[Robotron]

Fabian Rodriguez
Oscar Muneton
Adelaido Jimenez

I. INTRODUCTION

A. General Problem Summary

Our project aims at designing an autonomous robot that will compete in a head to head double elimination tournament called "Roborodentia". The objective of the competition is to have the robot collect small cans and push them into an end zone. Each side of the field contains 14 3oz cat food cans that are colored coded red and blue. A team's can located in a robot's end zone will be worth 1 point while cans stacked on each other will be increased by 2 points for each can in a stack. There is a center dividing wall located between the two sides of the playing field that will be removed 60 seconds into the match.

B. Client Overview

There was no particular client for this project. Our team used this competition as part of our senior project design. We had Professor John Seng from Cal Poly State University in San Luis Obispo as our advisor for the project. Professor Seng has in-depth experience with robotics and provided us with advice throughout our design, implementation, and testing phases of our project. Without his advice and support this project wouldn't have been successful, and for that we thank him.

C. Related Work

Several Roborodentia events have been held over the past couple years. Although the competitions have not always been the same, the main idea has not changed. This is to have participating teams build an autonomous robot that performs a certain task. Several designs in the competition include the BallReaper, Sexy Plexi, Awesom-O, Servo XS, Data-B, and Scorpio just to name a few.

II. PRODUCT DEFINITION

A. Need Statement

In order to participate in the 2012 Roborodentia competition, our group needed to have the resources to build a robot from scratch that would be able to perform a specific task. This specific task included retrieving cat food cans (3 oz.) from a playing field and moving them to the designated player's end-zone. The objective of the competition was to have the most cans on our own end-zone. Stacking the cans was another incentive, as the cans were worth more points when they were stacked.

B. Requirements

The requirements included to build a robot with specific limitations that would collect cans on the given playing field and move them to the robot's marked end zone. Some of those limitations were having the robot not exceed 12 inches in length by 12 inches in width. There was no height limitation. Another limitation that was placed on the robot was that the robot could not have any type of weapons or anything that resembled a weapon on it.

C. Criteria

The criteria for this project included navigating through a playing field where there was cans placed. The objective for the robot was to retrieve as many cans as it could in a certain

amount of time. The robot with the most points won the round. Each point was calculated as follows: 1 point for one can, 3 points for one can stacked on top of another can (1 stacked can), an additional 5 points for two stacked cans. For example, a stack of 3 cans would be worth 9 points. These points would be doubled if the opponent's cans were retrieved and stacked on our own end-zone.

III. DESIGN

A. Team Process

Our team process included proposing design specs to produce a prototype, in which we could test the design and modify as necessary. Once the prototype was built, we were able to test different aspects of the robot that we thought could use improvements. At this time we also went over some of the design decisions made previously and determined which ones needed modification.

After making the necessary adjustments on the prototype, we documented the changes and made the changes when producing the final robot. Some of these changes included changing the design of the claw that was used to pick up the cans from the playing field.

B. System Architecture

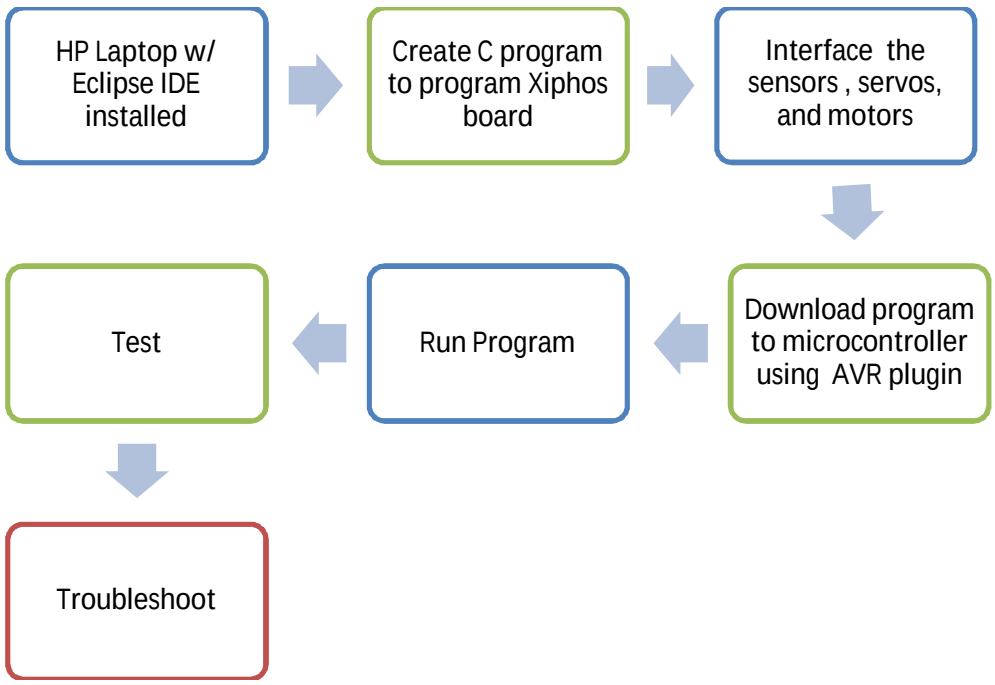

FIGURE 1: System architecture for our robot design.

Figure 1 above is an example of the system architecture we used to implement our autonomous robot. First, a programming tool such as Eclipse was installed and configured to work with the Xiphos board. Once the initial setup was done, step two consisted of writing a c program to allow communication with the microcontroller. The third, step was to interface the motors, servos, switches, and proximity sensors to make the robot work. Once a component was interface

the program was downloaded to the microcontroller and tested. If there was a problem then we had to troubleshoot and repeat the process starting at step three.

C. Hardware Block Diagram

See figure 2 below.

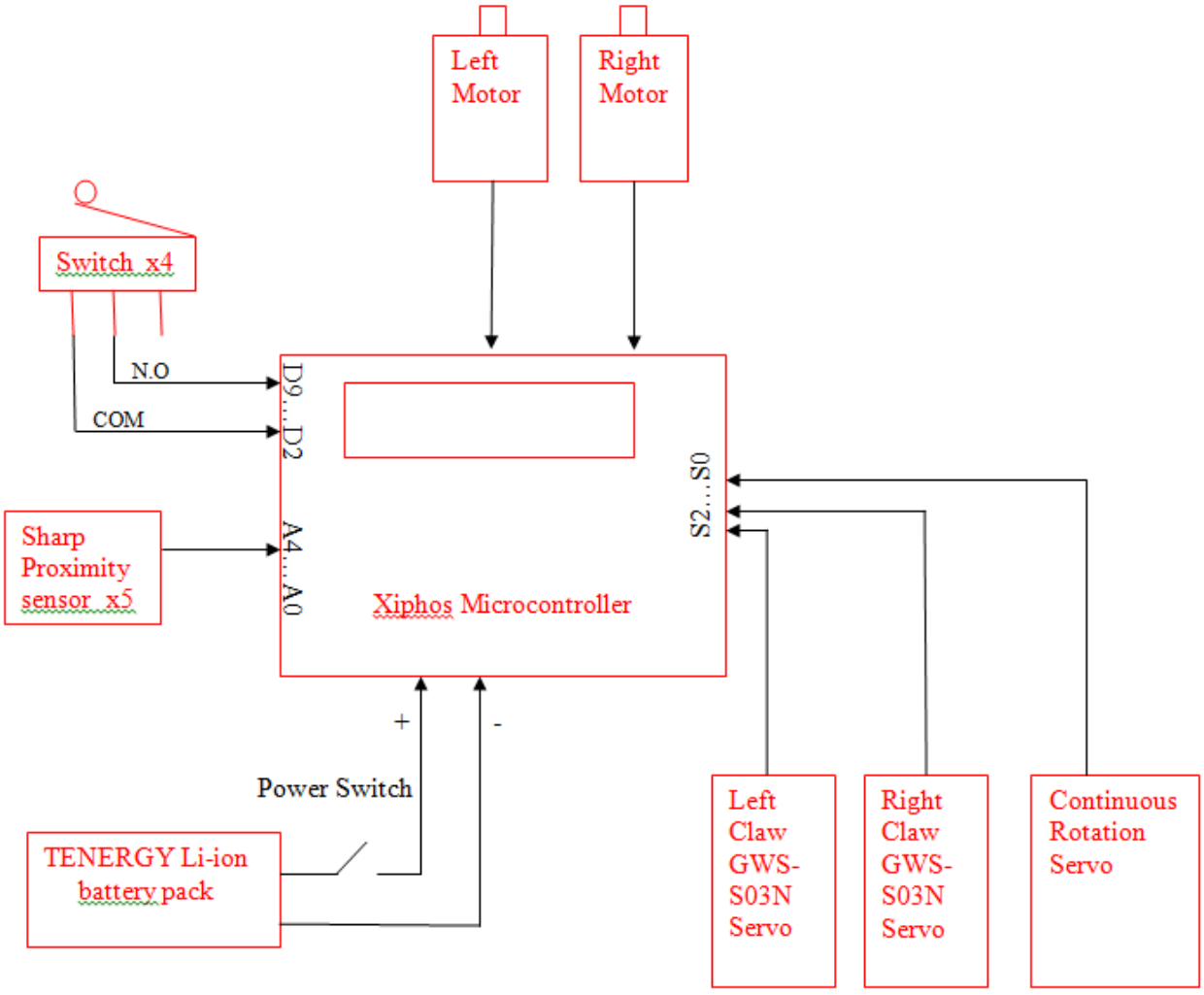

FIGURE 2: A block diagram for our system. It shows the subcomponents, such as the motors, power supply, switches, servos, proximity sensors, and the microcontroller. These subcomponents are important for the proper functionality of our robot.

D. Software Algorithms

Our software algorithms consisted of using a finite state machine (FSM) to have the robot in certain states to perform those certain tasks. Each state represented a function within our code.
These were the possible states that the robot could enter:

CALIBRATION,
FIND_CAN,
CAN_LEFT,
CAN_RIGHT,
TURN_AND_CENTER,
PICK_UP_CAN,
STACK_CAN,
DRIVE_TO_ENDZONE,
DROP_CANS,
DESTROY, POWER_OFF

Below are the functions that represented each state in the FSM.

- **calibrate()** - this function was called at the beginning of the FSM. It was used (as the name says it) to calibrate all the components of the robot.
- **findCan()** - this function was used to find the cans on the playing field. This was done by continuously getting feedback from the proximity sensors mounted on the front of the robot.
- **canLeft()** - once a can was found, it was determined on which side the can was found on. If the can was found to be on the left side of the robot, it would adjust accordingly and go into the next state (PICK_UP_CAN).
- **canRight()** this function is used when a can is detected on the right side of the robot.
- **turnAndCenter()** - this function is called whenever the robot reaches one of the ends of the playing field and needs to turn around.
- **pickUpCan()** - this function will usually be called whenever the robot is ready to physically pick up the can, i.e. after all the adjustments have been made.
- **stackCan()** - this function is called whenever the robot needs to stack cans.
- **driveToEndZone()** - this function is called whenever the robot has reached its maximum stacking limit and needs to drive back to its end-zone and drop the cans.
- **dropCans()** - this function is called whenever the robot is in the end-zone and is ready to drop the cans.
- **destroy()** - despite its' name the robot drops the opponent's stacked cans, that is it doesn't really destroy anything.

The finite state machine is shown in the figure on the next page.

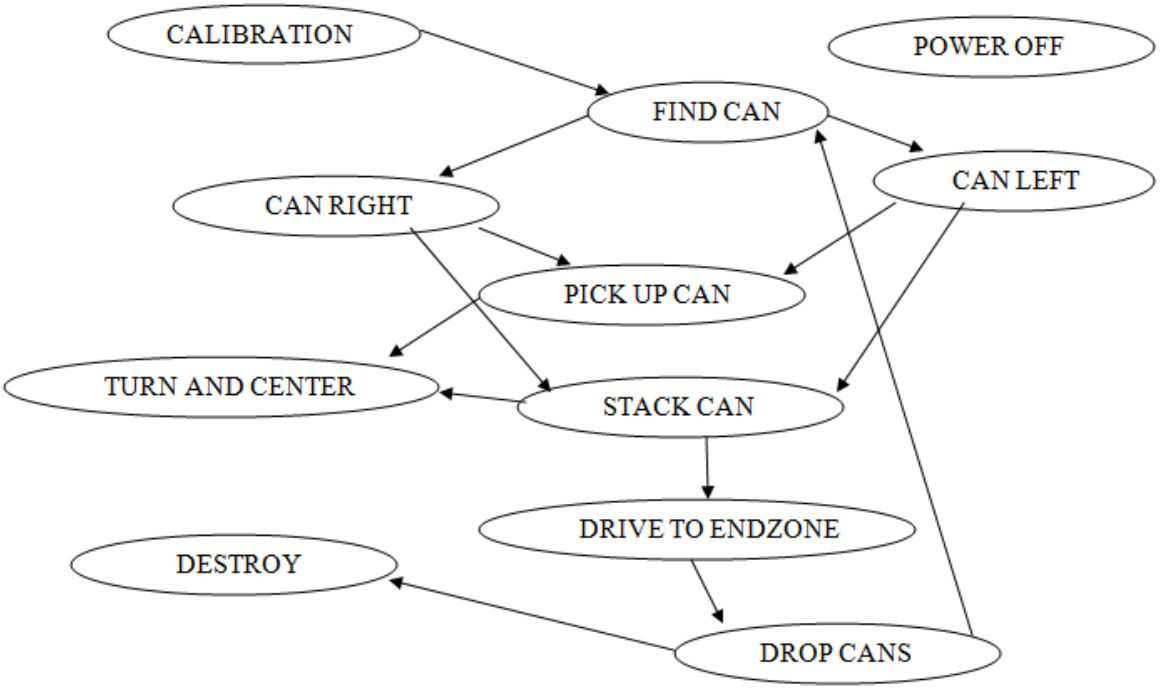

Figure 3. The finite state machine and possible states and possible state transition.

E. Design Decisions

For the initial design of our robot the criteria's which we mostly focused on were that the robot was restricted to a maximum dimension of 12" x 12", had to be fully autonomous, and would need to stack cans to score more points. To meet the stacking requirement we decided to design a claw system based on a forklift that would allow for multiple stacking. This initial design of our system contained a few flaws which required minor modifications. Unfortunately, during the manufacturing phase we ran into size constraint issues which did not allow for our pulley system to fit the required dimension so we were required to redesign the claw system. The new claw system was designed with certain criteria's in mind such as allowing guidance of the can to the claw, and fit required dimensions. The new claw system also had added sensors in place to improve efficiency and reduce error of the robots can stacking capabilities. Instead of having a fixed time count for the continuous sensor to lift the cans a switch was added to inform the microprocessor when the cans were fully lifted before a new can search could begin.

The most critical drawback of the development phase was designing the algorithm to have a fully functioning autonomous robot. Acquiring the data from all the sensors and deciding which sensors took priority proved to be a tedious and complex process. Without setting appropriate priority to certain sensors the robot could either continue driving and miss a can or end up dropping the stack of cans it has already collected, all which were experienced during the development phase.

IV. SYSTEM INTEGRATION ANDN TESTING

Individual Component Test Plan

Our original test plan included individually testing the components that were going to be added to the robot, i.e. the servos, proximity sensors, motors, switches, and the Xiphos board itself. After running several tests on these components it was deemed that some of the components (specifically some of the original servos) did not work. Therefore we needed to replace those components. The only critical components that we had trouble with (after testing) were the servos that were borrowed from the Capstone room. After finding that the servos were faulty, we decided to replace them with servos that were borrowed from the robotics club. After doing this we were able to continue with our testing of the overall system.

System Test Plan

For our system test plan it was pretty much trial and error. Especially at the beginning of the testing phase, we did not know what to expect. Even though we had implemented a prototype, it was not exactly the same (for the main reason that the prototype was made from wood and our final robot was made mainly from plexi-glass). For the main system testing, we obtained three to four cat food cans and setup our own playing field at the Capstone room. Originally, when we had planned to use the line sensors, we setup tape to be able to test the overall system and how it followed the lines. After determining that we were not using the line sensors we setup pseudo walls to represent the side of the playing field and scattered the cans around them. Doing this we were able to get a feel for how the actual playing field would be like. What helped during the testing and system integration was that an actual playing field was setup a week before the competition. This helped in determining what improvements we needed to make and it was a more accurate picture of the overall functionality of the system since it was tested on the actual platform and playing field where the competition would take place.

V. CONCLUSION

In conclusion this robot ended up being one of the top competitors in the competition. Our final robot design was the only robot in the competition to successfully implement a stacking system which allowed us to have a competitive edge over the other participants. Major obstacles that we overcame were developing a fully functioning autonomous lift system with all required sensors and components working in sync. Unfortunately due to the limited time and budget not all initial requirements and goals, placing first place, that we set were meet. Additionally, the navigation system has some minor errors due to the turning accuracy as well as the guesstimate used for the navigation system, which was predefined to a certain path and didn't have any output sensors to adjust for error. Overall our platform was able to navigate the playing field and distinguish cans from boundary obstacles. The final platform should provide a good foundation to build off from for next year's competition. Future work to work on would include a better navigation system that would provide accurate position of the robot in the playing instead of the current guesstimate we have in place, which only knows initial location and required final destination. An example would include to add wheel encoders. Also improve upon the turning capabilities of the wheels and find an optimum turns ration between the left and right wheels to allow for a better turn accuracy.

VI. BUDGET AND JUSTIFICATION

Our budget for this project was quite low since many of the items where provided by our advisor John Seng and the Cal Poly robotics club, which we are part of. Below is a summary of the key components used to accomplish our senior project design.

Component	Role
Xiphos board 1.0	used to control the motors, servos, switches, and to read data from the proximity sensors.
Parallax Continuous Rotation Servo	used to raise and lower the robots claw.
GWS S03N Standard Servo	used to control the opening and closing of each individual claw.
Sharp 2YOA21 52 proximity sensors	used to detect the cans and walls during the driving.
Zippy Coin Micro Switch	used for drivability purposes and to trigger the lowering and lifting of the claw.
Motors	used to for the drivability of the robot.
TENERGY Li-ion battery pack	used to supply power to the Xiphos microcontroller.
Various Connectors & Wires	various connectors and wires were needed to connect the switches, proximity sensors, motors, and servos.

VII. BILL OF MATERIALS

Item	Qty	Unit Price	SubTotal
Xiphos board	1	$0.00	$0.00
Motors	2	$21.95	$43.90
Metal Gear Motor Bracket (pair)	1	$7.45	$7.45
Mount Hubs (pair)	1	$6.95	$6.95
Wheels 60 x 8mm (pair)	1	$7.95	$7.95
Sharp 2YOA21 52 proximity Sensor	4	$0.00	$0.00
Zippy Coin Micro Switch	4	$0.00	$0.00
Power Switch	1	$1.50	$1.50
Tenergy Li-ion battery pack	1	$0.00	$0.00
Plexiglass	1	$20.99	$20.99
Aluminum round rod	1	$5.97	$5.97
Wood	1	$7.34	$7.34

4" Bolt with spacers and nuts	4	$10.99	$43.96
Rubber Wheel Swivel Plate Caster	1	$3.98	$3.98
Miscellaneous Materials	1	$20.00	$20.00
Tap and Die set	1	$25.50	$25.50
Servos	3	$0.00	$0.00
Total			$195.49

VIII. APPENDICES

Adelaido Jimenez My work was focused on helping with the design and building phase of the robot. I helped with the manufacturing of the robot's claw and frame. Once all the parts were produced I helped with the final assembly. I also helped Oscar with the programing part of the project.

Oscar Muneton
For this project my focus was mainly on programming the software algorithms that was used to run the robot. Though this was my main focus, I helped in contributing to the initial design of the robot, along with the actual production of it. One of my other focus points was testing the overall system. This took the majority of the time (other than actual production time) since the testing was done via trial and error.

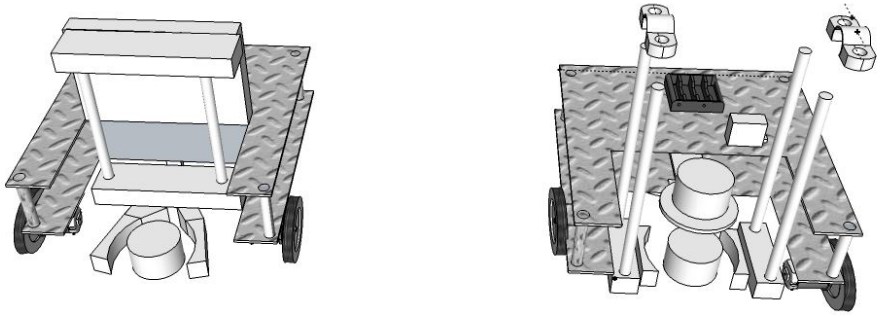

FIGURE 4: Google SketchUp of Robotron design. Right sketchUp was the initial design idea of how Robotron was going to be build. Left sketchUp was the final design of the robot.

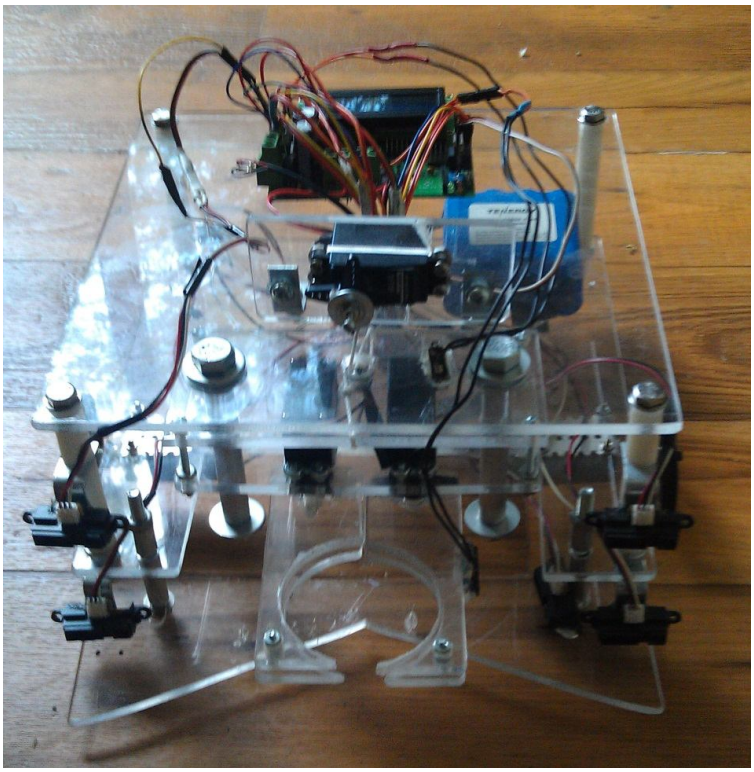

FIGURE 5: Front view of final assembly of Robotron.

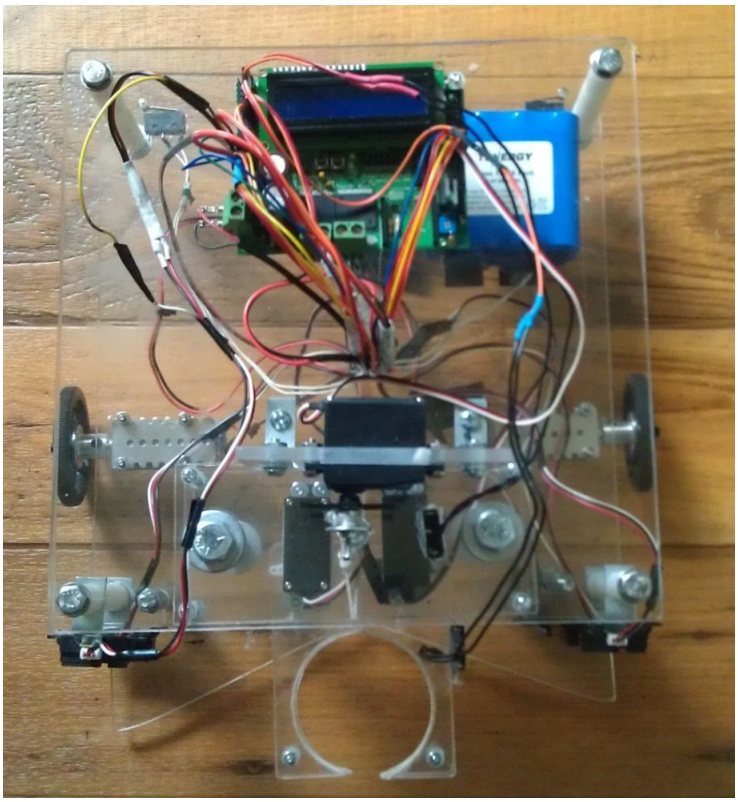

FIGURE 6: Top view of the final assembly of Robotron.

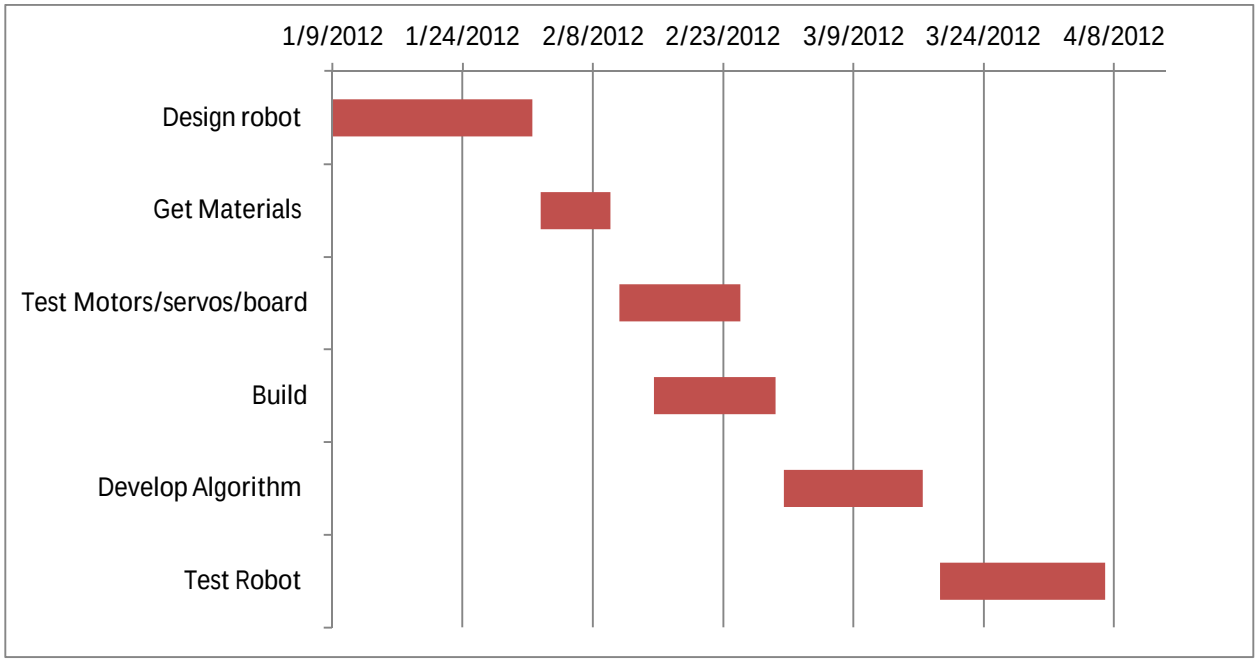

FIGURE 7: Gant chart showing the breakdown of our project and the completion date.

The following code segment is the file sp.c which contained most of the code that ran the system.

```
/*
 *  sp.c
 *
 *  Created on: Jan 15, 2012
 *  Author: Oscar Muneton
 *          Adelaido Jimenez
 *          Fabian Rodriguez
 */
#include <stdlib.h>
#include "projectGlobals.h"

#define DEBUG 1

enum STATE currState = CALIBRATION;

/* Keeping track of number of cans collected */
int numOfCans = 0;
int totalCans = 0; // Max cans for own side is 14

Speed *currSpeed;

int currDirection = NORTH;
int sideOfEndZone = LEFT_SIDE;
int adjust = NONE;
int currSpeedRight = FWD_SLOW_R;

/**
 * Calibrates and initializes all the required components
 * in order to run the AI of the robot.
 */
void calibrate() {

#ifdef DEBUG
      clearScreen();
      printString("Push button");
      lowerLine();
      printString("to start");
      buttonWait();
      printToScreen("Calibrating...");
#endif

      currSpeed->left  = FWD_SLOW_L;
      currSpeed->right = FWD_SLOW_R;

      /* Setting up claw platform switch */
      digitalDirection(STACK_SWITCH_IN, INPUT_PULLUP);
      digitalDirection(STACK_SWITCH_OUT, OUTPUT);
      digitalOutput(STACK_SWITCH_OUT, 0);

      /* Setting up claw switch */
      digitalDirection(CLAW_SWITCH_IN, INPUT_PULLUP);
      digitalDirection(CLAW_SWITCH_OUT, OUTPUT);
```

```
	digitalOutput(CLAW_SWITCH_OUT, 0);

	/* Setting up rear switches */
	digitalDirection(REAR_L_SWITCH_IN, INPUT_PULLUP);
	digitalDirection(REAR_L_SWITCH_OUT, OUTPUT);
	digitalOutput(REAR_L_SWITCH_OUT, 0);

	digitalDirection(REAR_R_SWITCH_IN, INPUT_PULLUP);
	digitalDirection(REAR_R_SWITCH_OUT, OUTPUT);
	digitalOutput(REAR_R_SWITCH_OUT, 0);

	closeClaw(FAST);
	raiseClaw();

	currState = FIND_CAN;

#ifdef DEBUG
	printToScreen("Calibrated");
	clearScreen();
#endif

/**
 * State that enables the robot to look for the given can.  This function will
 * be used to find the cans that will be randomly placed in the playing field.
 */
void findCan() {

	u08 left, right, wall_L, wall_R;
	int timer = 0, clawTimer = 0;
	bool isCanSetFlag = FALSE;
	u08 *result;

	driveFwd();

	// look for a can...
	while(!isCanSetFlag) {
		isCanSetFlag = isCanSet();

		if(isCanSetFlag) {
			adjust = NONE;
			stopMotors();
			if(numOfCans > 0) {
				delayMs(200);
				currState = STACK_CAN;
			}
			else {
				openClaw();
				clawTimer = lowerClawTimed();

				// claw did not go down all the way
				if(clawTimer >= CLAW_DOWN_TIME) {
					shakeClaw();
				}

				closeClaw(SLOW);
```

```
                        currState = PICK_UP_CAN;
                }
                free(result);
                return;
        }

        // look for cans on either side
        result = (u08 *)getAllProximityAvgs();

        left = result[LEFT_PROXIMITY_INDEX];
        right = result[RIGHT_PROXIMITY_INDEX];
        wall_L = result[WALL_FRONT_L_PROX_INDEX];
        wall_R = result[WALL_FRONT_R_PROX_INDEX];

        if(checkIsWall(left, wall_L) || checkIsWall(right, wall_R)) {
                stopMotors();
                currState = TURN_AND_CENTER;
                free(result);
                return;
        }

        if(left >= CAN_PROX_THRESHOLD || right >= CAN_PROX_THRESHOLD) {

                // on left
                if(left >= CAN_PROX_THRESHOLD) {
                        stopMotors();
                        turnLeft_Prox(left, 30);
                        driveFwd();
                        currState = CAN_LEFT;
                        free(result);
                        return;
                }

                // on right
                else {
                        stopMotors();
                        turnRight_Prox(right, 30);
                        driveFwd();
                        currState = CAN_RIGHT;
                        free(result);
                        return;
                }
        }

        if(result) {
                free(result);
        }
    }
}

void canLeft() {

    u08 left, right, wall_L, wall_R;
    int timer = 0;
    adjust = RIGHT;
```

```
// while can is not set or timer expires
while(1) {
	if(isCanSet()) {
		stopMotors();
		openClaw();
		timer = lowerClawTimed();
		if(timer >= CLAW_DOWN_TIME) {
			driveReverseSlow();
			delayMs(300);
			stopMotors();
			openClaw();
			timer = lowerClawTimed();

			if(timer >= CLAW_DOWN_TIME) {
				shakeClaw();
			}

			driveFwd();
			closeClaw(SLOW);
			delayMs(300);
			stopMotors();
		}
		closeClaw(SLOW);
		turnRight(ADJUST_TIME); // adjust right
		currState = PICK_UP_CAN;
		return;
	}
	timer++;

	if(timer >= 2000) {
		break;
	}
}

stopMotors();

if(timer >= 2000) {
	left = getProximityL();
	right = getProximityR();
	wall_L = getProximityWallFrontL();
	wall_R = getProximityWallFrontR();
	if(checkIsWall(left, wall_L) || checkIsWall(right, wall_R)) {
		stopMotors();
		currState = TURN_AND_CENTER;
		return;
	}

	if(numOfCans > 0) {
		turnRightU();
		delayMs(500);
		driveFwd();
		while(1) {
			if(isCanSet()) {
				currState = PICK_UP_CAN;
				break;
```

```
                        }

                        if(isWall()) {
                                currState = TURN_AND_CENTER;
                                return;
                        }
                }
                stopMotors();
        }
        else {
                // back up
                driveReverseSlow();
                delayMs(300);
                stopMotors();
                openClaw();
                timer = lowerClawTimed();
                if(timer >= CLAW_DOWN_TIME) {
                        shakeClaw();
                }

                driveFwd();
                closeClaw(SLOW);
                delayMs(300);
                stopMotors();
                currState = PICK_UP_CAN;
        }
        closeClaw(SLOW);
        turnRight(ADJUST_TIME); // adjust right
        return;
}
else {
        // if we are not stacking, open claw
        if(numOfCans == 0) {
                openClaw();
                timer = lowerClawTimed();
                if(timer >= 2000) {
                        raiseClaw();
                        driveReverseSlow();
                        delayMs(300);
                        stopMotors();
                        lowerClawTimed();
                        driveFwd();
                        closeClaw(SLOW);
                        delayMs(400);
                        stopMotors();
                }
                else {
                        closeClaw(SLOW);
                }

                turnRight(ADJUST_TIME); // adjust right
                currState = PICK_UP_CAN;
                return;
        }
        else {
```

```
                turnRight(ADJUST_TIME); // adjust right
                currState = STACK_CAN;
                return;
            }
        }
}

void canRight() {
        u08 left, right, wall_L, wall_R;
        int timer = 0, clawTimer = 0;
        adjust = LEFT;
        // while can is not set or timer expires
        while(1) {
            if(isCanSet()) {
                stopMotors();
                openClaw();
                clawTimer = lowerClawTimed();

                if(clawTimer >= CLAW_DOWN_TIME) {
                    driveReverseSlow();
                    delayMs(300);
                    stopMotors();
                    openClaw();
                    timer = lowerClawTimed();
                    if(timer >= CLAW_DOWN_TIME) {
                        shakeClaw();
                    }

                    driveFwd();
                    closeClaw(SLOW);
                    delayMs(300);
                    stopMotors();
                }
                closeClaw(SLOW);
                turnLeft(ADJUST_TIME); // adjust left
                currState = PICK_UP_CAN;
                return;
            }
            timer++;

            if(timer >= 2000) {
                break;
            }
        }

        stopMotors();

        if(timer >= 2000) {
            left = getProximityL();
            right = getProximityR();
            wall_L = getProximityWallFrontL();
            wall_R = getProximityWallFrontR();
            if(checkIsWall(left, wall_L) || checkIsWall(right, wall_R)) {
                stopMotors();
                currState = TURN_AND_CENTER;
```

```
                return;
        }

        if(numOfCans > 0) {
                turnRightU();
                delayMs(500);
                driveFwd();
                while(1) {
                        if(isCanSet()) {
                                currState = STACK_CAN;
                                break;
                        }

                        if(isWall()) {
                                currState = TURN_AND_CENTER;
                                stopMotors();
                                return;
                        }
                }
                stopMotors();
        }
        else {
                // back up
                driveReverseSlow();
                delayMs(300);
                stopMotors();
                openClaw();
                timer = lowerClawTimed();
                if(timer >= CLAW_DOWN_TIME) {
                        shakeClaw();
                }

                driveFwd();
                closeClaw(SLOW);
                delayMs(300);
                stopMotors();
                currState = PICK_UP_CAN;
        }
        closeClaw(SLOW);
        turnLeft(ADJUST_TIME); // adjust left
        return;
}
else {
        // if we are not stacking, open claw
        if(numOfCans == 0) {
                openClaw();
                timer = lowerClawTimed();
                if(timer >= 2000) {
                        raiseClaw();
                        driveReverseSlow();
                        delayMs(300);
                        stopMotors();
                        lowerClawTimed();
                        driveFwd();
                        closeClaw(SLOW);
```

```
                        stopMotors();
                }
                else {
                        closeClaw(SLOW);
                }

                turnLeft(ADJUST_TIME); // adjust left
                currState = PICK_UP_CAN;
                return;
            }
            else {
                turnLeft(ADJUST_TIME); // adjust left
                currState = STACK_CAN;
                return;
            }
        }
}

void turnAndCenter() {
        if(sideOfEndZone == RIGHT_SIDE) {
                sideOfEndZone = LEFT_SIDE;
                currSpeedRight = FWD_SLOW_R;
        }
        else {
                sideOfEndZone = RIGHT_SIDE;
                currSpeedRight = FWD_SLOW_R + 2;
        }
        driveReverse();
        delayMs(500);
        turn180(RIGHT_DIR);
        driveReverseCenter();
        while(!isWallRearBoth());
        currState = FIND_CAN;
}

/**
 * State that picks up a can.
 */
void pickUpCan() {

        raiseClaw();
        numOfCans++;
        totalCans++;

        // if we reached our limit on cans, drop them off
        if(numOfCans == MAX_CANS || totalCans == MAX_CANS_TOTAL) {
                currState = DRIVE_TO_ENDZONE;
        }
        else {
                currState = FIND_CAN;
        }
}

/**
 * State that stacks cans.
```

```
 */
void stackCan() {

	servo2(CLAW_PLATFORM_SERVO, STACK_SERVO_SPEED);
	delayMs(CLAW_DOWN_TIME_STACK);
	servoOff(CLAW_PLATFORM_SERVO);
	openClaw();

	int timer = lowerClawTimed();

	if(timer >= 3000) {
		raiseClaw();
		driveReverseSlow();
		delayMs(300);
		stopMotors();
		timer = lowerClawTimed();

		if(timer >= CLAW_DOWN_TIME) {
			shakeClaw();
		}
		driveFwd();
		closeClaw(SLOW);
		delayMs(300);
		stopMotors();
	}
	else {
		closeClaw(SLOW);
	}

	currState = PICK_UP_CAN;
}

void dropCans() {
	lowerClawTimed();
	servoOff(CLAW_PLATFORM_SERVO); // ensure servo is off
	openClaw();
	raiseClaw();

	if(totalCans == MAX_CANS_TOTAL) {
		// we collected all of our cans...go after our opponents' cans?
		currState = DESTROY;
	}
	else {
		driveReverse();
		delayMs(500);
		stopMotors();
		delayMs(100);
		if(sideOfEndZone == RIGHT_SIDE) {
			turn90(LEFT_DIR);
		}
		else {
			turn90(RIGHT_DIR);
		}

		currState = FIND_CAN;
```

```
	}

	numOfCans = 0;
}

void driveToEndZone() {

	driveReverseSlow();
	delayMs(500);
	stopMotors();
	if(sideOfEndZone == RIGHT_SIDE) {
		turn90(RIGHT_DIR);
	}
	else {
		turn90(LEFT_DIR);
	}

	driveFwd();
	while(!isWall());
	stopMotors();
	currState = DROP_CANS;
}

void destroy() {

	turn180(RIGHT_DIR);
	driveFwd();

	while(!isWall());
	stopMotors();
	turn90(RIGHT_DIR);

	// keep going until time runs out
	while(1) {
		driveFwd();
		while(!isWall());
		stopMotors();
		turn180(RIGHT_DIR);
	}

}

int main(int argc, char *argv[]) {

	initialize();

	// actual FSM
	while(currState != POWER_OFF) {
		switch(currState) {
			case CALIBRATION:
				calibrate();
				break;
			case FIND_CAN:
				findCan();
				break;
```

```
                        case CAN_LEFT:
                               canLeft();
                               break;
                        case CAN_RIGHT:
                               canRight();
                               break;
                        case TURN_AND_CENTER:
                               turnAndCenter();
                               break;
                        case PICK_UP_CAN:
                               pickUpCan();
                               break;
                        case STACK_CAN:
                               stackCan();
                               break;
                        case DRIVE_TO_ENDZONE:
                               driveToEndZone();
                               break;
                        case DROP_CANS:
                               dropCans();
                               break;
                        case DESTROY:
                               destroy();
                               break;
                        case POWER_OFF:
                        default:
                               return 0;
                }
        }
}
```

The following code segment is the spUtil.c utility file. This code was used in the sp.c source file to help with any utility functions used in the overall system.

```
/*
 *  spUtil.c
 *
 *  Created on: Mar 29, 2012
 *  Author: Oscar Muneton
 *          Adelaido Jimenez
 *          Fabian Rodriguez
 *
 */

#include "globals.h"

void stopMotors();
void printToScreen(char *str);
u08 getProximityL();
u08 getProximityR();
u08 getCanSetProximityAvg();
u08 getCanSetProximity();
bool isWall();
```

```
/* Turning functions */
void turnRightU();
void turnLeftU();

/**
 * Determines if the can is centered and ready to be picked up.
 * @return TRUE if the can is centered, FALSE otherwise.
 */
bool isCanSet() {
	return getCanSetProximityAvg() >= CAN_SET_THRESHOLD; //getCanSetProximityAvg()
>= CAN_SET_THRESHOLD;
}

/**
 * Opens the claw
 */
void openClaw() {
	servo2(LEFT_CLAW_SERVO, LEFT_CLAW_OPEN);
	servo2(RIGHT_CLAW_SERVO, RIGHT_CLAW_OPEN);
}

/**
 * Opens the claw slightly
 */
void openClawSlightly() {
	servo2(LEFT_CLAW_SERVO, LEFT_CLAW_CLOSE-10);
	servo2(RIGHT_CLAW_SERVO, RIGHT_CLAW_CLOSE-10);
}


/**
 * Closes the claw
 * @param speed Determines the speed to close the claw.  Possible values are:
 *              SLOW, FAST, UNBOUNDED.
 */
void closeClaw(enum CL_SPEED speed) {

	bool f0, f1;
	u08 left, right;

	// Left  claw - OPEN -> CLOSE == addition
	// Right claw - OPEN -> CLOSE == subtraction
	s08 lStart = LEFT_CLAW_OPEN + 1, rStart = RIGHT_CLAW_OPEN - 1;

	switch(speed) {
		case SLOW:
			while(lStart != LEFT_CLAW_CLOSE || rStart != RIGHT_CLAW_CLOSE) {

				if(lStart < LEFT_CLAW_CLOSE) {
					servo2(LEFT_CLAW_SERVO, lStart);
					lStart++;
				}

				if(rStart > RIGHT_CLAW_CLOSE) {
```

```
                                servo2(RIGHT_CLAW_SERVO, rStart);
                                rStart--;
                        }

                        delayMs(10);
                }
                break;

        default:
                servo2(LEFT_CLAW_SERVO, LEFT_CLAW_CLOSE);
                servo2(RIGHT_CLAW_SERVO, RIGHT_CLAW_CLOSE);
                break;
    }
}

void stopMotors() {
    brake0(255);
    brake1(255);
}

/**
 * Prints the String to the LCD screen by clearing the screen, resetting the
 * cursor and then displaying the text.
 */
void printToScreen(char *str) {
    clearScreen();
    printString(str);
}

/**
 * Returns the left proximity sensor's current reading.
 */
u08 getProximityL() {
    return analog(L_PROX_INPUT);
}

/**
 * Returns the right proximity sensor's current reading.
 */
u08 getProximityR() {
    return analog(R_PROX_INPUT);
}

/**
 * Returns the front, left wall proximity sensor's current reading.
 */
u08 getProximityWallFrontL() {
    return analog(WALL_FRONT_L_PROX);
}

/**
 * Returns the front, right wall proximity sensor's current reading.
 */
u08 getProximityWallFrontR() {
    return analog(WALL_FRONT_R_PROX);
```

```
}

/**
 * Returns the can set proximity sensor's current reading.
 */
u08 getCanSetProximity() {
	return analog(CAN_SET_PROX_INPUT);
}

/**
 * Returns an average of the can set proximity sensor's readings.
 */
u08 getCanSetProximityAvg() {
	int i;
	u16 avg = 0;

	// get average
	for(i = 0; i < NUM_OF_AVG_VALUES; i++) {
		avg += getCanSetProximity();
	}

	return avg / NUM_OF_AVG_VALUES;
}

/**
 * Returns an average of the right proximity sensor's readings.
 */
u08 getProximityRAvg() {
	int i;
	u16 avg = 0;

	// get average
	for(i = 0; i < NUM_OF_AVG_VALUES; i++) {
		avg += getProximityR();
	}

	return avg / NUM_OF_AVG_VALUES;
}

/**
 * Returns an average of the left proximity sensor's readings.
 */
u08 getProximityLAvg() {
	int i;
	u16 avg = 0;

	// get average
	for(i = 0; i < NUM_OF_AVG_VALUES; i++) {
		avg += getProximityL();
	}

	return avg / NUM_OF_AVG_VALUES;
}

/**
```

```
 * Returns an average of the front wall proximity sensor's readings.
 */
u08 getProximityWallFrontLAvg() {
	int i;
	u08 avg = 0;

	// get average
	for(i = 0; i < NUM_OF_AVG_VALUES; i++) {
		avg += getProximityWallFrontL();
	}

	return avg / NUM_OF_AVG_VALUES;
}

/**
 * Returns an average of the front wall proximity sensor's readings.
 */
u08 getProximityWallFrontRAvg() {
	int i;
	u08 avg = 0;

	// get average
	for(i = 0; i < NUM_OF_AVG_VALUES; i++) {
		avg += getProximityWallFrontR();
	}

	return avg / NUM_OF_AVG_VALUES;
}

/**
 * Retrieves all the proximity reading average values.
 * Must free the memory once you are done with it, since this
 * function mallocs the memory needed to store values.
 *
 * Note: Indexed values are as follows:
 *	Left    => 0
 *	Right   => 1
 *	WALL_FL => 2
 *	WALL_FR => 3
 */
u08 *getAllProximityAvgs() {
	int i;
	u16 avgL = 0, avgR = 0, avgWFL = 0, avgWFR = 0;
	u08 *result = (u08 *)malloc(4*sizeof(u08));

	// get averages
	for(i = 0; i < NUM_OF_AVG_VALUES; i++) {
		avgL   += getProximityL();
		avgR   += getProximityR();
		avgWFL += getProximityWallFrontL();
		avgWFR += getProximityWallFrontR();
	}

	// left = 0, right = 1, wall front R = 2,  wall front L = 3, wall rear = 4
	result[LEFT_PROXIMITY_INDEX]    = avgL   / NUM_OF_AVG_VALUES;
```

```
	result[RIGHT_PROXIMITY_INDEX]   = avgR   / NUM_OF_AVG_VALUES;
	result[WALL_FRONT_L_PROX_INDEX] = avgWFL / NUM_OF_AVG_VALUES;
	result[WALL_FRONT_R_PROX_INDEX] = avgWFR / NUM_OF_AVG_VALUES;

	return result;
}

/**
 * Determines if the numbers are within a certain range of each other
 * @param first the first number to check.
 * @param second the second number to check.
 * @param range the range to check for.
 * @return TRUE if the numbers are within the range, FALSE otherwise.
 */
bool isWithinRange(u08 first, u08 second, u08 range) {

	if((first + range <= second && first - range >= second) || (second + range <=
first && second - range >= first)) {
		return TRUE;
	}
	return FALSE;
}

bool checkIsWall(u08 side, u08 wall) {

	// if either value are within THRESHOLD units away from each other, then there
must be
	// a wall in front.
	if(wall < 120) {
		return FALSE;
	}

	if((side <= (wall + WALL_THRESHOLD) && side >= (wall - WALL_THRESHOLD)) ||
			((wall <= (side + WALL_THRESHOLD)) && (wall >= (side -
WALL_THRESHOLD)))) {
		return TRUE;
	}
	return FALSE;
}

/**
 * Determines if the robot is facing a wall.
 */
bool isWall() {
	if(getProximityWallFrontL() >= 120 || getProximityWallFrontR() >= 120) {
		if(checkIsWall(getProximityL(), getProximityWallFrontL()) ||
checkIsWall(getProximityR(), getProximityWallFrontR())) {
			return TRUE;
		}
	}
	return FALSE;
}

bool isWallRear() {
	return !digitalInput(REAR_L_SWITCH_IN) || !digitalInput(REAR_R_SWITCH_IN);
```

```
}

bool isWallRearBoth() {
	return !digitalInput(REAR_L_SWITCH_IN) && !digitalInput(REAR_R_SWITCH_IN);
}

/**
 * Enables the motors to drive forward.  The main difference between this function
 * and the other driveFwdSlow() is that this function gradually decreases the speed
 * starting at the fastest speed defined, until the desired speed is attained.
 */
void driveFwd() {
	u08 left = FWD_FAST, right = FWD_FAST;

	while(1) {
		motor1(right);
		motor0(left);
		delayMs(1);

		if(left != FWD_SLOW_L) {
			left -= 1;
		}
		if(right != currSpeedRight) {
			right -= 1;
		}

		if(left == FWD_SLOW_L && right == currSpeedRight) {
			break;
		}
	}
}

void driveReverse() {
	u08 left = REVERSE_FAST, right = REVERSE_FAST;

	while(1) {
		motor0(left);
		motor1(right);
		delayMs(1);

		if(left != REVERSE_SLOW_L) {
			left += 1;
		}
		if(right != REVERSE_SLOW_R) {
			right += 1;
		}

		if(left == REVERSE_SLOW_L && right == REVERSE_SLOW_R) {
			break;
		}
	}
}

void driveReverseSlow() {
	motor0(REVERSE_FAST);
```

```
	motor1(REVERSE_FAST);
	delayMs(50);
	motor1(REVERSE_SLOW_R);
	delayMs(50);
	motor0(REVERSE_SLOW_L);
}

void driveReverseFast() {
	motor0(REVERSE_FAST);
	motor1(REVERSE_FAST);
}

void driveReverseCenter() {
	motor0(REVERSE_CENTER);
	motor1(REVERSE_CENTER);
}

void turnRight_Prox(u08 right, int threshold) {
	turnRightU();
	while(getProximityRAvg() >= (right - threshold));
	stopMotors();
}

void turnLeft_Prox(u08 left, int threshold) {
	turnLeftU();
	while(getProximityL() >= (left - threshold));
	stopMotors();
}

void turnRightU() {

	u08 left = FWD_FAST, right = REVERSE_FAST;

	while(1) {
		motor0(left);
		motor1(right);
		delayMs(1);

		if(left != FWD_SLOW_L) {
			left -= 1;
		}

		if(right != REVERSE_SLOW_R) {
			right += 1;
		}

		if(left == FWD_SLOW_L && right == REVERSE_SLOW_R) {
			break;
		}
	}
}

void turnRight(int milliseconds) {
	turnRightU();
	delayMs(milliseconds);
```

```
	stopMotors();
}

void turnLeftU() {

	u08 left = REVERSE_FAST, right = FWD_FAST;

	while(1) {
		motor0(left);
		motor1(right);
		delayMs(1);
		if(left != REVERSE_SLOW_L) {
			left += 1;
		}

		if(right != FWD_SLOW_R) {
			right -= 1;
		}

		if(left >= REVERSE_SLOW_L && right <= FWD_SLOW_R) {
			break;
		}
	}
}

void turnLeft(int milliseconds) {
	turnLeftU();
	delayMs(milliseconds);
	stopMotors();
}

void turn90(int direction) {
	switch(direction) {
		case RIGHT_DIR:
			turnRight(800);
			break;
		case LEFT_DIR:
			turnLeft(800);
			break;
	}
}

void turn180(int direction) {
	switch(direction) {
		case RIGHT_DIR:
			turnRight(2000);
			break;
		case LEFT_DIR:
			turnLeft(2000);
			break;
	}
}

void turn360(int direction) {
	switch(direction) {
```

```
			case RIGHT_DIR:
				turnRight(5000);
				break;
			case LEFT_DIR:
				turnLeft(5000);
				break;
		}
}

void raiseClaw() {
	servo2(CLAW_PLATFORM_SERVO, PICK_UP_SERVO_SPEED);
	while(digitalInput(STACK_SWITCH_IN));
	servoOff(CLAW_PLATFORM_SERVO);
}

void raiseClawSlightly() {
	servo2(CLAW_PLATFORM_SERVO, PICK_UP_SERVO_SPEED);
	delayMs(800);
	servoOff(CLAW_PLATFORM_SERVO);
}

bool verifyCan() {
	return getCanSetProximity() >= 155;
}

void lowerClaw(bool withTime) {
	servo2(CLAW_PLATFORM_SERVO, DROP_OFF_SERVO_SPEED);

	if(withTime) {
		delayMs(CLAW_DOWN_TIME);
	}
	else {
		while(digitalInput(CLAW_SWITCH_IN));
	}
	servoOff(CLAW_PLATFORM_SERVO);
}

int lowerClawTimed() {
	int timer = 0;

	servo2(CLAW_PLATFORM_SERVO, DROP_OFF_SERVO_SPEED);
	while(digitalInput(CLAW_SWITCH_IN) && timer < CLAW_DOWN_TIME) {
		delayMs(1);
		timer++;
	}

	servoOff(CLAW_PLATFORM_SERVO);

	return timer;
}
```

The following code segment is the projectGlobals.h header file that was used in conjunction with the sp.c and spUtil.c source files.

```
//This file provides a place for users to add their own global defines,
//typedefs, and macros specific to this project. It enables multiple project
//folders to share a single copy of the XiphosLibrary, without needing to
//modify the library itself to change globals.h.

#ifndef PROJECTGLOBALS_H
#define PROJECTGLOBALS_H

#include "globals.h"
#include <stdlib.h>
#include <math.h>

/* Maximum number of cans to stack */
#define MAX_CANS        3
#define MAX_CANS_TOTAL 14

#define LEFT_SIDE  0
#define RIGHT_SIDE 1

#define ADJUST_TIME 200

/* Servo input definitions */
#define RIGHT_CLAW_SERVO     0
#define LEFT_CLAW_SERVO      1
#define CLAW_PLATFORM_SERVO  2

/* Analog inputs defined */
#define R_PROX_INPUT         0 // Right proximity sensor analog input
#define L_PROX_INPUT         1 // Left proximity sensor analog input
#define CAN_SET_PROX_INPUT   2
#define WALL_FRONT_R_PROX    3 // Wall front right proximity analog input
#define WALL_FRONT_L_PROX    4 // Wall front left proximity analog input

/* Claw definitions */
#define LEFT_CLAW_OPEN    -88
#define LEFT_CLAW_CLOSE   5
#define RIGHT_CLAW_OPEN   124
#define RIGHT_CLAW_CLOSE  12

#define LEFT_CLAW_CLOSE_HALF  -43
#define RIGHT_CLAW_CLOSE_HALF  67

/* Stacking cans initial time to lower claw */
#define CLAW_DOWN_TIME_STACK 2000

#define CLAW_DOWN_TIME 3500

/* Claw platform switch D8 -> D9 */
#define STACK_SWITCH_IN  9
#define STACK_SWITCH_OUT 8

/* Claw switch D6 -> D7 */
#define CLAW_SWITCH_IN  7
#define CLAW_SWITCH_OUT 6
```

```
/* Left rear switch D5 -> D4 */
#define REAR_L_SWITCH_IN  5
#define REAR_L_SWITCH_OUT 4

/* Right rear switch D3 -> D2 */
#define REAR_R_SWITCH_IN  3
#define REAR_R_SWITCH_OUT 2

/* Indices for when getting the proximity average values at once */
#define LEFT_PROXIMITY_INDEX    0
#define RIGHT_PROXIMITY_INDEX   1
#define WALL_FRONT_L_PROX_INDEX 2
#define WALL_FRONT_R_PROX_INDEX 3

/* Can set threshold */
#define CAN_SET_THRESHOLD 160
#define CLAW_DOWN_SET 114

#define WALL_THRESHOLD 10

#define CAN_PROX_THRESHOLD 120

/* Number of avg values when averaging analog inputs */
#define NUM_OF_AVG_VALUES  10

/* Servo speeds */
#define PICK_UP_SERVO_SPEED   50
#define DROP_OFF_SERVO_SPEED -50
#define STACK_SERVO_SPEED    -20
#define STACK_SERVO_SPEED_R   20

/* Forward speeds defined */
#define FWD_FAST 250

/* motor0 = Left motor */
#define FWD_SLOW_L 149

/* motor1 = Right motor */
#define FWD_SLOW_R 152

/* Gliding to a stop defined */
#define MOTOR_STOP 128

/* Reverse speeds defined */
#define REVERSE_FAST 20
#define REVERSE_CENTER 75
#define REVERSE_SLOW_L 105
#define REVERSE_SLOW_R 102

/* Turning directions */
#define LEFT_DIR  0
#define RIGHT_DIR 1

#define LEFT  0
#define RIGHT 1
```

```
#define NONE  2

/* "Compass" directions */
/* Note: takes about 5000 ms to turn 360 degrees */
#define NORTH      5000           // or 0
#define NORTH_WEST NORTH * (1/8)  // 625
#define WEST       NORTH * (2/8)  // 1250
#define SOUTH_WEST NORTH * (3/8)  // 1875
#define SOUTH      NORTH * (4/8)  // 2500
#define SOUTH_EAST NORTH * (5/8)  // 3125
#define EAST       NORTH * (6/8)  // 3750
#define NORTH_EAST NORTH * (7/8)  // 4375

extern int currSpeedRight;

enum STATE {
	CALIBRATION,
	FIND_CAN,
	CAN_LEFT,
	CAN_RIGHT,
	TURN_AND_CENTER,
	PICK_UP_CAN,
	STACK_CAN,
	DRIVE_TO_ENDZONE,
	DROP_CANS,
	DESTROY,
	POWER_OFF
};

/**
 * Claw close speeds.
 */
enum CL_SPEED {
	FAST,
	SLOW,
};

/**
 * Struct to keep track of current motor speeds.
 */
typedef struct speed {
	int right;
	int left;
} Speed;


/***** Line-following enums and structs NOT currently in use ******/
typedef enum SIDE {
	OPPONENT,
	OWN
} Side;

#endif //PROJECTGLOBALS_H
```