# Introduction to Programming With 3pi

Armstrong State University

# 1. Introduction to Programming

The following sections assume the user has little to no knowledge of the C programming language, but that the user has a basic understand of programming in general. This section seeks to introduce the user to the C programming language by programming the 3pi robot.

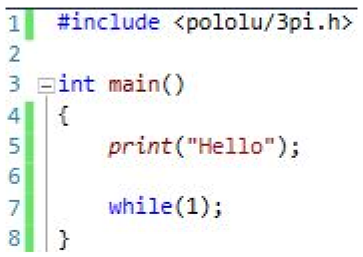
## 1.1 Basic Output of the 3pi

When you give the 3pi robot some kind of input, often times there is an output. Input is any type of information the robot can receive. Input can be many things like a number, string, or even the pressing of a button. All of these give the robot some kind of information that it can use to do certain actions.

Output is any type of information the robot gives back to you or an action it does based on the input it received. Like input, output can be many things like a number or a string. The robot can also use the screen (called an LCD) to display text, its lights (called LEDs), and the buzzer (plays music and sounds) for output. As you learn more about the robot, you will get to know more inputs and outputs.

### 1.1.1 Hello World

In this section you will be shown how to output values to the LCD. Many types of values can be outputted to the LCD but we will use strings for now. The program below shows how the string “Hello” is outputted to the LCD.

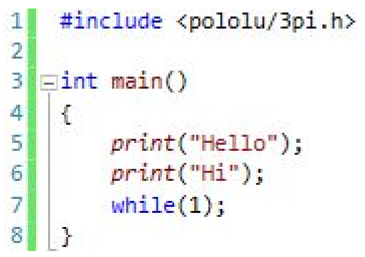


```
#include <pololu/3pi.h>

int main()
{
    print("Hello");

    while(1);
}
```

Prints “Hello” to the LCD

Using a function called print, one can output strings to the robot’s LCD. You will learn more about how these functions and how they work later. You should be able to print any character found on your keyboard, like question marks (?), percent signs (%), and dollar signs($). The “while(1);” is also an important part of the program. The “while” is called a loop and in this case it stops the program from ending. Programs for the 3pi robots should not reach the end as it may cause unpredictable behavior. You will also learn more about loops in future sections, just keep in mind statements like this are needed so programs do not reach the end.

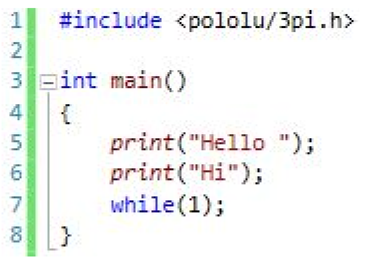
You can use the print function again to print more strings onto the LCD. The strings will be printed right after the last string that was printed. For example, you can use another print statement to print “Hi” to the LCD like so:



```
#include <pololu/3pi.h>

int main()
{
    print("Hello");
    print("Hi");
    while(1);
}
```

Prints “HelloHi” to the LCD

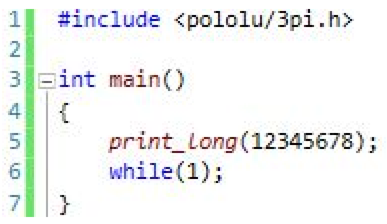
The strings will be printed exactly as you type them. With the code above, the LCD should appear as “HelloHi”. Putting a space after “Hello” will cause the LCD to leave a blank space and for “Hello” and “Hi” to be spaced out. This makes strings easier to read on the LCD. Here is how the code looks with an extra space in the “Hello” string:



```
#include <pololu/3pi.h>

int main()
{
    print("Hello ");
    print("Hi");
    while(1);
}
```

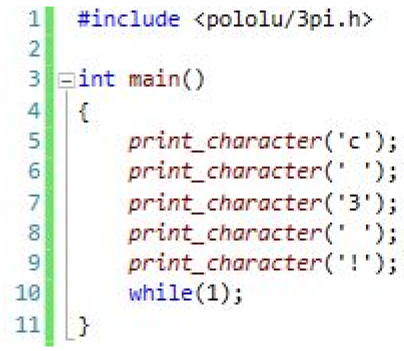
Prints “Hello Hi” to the LCD

Not only can you print strings, but you can also print numbers and characters with different printing functions. The two functions that print numbers and characters are “print_long” and “print_character”. “print_long” will print numbers. A “long” is a way to describe a big number, so print_long can print big numbers but also small ones as well. “print_character” will print characters, which are individual keyboard letters, numbers, and symbols surrounded by single quote (‘) marks. Here are what some characters look like: ‘a’, ‘b’, ‘3’, ‘!’. They are different from strings since strings use the double quote marks (“) and can have more than 1 letter, number, or symbol in between the quotation marks. Here are examples printing numbers and characters.



```
#include <pololu/3pi.h>

int main()
{
    print_long(12345678);
    while(1);
}
```

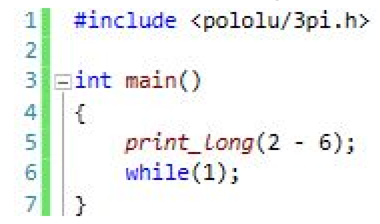
Prints 12345678 to the LCD.



```
#include <pololu/3pi.h>

int main()
{
    print_character('c');
    print_character(' ');
    print_character('3');
    print_character(' ');
    print_character('!');
    while(1);
}
```

Prints a few different characters to the LCD.

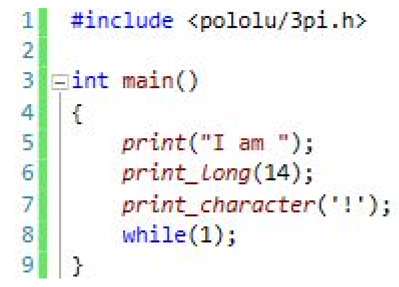
"print_long" can also be used to display negative numbers and the result of an expression. For example, print_long(3 + 3) would display 6 on the LCD, print_long(3 * 3) would display 9 on the LCD, print_long (3 * -3) would display -9 on the LCD, and print_long(18 - 19) would display -1 on the LCD. Here is an example using 2 - 6:



```
#include <pololu/3pi.h>

int main()
{
    print_long(2 - 6);
    while(1);
}
```

Prints the result of 2 - 6 to the LCD.

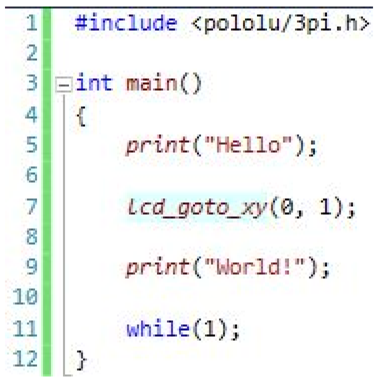
All 3 of these print functions can be used together to display text on the LCD like so:



```
#include <pololu/3pi.h>

int main()
{
    print("I am ");
    print_long(14);
    print_character('!');
    while(1);
}
```

Prints a message about age.

When trying to print something longer than "Hello", like "Hello World!", you will notice that the entire message does not fit on the LCD. The size of the LCD is 8x2 characters, meaning 8 characters are displayed on each row and there are 2 rows. If you try to print "Hello World!", only "Hello Wo" is displayed on the LCD. We can use a different function called "lcd_goto_xy" which allows us to pick where on the LCD a string can be printed. We will also use another call to the print function to split printing "Hello World!" between 2 print functions. An example is shown below.

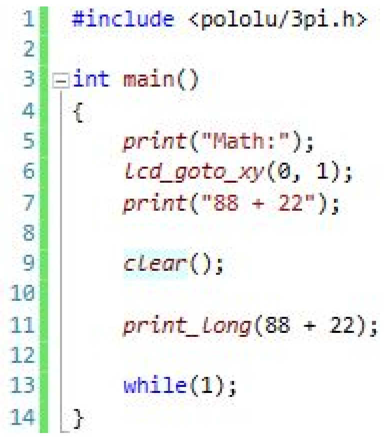


```c
#include <pololu/3pi.h>

int main()
{
    print("Hello");

    lcd_goto_xy(0, 1);

    print("World!");

    while(1);
}
```

Prints “Hello” on the first row then “World!” on the second row of the LCD.

The function lcd_goto_xy picked the first spot in the second row to start printing from, so when the function print was called to print “World!”, it was printed from the second row. Again, you will learn more about functions in future sections. The first number in the function lcd_goto_xy chooses which column to start printing from and the second number chooses which row, where 0 is the first column or row and 1 is the next column or row, and so on.

If you wish to print more text when the LCD is full or want to display text without any of the previous text, you can clear the entire LCD with the function “clear()”. After the screen is cleared, the next print function will start to print from the start of the LCD. The code below will print a string of a math expression, clear the LCD, then print the result of the expression.



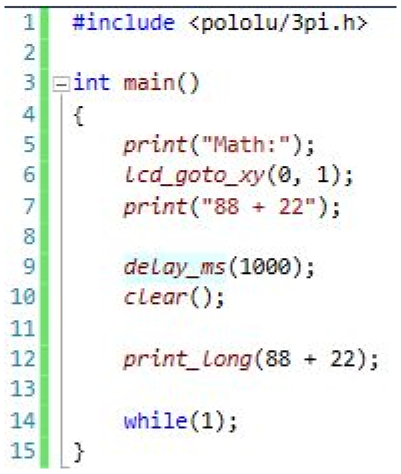
```c
#include <pololu/3pi.h>

int main()
{
    print("Math:");
    lcd_goto_xy(0, 1);
    print("88 + 22");

    clear();

    print_long(88 + 22);

    while(1);
}
```

Prints an expression then clears screen and prints answer

You may notice that the LCD only displays the result of the expression although according to the code the LCD should display the expression first. The code is displaying the text but the speed at which the code is executed is so fast you only see the result of the expression. A function can be used to help delay the execution of the “clear()” function.

This function is “delay_ms”. “delay_ms” delays the execution of the next lines of code by the specified amount of time (in milliseconds). 1000 milliseconds is equal to 1 second.

“delay_ms” also has another name that does the same thing called “delay”. Since it is more obvious that “delay_ms” will delay execution of the program by milliseconds, using “delay_ms” instead of “delay” could make reading code easier. We will typically be using “delay_ms” in our code. Here the delay function is used so that you can see the math expression from above for a second before you see the answer.



```
#include <pololu/3pi.h>

int main()
{
    print("Math:");
    lcd_goto_xy(0, 1);
    print("88 + 22");

    delay_ms(1000);
    clear();

    print_long(88 + 22);

    while(1);
}
```

Prints an expression, delays clearing of the LCD for 1 second, then prints the result.

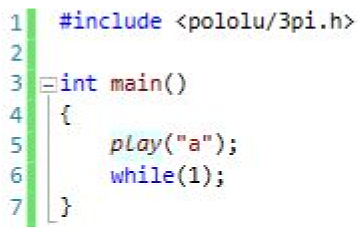
The delay function will not affect previously executed code. The text that was printed before the delay function will still be displayed on the screen. If the robot was playing music, the music would still be playing while the robot is delaying the next lines of code. You will see this in the next section.

List of introduced functions

- print(const char *str) - prints a string
- print_long(long value) - prints a long type number
- print_character(char c) - prints a character
- lcd_goto_xy(unsigned char col, unsigned char row) - changes where text will be printed on the LCD
- clear() - clears the LCD
- delay_ms(unsigned int milliseconds) - delays the execution of the next lines of code

## 1.1.2 Music

The “buzzer” of the 3pi robot allows it to play sounds and music. You can use this to make the robot more interesting and to help you notice when the robot has finished a part of the program. For example, you can play a sound at the start of a program so you know the program started running on the robot. Here is an example of the robot playing a single musical note:



```
#include <pololu/3pi.h>

int main()
{
    play("a");
    while(1);
}
```

Plays music note “a”.

The “play” function is used to play sounds and music. A string consisting of musical notes is used with “play” to make the 3pi play sounds and music.

## 1.1.3 LED

## 1.2 Variables

The C programming language gives us a handful of different variable types that are useful for solving a multitude of different problems. We can use variables to store values that we wish to use later in our program, or we can manipulate the variables if we need to keep track of changes with our 3pi, or display values that we want to monitor. Since the C programming language doesn’t have a “boolean” value, we will only cover two different types of variables: numerical, and text.

### 1.2.1 Numerical Values

We will begin with the simplest numerical data type: integers. Integers, represented as `int` in the C language, are used to store both positive and negative numbers. We initialize an integer by doing the following:

```
int x = 2;
```

In this example, we have set up a variable called ‘x,’ and we’ve set the value of x to 2. We can add, subtract, multiply, and divide these integers by other constants and variables. For instance:

```
int x = 2;
int y = 2+2;
int z = x+y;
```

In this example, our variable ‘x’ is equal to 2, our variable ‘y’ is equal to 2+2, which means that y is equal to 4, and our variable ‘z’ is equal to x+y, which means that z is equal to 6, because x = 2, and y = 4.

Integers have a weakness, however. If we want to represent the number (1/4) or (0.25), we can’t do this using an integer. Integer division removes everything after the decimal, meaning that the integer value of 0.25 is equal to 0.

Fortunately we have a way to represent 0.25 using two different data types: floats, and doubles. While these two data types may look as though they're different, they will be treated as though they are the same for the sake of this chapter. We can initialize doubles and floats by doing the following:

```
double x = 2.0;
float y = 3.0;
```

In this example the variable 'x' is a double with a value of 2.0, and the variable 'y' is a double with the value of 3.0. Just like integers, these variables can be added, subtracted, multiplied, and divided by other doubles and floats, as well as other variables. For instance:

```
double x = 1.5;
float y = 3.0;
double z = x*2;
float a = y++;
```

In this example the variable 'x' is equal to 1.5, the variable 'y' is equal to 3.0, the variable 'z' is equal to x*2 (x multiplied by 2) which equals 3.0, and the variable 'a' is equal to y++ which is equal to 4.0.

It's important to note that the '++' operator will increment integers, doubles, and floats by exactly 1. There are several other types of incrementation similar to this:

```
x++; // Increments x by 1
x--; // Decrements x by 1
x+=2; // Increments x by 2
x-=2; // Decrements x by 2
x*=2; // Equivalent to saying x=x*2
x/=2; // Equivalent to saying x=x/2
```

Finally, the last numerical data type we will use is called a long. A `long`, for the most part, is similar to an integer, except that it can hold a much larger integer value than the `int` data type can. We can initialize longs by doing the following:

```
long x = 2;
```

In this example the variable 'x' is a `long` with the value of 2.

### 1.2.2 Strings and Characters

Two other variable types that C allows, are strings and characters. Strings are actually an array of characters, but they aren't initialized like an array. We can initialize strings by doing the following:

```
string x = “This is a string”;
```

In this example the variable ‘x’ is a string with the value, “This is a string.” Strings can be added to other strings and even integers. If an integer is added to a string, the integer value is turned into a string value and appended immediately at the end of the string.

Characters, denoted as char, are essentially a piece of a string; they are one single ASCII character. We can initialize characters by doing the following:

```
char x = ‘a’;
```

Notice that in this example we are using single quotes instead of the double quotes we used to initialize the string -- this is necessary for the compiler to understand that this is a character. In this example, the variable ‘x’ is a character with the value ‘a’. It’s important to note that characters also have integer values, and they can be treated as though they are integers -- this will become important in later sections.

### 1.2.3 Finding and Displaying the Battery Life

Now that we know how to use strings and integers, let’s use them to help us figure out what the battery life of our 3pi robot is.

First, we set up an integer called ‘bat’ :

```
int bat;
```

Then, we use a method, provided by Pololu, called read_battery_millivolts(), and we set our ‘bat’ integer equal to the integer returned by this method :

```
bat = read_battery_millivolts();
```

If we want, we can consolidate these two steps into one:

```
int bat = read_battery_millivolts();
```

Next, we want to clear the screen on our 3pi, and print out our ‘bat’ value:

```
clear();
print_long(bat);
```

Next, let’s create a string called ‘units’, set it equal to “mV”, and display the units underneath the battery life:

```
string units = “mV”;
lcd_goto_xy(0, 1);
```

```
print(units);
```

Now, compile and run to see what the battery life of your 3pi robot is. Your final code should look like this:

```
int bat = read_battery_millivolts();
clear();
print_long(bat);
string units = “mV”;
lcd_goto_xy(0, 1);
print(units);
```

## 1.3 Selections and Loops

### 1.3.1 while, if-else, switch, and Buttons

In order to execute code after a button has been pressed, we have to use what is called a while loop. A while loop continues to loop a segment of code until some condition is met. In the context of the 3pi, Pololu has provided us with a method called button_is_pressed(), which will serve as the condition for the while loop :

```
while(!button_is_pressed(BUTTON_B)) {
      // do nothing
}
```

In this example, the while loop is waiting for the ‘B’ button on the 3pi to be pressed. The ‘!’ before the condition means that the condition is the *opposite* of whatever value we’re receiving. In this case, button_is_pressed() is false until we press the button.

While loops only continue looping their code if the condition is true. So, in order to loop infinitely while we wait for a button, we must invert the condition, thus we get !button_is_pressed(), which is true until we press the button.

We can execute this code in a different way, as well.

```
while(1) {
      if(button_is_pressed(BUTTON_B) {
            break;
      } else {
            // do nothing
      }
}
```

This example is exactly the same as the first example, except for the fact that this one uses what is called an if-else statement. The if statement works a bit like the while loop; it executes a bit of code provided that some condition is true. In this example, if the button is pressed, we do what's called a 'break', which means that we break out of our while loop.

In both examples, code that is placed after the while loop will run only after the 'B' button is pressed.

Another important type of loop is called a for loop. The for loop iterates through of a series of numbers until some condition is met:

```
int i;
for(i=0; i<10; i++) {
      // code to be run here
}
```

In this example we have an integer called 'i' that is set to 0. While the integer 'i' is less than the value 10, 'i' will be incremented by 1 until the condition is met. The above loop can also be written like this:

```
int i=0;
while(i<10) {
      // code to be run here
      i++;
}
```

Note that the code to be run is put *before* 'i' is incremented. This is exactly how the for loop works. The block of code is run first, then the integer is changed. Here is an example of how you could make a counter on the lcd screen

```
int i;
for(i=0; i<8; i++) {
      lcd_goto_xy(i, 0);
      print_long(i);
}
```

In this example we're creating an integer 'i' that starts at 0 and increments by one until it reaches the number 7 (Note: since the condition for this loop is '< 8', we will never reach the number 8). On each iteration, the integer value of 'i' will be printed on the LCD at its correct position on the screen.

Let's now return to the if-else statement. This statement is called a selection statement, and there are three different types of selection statements

```
if(condition) {
      // code
} else {
     //code
}


if(condition) {
     // code
} else if(condition) {
     // code
} else {
     // code
}


switch(variable) {
     case x:
          // code
     break;

     default:
          // code
     break;
}
```

The first selection statement is the if-else statement, which executes some code if some condition is true, and executes a different block of code if the condition is false. The next statement is essentially the same as the if-else statement, except that instead of one condition that can be true, there are now two conditions that can true. It's important to note that in the if-else, and the if-else if statements, there isn't actually a need to write the else, unless you only want that block of code to be executed if and only if the condition is false. For instance:

```
if(x == 2) {
     x++;
}

x *= 2;
```

In this example, x will *only* be incremented if x is equal to two, *and then* it will be multiplied by 2. However, if we change the code to this...

```
if(x == 2) {
     x++;
} else {
        x *= 2;
}
```

...the output is different. In this example x will *only* be incremented by one if x is equal to 2, but it *will not* be multiplied by two afterword, unlike in the previous example.

In regard to the if-else if statement, there can be as many ‘else if’ statements as necessary. It’s important to note that if multiple conditions are true, only the first condition will be executed.

```
if(x == 2) {
        x++;
} else if(x%2 == 0) {
        x--;
} else {
        x*=3;
}
```

In this example if x is equal to 2, there are two conditions that are met: ‘x==2’ and ‘x%2 ==0’. However, ‘x==2’ is the first condition that is met, so x will be incremented by 1.

Another useful selection statement is called a switch statement. The switch statement doesn’t take any conditions like the if-else statement, instead it takes integers, characters, and strings. It’s important to note that doubles and floats *do not* work with switch statements, as they are unreliable whenever they are compared.

```
int x=2;
switch(x) {
        case 2:
                x++;
        break;
}
```

In this example we have a switch statement that takes in the value of x. In the case that x is equal to 2, the x variable will be incremented. Note that this is exactly equivalent to:

```
if(x == 2) {
        x++;
}
```

Switch statements often have a ‘default’ case. This is the case that will be executed whenever no other case matches -- somewhat like an else statement. Switch statements are very useful when dealing with multiple cases, for instance:

```
switch(x) {
        case 2:
              x++;
        case 3:
        case 10:
              x++;
        break;

        default:
              x*=2;
        break;
  }
```

In this example we see that we’re dealing with case 2, 3 and 10. At case 2, x will be incremented by one, and, because there isn’t a break, the code for case 3 and 10 will *also* be executed. At case 3, the code for case 10 will be executed. At case 10, x will be incremented by one. After this code is run, if x=2, x will become 4, if x=3, x will become 4, and if x=10, x will become 11. If x is any other value, x will be multiplied by 2;

### 1.3.2 Play a Note With a Button

One important concept in robotics programming is an infinite loop. In the previous section we wrote:

```
while(1) {
      // code here
}
```

We see here that the condition for the while loop is always true (or 1), which means that the code within the brackets of the while loop will always run until we break. This is important in programs that are waiting for some sort of input, or some condition to be met.

```
while(1) {
      if(button_is_pressed(BUTTON_B) {
            play(“c”);
      }
}
```

In this example our 3pi will wait forever for us to press the B button. If the button is pressed, a note is played. If we release the button, the note will stop. These infinite loops have plenty of applications later on when we start navigating the Grid World.

## 1.4 Methods

### 1.4.1 Make a Method to Play a Song

## 2. Introduction to 3pi Navigation

2.1 Basic Navigation with the 3pi
- 2.1.1 gotoPoint()
- 2.1.2 gotoEdge()
- 2.1.3 gotoCorner()
- 2.1.4 move()

2.2 Navigation and Variables
- 2.2.1 Go to Locations using variables
- 2.2.2 Using Constants

2.3 Navigation with Selections and Loops
- 2.3.1 Zig-Zag
- 2.3.2 Square
- 2.3.3 Infinite Square
- 2.3.4 Square With Conditions

2.4 Navigation with Methods
- 2.4.1 Create a Zig-Zag method
- 2.4.2 Create a square(int w, int h) Method
- 2.4.3

## 3. Problems Using What We've Learned

Sdf