The Benefits of Modular Programming
David Culp
DeWitt Perry Middle School
culpd@cfbisd.edu

The Benefits of Modular Programming

1 Modular Programming

I have been involved with Botball for the last five years and one of the biggest frustrations to me is teaching kids to properly organize their code. In general most kids tend to program in a linear fashion and end up with giant, monolithic programs that are hard to maintain and edit. A much better method is to use modular programming techniques to make the code more readable, reusable and editable. If students can be taught to program in a modular manner, their frustration level will go down and their confidence and success rate will go up. This paper will show students (and teachers) some very basic modular programming techniques. This paper is not an exhaustive treatment of modular programming and only basic techniques will be covered.

1.1 Technique #1: using #define Statements

The easiest technique to make programs easier to read and edit is to use #define statements. In essence, a #define statement defines a word or piece of text to mean something to the compiler. The best way to understand a #define statement is to see an example in action. Consider the following typical program written by beginning programmers:

```
void main()
{
   motor(0,100);
   motor(2,100);
   sleep(2.0);

   motor(0,-100);
   motor(2,100);
   sleep(.75);

   motor(0,-100);
   motor(2,-100);
   sleep(3.0);

   ao();
}
```

The program is simple and straight forward. However, it is hard to read and understand exactly what is happening, for example, what do motors 0 and 2 correspond to? In addition, what happens if, during the course of development, the motors that correspond to ports 0 and 2 are moved to other ports, a lot of tedious and error prone editing must take place, especially if the program is very long. We can solve all of these problems by employing #define statements in their simplest form. Consider the rewritten program:

```
#define LEFT_MOTOR 0
#define RIGHT_MOTOR 2

void main()
{
   motor(LEFT_MOTOR,100);
   motor(RIGHT_MOTOR,100);
   sleep(2.0);

   motor(LEFT_MOTOR,-100);
   motor(RIGHT_MOTOR,100);
   sleep(.75);

   motor(LEFT_MOTOR,-100);
   motor(RIGHT_MOTOR,-100);
   sleep(3.0);

   ao();
}
```

The program is easier to read and understand and if a change is made to the robot, such as moving the left motor to port #3 only one change needs to be made at the top of the program.

Essentially what happens is that at compile time, the compiler searches through the program and replaces every instance of text “LEFT_MOTOR” with the text “0” and every instance of the text “RIGHT_MOTOR” with the text “2”. Note that I used all upper case in the #define statement but this is not necessary, however it is traditional in C programming to use all upper case with defines in order to differentiate them from variables when we read the program. #define statements are easy to use but they must appear at the top of the program.

1.2 Technique #2: Functions

Beginning programmers tend to write giant, linear, stream-of-consciousness programs. These programs grow huge, ungainly, difficult to edit and maintain. In addition, programmers (especially in Botball) tend to write the same pieces of code over and over again. I tell my students that laziness is one of the traits of great programmers, we don't like to do things over and over, and over again. Learning to use functions can cut down on the repetition of programming and make programs easier to read and edit. Go back and read our last program. *void main()*

After reading the program it appears the robot will do the following:

1. Go forward for two seconds
2. Turn left for .75 seconds
3. Drive backward for 3.0 seconds

It gets very boring and tedious to continuously type the exact same three lines of code every time you want the robot to maneuver in a particular way. In addition, imagine a 150 line program made of nothing by motor and sleep functions. The program is hard to read and difficult to edit. For example, if your robot does not do what you expect it to do 45 seconds into its routine, it is a pain to wade through 150 lines of motor and sleep functions to figure out where it went wrong and correct it. Fortunately, C provides a way for us to ease these problems and the solution is to use functions. The best way to understand functions is to see our program rewritten using functions:

```
#define LEFT_MOTOR 0
#define RIGHT_MOTOR 2

void main()
{
   forward();
   sleep(2.0);

   left();
   sleep(.75);

   reverse();
   sleep(3.0);

   ao();
}

void forward()
{
   motor(LEFT_MOTOR,100);
   motor(RIGHT_MOTOR,100);
}

void left()
{
   motor(LEFT_MOTOR,-100);
   motor(RIGHT_MOTOR,100);
}

void reverse()
{
   motor(LEFT_MOTOR,-100);
   motor(RIGHT_MOTOR,-100);
}
```

The program becomes much easier to read and understand and now, when we want our robot to maneuver we just need to use the function we have written for the particular direction we wish to go. When the program gets to a point in which we "call" or use our function it will jump down to our function and execute the code in the function contained between the open and close braces for that function.

A basic function in C looks like the following:

```
void function_name()
{
   stuff we want the function to do goes here
}
```

We can actually do more advanced things with functions. We can, if we wish, pass a value to a function and use that value. For instance, what if we wished to pass the amount of time we wish to sleep to the function instead of having to type a sleep function every time we use our movement functions. We can rewrite our functions and program as follows:

```
#define LEFT_MOTOR 0
#define RIGHT_MOTOR 2

void main()
{
   forward(2.0);
   left(.75);
   reverse(3.0);

   ao();
}

void forward(float sleep_time)
{
   motor(LEFT_MOTOR,100);
   motor(RIGHT_MOTOR,100);
   sleep(sleep_time);
}
```

```
void left(float sleep_time)
{
   motor(LEFT_MOTOR,-100);
   motor(RIGHT_MOTOR,100);
   sleep(sleep_time);
}

void reverse(float sleep_time)
{
   motor(LEFT_MOTOR,-100);
   motor(RIGHT_MOTOR,-100);
   sleep(sleep_time);
}
```

The addition of the “float sleep_time” argument in the function definition basically means “We are going to pass a floating point number to this function and call it sleep_time” we can then use the “sleep_time” variable in our function.

We can also get information back from a function. Consider the following function:

```
int add_two_ints(int a, int b)
{
   return a+b;
}
```

The name of the function is “add_two_ints” and takes two integer numbers called “a” and “b”. However, notice that instead of the word “void” in front we have replaced it with “int”. This means the function will return an int type. In the function the line “return a+b;” will return the integer type of a added to b. We could use the function in the following manner:

```
c=add_two_ints(12, 16);
```

There is a lot more possible with functions and the reader is advised to read the IC manual.

1.3 Technique #3: Comments

Using comments is not specifically a modular programming technique but a good programming technique to employ. Comments make a program more understandable and much easier to edit. I am stickler for good comments, in fact, if a student needs help I will not help them unless their program is fully and properly commented.

In essence, to do a single line comment in IC you use a // on the line. Every piece of text after the // will not be downloaded into your robot but is there to help you and others understand what the program is doing.

The other commenting technique in IC is the block comment. To start a block comment use /* and to end the block comment use */. Here is our program using both types of comments:

```
/*
Program name: Example.ic
Creation Date: June 4th, 2007
Author: David Culp (culpd@cfbisd.edu)
Purpose: To show different modular programming techniques
*/

#define LEFT_MOTOR 0  // Change these for your robot
#define RIGHT_MOTOR 2

void main()
{

   // Drive to the first goal
   forward(2.0);
   left(.75);
   reverse(3.0);
   //=========================

   // Drive to the second goal
   left(.2);
   reverse(1.75);
   forward(2.5);
   //=========================

   ao();
}
```

2 Conclusion

Modular programming is one of the single most powerful techniques students can use to increase their success in Botball. Using modular programming will make their programs easier to read, edit and maintain. In addition, in the high pressure atmosphere of the actual competition where last minute emergency code changes can make or break a team, modular programming can increase their chances of making the correct changes. This paper has only scratched the surface of modular programming techniques and the reader is encouraged to explore more, especially by reading the IC manual.