UNIVERSITY OF VICTORIA

Faculty of Engineering

ENGR 120 B03

VEX Robotics Project - G89

Software Specification Handbook

Jayden Chan, Cobey Hollier, Gabe Goerzen

Updated March 24, 2018

Contents

1 Introduction

Thank you for reading our code specification handbook. This document is designed to serve as reference material for both ourselves and for others who will read the code; either for grading purposes or after it has been published. It contains information for every source and header file as well as an overall project summary. In addition to the technical descriptions provided in section 3, we have included a Finite State Machine as well as a brief introduction to PID controllers.

2 Project Outline

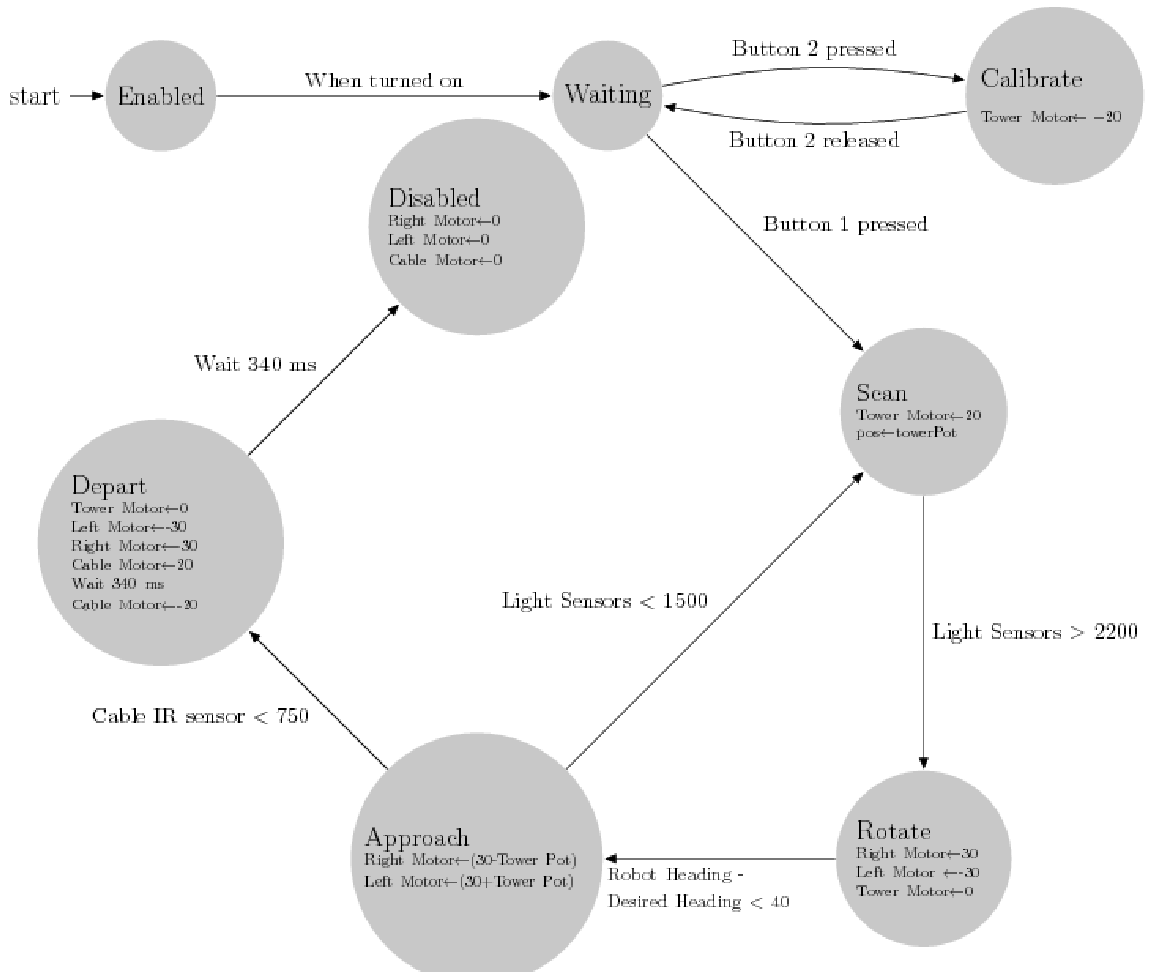
Our primary goal for the software was to make it as modular and adaptable as possible. This made it much easier to add and remove robot functions as needed for the various milestone requirements. In order to achieve this goal, we implemented a Finite State Machine. As the name suggests, it is a state-based framework which controls the robot's actions at any given point in its routine. This modular approach enables us to build a complex sequence of actions from many simple ones. It also allows the robot to perform a non-linear routine (for example, it may return to the "Scan" state if it loses the beacon on its approach).

2.1 Brief Note Regarding PID Controllers

The majority of our robot's driving and scanning functions are controlled by closed-loop PID controllers. As a result, we cannot determine the motor powers at any given point in time as that is solely determined by the PID controllers at runtime. In our Finite State Machine we have provided a rough estimate of what the motors would be outputting, though this is not entirely accurate. A more detailed outline of PID controllers can be found in Appendices A and B.

2.2 Finite State Machine

A diagram of our robot's final Finite State Machine is shown below.



3 Source Specification

3.1 main.c

Author: Jayden Chan, Cobey Hollier

Date Created: January 10, 2018

Description: This file contains the robot's configuration settings as well as the main entry point for the code. It contains the finite state machine code that determines the robot's actions at any given point in time.

Includes: `Okarito.c`

Method Summary

Return Type	Name & Description
`task`	`main()` This file contains the robot's configuration settings as well as the main entry po
`void`	`init()` The main method for the program. Contains the finite state machine to control the robot's actions.
`void`	`cleanup()` Initialization code for the robot to execute when it begins its routine. Clears the debug stream, initializes the drivebase, and waits for a small amount of time to let any sensor values settle.

Method Detail

main

```
task main()
```

This file contains the robot's configuration settings as well as the main entry po

init

```
void init()
```

The main method for the program. Contains the finite state machine to control the robot's actions.

cleanup

```
void cleanup()
```

Initialization code for the robot to execute when it begins its routine. Clears the debug stream, initializes the drivebase, and waits for a small amount of time to let any sensor values settle.

3.2 LightHouse.c

Author: Jayden Chan

Date Created: March 4, 2018

Description: This class is responsible for controlling the robot's "Lighthouse" assembly, which includes a flashlight and visible light (540 nm) phototransistor. It includes functions for finding the beacon and rotating the assembly to face it.

Includes: `Constants.h, PIDController.c, Utils.c, DriveBase.c`

Method Summary

Return Type	Name & Description
`float`	`getLeftLight()` Gets the value of the left light sensor after averaging the last few values.
`float`	`getRightLight()` Gets the value of the right light sensor after averaging the last few values.
`void`	`rotateToDeg(float degrees, int maxSpeed, int safeRange, int safeThreshold)` Rotates the lighthouse assembly to a specific angle relative to the back of the robot using a PID loop.
`void`	`lightHouseInit()` Initialization code for the lighthouse assembly PID controller.
`void`	`fastScan()` Scans for the target object less methodically by simply stopping when the light sensors exceed a certain threshold. This is much faster than scanning the entire field of view but can sometimes result in some error.
`void`	`autoTrackBeacon()` Automatically aligns the lighthouse assembly with the beacon. The rotation value of the lighthouse can then be used to automatically adjust the heading of the robot to make sure it stays on track at all times. This function only works if the sensor is already mostly aligned with the beacon; it's used mainly for fine adjustment, not finding the beacon from a dead start.

Method Detail

getLeftLight

```
float getLeftLight()
```

Gets the value of the left light sensor after averaging the last few values.

Returns: The value of the sensor.

getRightLight

```
float getRightLight()
```

Gets the value of the right light sensor after averaging the last few values.

Returns: The value of the sensor.

rotateToDeg

```
void rotateToDeg(float degrees, int maxSpeed, int safeRange, int
safeThreshold)
```

Rotates the lighthouse assembly to a specific angle relative to the back of the robot using a PID loop.

Parameters:

- `degrees` - The angle to rotate to.
- `maxSpeed` - The max allowed speed.
- `safeRange` - The range tollerance.
- `safeThreshold` - The time needed to be in the safe zone before finishing.

lightHouseInit

```
void lightHouseInit()
```

Initialization code for the lighthouse assembly PID controller.

fastScan

```
void fastScan()
```

Scans for the target object less methodically by simply stopping when the light sensors exceed a certain threshold. This is much faster than scanning the entire field of view but can sometimes result in some error.

autoTrackBeacon

```
void autoTrackBeacon()
```

Automatically aligns the lighthouse assembly with the beacon. The rotation value of the lighthouse can then be used to automatically adjust the heading of the robot to make sure it stays on track at all times. This function only works if the sensor is already mostly aligned with the beacon; it's used mainly for fine adjustment, not finding the beacon from a dead start.

3.3 LEDController.c

Author: Jayden Chan
Date Created: March 3, 2018
Description: This class is for controlling the robot's LEDs. It includes functions for easily toggling the state of the robot's LEDs.
Includes: None

Method Summary

Return Type	Name & Description
`void`	`toggleRedLED()` Toggles the red LED on or off.
`void`	`toggleRainbowLED()` Toggles the rainbow LED on or off.
`void`	`turnOffAllLED()` Turns off all LEDs on the robot. Used in the initialization code for the robot.

Method Detail

toggleRedLED

```
void toggleRedLED()
```

Toggles the red LED on or off.

toggleRainbowLED

```
void toggleRainbowLED()
```

Toggles the rainbow LED on or off.

turnOffAllLED

```
void turnOffAllLED()
```

Turns off all LEDs on the robot. Used in the initialization code for the robot.

3.4 CableGuide.c

Author: Jayden Chan
Date Created: March 22, 2018
Description: This class provides code for controlling the robot's cable guide mechanism which is responsible for preventing the cable from getting caught in the drivetrain as well as to separate the cable from the robot at the end of the connection sequence.
Includes: None

Method Summary

Return Type	Name & Description
`void`	`cableGuideDown()` Lowers the cable guide.
`void`	`cableGuideUp()` Raises the cable guide.

Method Detail

cableGuideDown

`void cableGuideDown()`

Lowers the cable guide.

cableGuideUp

`void cableGuideUp()`

Raises the cable guide.

3.5 PIDController.c

Author: Jayden Chan
Date Created: January 16, 2018
Description: This class contains all the code needed for the PID controllers on the robot. It includes a struct typedef to enable ”pseudo object oriented” code in RobotC.
Includes: `Utils.c, Constants.h`

Method Summary

Return Type	Name & Description
`void`	`PIDInit(PID &pid, float kP, float kI, float kD, float integralLimit, float epsilon, float slewRate, bool zeroOnCross, int refreshRate)` Initializes the PID controller with the provided values. There are many.
`void`	`PIDReset(PID &pid)` Resets all critical values for the PID controller provided.
`float`	`PIDFilter(PID &pid)` Filters the PID output. This includes limiting the rate of change of the output (slew rate) and ensuring that the output does not exceed 127, the max value.

`float`	`PIDCalculate(PID &pid, float error)` Computes the overall PID output using the provided error. This function is pretty complicated so I have included some extra comments to explain what's going on.

Method Detail

PIDInit

```
void PIDInit(PID &pid, float kP, float kI, float kD, float integralLimit,
float epsilon, float slewRate, bool zeroOnCross, int refreshRate)
```

Initializes the PID controller with the provided values. There are many.

Parameters:

`pid` - The PID controller to initialize.
`kP` - The P value for the controller.
`kI` - The I value for the controller.
`kD` - The D value for the controller.
`integralLimit` - The integral limit.
`epsilon` - The epsilon value.
`slewRate` - The slew rate.
`zeroOnCross` - Whether or not to reset the integral term when the error changes signs.
`refreshRate` - The delay between loops.

PIDReset

```
void PIDReset(PID &pid)
```

Resets all critical values for the PID controller provided.

Parameters:

`pid` - The controller to reset.

PIDFilter

```
float PIDFilter(PID &pid)
```

Filters the PID output. This includes limiting the rate of change of the output (slew rate) and ensuring that the output does not exceed 127, the max value.

Parameters:

`pid` - The controller to filter.

Returns: The filtered output.

PIDCalculate

```
float PIDCalculate(PID &pid, float error)
```

Computes the overall PID output using the provided error. This function is pretty complicated so I have included some extra comments to explain what's going on.

Parameters:

`pid` - The PID controller to use.
`error` - The error to use for the calculation.

Returns: The output value of the PID controller.

3.6 Arm.c

Author: Jayden Chan, Cobey Hollier

Date Created: February 16, 2018

Description: Wrapper class for the arm mechanism. This code is used to easily determine information about the state of the robot's arm.

Includes: `Constants.h`

Method Summary

Return Type	Name & Description
`bool`	`isCableDetached(float defaultValue)` Returns true or false depending on whether or not the cable has been successfully attached to the beacon. The function does this by detecting a difference in the sensor values. If the sensor value changes by more than the specified value in the constants file, we know the cable has been connected successfully.

Method Detail

isCableDetached

```
bool isCableDetached(float defaultValue)
```

Returns true or false depending on whether or not the cable has been successfully attached to the beacon. The function does this by detecting a difference in the sensor values. If the sensor value changes by more than the specified value in the constants file, we know the cable has been connected successfully.
Parameters:

`defaultValue` - The sensor value when the cable is still being held by the robot.

Returns: Whether the cable is connected or not.

3.7 Okarito.c

Author: Jayden Chan, Cobey Hollier
Date Created: February 16, 2018
Description: This is the file which contains the main robot code for the robot. All functions from other files come together here to make the robot actually perform the functions it needs to.
Includes: `DriveBase.c, RobotStates.h, LEDController.c, LightHouse.c, CableGuide.c`

Method Summary

Return Type	Name & Description
`void`	`testPeriodic()` Function used for testing only.
`void`	`callibrate()` Callibrates the robot's lighthouse assembly so that it reads the correct values every time.
`void`	`waitingForButtons()` Checks the two buttons on the robot to see if they have been pressed. When a button is pressed the function will change the state of the robot to whatever we want.

`void`	`scanForBeacon()` Performs the scan for the target object then changes the state of the robot to rotate towards the found object.
`void`	`rotateToBeacon()` Rotates the robot towards the beacon using the sensor value obtained from the scanForBeacon function.
`void`	`approachTarget()` Approaches the beacon using an ultrasonic sensor and terminates the approach when the cable has been connected successfully.
`void`	`departTarget()` Backs away from the target and turns after the cable has successfully been connected.

Method Detail

testPeriodic

```
void testPeriodic()
```

Function used for testing only.

callibrate

```
void callibrate()
```

Callibrates the robot's lighthouse assembly so that it reads the correct values every time.

waitingForButtons

```
void waitingForButtons()
```

Checks the two buttons on the robot to see if they have been pressed. When a button is pressed the function will change the state of the robot to whatever we want.

scanForBeacon

```
void scanForBeacon()
```

Performs the scan for the target object then changes the state of the robot to rotate towards the found object.

rotateToBeacon

```
void rotateToBeacon()
```

Rotates the robot towards the beacon using the sensor value obtained from the scanForBeacon function.

approachTarget

```
void approachTarget()
```

Approaches the beacon using an ultrasonic sensor and terminates the approach when the cable has been connected successfully.

departTarget

```
void departTarget()
```

Backs away from the target and turns after the cable has successfully been connected.

3.8 Utils.c

Author: Jayden Chan
Date Created: January 12, 2018
Description: This is a utility class that is designed to support other parts of the code such as the PID controller and ultrasonic filtering class. It includes mostly mathematical functions.
Includes: None

Method Summary

Return Type	Name & Description
`float`	`sign(float input)` Returns the sign of the input. If the input is positive, its sign is (+1), if it's negative, its sign is (-1). If it is zero, its sign is (0).
`float`	`clamp(float input, float clamp)` Clamps the input value between the specified limit.
`float`	`clamp2(float input, float min, float max)` Clamps the input value between the specified minimum and maximum.

Method Detail

sign

```
float sign(float input)
```

Returns the sign of the input. If the input is positive, its sign is (+1), if it's negative, its sign is (-1). If it is zero, its sign is (0).
Parameters:

`input` - The value to get the sign of.

Returns: The sign of the input.

clamp

```
float clamp(float input, float clamp)
```

Clamps the input value between the specified limit.

Parameters:

`input` - The input value to clamp.
`clamp` - The min and max value to clamp at.

Returns: The clamped value of the input.

clamp2

```
float clamp2(float input, float min, float max)
```

Clamps the input value between the specified minimum and maximum.

Parameters:

`input` - The input value to clamp.
`min` - The lower bound of the clamp.
`max` - The upper bound of the clamp.

Returns: The clamped value of the input.

3.9 Ultrasonic.c

Author: Jayden Chan

Date Created: February 16, 2018

Description: This is a wrapper class for the ultrasonic sensor. It is responsible for filtering the bad values from the sensor and making it easy for us to get a clean value.

Includes: `Utils.c`

Method Summary

Return Type	Name & Description
`float`	`getUltraSonic()` Returns the value of the ultrasonic sensor after doing some processing to a

Method Detail

getUltraSonic

```
float getUltraSonic()
```

Returns the value of the ultrasonic sensor after doing some processing to a

3.10 DriveBase.c

Author: Jayden Chan
Date Created: January 13, 2018
Description: Class for controlling the robot's drivetrain. This file combines code from several other files in order to tell the drivetrain what to do in specific circumstances. All driving functions are controlled with closed-loop PID controllers. Most of them are also using L-R compensation PID to ensure that the robot stays on its intended course.
Includes: `PIDController.c, Ultrasonic.c, Arm.c, LEDController.c, Constants.h, Lighthouse.c`

Method Summary

Return Type	Name & Description
`void`	`driveInit()` Initialization code for all of the PID controllers associated with the drivebase.
`void`	`driveReset()` Resets all encoders and PID loops.
`void`	`setRaw(float left, float right)` Sets the values for the left and right side of the drivetrain. Written to simplify drive functions.

`void`	`stopMotors()` Stops the motors.
`void`	`driveStraight(int distance, int maxSpeed, int safeRange, int safeThreshold)` Drives in a perfectly straight line using distance PID and L-R compensation PID.
`void`	`arcTurn(float radius, float orientation, bool turnRight, int safeRange, int safeThreshold)` Performs an arc turn with the specified radius and max speed. Radius is measured from the INSIDE wheel.
`void`	`cableApproach()` Approaches the target using the ultrasonic sensor and terminates when the cable has been connected successfully.
`void`	`rotate(float degrees, float maxSpeed, int safeRange, int safeThreshold)` Rotates in place for the specified number of degrees.
`bool`	`realTimeApproach(int maxSpeed)` Tracks the beacon in real time using the lighthouse assembly as a slave PID controller for the drivetrain and the ultrasonic sensor as the primary error.

Method Detail

driveInit

```
void driveInit()
```

Initialization code for all of the PID controllers associated with the drivebase.

driveReset

```
void driveReset()
```

Resets all encoders and PID loops.

setRaw

```
void setRaw(float left, float right)
```

Sets the values for the left and right side of the drivetrain. Written to simplify drive functions.

Parameters:

`left` - Power level for the left side.
`right` - Power level for the right side.

stopMotors

```
void stopMotors()
```

Stops the motors.

driveStraight

```
void driveStraight(int distance, int maxSpeed, int safeRange, int
safeThreshold)
```

Drives in a perfectly straight line using distance PID and L-R compensation PID.

Parameters:

`distance` - The distance in cm.
`maxSpeed` - The max allowed speed.
`safeRange` - The acceptable range around the target to finish in.
`safeThreshold` - The time required to be inside the safe zone before exiting the function.

arcTurn

```
void arcTurn(float radius, float orientation, bool turnRight, int safeRange,
int safeThreshold)
```

Performs an arc turn with the specified radius and max speed. Radius is measured from the INSIDE wheel.

Parameters:

`radius` - The radius for the arc.
`orientation` - The ending orientation for the arc.
`turnRight` - Whether to turn right.
`safeRange` - The acceptable range to end in.
`safeThreshold` - The time required to be inside the safe zone before exiting the function.

cableApproach

```
void cableApproach()
```

Approaches the target using the ultrasonic sensor and terminates when the cable has been connected successfully.

rotate

```
void rotate(float degrees, float maxSpeed, int safeRange, int safeThreshold)
```

Rotates in place for the specified number of degrees.

Parameters:

`degrees` - The number of degrees to turn.
`maxSpeed` - The max allowed speed during the turn.
`safeRange` - The acceptable range to around the target to finish the turn in.
`safeThreshold` - The amount of time neede to be inside the safe zone before exiting the function.

realTimeApproach

```
bool realTimeApproach(int maxSpeed)
```

Tracks the beacon in real time using the lighthouse assembly as a slave PID controller for the drivetrain and the ultrasonic sensor as the primary error.
Parameters:

`maxSpeed` - The maximum speed allowed.

Returns: Whether or not the connection was successful.

4 Header Specification

4.1 Constants.h

Description: This class contains all of the constants used throughout the robot code.

Constants Summary

Type	Units	Name & Description
`int`	none	`MAX_SPEED` The maximum value that can be assigned to a motor.
`float`	ticks	`TICKS_PER_DEG` The number of ticks read by the potentiometer per one degree of rotation of the LightHouse assembly.
`float`	ticks	`TICKS_PER_ROT` The number of ticks read by the motor encoders per one rotation of the drive shaft.
`float`	cm	`WHEEL_CIRC` The exact circumference of the robot's wheels.
`float`	ticks	`TICKS_PER_CM2` The number of ticks recorded by the encoders after 1 centimeter of driving.
`float`	cm	`DRIVETRAIN_WIDTH` The exact width of the robot's drivetrain.
`float`	none	`MATH_PI` Pi.

float	cm	ULTRASONIC_THRESH The distance from the beacon where the robot should stop when connecting the cable.
int	none	CABLE_SENSOR_DELTA The change in sensor value required to trigger the ”cable attached” condition.
int	ticks	POT_TRACKING_THRESH The potentiometer reading when the LightHouse mechanism is exactly centered.
int	none	BEACON_FOUND_THRESH The minimum sensor reading required of both light sensors before the beacon searching function terminates.
int	none	L_SENSOR_DIFF The correction value to ensure that both light sensors are reading the same value when in the same conditions.
int	ticks	POT_OFFSET The amount of ticks to subtract from the potentiometer such that it reads zero when the LightHouse assembly is facing exactly backwards.

Appendices

A PID Controller Overview

A PID controller offers a more sophisticated approach to controlling a motor when compared to the basic “bang-bang” style. A typical PID controller consists of the following elements:

1. Setpoint - The desired final position of the motor
2. Error - The difference between the current position of the motor and the setpoint
3. Slew Rate Controller - A filtering algorithm that prevents the output of the controller from changing too rapidly, resulting in wheel skidding or other unwanted behaviour
4. “P”, “I”, and “D” Terms - Outlined in Appendix B.
5. Output Sum - The final calculated output of the controller.

The PID controller operates in a loop. On each iteration, the derivative and integral of the error with respect to time are calculated. The error itself, along with these calculated terms are then multiplied by tuned constants and summed to form the preliminary output. In essence, a PID controller is the sum of the error, the derivative of the error, and the integral of the error. This “final” output sum is then sent to the slew rate controller, which prevents it from changing too rapidly. The processed value from the slew rate controller is then sent to the motors.

B PID Constants Summary

Name	Description
kP	(Stands for: **Proportional**) The constant by which to multiply the error when generating the PID output sum.
kI	(Stands for: **Integral**) The constant by which to multiply the integral of the error when generating the PID output sum.
kD	(Stands for: **Derivative**) The constant by which to multiply the derivative of the error when generating the PID output sum.
kS	(Stands for: **Slew Rate**) The maximum rate of change of the output of the PID controller.
kR	(Stands for: **Refresh Rate**) The refresh rate of the PID controller in milliseconds.