.1 second before repeat
  }
}
```

In retrospect, since I did not notice while working with this, I wish the instantiation of the button variable was done in main and only had the assignment be part of the endless loop. This program takes the voltage read from pin 3 and interprets it as a high or low signal represented by a 1 or 0 respectively. It then takes that value and outputs it to the console window. Without circuit design, we have an active high signal so when the button is pushed we should see 1 in the console. Active low would give 0.

We then modify the program so we can see a flashing light when the button is pressed.

```
/*
  Button Display.c
  Displays the state of the P3 button in the SimpleIDE Terminal.
  1 -> button pressed, 0 -> button not pressed.
  http://learn.parallax.com/propeller-c-simple-circuits/check-pushbuttons
*/

#include "simpletools.h" // Include simpletools

int main() // main function
{
  while(1) // Endless loop
  {
    int button = input(3); // P3 input -> button variable
    print("button = %d\n", button); // Display button state
    if(button == 1){
      high(26);
      pause(50);
      low(26);
      pause(50);
    }
    else{
      pause(100);
    }
    pause(100); // Wait 0.1 second before repeat
  }
}
```

Last, we implement the second button in the given design and make it control the other built in LED, again active high. Below is my final code to do so, given the second button is connected to pin 4.

```

/*
  Button Display.c

  Displays the state of the P3 button in the SimpleIDE Terminal.
  1 -> button pressed, 0 -> button not pressed.

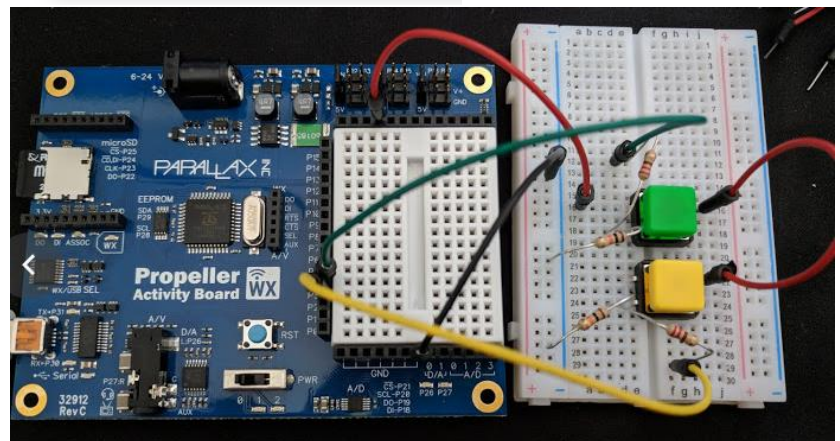
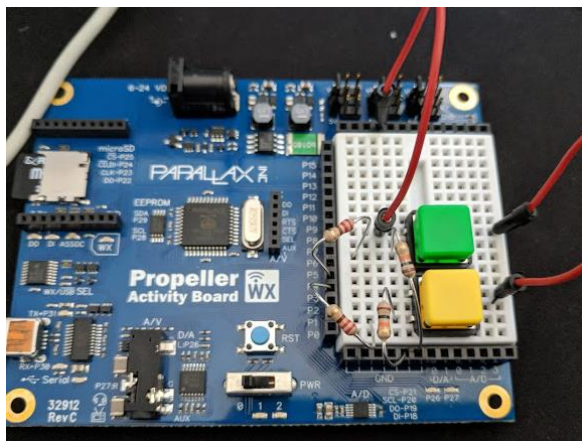
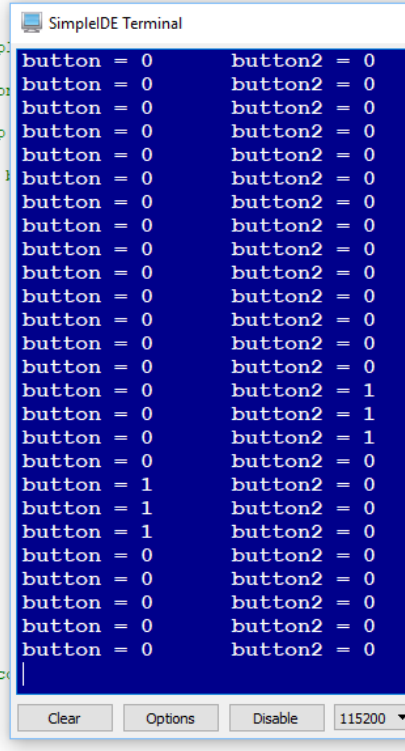
  http://learn.parallax.com/propeller-c-simple-circuits/check-pushbuttons
*/

#include "simpletools.h"           // Include simpletools.h

int main()                       // main function
{
  while(1)                       // Endless loop
  {
    int button = input(3);        // P3 input -> button
    int button2 = input(4);
    print("button = %d\tbutton2 = %d\n", button, button2);

    if(button == 1){
      high(26);
      pause(50);
      low(26);
      pause(50);
    }
    else{
      pause(100);
    }
    if(button2 == 1){
      high(27);
      pause(50);
      low(27);
      pause(50);
    }
    else{
      pause(100);
    }
    pause(100);                  // Wait 0.1 seconds
  }
}

```



## PART 2

### OVERVIEW

Part 2 is where we explore some basic circuits to try and understand fundamentals of this platform. We will learn from these projects before delving deeper in the next section: Piezo Beep, Measure Volts, Set Volts, Sense Light, Standard Servo, Seven Segment Display.

## LAB DISCUSSION

### WORK PERFORMED / SOLUTION:

#### Piezo Beep –

Continuing the trend of tool expansion, we will now follow a walkthrough of operating a piezoelectric speaker. Below is the code as originally provided:

```
#include "simpletools.h"           // Include simpletools

int main()                        // main function
{
    freqout(4, 1000, 3000);       // pin, duration, frequency
}
```

The freqout() command takes three arguments. The first is the pin the signal should be sent to, in our case the speaker is connected to pin 4. The second is the length of time in milliseconds to play a tone, and the third is the frequency of the tone we would like to play in hertz. The above example plays a single 3000Hz note for 1 second.

```
#include "simpletools.h"           // Include simpletools

int note[] = {1047,1175,1319,1397,
              1568,1760,1976,2093};

int main()                        // main function
{
    for (int i = 0; i < 8; i++)
    {
        freqout(4, 1000, note[i]); // pin, duration, frequency
        pause(68);
    }
}
```

The above example expands on the original code by establishing an array of integers to be later interpreted as frequencies we would like to play. In the for loop we then use the freqout() command just like before but pass the indexed array element as the third argument so we can change the pitch played.

The final example below makes a few more changes to play a bit of the batman theme. First change is to use an array of floats for a more accurate note representation. After this, I changed the for loop requirements to be more flexible in case I wanted to change the tune later, by using `sizeof()` I can safely step through the array without knowing its length. Last I added some conditional statements within the for loop for two special cases, so the last two notes are varied in length and pause between notes.

```
#include "simpletools.h"           // Include simpletools

int note[] = {1174.7,1174.7,1108.7,1108.7,
              1046.5,1046.5,1108.7,1108.7, 1174.7, 1174.7};

int main()                       // main function
{
    for (int i = 0; i < (sizeof(note)/sizeof(int)); i++)
    {
        if(i==8)
        {
            freqout(4, 100, note[i]);           // pin, duration, frequency
            pause(210);
        }
        else if(i==9)
        {
            freqout(4, 700, note[i]);           // pin, duration, frequency
            pause(10);
        }
        else
        {
            freqout(4, 300, note[i]);           // pin, duration, frequency
            pause(10);
        }
    }
}
```



## Measure Volts –

We now make use of some knowledge we gained during the step 1 introduction about the analog to digital converters built in to the board. Below is the initial code as provided and its output:

```
Measure Volts.c

Make voltmeter style measurements with the ADC pins.
http://learn.parallax.com/propeller-c-sim
*/

#include "simpletools.h"
#include "adcDCpropab.h"


int main()
{
    adc_init(21, 20, 19, 18);

    float v2, v3;

    while(1)
    {
        v2 = adc_volts(2);
        v3 = adc_volts(3);

        putchar(HOME);
        print("A/D2 = %f V%c\n", v2, CLREOL);
        print("A/D3 = %f V%c\n", v3, CLREOL);

        pause(100);
    }
}
```



In the example above, we make use to two ADC pins so we can take two voltage measurements. In the example below, the wire connected to ADC pin 2 has been moved from ground to the 3.3V output in the board.

```
Make voltmeter style measurements with the ADC pins.
http://learn.parallax.com/propeller-c-sim
*/

#include "simpletools.h"
#include "adcDCpropab.h"

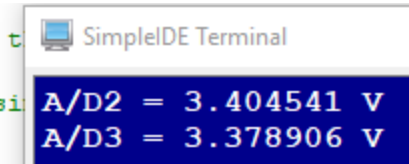
int main()
{
    adc_init(21, 20, 19, 18);

    float v2, v3;

    while(1)
    {
        v2 = adc_volts(2);
        v3 = adc_volts(3);

        putchar(HOME);
        print("A/D2 = %f V%c\n", v2, CLREOL);
        print("A/D3 = %f V%c\n", v3, CLREOL);

        pause(100);
    }
}
```



As it is important to know how this data works we now change the code to give an additional display, the raw value the ADC pins report. For 0 it looks pretty much the same but for 5V there is an obvious disparity. The ADC pins report with an LSB of (5/4090), we could take this raw data and multiply it by that value to get volts but there is a function already built in to do that so we use it instead, `adc_volts()`;

```

// MAKE VOLTAGE STYLE MEASUREMENTS WITH THE ADC
// http://learn.parallax.com/propeller-c-simple-
*/

#include "simpletools.h"
#include "adcDCpropab.h"

int main()
{
    adc_init(21, 20, 19, 18);

    float v2, v3;
    int ad2, ad3;

    while(1)
    {
        v2 = adc_volts(2);
        v3 = adc_volts(3);
        ad2 = adc_in(2);
        ad3 = adc_in(3);

        putchar(HOME);
        print("ad2 = %d%c\n", ad2, CLREOL);
        print("A/D2 = %f V%c\n", v2, CLREOL);
        print("ad3 = %d%c\n", ad3, CLREOL);
        print("A/D3 = %f V%c\n", v3, CLREOL);

        pause(100);
    }
}

```

SimpleIDE Terminal

```

ad2 = 0
A/D2 = 0.000000 V
ad3 = 4090
A/D3 = 4.998779 V
|

```

Last, our job is to modify this program so we can represent the output a temperature sensor would give as too hot, too cold, or just right for a certain habitat. To do this, I used a couple conditional statements to set the ranges we are looking out for. Within those ranges I then display if the reading represents too hot, too cold, or just right and control the build in LED's for a visual indicator outside the terminal window. If too cold, no LED is shown, if too hot, they blink in alternation, and if just right they both remain on. Notable mention as well for the `putChar(HOME)` command they used, it puts the cursor at the very beginning of the terminal window allowing everything previously displayed to be overwritten instead of scrolling down forever, it is much easier to read.

```
#include "simpletools.h"           // Include simpletools
#include "adcDCpropab.h"         // Include adcDCpropab

int main()                       // Main function
{
    adc_init(21, 20, 19, 18);

    float v3;                    // V

    while(1)
    {
        v3 = adc_volts(3);
        putChar(HOME);
        print("A/D3 = %f V%c\n", v3, CLREOL);
        if(v3 > 1.0)
        {
            print("***TOO HOT***");
            high(26);
            low(27);
            pause(20);
            high(27);
            low(26);
            pause(20);
        }
        else if (v3 < .5)
        {
            print("***TOO COLD***");
            low(27);
            low(26);
            pause(40);
        }
        else
        {
            print("***Just right***");
            high(27);
            high(26);
            pause(40);
        }
    }
}
```



### Set Volts –

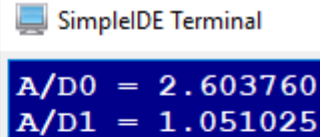
Here, we will now both convert digital to analog and back, in the given code below we output certain values to both pin 26 and pin 27 to represent what voltage to provide. The range of this output is 0V to 3.3V and is broken up into 256ths of its maximum value. The third parameter given in `dac_ctr()` represents the number of 256s to multiply by 3.3V to output.

```
#include "simpletools.h"
#include "adcDCpropab.h"

int main()
{
    adc_init(21, 20, 19, 18);

    dac_ctr(26, 0, 194);
    dac_ctr(27, 1, 78);

    print("A/D0 = %f\n", adc_volts(0));
    print("A/D1 = %f\n", adc_volts(1));
}
```



SimpleIDE Terminal

```
A/D0 = 2.603760
A/D1 = 1.051025
|
```

Next, is to prove this point by changing the values given as third parameters to `dac_ctr()` to their counterparts and to monitor what happens to the voltages and LED's. This can be seen below:

```
*/
#include "simpletools.h"
#include "adcDCpropab.h"

int main()
{
    adc_init(21, 20, 19, 18);

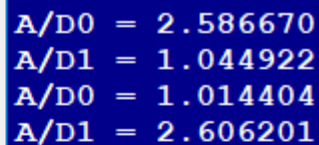
    dac_ctr(26, 0, 194);
    dac_ctr(27, 1, 78);

    print("A/D0 = %f\n", adc_volts(0));
    print("A/D1 = %f\n", adc_volts(1));

    pause(2000);

    dac_ctr(26, 0, 78);
    dac_ctr(27, 1, 194);

    print("A/D0 = %f\n", adc_volts(0));
    print("A/D1 = %f\n", adc_volts(1));
}
```



```
A/D0 = 2.586670
A/D1 = 1.044922
A/D0 = 1.014404
A/D1 = 2.606201
|
```

Last, we are instructed to make the LED's pulsate using the analog to digital method given before. Knowing the full range of output these pins have, we can use a loop to gradually step down or step up the signal given to the LED's. This makes it appear as a smooth transition to the eye since  $1/256^{\text{th}}$  is such a small difference to be made at any given moment.

```

/*
  Set and Measure Volts.c

  Set D/A0 to 2.5 V and D/A1 to 1 V, and measure with A/D0 and A/D1.

  http://learn.parallax.com/propeller-c-simple-circuits/set-volts
*/

#include "simpletools.h"           // Include simpletools
#include "adcDCpropab.h"         // Include adcDCpropab

int main()                       // main function
{
  adc_init(21, 20, 19, 18);      // CS=21, SCL=20, DO=19, DI=18

  dac_ctr(26, 0, 194);           // D/A ch 0 -> 2.5 V to D/A 0
  dac_ctr(27, 1, 78);           // D/A ch 1 -> 1 V to D/A 1

  print("A/D0 = %f\n", adc_volts(0)); // Display A/D0 volts
  print("A/D1 = %f\n", adc_volts(1)); // Display A/D1 volts

  pause(2000);

  dac_ctr(26, 0, 78);
  dac_ctr(27, 1, 194);

  print("A/D0 = %f\n", adc_volts(0)); // Display A/D0 volts
  print("A/D1 = %f\n", adc_volts(1)); // Display A/D1 volts

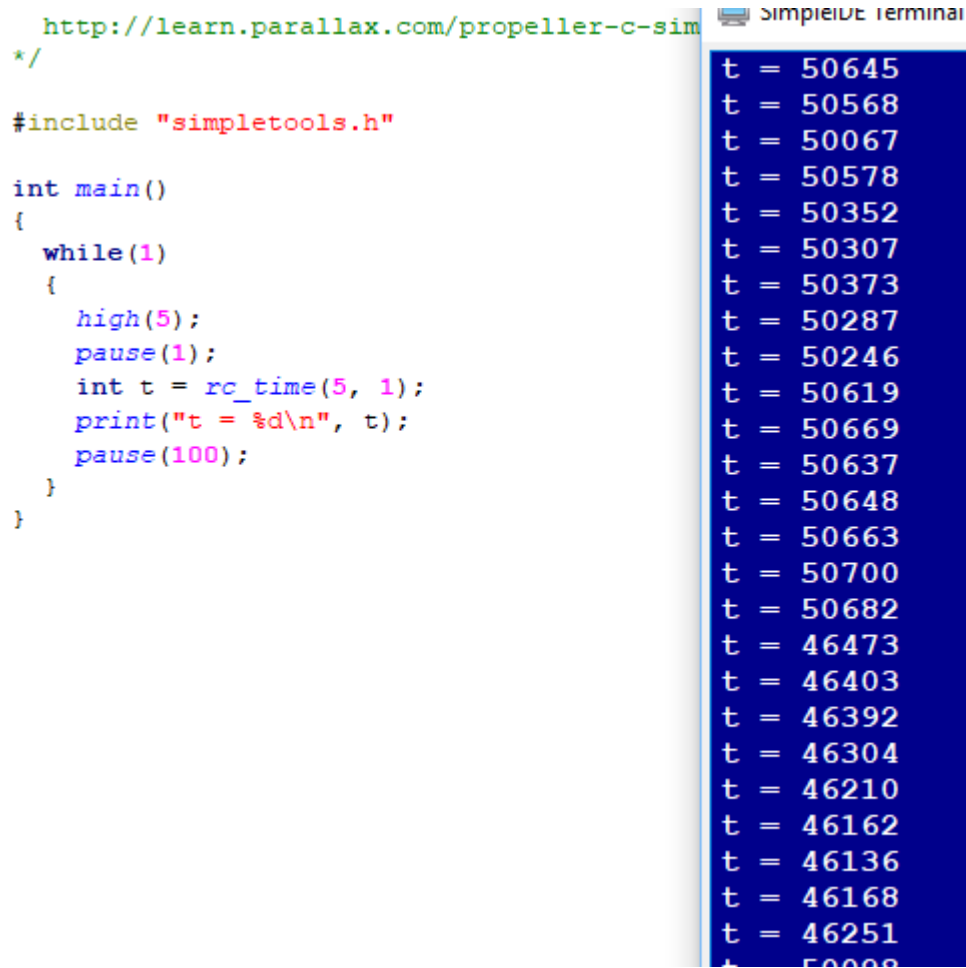
  pause(2000);

  while(1)
  {
    for(int v = 0; v < 256; v++)
    {
      dac_ctr(26, 0, v);
      dac_ctr(27, 1, 255-v);
      pause(4);
    }
  }
}

```

## Sense Light –

We now build a charge transfer circuit to give the much broader spectrum of values that can be obtained by mean outside of analog to digital conversion. This example will use a variable nozzle in parallel with a capacitor to see how fast the circuit is discharged. Our nozzle in this case is the phototransistor, in this case the more light it sees, the greater amount of current that will be allowed through and therefore the faster the capacitor is able to discharge.



```

http://learn.parallax.com/propeller-c-sim
*/

#include "simpletools.h"

int main()
{
    while(1)
    {
        high(5);
        pause(1);
        int t = rc_time(5, 1);
        print("t = %d\n", t);
        pause(100);
    }
}

```

```

SimpleIDE terminal
t = 50645
t = 50568
t = 50067
t = 50578
t = 50352
t = 50307
t = 50373
t = 50287
t = 50246
t = 50619
t = 50669
t = 50637
t = 50648
t = 50663
t = 50700
t = 50682
t = 46473
t = 46403
t = 46392
t = 46304
t = 46210
t = 46162
t = 46136
t = 46168
t = 46251
t = 50000

```

In the example above I turned off all lights but my computers monitor and instantly saw the values shoot upwards.

```
#include "simpletools.h"

int main()
{
    while(1)
    {
        high(5);
        pause(1);
        int t = rc_time(5, 1);
        print("t = %d\n", t);
        pause(100);
    }
}
```

```
t = 1409
t = 1409
t = 1409
t = 1409
t = 1406
t = 1408
t = 1406
t = 1406
t = 1407
t = 1403
t = 1403
t = 1402
t = 1402
t = 1403
t = 1402
```

In the attempt above I turned on 6 daylight temperature LED bulbs

```
int main()
{
    while(1)
    {
        high(5);
        pause(1);
        int t = rc_time(5, 1);
        print("t = %d\n", t);
        pause(100);
    }
}
```

```
t = 9400
t = 9397
t = 9401
t = 9396
t = 9405
t = 9397
t = 9402
t = 9394
t = 9398
t = 9398
t = 9398
t = 9398
t = 9398
t = 9401
```

In the example above I still had the 6 LEDs on but used a capacitor with 10 times the capacity as the first, giving a roughly 10 times increase in the amount of time required to discharge.

Putting this together with the piezo knowledge gained earlier I then mapped a fraction of this input to the frequency delivered to the speaker. The result in a tone that changes as the amount of light in the room changes. A brighter room represented a lower tone.

```
while(1)
{
    high(5);
    pause(1);
    int t = rc_time(5, 1);
    print("t = %d\n", t);
    freqout(15, 20, t/30);
}
}
```

## Standard Servo –

Below is the code as provided for this portion of the lab. Having already completed the Arduino lab, I can say that I know what the servo is supposed to do and observe things a bit closer as I make my way through this iteration. Right off the bat however, I would like to note just how much cleaner this code is one again than the Arduino library, which I find to be a strange theme seeing as how they both are based off of the C language.

```
#include "simpletools.h"           // Include simpletools header
#include "servo.h"                 // Include servo header

int main()                        // main function
{
    servo_angle(16, 0);           // Pl6 servo to 0 degrees
    pause(3000);                  // ...for 3 seconds
    servo_angle(16, 900);         // Pl6 servo to 90 degrees
    pause(3000);                  // ...for 3 seconds
    servo_angle(16, 1800);        // Pl6 servo to 180 degrees
    pause(3000);                  // ...for 3 seconds
    servo_stop();                 // Stop servo process
}
```

Above, we have three servo movements executed by three calls to the function `servo_angle()`. The first parameter is the servo header in use, and the second is the angle from position zero we want the servo to spin to in increments of .1 degrees. ( $1800 * .1 = 180$  degrees). This is all followed by a stop function which unlocks the servo motor.

```
#include "simpletools.h"           // Include simpletools header
#include "servo.h"                 // Include servo header

int main()                        // main function
{
    servo_angle(16, 0);           // Pl6 servo to 0 degrees
    pause(3000);                  // ...for 3 seconds
    servo_angle(16, 900);         // Pl6 servo to 90 degrees
    pause(3000);                  // ...for 3 seconds
    servo_setramp(16,8);          // Pl6 servo to 180 degrees
    servo_angle(16, 1800);        // ...for 3 seconds
    pause(3000);                  // Pl6 servo to 90 degrees
    servo_angle(16, 900);         // ...for 3 seconds
    pause(3000);                  // Stop servo process
    servo_stop();
}
```

Above, we can see some changes have been made. Most obviously, one additional servo movement has been added to return the motor to the 90 degree mark instead of stopping at 180 like before but the second change is the focus here. The `servo_setramp()` function is one who controls how fast the servo motor moves. The reason it was placed before two servo movement is so we can realize it's a change that stays in effect until changed again. The first parameter is the servo header again but the second is how many tenths of a degree to move for every fiftieth of a second.



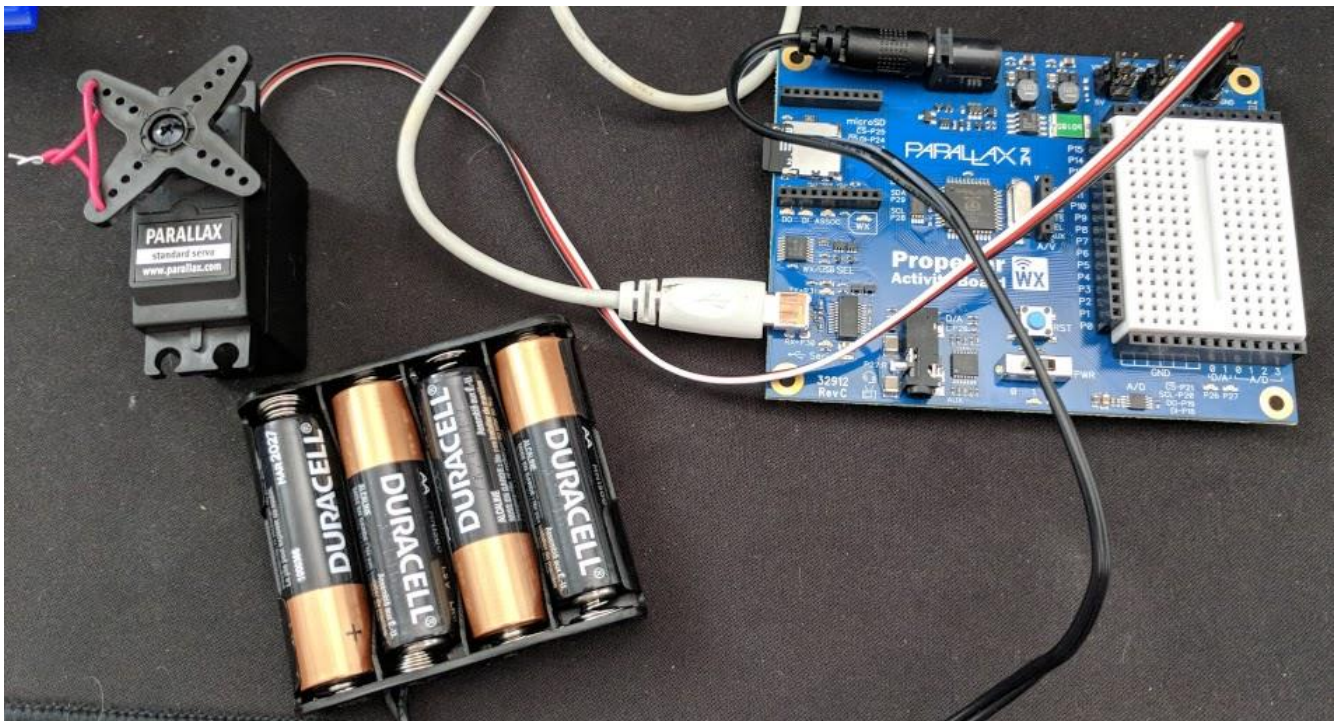
```

#include "simpletools.h"           // Include simpletools header
#include "servo.h"                // Include servo header

int main()                       // main function
{
    int p[] = {100,200,300,500,700,1100};|
    for(int i = 0; i < sizeof(p)/sizeof(int); i++)
    {
        servo_angle(16, p[i]);    // P16 servo to 0 degrees
        pause(3000);              // ...for 3 seconds
    }
    servo_stop();                 // Stop servo process
}

```

Finally, we are reminded of the convenience of arrays and how we can store sequences for devices to follow much like the piezo speaker song. In the above example I decided to make use of the sizeof trick again in case more angles should be added to the array. This approach is much more compact and thus easier to read. Below, is my circuit built as the walkthrough recommended.



## Seven Segment Display –

This first snippet of code is loaded with information. First, we are introduced to a new way to declare a contiguous chunk of pins as input or outputs by using `set_directions()`. The first two parameters establish the range of pins to affect, and the last parameter is a bit mask to tell the system if that pin should be an input or an output. To start, each bit in the mask represents each individual pin following the same order provided. Using the example below reading left to right that means pin 15 is 1, pin 14 is 1, so on and so forth until the final bit for pin 8 is also 1. 1 represents output and 0 represents input.

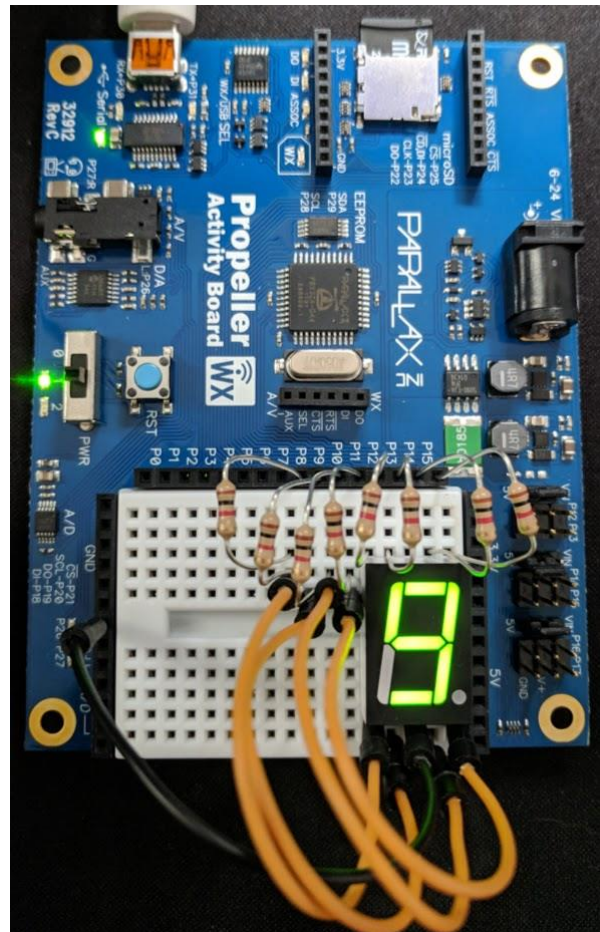
The next part of this code is the repeated sections of `set_outputs()` and `pause()`. We already know what `pause()` does so I'll skip the explanation, `set_outputs()` however is new. This new function is similar to the last except it batch assigns either a high or low signal to the range of pins given using the same bit mask idea.

```
#include "simpletools.h"           // Include simpletools

int main()                       // main function
{
    set_directions(15, 8, 0b11111111); // P15...P8 -> output

    set_outputs(15, 8, 0b11100111); // 0 -> 7-segment display
    pause(500);
    set_outputs(15, 8, 0b10000100); // 1
    pause(500);
    set_outputs(15, 8, 0b11010011); // 2
    pause(500);
    set_outputs(15, 8, 0b11010110); // 3
    pause(500);
    set_outputs(15, 8, 0b10110100); // 4
    pause(500);
    set_outputs(15, 8, 0b01110110); // 5
    pause(500);
    set_outputs(15, 8, 0b01110111); // 6
    pause(500);
    set_outputs(15, 8, 0b11000100); // 7
    pause(500);
    set_outputs(15, 8, 0b11110111); // 8
    pause(500);
    set_outputs(15, 8, 0b11110110); // 9
    pause(500);
}
```

The circuit itself is very straight forward. The seven segment display can be thought of as simply a group of LEDs that have a common anode pin and a common cathode pin for them to all share one signal on that side of the diode. All the resistors shown are just current limiting resistors as would be used with any LED. All of the remaining pins besides the common anode and cathode ones correspond to a certain sections LED. In this case the common cathode was grounded, and we send a high signal to the segment we want to light.



(My remaining examples will be much briefer now that these ideas have been explained.)

Below we change the code a bit to make it more compact by again using an array. Since there was so much repetition in the previous code it is easy to convert these statements into a loop structure.

```

Display digits on a 7 segment (common cathode) LED display
http://learn.parallax.com/propeller-c-simple-circuits/s
*/

#include "simpletools.h"           // Include s

int n[] = {0b11100111,0b10000100,0b11010011,0b11010110,
           0b10110100,0b01110110,0b01110111,0b11000100,
           0b11110111,0b11110110};

char s[9];

int main()                       // main func
{
    set_directions(15, 8, 0b11111111);    // P15...P8
    pause(1000);

    for (int i = 0; i<10; i++)
    {
        set_outputs(15, 8, n[i]);          // 0-9
        print("n[%d] displayed.\n", i);
        pause(500);
    }
}

```

```

SimpleIDE Terminal
n[0] displayed.
n[1] displayed.
n[2] displayed.
n[3] displayed.
n[4] displayed.
n[5] displayed.
n[6] displayed.
n[7] displayed.
n[8] displayed.
n[9] displayed.

```

The last task of this section is to apply what we learned from the measure volts section to create a basic voltmeter. Since we only have this one 7 segment display to use I used a quick rounding method (only good for positive values) to take account for the integer truncation.

```

http://learn.parallax.com/propeller-c-simple-circuits/s
*/

#include "adcDCpropab.h"         // Include
#include "simpletools.h"         // Include
#include "math.h"

float v2, v3;                   // Voltage va
float diff;
int int_diff;

int n[] = {0b11100111,0b10000100,0b11010011,0b11010110,
           0b10110100,0b01110110,0b01110111,0b11000100,
           0b11110111,0b11110110};

int main()                       // main func
{
    adc_init(21, 20, 19, 18);    // CS=21, :
    set_directions(15, 8, 0b11111111);    // P15...P8
    pause(1000);

    while(1)                     // Loop repe
    {
        v2 = adc_volts(2);
        v3 = adc_volts(3);
        diff = v3-v2;
        int_diff = diff + .5;
        putChar(HOME);           // Cursor ->
        print("Voltage = %f V%c\n", diff, CLREOL);    // Disp
        set_outputs(15, 8, n[int_diff]);          // 0-9
        print("n[%d] displayed.\n", int_diff);
        pause(100);
    }
}

```

```

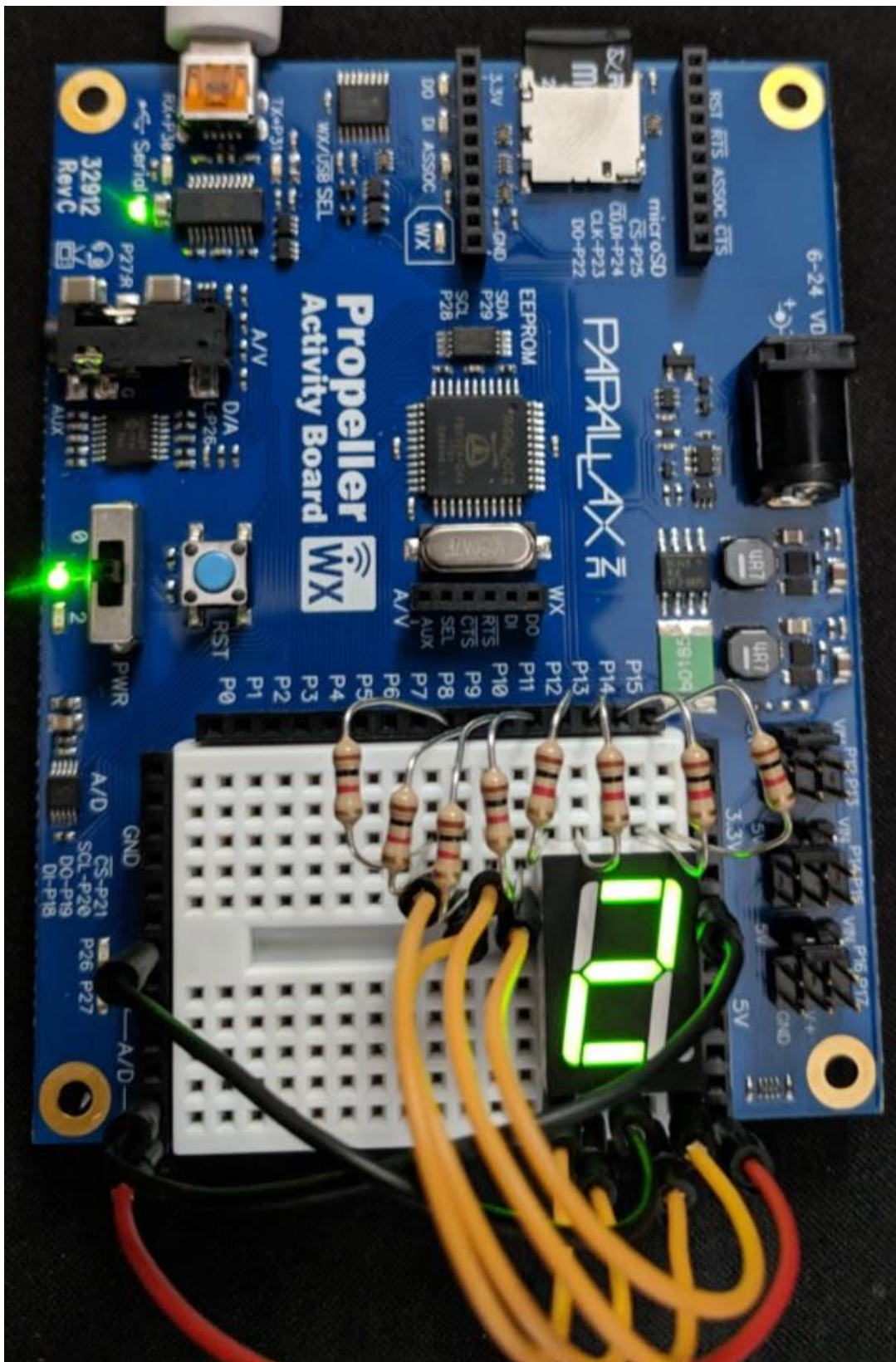
Voltage = 4.992676 V
n[5] displayed.

```

Clear Options Disable



Here is the functional circuit. The low potential prong is in 3.3V and high potential prong in 5V yielding a rounded difference of 2 volts.



## PART 3

### OVERVIEW

This section will focus a lot more on reading, with some doing sprinkled in. We will learn these topics and more: SD card data and play WAV files, propeller C functions, multicore approaches, what's a multicore?, library studies, counter modules applied and execution of code, examine the C interface with parallax assembly language. Given the nature of this section, most of the discussion topics will be summaries, insights, or both from what I was instructed to read.

## LAB DISCUSSION

WORK PERFORMED / SOLUTION:

## SD Card Data and Play WAV Files –

```
/*
  SD Minimal.side

  Create test.txt, write characters in, read back
  http://learn.parallax.com/propeller-c-sim
*/

#include "simpletools.h"

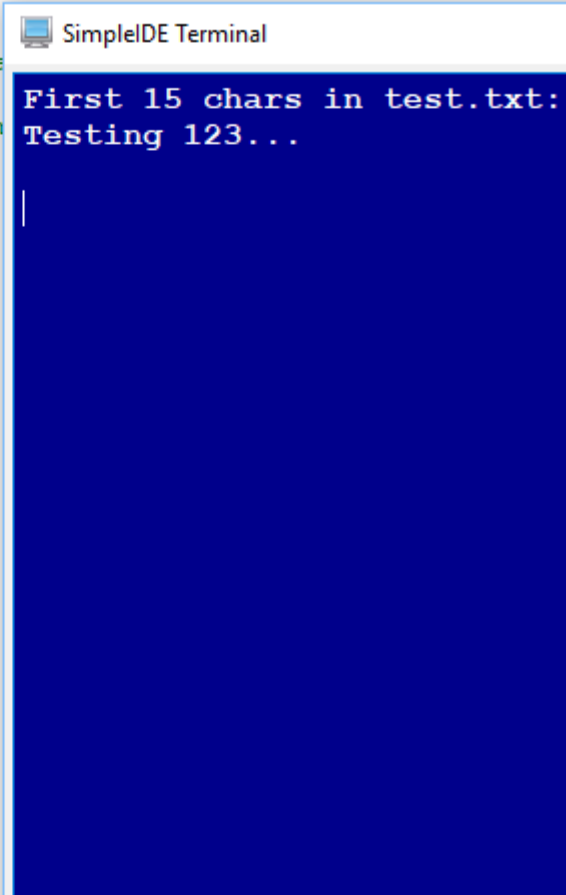
int DO = 22, CLK = 23, DI = 24, CS = 25;

int main(void)
{
  sd_mount(DO, CLK, DI, CS);

  FILE* fp = fopen("test.txt", "w");
  fwrite("Testing 123...\n", 1, 15, fp);
  fclose(fp);

  char s[15];
  fp = fopen("test.txt", "r");
  fread(s, 1, 15, fp);
  fclose(fp);

  print("First 15 chars in test.txt:\n");
  print("%s", s);
  print("\n");
}
```

A screenshot of the SimpleIDE Terminal window. The terminal has a blue background and white text. It displays the output of the code: "First 15 chars in test.txt:" followed by "Testing 123..." on the next line. A white cursor is visible on the line following the output.

Using the code provided above I was able to test the SD card reading and writing capabilities of my board. The code helped me remember that a file pointer needs to be created to be able to close the file when complete.

```

/*
SD Minimal.side

Create test.txt, write characters in, read back out. display.

http://learn.parallax.com/propeller-c-simp
*/

#include "simpletools.h"

int DO = 22, CLK = 23, DI = 24, CS = 25;

int main(void)
{
    sd_mount(DO, CLK, DI, CS);

    FILE* fp = fopen("test.txt", "w");

    int val = 500;
    fwrite(&val, sizeof(val), 1, fp);    // 1
    val = -100000;
    fwrite(&val, sizeof(val), 1, fp);    // 2
    fclose(fp);

    //char s[15];
    fp = fopen("test.txt", "r");

    fread(&val, 4, 1, fp);
    print("val = %d\n", val);
    fread(&val, 4, 1, fp);
    print("val = %d\n", val);
    fclose(fp);

    //print("First 15 chars in test.txt:\n");
    // print("%s", s);
    // print("\n");
}

```

SimpleIDE Terminal

```

val = 500
val = -100000

```

Clear

Options

Some small modifications to the program provided before gave way to the next example above. This demonstrates the ability to store separate pieces of data of an indicated size to be able to be retrieved sequentially later.

```

#include "simpletools.h"
#include "adcDCpropab.h"

int DO = 22, CLK = 23, DI = 24, CS = 25;

int main(void)
{
    adc_init(21, 20, 19, 18);
    sd_mount(DO, CLK, DI, CS);

    FILE* fp = fopen("test.txt", "w");
    float val;

    for(int i = 0 ; i < 10; i++)
    {
        val = adc_volts(2);
        fwrite(&val, sizeof(val), 1, fp);    // Adc
    }

    fclose(fp);

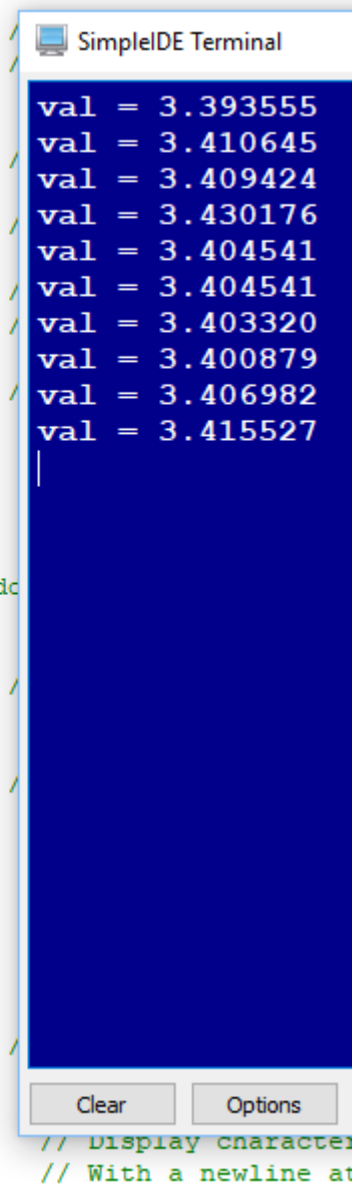
    //char s[15];
    fp = fopen("test.txt", "r");

    for(int i = 0 ; i < 10; i++)
    {
        fread(&val, 4, 1, fp);
        print("val = %f\n", val);
    }

    fclose(fp);

    //print("First 15 chars in test.txt:\n");
    // print("%s", s);
    // print("\n");
}

```



```

SimpleIDE Terminal

val = 3.393555
val = 3.410645
val = 3.409424
val = 3.430176
val = 3.404541
val = 3.404541
val = 3.403320
val = 3.400879
val = 3.406982
val = 3.415527

Clear Options
// Display character
// With a newline at

```

Last, I was tasked with making further modifications to store incoming values from the boards pins to the SD card. I chose to use the measure volts example from earlier to measure the 3.3V header 10 times. I then displayed the results after re-opening the file for reading.



```

/*
  Test WAV Volume.c

  Play back a .wav file and try a few different volume settings.

  http://learn.parallax.com/propeller-c-simple-devices/play-wav-files
*/

#include "simpletools.h"
#include "wavplayer.h"

int main()                                // main function
{
  int DO = 22, CLK = 23, DI = 24, CS = 25; // SD I/O pins
  sd_mount(DO, CLK, DI, CS);              // Mount SD card

  const char techloop[] = {"techloop.wav"}; // Set up techloop string
  wav_play(techloop);                      // Pass to wav player

  wav_volume(6);                           // Adjust volume
  pause(3500);                             // Play for 3.5 s
  wav_volume(4);                           // Repeat twice more
  pause(2000);
  wav_volume(8);
  pause(3500);

  wav_stop();                             // Stop playing
}

```

This exercise was to demonstrate the ability to process and play .wav files using our board. After downloading the .wav file provided and using the code provided as displayed above I was able to hear some music played at a few different volumes. It seems the volume modifying function stays in effect until changed again.

```

/*
  Test WAV Volume.c

  Play back a .wav file and try a few different volume settings.

  http://learn.parallax.com/propeller-c-simple-devices/play-wav-files
*/

#include "simpletools.h"
#include "wavplayer.h"

int main()                                // main function
{
  int button1, button2;
  int DO = 22, CLK = 23, DI = 24, CS = 25; // SD I/O pins
  sd_mount(DO, CLK, DI, CS);              // Mount SD card

  button1 = input(7);                      //pull down externally
  button2 = input(8);                      //pull down externally

  const char track1[] = {"track1.wav"};    // Set up techloop string
  const char track2[] = {"track2.wav"};    // Set up techloop string

  if(button1)
  {
    wav_play(track1);                      // Pass to wav player

    wav_volume(8);
    pause(3500);

    wav_stop();                            // Stop playing
  }

  else if(button2)
  {
    wav_play(track2);                      // Pass to wav player

    wav_volume(8);
    pause(3500);

    wav_stop();                            // Stop playing
  }
}

```

The new example above expands on the previous code by providing conditional logic linked to button status. If one button is pressed, then one tune is played, if the other button is pressed then another sound clip is played instead.

## Propeller C Functions –

Send a simple hello message to the console  
one of the messages.

<http://learn.parallax.com/propeller-c-functions>  
\*/

```
#include "simpletools.h"
```

```
void hello(void);
```

```
int main()
{
    hello();
    print("Hello again from main!\n");
}
```

```
void hello(void)
{
    print("Hello from function!\n");
    pause(500);
}
```

```
Hello from function!
Hello again from main!
```

I'm not entirely sure why this concept was introduced so late in the process but here we are, the idea of calling your own defined functions from within main. The terminal outputs are just a way to prove that control can jump from one place to another (through the modification of the instruction pointer).

```
/*
    Function with Parameter.c

    Call a function that displays a value passed to it.

    http://learn.parallax.com/propeller-c-functions
*/
```

```
#include "simpletools.h"
```

```
void value(int a);
```

```
int main()
{
    value(6);
}
```

```
void value(int a)
{
    print("The value is: a = %d\n", a);
}
```

SimpleIDE Terminal

```
The value is: a = 6
```

Now we are shown that functions can be given parameters to communicate information. In this case we use an immediate value, but variables can be used as well.

In the last example below, we demonstrate the use of global variables – variable that can be accessed from anywhere, as another way to communicate data between functions.

```
/*
  Global Exchange.c

  Functions exchange information with glob

  http://learn.parallax.com/projects/SimpleIDE
*/

#include "simpletools.h"

void subtracter(void);

int a, b, n;

int main()
{
  a = 96;
  b = 32;
  subtracter();
  print("n = %d\n", n);
}

void subtracter(void)
{
  n = a - b;
}
```

A screenshot of a SimpleIDE window showing the output of the program. The text "n = 64" is displayed on a dark blue background.

## Multicore Approaches –

Requirements for the function `cog_run`: function cannot have parameters, return a value, should have all instructions in a `while(1)` loop. Should not use `print`, `scan`, or other calls using simpleIDE terminal. Must do so by first handling over terminal control.

To the right is a very simple example given of the way cogs operate. As can be seen, this will cause the LED's to blink, note there is no end to these cogs and thus the program will not end, not will it release the cores or memory in use. Below is the terminal output for this program.

```
Just one cog running...
Cog launched to blink P26 LED
Cog launched to blink P27 LED
```

The final example below is me playing with the timings for the blink to end and to see that a cog can end itself too. If the end is external, then a pause or a trigger of some sort must be used. If the end is immediate then the statements are executed so rapidly it's like the cog was never really running to begin with.

```
#include "simpletools.h"

void blink();
int * cog;

int main()
{
    cog = cog_run(blink, 128);
    //pause(3000);
    //cog_end(cog);
}

void blink()
{
    for(int n = 1; n <= 15; n++)
    {
        high(26);
        pause(100);
        low(26);
        pause(100);
    }
    cog_end(cog);
}
```

```
#include "simpletools.h"

void blink();

void blink2()
{
    while(1)
    {
        high(27);
        pause(223);
        low(27);
        pause(223);
    }
}

int main()
{
    cog_run(blink, 128);
    cog_run(blink2, 128);
}

void blink()
{
    while(1)
    {
        high(26);
        pause(100);
        low(26);
        pause(100);
    }
}
```

## PART 4

### OVERVIEW

This lab is now ending, part 4 is to demonstrate the knowledge gained from prior exercises by designing a project of our own. I decided to use the joystick as I did in the Arduino lab but for a different use this time. One of the main focuses of this lab was using a multicore approach, making that idea the cornerstone of my design.

## LAB DISCUSSION

### WORK PERFORMED / SOLUTION:

Knowing I would be running multiple cores I decided to design from the bottom up, defining functions to use first, then wrapping a control around them to make them work together. My end goal was to have a joystick control a servo, while also having a button function as a sort of enable line for the system to be functional. With this idea I now knew I had 3 components and would consequently write 3 controlling functions. The button function needed to have control over the joystick and servo operation. The joystick function would simply output data that correlates to the angle the servo should move to. Last, the servo function interprets the angle given by the joystick and moves there.

To achieve these tasks, I needed to have a volatile global variable for indicating whether the system is currently on or off, and another one for communicating the angle from the joystick to the servo.

Combined with a global volatile variable, the button function would also need control over starting and stopping the servo and joystick control, so I made their cog pointers global as well. After the main function initializes the system to on, all remaining control is given to the button.

As outlined in the listing files below, assembly is very straight forward. Even more so than given in the diagram for two reasons; First, the indicator LED used is build into the board. Second, the button used is built into the joystick.

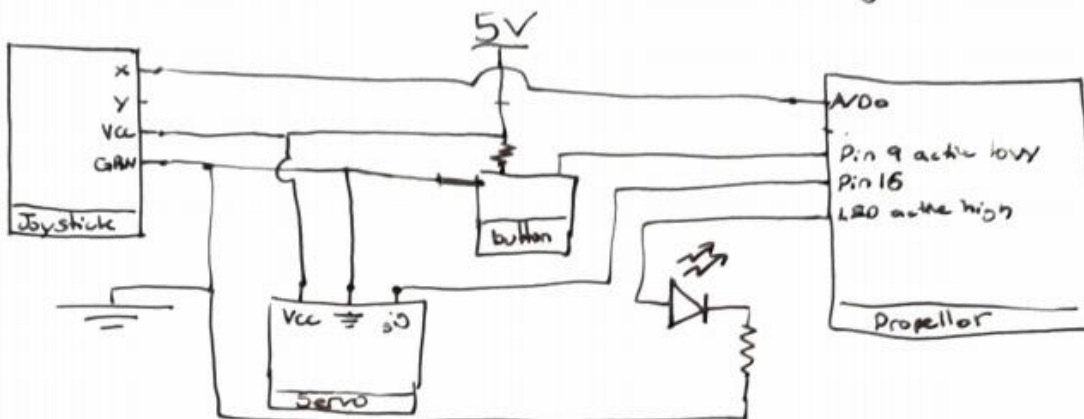
For this reason, and for my previous bouts with the joystick in the Arduino lab – this build went swimmingly and worked as expected the very first time. I'd like to use this code again for an even faster responding servo, so the joystick input and servo angle would have very little delay. I believe this type of approach plus perhaps some sort of jitter reduction or wiggle room in the joystick would give very precise control of structures like cranes, RC cars, or simulation yolks.

LISTING FILES(S):

## Part 4)

idea: Joystick controlled servo w/  
active/inactive indication LED.  
System turned on & off via pushbutton

volatile global variables for : main in cog 0  
Joystick in cog 1  
servo in cog 2  
LED in cog 3  
Pushbutton in cog 4



Joystick range 0-5V , servo range 0-180 degrees  
 $\Rightarrow \left(\frac{x}{5}\right) \cdot 180 \Rightarrow \text{degrees} * 10 \Rightarrow \text{servo angle (0-1800)}$

```

/*
Blank Simple Project.c
http://learn.parallax.com/propeller-c-tutorials
*/
#include "simpletools.h"          // Include simple tools
#include "servo.h"               // Include servo header
#include "adcDCpropab.h"         // Include adcDCpropAB

int* button_p, joystick_p, led_p, servo_p;
volatile int systemStatus, angle;

void joystick()
```



```

{
  while(1)
  {
    angle = (adc_volts(0)/5) * 1800;
  }
}

void led()
{
  while(1)
  {
    if(systemStatus)
    {
      high(26);
    }
    else
    {
      low(26);
    }
  }
}

void servo()
{
  while(1)
  {
    servo_angle(16, angle);
  }
}

void button()
{
  while(1)
  {
    if(!input(9))
    {
      if(systemStatus == 0)
      {
        joystick_p = cog_run(joystick, 128);
        servo_p = cog_run(servo, 128);
        systemStatus = 1;
        pause(200);
      }
      else
      {
        cog_end(joystick_p);
        cog_end(servo_p);
        servo_stop();
        systemStatus = 0;
        pause(200);
      }
    }
  }
}

int main()                // Main function
{
  adc_init(21, 20, 19, 18);      // CS=21, SCL=20, DO=19, DI=18
  systemStatus = 1;
  joystick_p = cog_run(joystick, 128);
  led_p = cog_run(led, 128);
  servo_p = cog_run(servo, 128);
  button_p = cog_run(button, 128);
}

```

