Software Handbook

Table of Contents

REVISION HISTORY

DATE	DESCRIPTION OF CHANGE
V171102	RJV-added abbreviations to software style guide, updated best practices
V170713	RJV-updated software style guide, code development process
V160420	RJV-updated from season lessons learned
V151223	RJV-updated detailed design process, standard software templates, revised variable naming, added UML models
V151107	RJV-Added information flow
V150920	RJV-updated design process, procedures, style guide
V150302	Original

1 INTRODUCTION

1.1 Purpose

The Purpose of this document is to provide best practices and references for the software sub-team.

1.2 Scope

The scope of this document includes:

1) The software design process
2) Software procedures
3) JAVA and JAVA library
4) JAVA style guide for FIRST robot programming
5) Trouble shooting

1.3 Audience

The intended audience is the software sub-team and mentors.

1.4 References

JAVA Web site
Eclipse Web Site
FIRST FRC
Writing Robust JAVA Code, AmbySoft Inc Coding Standards for JAVA, V17.01d, Jan 2000
SUN JAVA Coding Style Guide 1998 - SUN Microsystems
SUN JAVA Code Conventions 1997- SUN Microsystems
JAVA Software Coding Standards Guide - SourceFormatX
JAVA Standards - Government of Bahrain

1.5 Definitions

CID	Control Interface Document (Developed by the electrical team)
FMS	Field Management System
I/O	Input/Output
IP	Internet Protocol
P&ID	Process and Instrumentation Diagram
UML	Universal Modeling Language

2 OVERVIEW

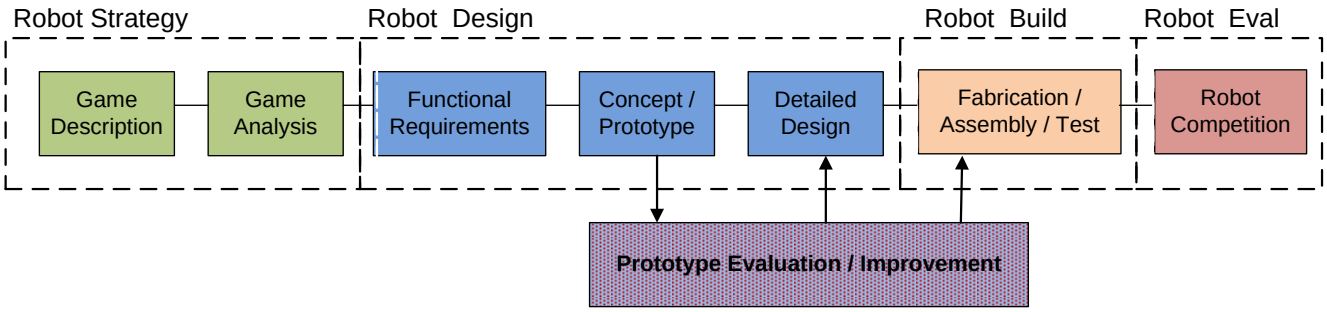
2.1 Description

This document provides software sub team reference material for:

1. Software Design Process
2. Software coding best practices
3. Software procedures
4. Java program style guide
5. Software trouble shooting
6. Software links

3 SOFTWARE DESIGN PROCESS

3.1 Overall Robot Design Process Steps



3.2 Robot Strategy

3.2.1 Team Development of Game Strategy

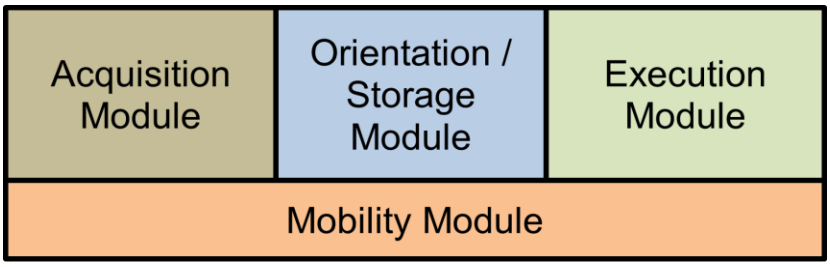
After the kickoff video and READING the game manual, the team will develop the following:

1. A game strategy that includes how the robot will move on the playing field and what actions the robot has to do to play the game.
2. The team will also define the constraints of the robot from the Game Manual.

3.3 Robot Concept Design

3.3.1 Technical Team Robot Module Requirements

The technical team(Mechanical/Electrical/Software) will group the robot actions developed by the team with respect to the basic robot module model shown below. The robot constraints listed by the team will also be grouped with respect to the robot model.



3.3.2 Technical Team Robot Concept Development

During the robot concept development the electrical team is responsible for developing the sensor input and actuator output list for the robot mechanism concepts.

The software team is responsible for understanding how the robot will function.

3.4 Software Design Process Steps

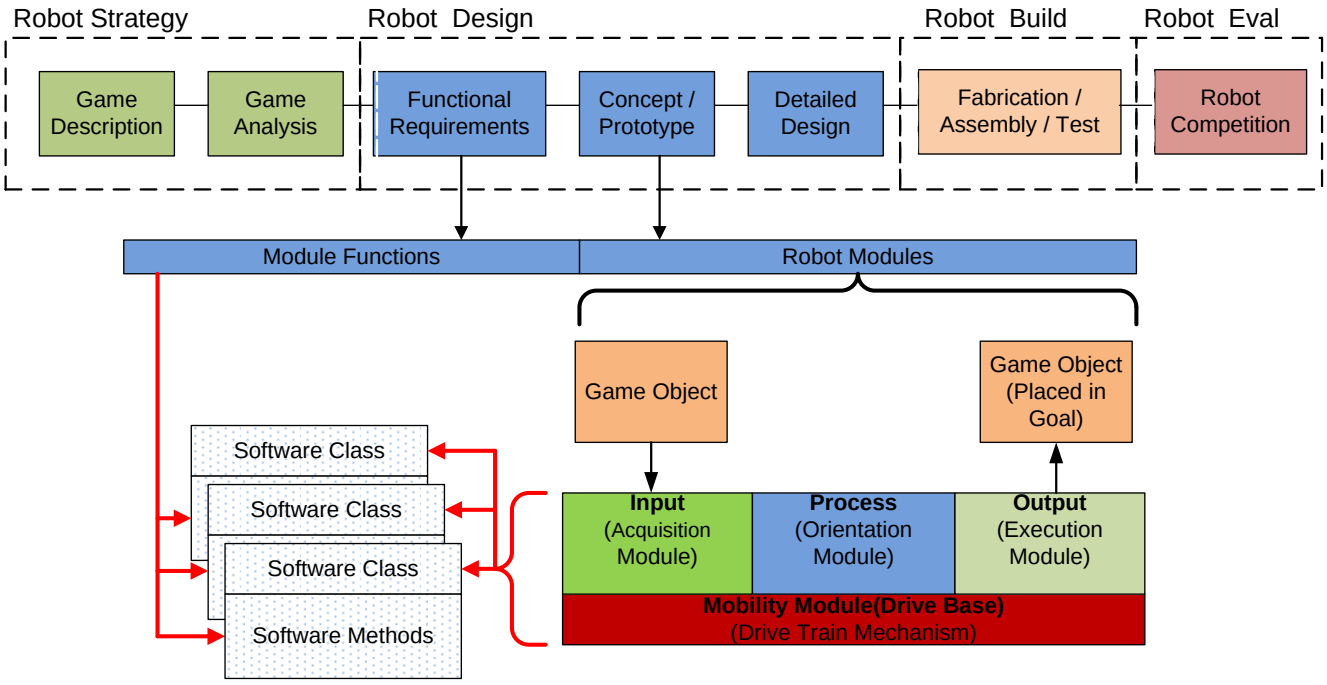
There are seven steps to code development:

1. **Determine the functions the robot has to perform.** This list of functions is developed from a software requirements document or the mechanical description of the robot module.
2. **Link the electrical hardware to computer I/O:** From the electrical control interface document (CID) develop the constants map (robotMap.java)
3. **Link robot functions/definition to robot classes.** The typical classes we have for modules are: drivebase module, acquisition module, orientation module, execution module. Two standard classes: Robot(main program clase) and RobotMap(program constants) conform to FIRST standards.
4. **Develop code logic flow.** THIS IS THE SOFTWARE PLAN. Write in English the sequence of logic the program has to follow. This can be accomplished with the following methods:
    a. Structure English: Describe logic flow a series of flow statements. Flow statements would contain software constructs like if-then, while, get, set, etc.
    b. Psedo-Code: English description organized in a coding format with no computer language syntax.
    c. Flow Chart: use of symbols and text in symbols showing logic flow
    d. UML Model: Software industry standard for planning and describing code (class diagram, sequence diagram, state diagram, use-case document)
7. **Write Code.** Code to be written per this handbook style guide. Code development sequence:
    a. Build module class files and class structures(class-constructor-variable declarations-methods)
    b. Code method stubs – then use eclipse to build method documentation header
    c. Code initialization: variable-object declarations
    d. Update robotMap constants
    e. Code method content. For every line of code should have a line of comment. Remove comments for obvious statements after coding.
    f. Code test method to be used in robot.java “testPeroidic”

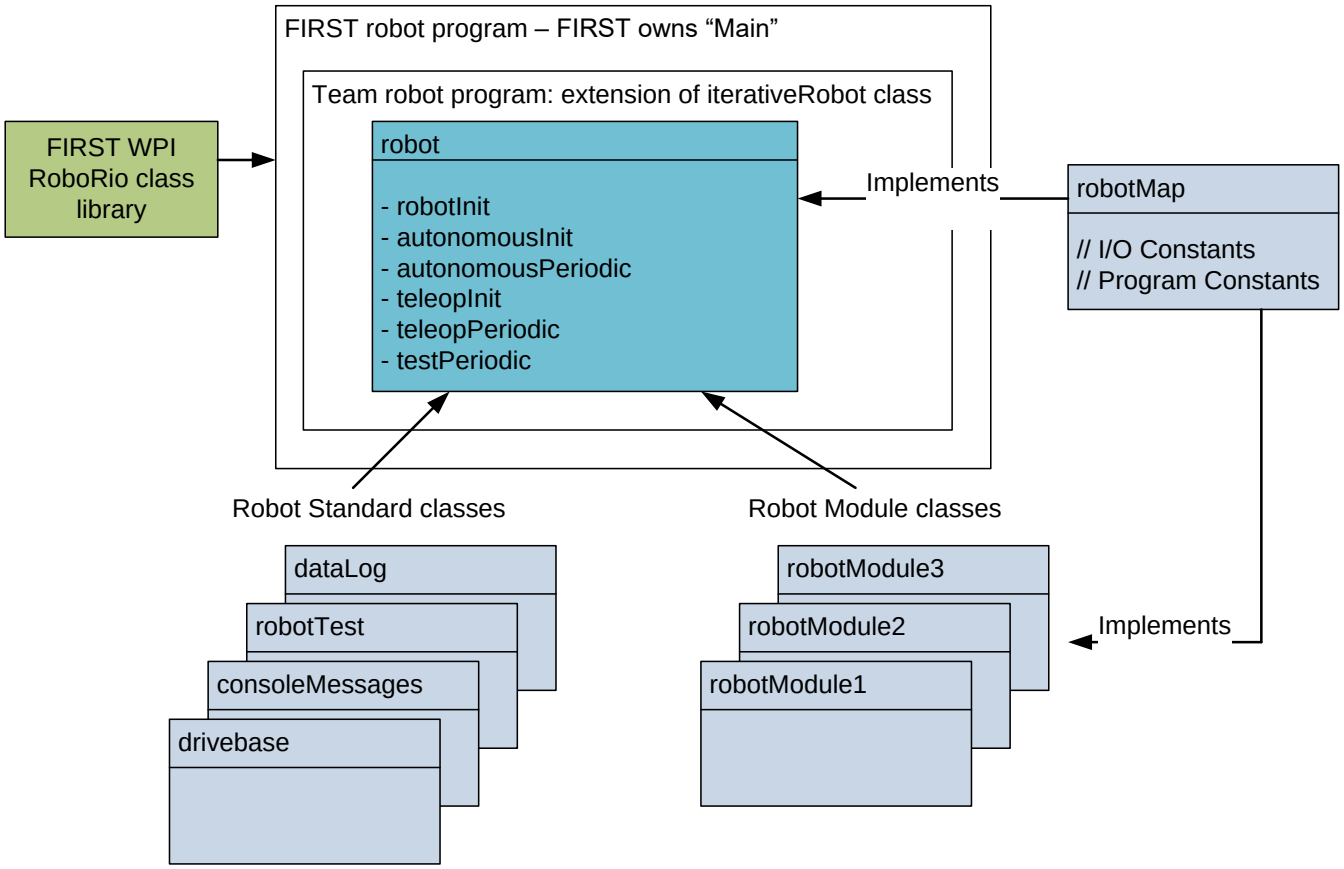
5. **Code Review**: The code developer walks through code with peers and mentors. This process will fine the vast majority of errors and missing code needed for the robot.
6. **Test Code:** There should be code to see the state of inputs and code to execute outputs. After this is accomplished, module code logic is tested.

The code development should comply with the FIRST code template and coding standards of this handbook. NOTE: The comments in the code should follow the software plan developed.

The following is software design process block diagram:



3.5 FIRST Class and Program Architecture



3.5.1 Module Software Requirements

From the sensor/actuator list and robot mechanism functionality the software team will develop a software requirements document. The document should include:

1) Mechanism object definition map
2) Driver station object definition map
3) Robot Map definition (RoboRio port definitions)
4) Logic description of each mechanism(written sequence list, flow chart, UML diagrams)
5) Failure description of the robot modules (e.g. mechanism reaches a movement limit, or robot told to go faster than is should)
6) Method on how to test mechanism logic

3.5.2 Code Development Team

After a review of the software requirements for the robot(software sub-team members / mentors), the coding development will assigned to software sub-team members.

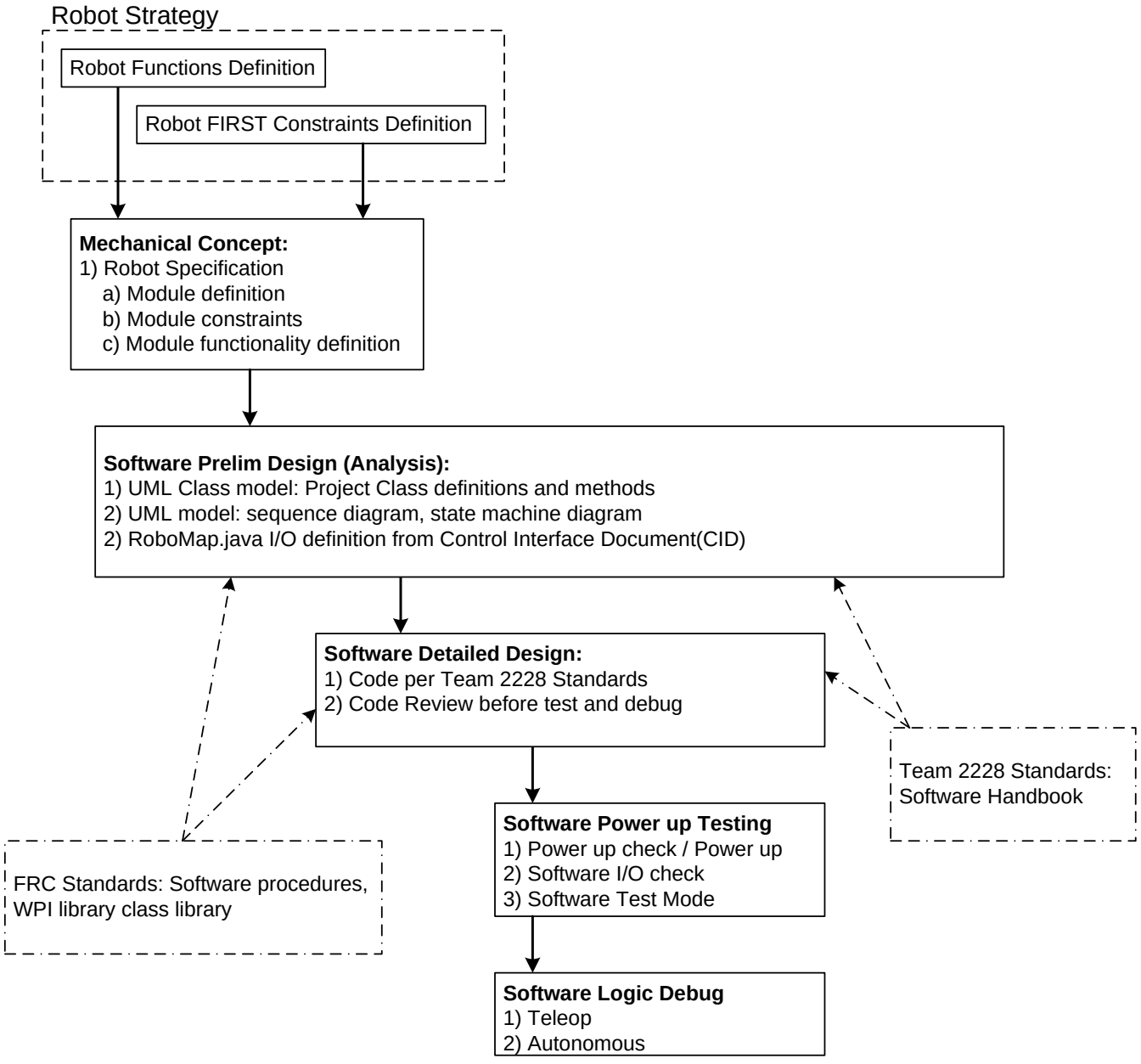
3.5.3 Robot Map

A robot map JAVA file should be developed that defines the I/O used on the RoboRio, its addressing and naming. Information for this map is obtained from the electrical CID table.

3.5.4 Coding Development Process

3.5.5 Design Information Cascade

The general flow of a robot design has a cascade effect on when technical disciplines engage in the design and fabrication process. The cascade effect is shown in the following diagram.



4 BUILD SEASON SOFTWARE TASKS AND DELIVERABLES

4.1 Week 1:

Working with concept teams develop module classes and define module functions.

4.2 Week 2:

Develop drive base software and unit test. At the end of this week the drivers will be running the robot chassis and drive train.

4.3 Week 3:

Support drivers and drive train testing. Complete robot module functional specification.

4.4 Week 4:

Develop robot module code and unit test.

4.5 Week 5:

Working with the electrical team perform I/O testing and software functional testing.

4.6 Week 6:

Support robot system testing and robot program improvement.

5 SOFTWARE BEST PRACTICES

5.1 General

1) The software program should followed the team style guide
2) Software developer should consider how to know the code is functioning in debug.
3) A logger should be used for process control data for charting
4) Software should be developed to test that sensors are functioning
5) Software should be developed to test functions individually

5.2 Comments

Comments should be used for understanding of what not how. To minimize comments the use of prober variable naming should be used per the team software style guide.

5.3 Range Checking

All inputs and outputs should be checked for out of range conditions.

5.4 Data logger

There are two types of data loggers; 1) programming process and 2) Control data for process charting. Typically print statements are used for debug that shows the user the program execution state. Control data is typically logging process data verses time.

5.5 Software Design Review

The software should have a design review walkthrough before testing to check for:

1) Logic errors
2) Out of range errors
3) Communications errors
4) System input fault errors

The design review process consists of the following:

1) The author reads the code to a review team which should consist of the mentor and several software team members that may have to maintain the software.

2) Review members are **allowed to ask** the following questions:
    a. Questions with regard to the team software style guide
    b. Questions with regard to standard industry code practices (For example: use of *case* structure instead of a string of *if else*… statements)
    c. Questions to help review team members understand the code organization or code logic
    d. Questions with regard to "How can this software fail" , i.e what conditions would make the software fail (out of range values, call statement returns a different data type, hung in a loop(need for time outs), etc)
3) Review members **questions which are not allowed**:
    a. "I wouldn't have written this code this way"
4) Remember: Code Complexity requires more documentation!!!!!!!! Reviewers can request that more comments are required to make the code understandable.

DESIGN REVIEW ETIQUETTE:
There are many ways to solve any problem. The question is!! - Does the code solve the problem and is it documented so you can understand what is going on.

6 SOFTWARE PROCEDURES

6.1 Configure Router-Modem

- Programming your radio for home use
- Programming Radios for FMS Offseason

6.2 Configuring the roboRIO

Before a brand new roboRIO can be put into action, you must first install the latest roboRIO firmware and then re-image the software using the latest version. Before you begin, ensure that the NI (National Instruments) suite is updated.

- Imaging your roboRIO
- Updating RoboRio Firmware
- Installing Java 8 on the roboRIO using the FRC roboRIO Java Installer
- Recovering a roboRIO using Safe Mode

6.3 RoboRIO Networking

- Windows Firewall Configuration
- RoboRIO Networking
- RoboRIO Network Troubleshooting
- Running a 2015 Week Zero
- IP Networking at the Event
- Troubleshooting Dashboard Connectivity

- FMS Whitepaper

6.4 Software-Firmware updates

- Installing the FRC 2015 Update Suite
- Current Software Revisions

6.5 Creating and Running Robot Programs

- Creating your Benchtop Test Program
- Building and downloading a robot project to the roboRIO
- Using Riolog to view console output
- Debugging a robot program

- Running your Benchtop Test Program - Tethered
- Running your Benchtop Test Program - Wireless

- RobotBuilder

6.6 Configuring and Updating Pneumatics and Power Distribution

- Updating and Configuring Pneumatics Control Module and Power Distribution Panel

6.7 Setting up the JAVA Development Environment

- Installing Eclipse (C++/Java)
- Installing the FRC 2015 Update Suite (All Languages)
- Installing Java 8 on the roboRIO using the FRC roboRIO Java Installer (Java only)

6.8 Troubleshooting

- Status Light Quick Reference
- Driver Station Log File Viewer
- RoboRIO Brownout and Understanding Current Draw

7 FRC JAVA and JAVA Library

7.1 JAVA

- FRC Java Programming
- C++\Java Porting Guide - 2014 to 2015
- FRC Java WPILib API Documentation
- Java conventions for objects, methods and variables

- Choosing a Base Class

7.2 WPI Library

- What is WPILib
- Online WPILibJ documentation

7.2.1 Sensors

- WPILib Sensor Overview
- Switches - Using limit switches to control behavior
- Analog inputs
- Potentiometers - Measuring joint angle or linear motion
- Analog triggers
- Accelerometers - measuring acceleration and tilt
- Gyros - Measuring rotation and controlling robot driving direction
- Ultrasonic Sensors - Measuring robot distance to a surface
- Counters - Measuring rotation, counting pulses and more
- Encoders - Measuring rotation of a wheel or other shaft
- Operating the robot with feedback from sensors (PID control)

7.2.2 Actuators

- Actuator Overview
- Driving motors with speed controller objects (Victors, Talons and Jaguars)
- Getting your robot to drive with the RobotDrive class
- Repeatable Low Power Movement - Controlling Servos with WPILib
- Driving a robot using Mecanum drive
- Using the motor safety feature
- On/Off control of motors and other mechanisms - Relays
- Operating a compressor for pneumatics
- Operating pneumatic cylinders - Solenoids

7.2.3 Using the CAN subsystem with the RoboRIO

- Using the CAN subsystem with the RoboRIO
- Power Distribution Panel
- Pneumatics Control Module
- Jaguar speed controllers

7.2.4 Driver Station

- FRC Driver Station
- Driver Station Input Overview
- Joysticks
- Displaying Data on the DS - Dashboard Overview
- SmartDashboard Manual
- SmartDashboard 2.0 SFX

- Smart Dashboard Details
- Using the SmartDashboard Vision installer
- Test Mode and Live Window

8 SOFTWARE DESIGN DOCUMENTATION

At the end of the robot design and build project the following software documentation should be developed:

1. I/O map(started by the electrical sub-team) with sensor/actuator names used in the software program
2. Robot Map JAVA Class with the definition of sensor/actuator port assignments
3. JAVA docs document that has extracted the following information from all Class JAVA programs:
    a. The name of the class
    b. What the class does
    c. Name of methods, how to use them, and what they do

9 Robot Program Archive

The robot program project should be archived on GitHub. See Appendix B for procedure.

10 PROGRAMMING STYLE GUIDE

10.1 Introduction

The intent of a program style is to provide greater consistency between all programmers. Greater consistency means code is easier to read and maintain.

No programming style is perfect. If you go against the standard style you should document what you are doing.

10.2 Naming Conventions

10.2.1 Good Naming Convention

1. A name should use complete English words.
2. Abbreviations should be avoided. If an abbreviation is used it should be self evident to any reader or well defined. Abbreviations to be a minimum of three letters. **NO one letter names.**
3. Class names should start with a capital letter
4. All names should be concatenated with the first letter being lower case and the first letter of the next word being capitalized. (e.g. driveTrainLeftFrontMotor)
5. Avoid long names
6. All names should be developed in directory style:

Module object name - Object adjective - Object type

Object type examples: Sensor, Cylinder, Motor, Servo, Switch, Encoder, Potentiometer, Solenoid, Gyro,
Object name examples: elevatorUpperLimitSwitch (elevator-UpperLimit-Switch), driveTrainLeftMotor (drivetrain-Left-Motor), shooterFlyWheelMotor (shooter-FlyWheel-Motor)

10.2.2 Constant Names

1. For "final" constants all upper case letters should be used.
   For example:
   Static final int SPEED_LIMIT = 25; //MPH(Miles Per Hour)

   or

   static int kGallonsPerMinute = 25; //GPM(Gallons Per Minute)

   Typically a name in capitals is associated with locked constants and k<name> for constants that can be in code.

2. All constant declarations should a comment that includes a unit definition
3. If a static variable is use that is a constant that may change by the program or user input the suffix Stpt(set point) should be used. This helps in readability to understand that the variable is not an analog input or expression variable.
   For example: int roomTemperatureStpt = 75; //Room temperature(deg F)

4. For I/O addresses the word "CHAN" should be used at the end of the I/O channel address.
   For example: DRIVE_TRAIN_RIGHT_MOTOR_CHAN

5. Typical Boolean variables in JAVA start with an “is” (for example: isRobotEnabled). This helps in software readability. The English sentence “If the robot is enabled do….”, code: if(isRobotEnabled) then {…}. Two built in Boolean constants are "true" and "false".

10.2.3 Variable Names

1. Variable names should be developed in a directory style.
   For exanple:
   int shooterEncoderCount; //Encoder count since last reading(counts)
   int elevatorPositionValue; //Elevator position(0-5volts, or 0-100%)

   Note: robot module - module object - data type

2. For improved readability variables should have a suffix indicating a varying value (e.g value, count, . For example: timerValue, shooterEncoderCount.

3. To improve readability for variables that are constants that can be changed the following suffix should be used: “stpt”. For example: lifterDownPositionStpt. Or the prefix “k”. For example: kGallonsPerMinute
4. All variable declarations require an inline comment describing the variable and its units

10.2.4 Boolean Names

Boolean variables are typically used as an indicator of a state that is true or false. The Boolean name typically starts with “is” to improve code readablility.

For example:
isReverse – is a motor setup in reverse mode; if isReverse then ……
isAutoEnabled – is the system setup in auto mode; if isAutoEncabled then….

10.2.5 Abbreviations

Abbreviations should be avoided **unless the abbreviation is universally well understood**! By not using abbreviations, the need to comment is reduced. The code reads like an English sentence.

Abbreviations **should be at least three letters**.

Examples of Universally understood abbreviations: average-avg, forward-fwd, down-dwn, setpoint-stpt, motor-mtr, encoder-enc, master-mstr, reset-rst, initialize-init, command-cmd

For example:
What is this: R1; a constant?, Variable?, Boolean?, object?
It is an object: driveRightMasterMotor; *Note: class-object name- object type*

This could be driveRgtMstrMtr, however, driveRightMasterMotor is better code in that the code will read like an English statement.

10.2.5.1 How to Develop an Abbreviation

1. Remove all vowels
2. Save the first letter, then save constants that resemble the original word and remove other constants until you have a minimum of three letter constants

10.2.6 Field Names

Field names are typically nouns. FIRST recommendation is that all field variables use the following prefix: m_FieldVariableName. All field names should be "private". Use "getters"/"setters" methods to access.

For example: private int m_StudentNumber

10.2.7 Method Names

1. The first word of a method name should be a strong verb.

   For example:
   open/close, add/drop, get/set

2. To work with object fields the first word verb should be "set<InstanceVariable>" and "get<InstanceVariable>".
3. To read the status of a Boolean field, the method prefix should be "is". For example: isFwdLimitSwitchClosed

10.2.8 Naming Loop Counters

For loop counters it is acceptable to use single characters. Typical variables are i,j,k

10.3 Source Code Style Guidelines

10.3.1 Code Line Length

The limit on the length of lines is 80 columns and this is a strongly preferred limit.

10.3.2 Program Statements

1. There should be one program statement per line.
2. For a control flow statement that has more than one comparison, the comparisons should be listed in a column.

```
For example:
IF(foo1 >bar1) || (foo2 <= bar3) && (foo6 != bar2) then {
}

should be written as:

IF(foo1 > bar1)
   || (foo2 <= bar3)
   && (foo6 != bar2) then {
}
```

10.3.3 Indentation

Use the indentation standard of your IDE. The JAVA standard is a tab/space of 4 characters. Deeper indentations should also be tab/space of 4 characters.

10.3.4 Bracing

The JAVA standard is to put the opening brace last on the line, and put the closing brace first.

```
For example:
if (foo is true) {
   we do bar
}
```

10.3.5 White Space in Code

White space should be used around all operation characters (i.e. +,-,<,>,=,||,*,%, etc)

10.3.6 Parenthesis

1. Parenthesis should be used around all logical comparisons
   For example:
   ```
   IF(foo1 < bar1) then {
   }
   ```
2. Parenthesis should be used to set the exact order of operations.
   For example:

```
foo1 = (foo2 + bar3) * bar2;
```

10.3.7 Magic numbers

Magic numbers - *"numbers used in the code with no explanation what the number means",* should not be used. variable constants, and enums, should be used instead.
For example:

```
if(dayCountValue == FRIDAY) then{  //FRIDAY is an enum type with value 5
                                   // enums should be in capital letters
}

final int TRASH_DAY = 4; //Trash day is on wendsday, 4th day of the week, sunday = 0
or
 final int kTrashDay = 4; //Trash day is on wendsday, 4th day of the week, sunday = 0

if(dayCountValue == TRASH_DAY) then{
}
```

If a number is used in the code it must have a comment associated with the number.
For exmple:

```
//Setting the speed to 50%
setSpeed(.5);
```

10.3.8 Flow Control Statements

There are two types of flow control statements:

1. Decision: if-then-else, switch-case
2. Loops: while, do-while, and for

The comparison should be organized in the following fashion:

1. The comparison should be enclosed in parenthesis.
2. The variable that you are checking should always be on the left side of the comparison

10.4 Program Documentation

10.4.1 Types of Comments

1. For documentation purposes using "javadocs" The comments should be enclosed with

```
/**
Multiline comments....
....
*/
```

2. A C-style multiline comment should be used for multiline comments in the code and for "commenting out" code that you do not want to erase but you do not want to run.
3. The standard single line comment: "//" for code comments

10.4.1.1 Class Prologue Documentation

Each class file should have a prologue after the import statements that has the following information:

1. The class name and the reason for the class
2. List of methods, method descriptions and invoking a method format
3. A list of all abbreviations and their description should be in the Class header

4. Revision history with the latest revision at the top of the list

10.4.2 Method Header

Each method should have a header that includes the following information:

1. what the method does
2. Description of parameters passed to the method and the data type
3. what is returned from the method
4. An example of how to invoke the method
5. Revision history

10.4.2.1 Code Comments

Comments should be added to code help other team members understand why the code is written the way it is. Comments should be used for all decision statements, and a sequence of several statements to do one function as a minimum. All complex code should be commented.

In a perfect world it would be best to write your comments first and then write your code. The comments should reflect what you signed up for in the functional description of the program requirements document. You now have a direct correlation of requirements to code. This speeds up code reviews to verify all requirements are met.

("**If your program isn't worth documenting, it probably isn't worth running" (Nagler, 1995))**

Comments should be placed over a JAVA statement with a line space before the comment. Comments at the end of the line should be avoided.

For example:

```
....previous code
}

//Continue ramping the speed on the well pump motor until we have reached the
//specified gallons per minute rate
IF(wellPumpMotorGPM  <=  gallonsPerMinuteStpt){
   wellPumpMotorSpeedStpt++
}//end if
```

For an ending brace the optional comment that denotes the end of a control flow statement is helpful know what control flow statement is the ending brace is associated with.

11 POWER UP / TROUBLESHOOTING

11.1 Power up

11.1.1 Software Power Up Steps

At this point the electrical sub-team has wired, checked for shorts and has powered on the robot. The following is a power up sequence for software sub-team

1. First thing always – is everything connected / plugged in? (The electrical team should have tested the cabling and the I/O device before the software team arrives.)
2. Is the power turned on? – USE VERBAL POWER ON PROTOCOL!
3. Do the LED indicators on devices read the correct value?
4. Configure the wireless bridge
5. Re-confirm CAN addresses to CID document
6. Download robot program
7. Enable RoboRio – USE VERBAL ROBOT ACTIVE PROTOCOL!
8. Power up Driver Console and enter "Test Mode"
9. Test sensors and actuators for functionality
10. Test robot program

Also see Status Light Quick Reference

11.1.2 Software Power Up Verbal Protocol

1. A safety observer shall be assigned.
2. The person on the "**Robot Enable Button**" Shall proclaims "**Robot ON**"
3. Only the safety observer after assessing the situation responses "**All Clear**"
4. The person on the "**Robot Enable Button**" final response should be "**Robot Active**" (This response is a verification of the safety observer.)

11.1.3 Test Mode and Live Window

The first step in powering up the robot is to check the functionality of the sensors and actuators. The following shows how to use the test mode and Live Windows:
Test Mode and Live Windows usage

11.2 Control System Troubleshooting

Control System Troubleshooting steps:

1. First thing always – is everything connected / plugged in? (The electrical team should have tested the cabling and the I/O device before the software team arrives.)
2. Is the power turned on?
3. Do the LED indicators on devices read the correct value?
4. Does the I/O device function? (There should be a test program for each I/O device to check its operation – the test program should have been tested during debug.)
5. If the system was functioning, the main question is "WHAT HAS CHANGED?"

12 SOFTWARE LINKS

1. Talon SRX users guide and software manual
2. Spike users guide
3. Pneumatics Control Module users guide
4. Power Distribution Panel users guide
5. Voltage Regulator Module users guide
6. Victor SP quick start guide

7. RoboRio user manual
8. FRC JAVA Programming

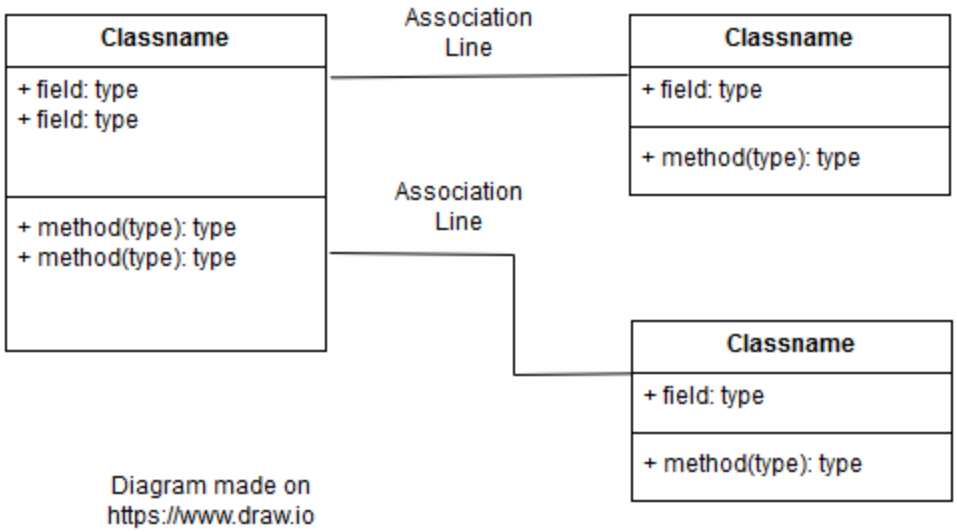
13 SOFTWARE TRAINING

13.1 Level1 Training

1. Level 1 training consists of:
    a. Understanding the software control components
    b. Understanding the mechanics of how JAVA programs are organized in FIRST
    c. Understand the basics software constructs of the JAVA language
2. Level 2 Training consists of:
    a. Understand the software procedures for loading tools
    b. Understand the software procedures to configure components of the control system
3. Level 3 Training consists of:
    a. Understand advanced control algorithms (e.g. PID, Feedback control systems).

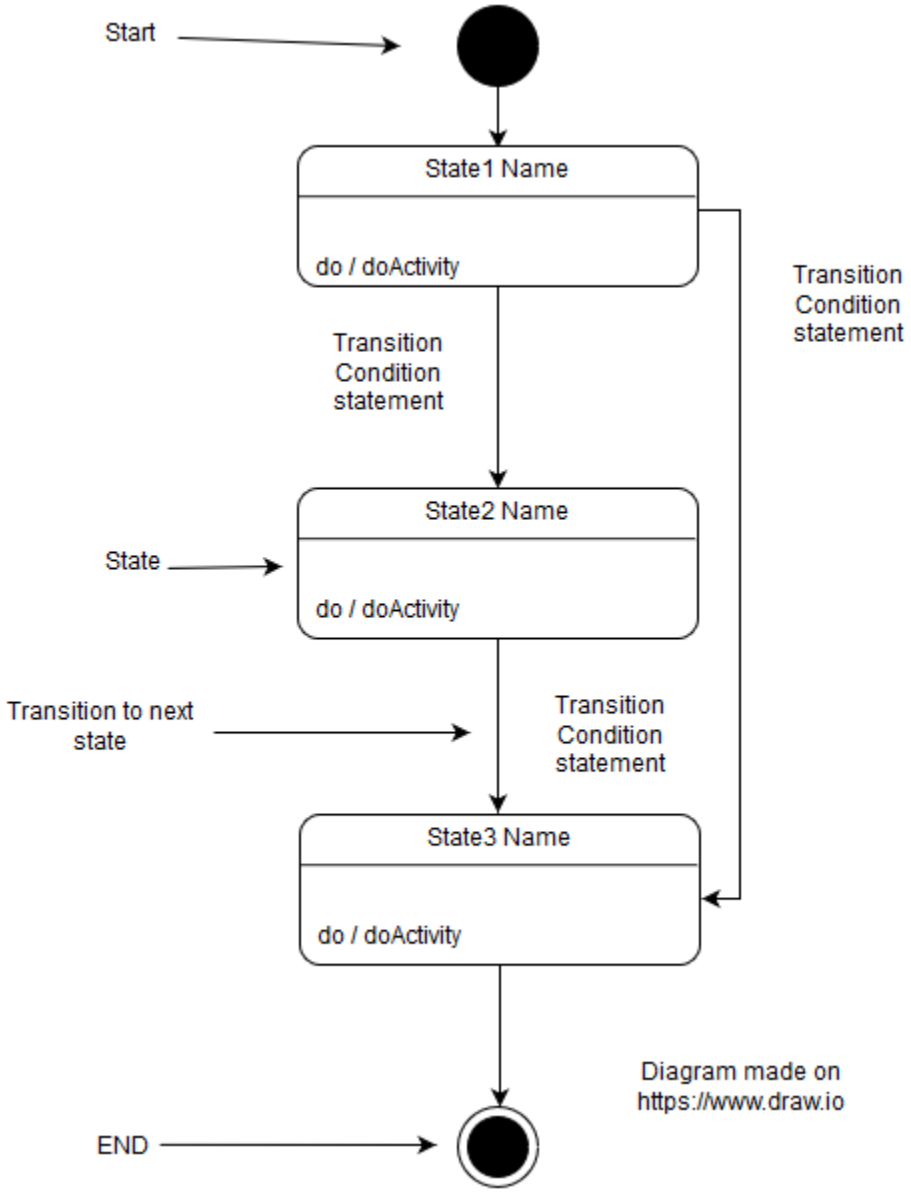
14 APPENDIX - UML MODELS

14.1 Class Model



There is an add in for eclipse to make class

14.2 State Model



15 APPENDIX - FIRST robot.java PROGRAM ORGANIZATION

```
package org.usfirst.frc.team2228.robot;
//-- CLASS HEADER --
/**
*  Robot.java // This is the main program class for the robot
*  @FRC season theme name(e.g. Recyle-Rush 2015)
*  @Author (e.g. John Doe, Team 2228)
*  @Class methods:
*       robotInit()            // called by IterativeRobot on robot power up
*       autonomousInit()       // called by IterativeRobot on start of autonomous
*       autonomousPeriodic()   // called by IterativeRobot every ~20ms during autonomous
*       teleopInit()           // called by IterativeRobot on start of teleop
*       teleopPeriodic()       // called by IterativeRobot every ~20ms during teleop
*       testPeriodic()         // called by IterativeRobot every ~20ms during test
*
*       Definitions / Abbreviations:
*       PCM     Pneumatics Control Module
*       PDP     Power Distribution Panel
*       TALON   Talon motor controller
*/

//-- CLASS IMPORTS FROM FIRST WPILIBJ --
import edu.wpi.first.wpilibj.IterativeRobot;
import edu.wpi.first.wpilibj.Timer;

//-- CLASS IMPORTS FROM TEAM 2228 --
import org.usfirst.frc.team2228.modules.MobilityModule;
import org.usfirst.frc.team2228.modules.AcquisitionModule;
import org.usfirst.frc.team2228.modules.OrientationModule;
import org.usfirst.frc.team2228.modules.ExecutionModule;
import org.usfirst.frc.team2228.modules.RobotMap;

/**
 * MainClass
 */
public class Robot extends IterativeRobot{

        //-- CLASS VARIABLE DEFINITIONS --
        double  m_teleopTime = 0;      // Time in ms
        double   m_oldTeleopTime = 0;  // Time in ms
        double  m_newTeleopTime = 0;    // Time in ms
        long    m_autonomousTime;      // Time in ms

        //-- FRC OBJECT NAME DEFINITIONS --
        Date                                nowTime;

       //-- MODULE OBJECT NAME DEFINITIONS
        MobilityModule                      _driveTrainModule;
        AquisitionModule                    _gatherModule;
        OrientationModule                   _elevatorModule;
        ExecutionModule                     _pusherModule;



        /**
     * This function is run when the robot is first started up and should be
     * used for any initialization code.
     */
    public void robotInit() {
         //Construct robot module objects
        _driveTrainModule  = new MobilityModule;
        _gatherModule      = new AquisitionModule;
        _elevatorModule    = new OrientationModule;
        _pusherModule      = new ExecutionModule;

        // Construct robot module objects in robot
        _driveTrainModule.init();
```

```
    _gatherModule.init();
    _elevatorModule.init();
    _pusherModule.init();


    //--SET UP SMARTSCREEN??

}

 /**
 * This function is called when going into autonomous mode
 */
    public void autonomousInit()
    {
        // initialize parameters in robot modules for autonomous mode
        _driveTrainModule.autoInit();
        _gatherModule.autoInit();
        _elevatorModule.autoInit();
        _pusherModule.autoInit();

    }
/**
 * This function is called periodically during autonomous( approx. every 20ms)
 */
public void autonomousPeriodic() {

    // update robot modules in autonomous mode
     _driveTrainModule.autoUpdate();
     _gatherModule.autoUpdate();
     _elevatorModule.autoUpdate();
     _pusherModule.autoUpdate();
}

/**
    * This function is called when going into teleop mode
    */
public void teleopInit(){
     // Teleoperation robot module initialization
     _driveTrainModule.teleInit();
     _gatherModule.teleInit();
     _elevatorModule.teleInit();
     _pusherModule.teleInit();
}

/**
 * This function is called periodically during operator control (approx.. every 20ms)
 */
public void teleopPeriodic() {

    // capture teleop cycle time
    m_newTeleopTime = Timer.getFPGATimestamp();
    m_teleopTime = m_newTeleopTime - m_oldTeleopTime;
    m_oldTeleopTime = m_newTeleopTime;

   // Update robot module objects in teleoperation mode
    _driveTrainModule.teleUpdate();
    _gatherModule.teleUpdate();
    _elevatorModule.teleupdate();
    _pusherModule.teleUpdate();

}

/**
 * This function is called periodically during test mode. (approx.. every 20ms)
 */
public void testPeriodic() {
    // update robot modules in test mode
    _driveTrainModule.testUpdate();
    _gatherModule.testUpdate();
```

```
        _elevatorModule.testUpdate();
        _pusherModule.testUpdate();
    }

}
```

16 APPENDIX – ModuleClass.java ORGANIZATION

```
package org.usfirst.frc.team2228.robot;
//-- CLASS HEADER --
/**
*  ClassModule.java // This is the typical layout of a module class
*  @FRC season theme name(e.g. Recyle-Rush 2015)
*  @Author (e.g. J. Doe, Team 2228)
*  @Class methods:
*       init()              // called by Robot.java on robot power up
*       autoInit()          // called by Robot.java on start of autonomous
*       autoUpdate()        // called by Robot.java every ~20ms during autonomous
*       teleopInit()        // called by Robot.java on start of teleop
*       teleopUpdate()      // called by Robot.java every ~20ms during teleop
*       testUpdate()        // called by Robot.java
*
*       Definitions / Abbreviations:
*       PCM     Pneumatics Control Module
*       PDP     Power Distribution Panel
*       TALON   Talon motor controller
*/

//-- CLASS IMPORTS FROM FIRST WPILIBJ -
//-- For example:
import edu.wpi.first.wpilibj.DigitalInput;
import edu.wpi.first.wpilibj.Solenoid;
import edu.wpi.first.wpilibj.CANTalon;

//-- CLASS IMPORTS FROM TEAM 2228 --
import org.usfirst.frc.team2228.modules.MobilityModule;
import org.usfirst.frc.team2228.modules.RobotMap;


//--CLASS ClassModuleName --
public class ClassModuleName{

    //-- CLASS VARIABLE DEFINITIONS -
    //-- For example:
    Private int m_elevatorSpeedStpt;

    //-- FRC OBJECT COMPONENT NAME DEFINITIONS -
    //-- For example:
    DigitalInput       _elevatorAtBottomSwitch;
    Solenoid           _elevatorBrakeCylinder;
    CANTalon           _elvatorLiftMotor;
    RobotMap           _cnst;

    //-- OBJECT CONSTRUCTOR
    public ClassModuleName() {
    {

    //-- CREATE ClassModuleName OBJECTS
    Public void init(){
       //For example:
       _elevatorBrakeCyclinder =      new Solenoid(CAN_ID_PCM, BRAKE_EXTEND_SOLENOID_CHAN);
       _elvatorLiftMotor =            new CANTalon(CAN_ID_LIFT_TALON)
       _elevatorAtBottomSwitch =      new DigitalInput(ELEVATOR_AT_BOTTOM_SWITCH_DIN_CHAN)

       //-- Point to RobotMap object - There is only one global object, so we have to point
       //   to it.
       _cnst = RobotMap.getInstance( );

     //-- Usage example: _cnst.ELEVATOR_AT_BOTTOM_SWITCH_PORT

    }

    //--ClassModule "get" - "set" - "is" METHODS
    //--For example:
```

```
    Public void setElevatorSpeed(int elevatorSpeed){
        m_elevatorSpeedStpt = elevatorSpeed;
    }

     /**
     * This function is called when going into autonomous mode
     */
        public void autoInit()
        {

        }
    /**
     * This function is called periodically during autonomous
     */
    public void autoUpdate() {

    }

    /**
        * This function is called when going into teleop mode
        */
    public void teleopInit(){

    }

    /**
     * This function is called periodically during operator control
     */
    public void teleopUpdate() {

    }

    /**
     * This function is called periodically during test mode
     */
    public void testUpdate() {

    }

}
```

17 APPENDIX - RobotMap.java ORGANIZATION

```
package org.usfirst.frc.team2228.robot;

/**
 * The RobotMap is a mapping from the ports sensors and actuators are wired into
 * to a variable name. This provides flexibility changing wiring, makes checking
 * the wiring easier and significantly reduces the number of magic numbers
 * floating around.
 */
public Class RobotMap {

        // --- CREATE SINGLETON OF ROBOTMAP ----
        // The following JAVA code allows only one instance of RobotMap
        private static RobotMap instance = null;
           private RobotMap() {
              // Exists only to defeat instantiation by others
           }

           // This method is how other objects point to RobotMap
           public static RobotMap getInstance() {
              if(instance == null) {
                 instance = new RobotMap();
              }
              return instance;
           }


    // ---DIGITAL I/O---
        final int DIGITAL_IO_CHANNEL0 = 0;

        // For example
        final int ELEVATOR_AT_BOTTOM_SWITCH_PORT = 1;

        final int DIGITAL_IO_CHANNEL2 = 2;
        final int DIGITAL_IO_CHANNEL3 = 3;
        final int DIGITAL_IO_CHANNEL4 = 4;
        final int DIGITAL_IO_CHANNEL5 = 5;
        final int DIGITAL_IO_CHANNEL6 = 6;
        final int DIGITAL_IO_CHANNEL7 = 7;
        final int DIGITAL_IO_CHANNEL8 = 8;
        final int DIGITAL_IO_CHANNEL9 = 9;

        // ---DIGITAL I/O MXP---
        final int DIGITAL_IO_CHANNEL10 = 10
        final int DIGITAL_IO_CHANNEL11 = 11;
        final int DIGITAL_IO_CHANNEL12 = 12;
        final int DIGITAL_IO_CHANNEL13 = 13;
        final int DIGITAL_IO_CHANNEL14 = 14;
        final int DIGITAL_IO_CHANNEL15 = 15;
        final int DIGITAL_IO_CHANNEL16 = 17;
        final int DIGITAL_IO_CHANNEL17 = 18;

       // ---ANALOG INPUT---
        final int ANALOG_INPUT_CHANNEL1 = 0;

        //For example
        final int GYRO_ANALOG_INPUT = 1;

        final int ANALOG_INPUT_CHANNEL2 = 2;
        final int ANALOG_INPUT_CHANNEL3 = 3;

        // ---ANALOG INPUT MXP---
        final int ANALOG_INPUT_CHANNEL4 = 4;
```

```
final int ANALOG_INPUT_CHANNEL5 = 5;
final int ANALOG_INPUT_CHANNEL6 = 6;
final int ANALOG_INPUT_CHANNEL7 = 7;

// ---ANALOG OUTPUT---
final int ANALOG_OUTPUT_CHANNEL0 = 0;
final int ANALOG_OUTPUT_CHANNEL1 = 1;

// ---RELAYS---
final int RELAY_CHANNEL0 = 0;
final int RELAY_CHANNEL1 = 1;
final int RELAY_CHANNEL2 = 2;
final int RELAY_CHANNEL3 = 3;

 // --PWM--
final int PWM_CHANNEL0 = 0;
final int PWM_CHANNEL1 = 1;
final int PWM_CHANNEL2 = 2;
final int PWM_CHANNEL3 = 3;
final int PWM_CHANNEL4 = 4;
final int PWM_CHANNEL5 = 5;
final int PWM_CHANNEL6 = 6;
final int PWM_CHANNEL7 = 7;
final int PWM_CHANNEL8 = 8;
final int PWM_CHANNEL9 = 9;

final int ZERO_SPEED = 0;
final int MAX_FWD_SPEED = 1;
final int MAX_REV_SPEED = -1;

// ---JOYSTICK---
final int JOYSTICK_PORT0_PB0 = 0; // Push buttons
final int JOYSTICK_PORT0_PB1 = 1;
final int JOYSTICK_PORT0_PB2 = 2;
final int JOYSTICK_PORT0_PB3 = 3;
final int JOYSTICK_PORT0_PB4 = 4;
final int JOYSTICK_PORT0_PB5 = 5;

final int JOYSTICK_PORT1_PB0 = 0; // Push buttons
final int JOYSTICK_PORT1_PB1 = 1;
final int JOYSTICK_PORT1_PB2 = 2;
final int JOYSTICK_PORT1_PB3 = 3;
final int JOYSTICK_PORT1_PB4 = 4;
final int JOYSTICK_PORT1_PB5 = 5;

// ---CAN ADDRESS(0-62)---
final int CAN_ID_PDP = 0;
final int CAN_ID_ROBO = 2;
final int CAN_ID_PCM = 3;
final int CAN_ID_LIFT_TALON = 32;

 // for example
final int CAN_ID_JAGUAR_BACK_RIGHT_WHEEL = 11;
final int CAN_ID_JAGUAR_BACK_LEFT_WHEEL = 12;
final int CAN_ID_JAGUAR_FRONT_RIGHT_WHEEL = 13;
final int CAN_ID_JAGUAR_FRONT_LEFT_WHEEL = 14

// ---STANDARD CONSTANTS---
final int ZERO_SPEED = 0;
final int MAX_FWD_SPEED = 1;
final int MAX_REV_SPEED = -1;
final int COUNTS_PER_REV = 256;

// ---PNEUMATICS---
final int BRAKE_EXTEND_SOLENOID_CHAN = 0;
final int SOLENOID_CHANNEL1 = 1;
final int SOLENOID_CHANNEL2 = 2;
final int SOLENOID_CHANNEL3 = 3;
final int SOLENOID_CHANNEL4 = 4;
```

```
final int SOLENOID_CHANNEL5 = 5;
final int SOLENOID_CHANNEL6 = 6;
final int SOLENOID_CHANNEL7 = 7;
final boolean EXTEND = true;
final boolean RETRACT = false;
```

18 APPENDIX – STANDARD ROBOT SINGLETON CLASSES

A singleton class is a special class that has only one instance in the program package. Each robot module object will create a pointer to the singleton objects to use their methods.

18.1 RobotMap.java

This class is a singleton class that all modules will access to obtain I/O constants.

18.2 Message.java

This class is a singleton class that all modules will call to send messages to the console. It has logic to prevent messages during competition matches.

18.3 Logger.java

This class is a singleton class that all class robot modules will call to send data to be logged. It has logic to prevent logging data during competition matches.

19 APPENDIX - FAILSAFE WIRING

Failsafe wiring is a failure analysis process of selecting the state of sensors to the control system such that a failure of the sensor or the wiring would indicate a failure state and/or no action state to the robot controller.

Example1:
On the motor controllers the end of travel limit switches stop the mechanism from moving in that direction. The switch is wired as normally closed. The system is wired such that if the wire broke, came loose or the limit switch failed and opened the mechanism motor control would stop the motor from moving in the present direction.

Example2:
A reflective sensor is used to detect that a ball is present before the control system will allow any action to occur on the ball. A positive signal is needed to enable the logic. If the sensor fails and/or looses power, the wires are broken or come loose; no action can be performed.

Example3:
A bumper switch is activated to indicate a game object is in place to perform the next operation. The switch is wired as a normally open(NO). If the switch breaks and cannot be closed or the wire breaks and/or comes loose no action will be enabled to do the next action.

To anticipate failure, software can be written to check the state of a sensor before a robot action and then the state of a sensor after the action to see that the sensor changes state. If a sensor/wiring failure is detected there may be ways to compensate and still have the robot continue to operate until the issue is resolved.

20 APPENDIX - ARCHIVING A ROBOT PROJECT PROCEDURE