Modularization of Botball Libraries
Kevin Cotrone, Garrett Sickles, and Marshall Parker
Norman Advanced Robotics
kevincotrone@gmail.com, garrettsickles@gmail.com, runningjmp@gmail.com

Modularization of Botball Libraries

Introduction

The point behind modular coding in Botball is to allow programmers to reflect the mechanical design of their robot, but more importantly enable code to be easily handed down and assembled year after year no matter who the programmer may be. By producing code in modules, teams are able to easily organize and produce standardized libraries. Maybe you create a programming module to gather and interpret E.T. data and then pass the results to your main program logic which then operates a servo module. Basically, it is a lot easier to build a house from bricks (Modules) and mortar (logic) you have already created than to spend countless hours working out how to build a brick and how to make mortar every year.

What is Modular Code

Modular Code separates different functionalities from each other in a logical, convenient manner. One way we use modular code is to affect multiples of the same or similar function. What this means is instead of making very specific, single use code, take the time to build functions with robust, multi purpose functionality that can be used in varying circumstances. In the long run this cuts development and debugging time by increasing the effectiveness of your code.This allows programmers to easily find what needs to be changed because it is not the functions' actual construction, but instead the parameters you are passing to it. In the future it will be much easier to interpret your code if it is engineered modularly because it allows you to see how the changes made to each module affect the outcome of your overall program. [1]

Why Modular Code is Best

The modular open source code libraries Norman Advanced has created offer code for everything from reading and time averaging sensor values to moving servos with variable speed and pinpoint precision. The libraries offer also a pseudo middle ground between object oriented code, like Java or C++, and procedural code, like C. We achieve this by structuring the different components of the robot into memory organizations like structures and unions. Our modular code allows you to include only what you need while using code that every robot needs. While you might not use every function of every library, having the code to use a servo in every way, read a sensor in different ways, or to move distances with a CBC can clean up the main program that runs a robot and keeps code consistent year to year. This means that each robot can have the same code for servos and sensors, enabling you to contribute and work together with identical code and therefore reducing programming and testing time.
In the same way that Java offers modularization in the form of packages, Botball Open Code offers modularization through our header files. You can use both C and Java with the CBC but the Botball Open Code was created in and intended for use in C. Using the Botball Open Code libraries enables code to be fluent in structure and easier to manipulate. With libraries for create movement, create scripting, CBC navigation, servo movement, and both analog and digital sensors, you're able to control all pieces of the Botball kit more efficiently than with the standard

libraries.

Something else we have incorporated into the design is pseudo-object oriented functionality to limit the abstraction of servos, motors, and sensors to a small, well-defined parameter set which is both intuitive and easily to manipulate. The functions we created to control different aspects of the robot still provide robust access to all the robot's features.

Influences of Mechanical Design on Modular Code

The entire reason we believe modular code is especially effective in Botball is because it mirrors the same structure and design of the physical robot. Builders on most Botball teams reuse specific designs for motors, servos, claws, arms, etc... and they may even recycle certain designs year after year. Modular code is the same as these mechanical designs except the libraries we have developed are open source and formulaic, allowing any botball team to utilize their functionality.

Modular code can also also be useful though because it enables programmers to build and share code in a very easy manner.

How to Apply Modular Code to Your Robot

Modular code can be exciting to apply to your robot because it can be challenging and push you to learn and think more critically about how you design and build robots and code. For instance, if you were to build a claw and want to write code to raise and lower this claw, instead of making completely specific functions to do this, you could use preassembled servo functions that can already operate your claw to modularize code and have more coherence. The other way to apply modular code to your robot is by organizing the different code modules into multiple header files. By doing so, you are able to improve the coherence of your library in an extremely orderly, logical, and modular way.

Examples

Using the Open Code is a little confusing at first but it's easy to grasp the concept.

The movement of a CBC robot with the cbcnavlib.h library is fairly simple after measuring the robot's wheel diameter and wheel base.

```
int main()
{
    struct cbcRobot robot = {0,3,150,55,1100};  //Define the robot. 0 and 3 are the motor ports,
                                                //150 mm is the wheel distance
                                                //55 is wheel diameter, 1100 is ticks per rotation
    set_servo_position(arm_servo, top_pos);

    msleep(5000);

    shut_down_in(118.0);

    cbc_align(400);                             //Align with both of the top hats on the black line
```

Cbcnavlib.h also allows you to move in the same manner as the create using the cbc_arc, cbc_straight, and cbc_spin functions. The cbcnavlib allows you to use distances instead of abstract values such as ticks.

Similarly, building and moving a servo is simple.

```
    ////////////////////////////////////
    //Score both of the balls in the MPA//
    ////////////////////////////////////
    create_spin_angle(200,-132); //Spin CW towards the MPA

    ballarm = build_servo(1,0,1800,12,8); //Rebuild the ballarm servo
    msleep(1000); //Sleep because we can
    wait_servo(ballarm,260); //Lower the ballarm, depositing both balls into the MPA
    wait_servo(ballarm,1600); //Raise the ballarm
    msleep(1000); //Sleep because we can
```

This code efficiently creates a servo that you can call quickly with the wait_servo function that moves the servo fluidly to the desired position.

The Open Code library has functions for create movement as well.

```
int main()
{
    create_connect(); //Connect to the create

    servo kelparm = build_servo(0, 0, 1911, 20, 5); //Build the kelparm servo
    servo ballarm = build_servo(1, 0, 1401, 20, 5); //Build the ballarm servo

    ////////////////////////////////////
    //Move into Kelp Grabbing Position//
    ////////////////////////////////////
    create_drive_segment(200,400); //Exit the Start Box
    create_drive_arc(200, 375, 180.5); //Arc to the Kelp
    create_drive_segment(200, 226); //Center on the Kelp
    create_spin_angle(200, 85); //Face the Kelp
    create_drive_touch(-200, -200, 14, 13); //Bump sense the kelp wall
    create_drive_segment(200, 300); //Drive away from the kelp wall

    wait_servo(liftarm, 800); //Undo the kelparm quick release
    wait_servo(liftarm, 1910); //Lower the Kelparm to the minimum position

    create_drive_segment(200, -304); //Drive into the kelp wall

    ////////////////////////////////////
    //Grab the kelp with the kelparm//
    ////////////////////////////////////
    get_kelp(); //Run the sequence to aquire kelp

    ////////////////////////////////////
    //Score the kelp in the start box//
    ////////////////////////////////////
    create_drive_segment(100, 50); //Back away from the kelp wall
    create_spin_angle(120, 173);    //Turn towards the start box

    wait_servo(liftarm, 1250); //Stabalize the kelp on the ground

    create_drive_segment(75, -90); //Drive to start box

    wait_servo(liftarm, 1300); //Move the Kelparm into the dump position

    create_drive_segment(170, -215); //Deposit the kelp in the start box

    liftarm = build_servo(0, 0, 1710, 20, 10); //Rebuil the Servo to move more quickly
```

You're able to use functions that use distances instead of speed and time. These functions clean up code and allow you to arc and spin easier than with timed functions.

The modular code libraries can be cloned from https://github.com/garrettsickles/BotballOpenCode.git

Sources

[1] http://www.eng.fsu.edu/~dommelen/courses/cpm/notes/progreq/node2.html