Effective Coding Practices
Benjin Dubishar
Dead Robot Society
shadowthunder@gmail.com

Effective Coding Practices

1. Introduction

Writing software without proper and consistent coding conventions is, quite simply, a bad idea. Not only does it cause immediate and up-front issues such as being significantly more difficult to code correctly the first time through, but it causes the programmer to shoot himself/herself in the foot in the long run as well. This paper explains the benefits of several coding practices that the Dead Robot Society's software team makes use of to be as effective during the Botball season and during competition as possible.

2. State Machine

2.1. Description of a State Machine

As the Dead Robot Society uses it, a State Machine is the highest level block of code, taking up the entire main() of our programs. It consists of three primary parts: the opening sequence (or "initialization"), the conditional statements, and the closing sequence ("shutdown").

The initialization prepares the robot for that round by enabling servos and the camera, holding all arms, cages, claws, etc. in their respective starting positions, calibrating sensors and color models, and finally waiting for the start light. Once *wait_for_light(int port)* has been called, the robot sits there, patiently awaiting once-overs of the board and hardware and shaking hands with the other team before launching into the bulk of the program.

The set of conditional statements is a series of methods which will run in an order dependent on how the previous instruction reacted to the game environment.

```
if(state == inStartBox_E)   state = moveToIsland();
```

The above line is a sample State Machine conditional. This particular line would likely be the first to execute because it would only be run when the robot is in its starting position and facing East (an arbitrary direction on the game board to assist with navigation). At the end of *moveToIsland(),* the next state – something helpfully-named such as "atIsland_S" - would be returned and the appropriate method would then be run.

The shutdown sequence's primary purpose is to do everything possible to avoid being disqualified by twitching, spinning, grabbing, etc. after the allotted time has run out. After the last state has been executed, "finished_U" (with the 'U' standing for Unknown position) is returned, and this signals the robot to disable servos, stop all motors, and disconnect from the

Create. The Dead Robot Society makes sure to call this method with ample time remaining in the game to complete all of the appropriate shutdown tasks; five seconds before the end is sufficient.

2.2. Benefits of a State Machine

2.2.1. Organization and Cleanliness

Having a dominant and consistent structure for code allows the programmer to quickly "drill down to" or find a specific line or method that is causing issues. A properly-coded State Machine mandates that all variables and methods are contextually-named which enables any other programmer – one less familiar with the code – to pick up where the previous person left off with a minimal learning curve. If an entire sequence needs removal for testing purposes, only one line needs to be commented out instead of twenty (or more), and it's obvious which line that is.

2.2.2. Code Branching

When "testing the snot out of" code (as one of our mentors says), not everything prior to that one section needs to be run each time. With a State Machine, altering the starting location in your code is literally a one-line change – have the initialization state return "facingOilSlicks_N" instead of "inStartBox_E". A more complex scenario might involve two distinctly different courses of action depending on whether or not Botguy is in the expected location. There's no reason to 'move Botguy to your start-box' if there is, in fact, no Botguy in your possession to move; perhaps you'd prefer to attempt to steal Botguy back, and *then* proceed with scoring. In a State Machine, a different output state can be returned depending on what was accomplished within that state – therefore "branching" the code – and, once Botguy is back within your grasps, pick up where you left off.

2.2.3. Error States

Similar to Code Branching, Error States are permanent branches; that is, they never rejoin with the main program. Also called "Fail States", these are used when something goes irrecoverably wrong, and you can't risk this rogue robot causing a disqualification or causing the other robot to lose orientation. Sometimes, it's simply much safer for a robot to find its way into a corner – any corner – and run the shutdown code immediately.

2.3. State Machine Example

```
ÏÞßàvoid main(){
ÏÏ¨¹¹Ïprintf("Initializing Code\n");
ÏÏ¨¹¹Ïstate = initialize();//prepare for battle, check ports, raise fence...
ÏÏ¨¹¹Ïwait_for_light();//calibrate the start light sensor and wait for the beginning of the game
ÏÏ¨¹¹Ïshut_down_in(118);//End a few seconds before the end of the time allowed just in case.
ÏÏ§
ÏÏ§//while we aren't finished continue looping and checking your state value
ÏÏ¨¹¹±while(state != finished_U){
ÏÏ§ÏÏ5//signal a new loop
ÏÏ§ÏÏ7¹¹Ïbeep();
ÏÏ§ÏÏ7¹³¹if(state == inStartBox_W)       state = moveToIsland();
ÏÏ§ÏÏ7¹³¹if(state == atIsland_E)         state = squareOnIsland();
ÏÏ§ÏÏ7¹³¹if(state == squaredOnIsland_E)  state = sweepIsland();
ÏÏ§ÏÏ7¹³¹if(state == sweptIsland_SE)     state = moveToFrogs();
ÏÏ§ÏÏ7¹³¹if(state == atFrogs_S)          state = gatherFrogs();
ÏÏ§ÏÏ7¹³¹if(state == canHasFrogs_S)      state = moveFrogsToStartBox();
ÏÏ§ÏÏ7¹³¹if(state == scoredFrogs_S)      state = moveToWaitPosition();
ÏÏ§ÏÏ°}
ÏÏ§//done
ÏÏ¨¹¹Ïshutdown();
ÏÏ©}
```

3. Split Code Into Smaller Methods

3.1. Benefits of Splitting into Methods

Every novice coder has had this issue: lines of code are written one after another for possibly hundreds in a row until even the trained eye sees little other than an intimidating screen full of blocky code (but don't worry; of *course* it's commented, right?). In order to figure out where in the program you're looking, you'd literally have to trace your way through the entire thing and, if you were to get lost, you'd have to start over. Every programming language allows for user-defined methods. Calling *wait_for_light(int port)* doesn't magically know to check the value of the specified analog port whenever a specific button is pressed, then wait until the value peaks again; it calls a set of twentyish lines of code which do each of those instructions individually, but it only takes one line of code to call the entire sequence.

Instead of having every single instruction follow immediately after the previous one, put all of the code required to move from the start box to the first objective in one method called "*moveToDirtyDucks()*". And there's no reason to stop there. Replace all of your motor commands with "*drive(int speed)*" methods that call the lower-level motor commands. A good 'rule of thumb' is to put code into a method if it is used more than once. This way, you only have to change code one place in order to update everywhere that it's used.

3.2. Method-Based Code Example

In this example, almost every method called is one that was written by the Dead Robot Society software team, rather than a lower-level instruction taken directly out of the KIPR-provided library. The *move(int speed, int distance)*, *turn(int speed, int degrees)*, and *setArm(int armPosition)* commands are comprised of those KIPR instructions, but it is easier to code, debug, and read with them compiled into methods. All of this code is contained in the *moveToIsland()* command so that the State Machine can perform many basic actions with only the one line.

```
// Output: Returns the current location in the state machine (that the robot
is facing the island)
// Side Effects: ducks pushed out of the way, jackscrew arm raised, in
position in front of the island
ÏÞßàint moveToIsland(){
ÏÏ¨¹¹Ïprintf("moveToIsland\n");
ÏÏ¨¹¹ÏtimeOfStart = seconds(); // start a counter for timing the run
ÏÏ¨¹¹Ïmove(70, 3);// positive numbers move the Createbot forward
ÏÏ¨¹¹ÏsetArm(armUp); // begin raising the jackscrew arm (non-blocking)
ÏÏ¨¹¹Ïturn(-70,-85);// negative numbers turn the Createbot clockwise
ÏÏ¨¹¹Ïmove(70, 13);
ÏÏ¨¹¹ÏsetFence(duckSetting); // plow any ducks out of the way
ÏÏ¨¹¹Ïmove(70, 8);
ÏÏ¨¹¹Ïturn(-70, -45);
ÏÏ¨¹¹Ïmove(70, 12);
ÏÏ¨¹¹ÏsetFence(upSetting);// raise fence to get a better square up
ÏÏ¨¹¹Ïmove(70, 4);
ÏÏ¨¹¹Ïturn(-70, -23);// finish turning toward the island
ÏÏ¨¹¹Ïmove(70, 14);
Â¹Ä¹¹Ïreturn atIsland_E;// robot is now at the island, facing east
ÏÏ©}
```

4. Libraries

4.1. Description of a Library

In Botball – and, indeed, almost any other coding realm – code should be written from the top down, with all of your main objectives explicitly listed in the State Machine, then the tasks needed to complete each objective in the methods, and finally the individual instructions necessary for each of those tasks divided into libraries depending its type. A code library contains all of the code needed for the most basic functions of a robot and provides a convenient way to recycle often-used code.

4.2. Benefits and Usage of Libraries

The Dead Robot Society operates using four libraries: movement, actuators, sensors, and camera. The table below categorizes different types of actions that a robot may perform by which library it would belong in.

Library Name	Types of Actions
Movement	Drives (forward/backward, blocked/unblocked) Rotations (clockwise/counter-clockwise, blocked/unblocked) Pivots (clockwise/counter-clockwise)
Actuators	Claws (open/close) Arms (raise/lower) Cages (open/close) Flaps and Drag Fins (extend/retract)
Sensors	Line-following Squaring up (infrared/physical) Bump triggers Wall-following
Camera	Blob-tracking Navigation Get-well points Look approximations

Each instruction called in the methods from the previous section is contained in one of these and, similar to the methods, this minimizes the amount potential error when updating variables such as timings and distances.

5. References

Dead Robot Society. (2010). Create Documentation Drop Code [Computer program]. Herndon, Virginia.

Gras, Jon; Newcastle, Les; Newcastle, Karen. (2010). [Discussion on function of State Machine].