```
1  /* ============================================================*
2   * Project: 3                                                 *
3   * Title: Deliberating and Acting                             *
4   * Team: 10                                                   *
5   *     Eskridge, Brent                                        *
6   *     Lopez, Tony                                            *
7   *     Maole, Amit                                            *
8   *     Utley, Klo                                             *
9   *                                                            *
10  *============================================================*/
11
12  // Import the ability to use cmu camera library
13  #use "cmucamlib.ic"
14
15  /* ------------------------------------------------------------*
16   * Structures
17   * ------------------------------------------------------------*/
18
19  // A position in the arena
20  struct coordinate {
21      float x;
22      float y;
23  };
24
25  //A segment of a planned path
26  struct path {
27      float distance;
28      int direction;
29  };
30
31  // A destination's information
32  struct destInfo {
33      struct coordinate position;
34      int confidence;
35      int visitedFlag;
36  };
37
38  // A target's information
39  struct targetInfo {
40      struct coordinate position;
41      int confidence;
42      int closestDestIDX;
43      int retrievedFlag;
44  };
45
46  // Information used for sorting destinations
47  struct destDistance {
48      int destIdx;
49      float distance;
50  };
51
52  //Information used for sorting targets
53  struct targetDistance {
54      int targetIdx;
55      float distance;
56  };
57
58  /* ------------------------------------------------------------*
```

```
 59   * Constants
 60   * ------------------------------------------------------------*/
 61
 62  // Boolean
 63  int TRUE = 1;
 64  int FALSE = 0;
 65
 66  // Relative directions
 67  int LEFT = 1;
 68  int RIGHT = 2;
 69
 70  //servo constants
 71  int SERVO_CENTER = 2675;
 72  int SERVO_LEFT = 1200;
 73  int SERVO_RIGHT = 4060;
 74  int TICKS_PER_DEGREE = 16;
 75
 76  //heading constants
 77  int NONE = -1;
 78  int NORTH = 0;
 79  int EAST = 1;
 80  int SOUTH = 2;
 81  int WEST = 3;
 82
 83  // X or Y value empty value
 84  float EMPTY = -1.0;
 85
 86  // The maximum distance
 87  float MAX_DISTANCE = 1000.0;
 88
 89  //Path setting states
 90  int FIND_DESTINATION_PATH = 0;
 91  int FIND_TARGET_PATH = 1;
 92
 93  // States of the robot
 94  int STATE_IDLE = 0;
 95  int STATE_FIND_DESTINATION = 1;
 96  int STATE_RETRIEVE_TARGETS = 2;
 97  int STATE_RECALIBRATE_POSITION = 3;
 98  int STATE_RECALIBRATE_HEADING = 4;
 99  int STATE_PURSUE = 5;
100
101  // Retrieval sub-states
102  int SUB_STATE_IDLE = 0;
103  int SUB_STATE_LOCATE_TARGET = 1;
104  int SUB_STATE_TRAVEL_TO_TARGET = 2;
105  int SUB_STATE_RETURN_TARGET = 3;
106
107
108  // Position with respect to destinations
109  int POSITION_OUTSIDE = 0;
110  int POSITION_INSIDE = 1;
111
112
113  // Motor control states
114  int MOTOR_CONTROL_IDLE = 0;
115  int MOTOR_CONTROL_TURNING = 1;
116  int MOTOR_CONTROL_DRIVING = 2;
```

```
117
118 long MOTOR_ON_SLEEP = 100L;
119 long MOTOR_OFF_SLEEP = 60L;
120
121 // Sensor IDs
122 int IR_LEFT_ID = 2;
123 int IR_RIGHT_ID = 3;
124 int ENCODER_LEFT_ID = 0;
125 int ENCODER_RIGHT_ID = 1;
126 int LEFT_TOUCH_SENSOR = 11;
127 int RIGHT_TOUCH_SENSOR = 15;
128
129 // Motor IDs
130 int MOTOR_LEFT_ID = 0;
131 int MOTOR_RIGHT_ID = 2;
132
133 //MOTOR variables
134 int ENCODER_TOLERANCE = 2;
135 float EVENTS_PER_FOOT = 116.0;
136 int EVENT_TOLERANCE = 6;
137
138
139 // Motor power settings
140 int MOTOR_PWR_STD = 50;
141 int MOTOR_LEFT_TURN_PWR = 67;
142 int MOTOR_RIGHT_TURN_PWR = 67;
143 int MOTOR_REVERSE = -1;
144 int MOTOR_PWR_CHANGE = 2;
145 int TURN_90_TICKS = 60;
146
147 // Confidence values
148 int CONFIDENCE_UNKNOWN = 0;
149 int CONFIDENCE_HOUGEN = 1;
150 int CONFIDENCE_SENSED = 2;
151
152 // Ir Threshold Value
153 int threshold = 175;  /* value of top hat near white edge of tape */
154 int Rhat = analog(IR_RIGHT_ID);
155 int Lhat = analog(IR_LEFT_ID);
156
157 /* ------------------------------------------------------------*
158  * Global variables
159  * ------------------------------------------------------------*/
160
161 // There are 0-4 destinations
162 // Add an additional 4 in case all are incorrect
163 int destinationCount = 8;
164 struct destInfo destinations[8];
165
166 // There are 0-16 targets
167 // Wrong targets will be caught in cleanup mode
168 int targetCount = 16;
169 struct targetInfo targets[16];
170
171 //This array of paths can hold up to 10 steps, 0 to begin with
172 int planSize = 0;
173 struct path pathPlan[10];
174
```

```
175  //Global Arrays used to sort distances
176  int indices8[8];
177  int indices16[16];
178
179  // Our last known position
180  struct coordinate lastKnownPosition = { -1.0, -1.0 };
181
182  // Our current heading
183  int currentHeading = NORTH;
184
185  // The time of travel since the last known position
186  long travelTime = 0L;
187
188  // The current destination of interest
189  int destOfInterestIdx = -1;
190
191  // The current target of interest
192  int targetOfInterestIdx = -1;
193
194  // The robot's current state
195  int currentState = STATE_IDLE;
196
197  // The robot's current retrieval sub-state
198  int currentRetrievalSubState = SUB_STATE_LOCATE_TARGET;
199
200  // The state of the robot's motor control
201  int currentMotorControlState = MOTOR_CONTROL_IDLE;
202
203  // Motor power levels
204  int leftPower = 50;
205  int rightPower = 67;
206
207
208  // The position of the IR sensors
209  int irLeftPosition = POSITION_OUTSIDE;
210  int irRightPosition = POSITION_OUTSIDE;
211  int irLastLeftValue = 0;
212  int irLastRightValue = 0;
213
214  // The encoder values
215  int encoderLastLeftValue = 0;
216  int encoderLastRightValue = 0;
217
218  // Dr. Hougen's input
219  float initialData[42] = { 3.0, 5.0,  6.0, 10.0, 7.0, 6.0,
220        2.0, 8.0,  3.0, 1.0, 6.0, 3.0,
221        7.0, 3.0,  1.5, 6.5, 5.0, 3.0,
222        2.0, 3.0,  -1.0, -1.0, -1.0, -1.0,
223        -1.0, -1.0,  -1.0, -1.0, -1.0, -1.0,
224        -1.0, -1.0,  4.0, 6.0, 7.0, 2.0,
225        7.0, 8.0,  -1.0, -1.0, 1.0, 2.0 };
226
227
228
229  /* ------------------------------------------------------------*
230   * Functions
231   * ------------------------------------------------------------*/
232
```

```
233 /*
234  * Main
235  */
236 void main()
237 {
238     // Prompt the user to start us up
239     printf("\nReady...");
240     start_press();
241 
242     // Initialize the robot
243     initialize();
244 
245     //To begin, we need to find the closest destination
246     currentState = STATE_FIND_DESTINATION;
247 
248     // Main execution loop
249     while( !stop_button() )
250       {
251 
252         // Find a destination
253         if( currentState == STATE_FIND_DESTINATION )
254           {
255             printf("\nFinding destination");
256             msleep(1000L);
257             findDestination();
258         }
259 
260 
261         // Retrieve targets
262         if( currentState == STATE_RETRIEVE_TARGETS )
263           {
264             printf("\nRetrieving Targets for Dest %d", destOfInterestIdx);
265             msleep(1000L);
266             retrieveTargetsForCurrentDestinatior();
267             //After a target has just been retrieved, we need another one
268         }
269         /*
270         // Pursue the other robot
271         if( currentState == STATE_PURSUE )
272           {
273             printf("\nDone!!!");
274         }
275         */
276 
277     }
278 
279 }
280 
281 
282 /*
283  * Initialize the robot
284  */
285 void initialize()
286 {
287     // Initialize the CMU cam and set the lighting
288     //init_camera();
289     //clamp_camera_yuv();
290 
```

```
291      //Initialize servo
292      init_expbd_servos(1);
293
294      //disable, then enable both encoders
295      disable_encoder(ENCODER_LEFT_ID);
296      disable_encoder(ENCODER_RIGHT_ID);
297      enable_encoder(ENCODER_LEFT_ID);
298      enable_encoder(ENCODER_RIGHT_ID);
299
300      // Populate the world model
301      populateWM();
302
303  }
304
305
306  /*
307   * Populate the world model from the initial data
308   */
309  void populateWM()
310  {
311      int i = 0;
312
313      // The first 32 pieces of data are target info
314      for( i = 0; i < 16; i++ )
315        {
316          targets[i].position.x = initialData[ 2 * i ];
317          targets[i].position.y = initialData[ 2 * i + 1];
318          targets[i].confidence = CONFIDENCE_HOUGEN;
319          targets[i].closestDestIDX = -1;
320          targets[i].retrievedFlag = FALSE;
321      }
322      // The next 8 pieces of data are destination info
323      for( i = 0; i < 4; i++ )
324        {
325          destinations[i].position.x = initialData[ 2 * i + 32];
326          destinations[i].position.y = initialData[ 2 * i + 33];
327          destinations[i].confidence = CONFIDENCE_HOUGEN;
328          destinations[i].visitedFlag = FALSE;
329      }
330
331      // Initialize the rest of the destinations with empty values
332      for( i = 4; i < 8; i++ )
333        {
334          destinations[i].position.x = EMPTY;
335          destinations[i].position.y = EMPTY;
336          destinations[i].confidence = CONFIDENCE_UNKNOWN;
337          destinations[i].visitedFlag = FALSE;
338      }
339      // The last 2 pieces of data are our initial position
340      lastKnownPosition.x = initialData[40];
341      lastKnownPosition.y = initialData[41];
342
343      //we now attempt to set the closest destination index for each variable
344      for ( i = 0; i < 16; i++ )
345        {
346          if (targets[i].position.x != EMPTY)
347            {
348              targets[i].closestDestIDX = findClosestDestinationIdxForTargetIdx(i);
```

```
349         }
350     }
351 
352     //initialize all of the pathPlan's 10 elements to be empty
353     for ( i = 0; i < 9; i++)
354     {
355         pathPlan[i].direction = NONE;
356         pathPlan[i].distance = EMPTY;
357     }
358     printf("\nWorld Populated");
359     msleep(1000L);
360 }
361 
362 
363 /*
364  * Find the closest unvisited destination and move towards it
365  */
366 void findDestination()
367 {
368     // Get the closest unvisited destination index
369     int closestIdx = getClosestUnvisitedDestinationIdx();
370 
371     //if the index is -1, there are no unvisited destinations
372     if (closestIdx == -1)
373       {
374         printf("\nNo More Dest");
375         msleep(1000L);
376         //TODO: change to cleanup
377         currentState = STATE_PURSUE;
378     }
379     else
380       {
381         printf("\nDest found: %d", closestIdx);
382         msleep(1000L);
383         //if the index is not -1, we have a destination to visit
384         destOfInterestIdx = closestIdx;
385         // Plot a course
386         setPath(destinations[destOfInterestIdx].position.x, destinations[
            destOfInterestIdx].position.y, FIND_DESTINATION_PATH);
387         //Run the course
388         executePath();
389         //Move on to retrieving targets
390         currentState = STATE_RETRIEVE_TARGETS;
391     }
392 }
393 
394 
395 /*
396  * Retrieve the targets for the current destination of interest
397  */
398 void retrieveTargetsForCurrentDestinatior()
399 {
400     // Determine what current target of interest is.  If we don't have one,
401     // either find one, or move on to the next state.
402 
403     int targetIDX = -1;
404     float newBearing = 0.0;
405     int verified = FALSE;
```

```
406
407       // Do we need to locate a target?
408       if( currentRetrievalSubState == SUB_STATE_LOCATE_TARGET )
409         {
410           targetIDX = getClosestUnretrievedTargetIdx();
411           //TODO: modify what happens if there is no target
412           if (targetIDX != -1)
413             {
414               newBearing = getBearingToCoordinate
415                 (targets[targetIDX].position.x, targets[targetIDX].position.y);
416               turnCamera(newBearing);
417
418               //TODO: verify cube presence with camera
419               verified = TRUE;//verifyCube();
420
421               if (verified)
422                 {
423                   setPath(targets[targetIDX].position.x,
                      targets[targetIDX].position.y, FIND_TARGET_PATH);
424
425                   //move onto traveling towards cube
426                   currentRetrievalSubState = SUB_STATE_TRAVEL_TO_TARGET;
427
428               }
429               else
430                 {
431                   currentRetrievalSubState = SUB_STATE_LOCATE_TARGET;
432                   targets[targetIDX].retrievedFlag = TRUE;
433
434               }
435
436           }
437           else
438             {
439               printf("\nNo more targets");
440               msleep(1000L);
441               destinations[destOfInterestIdx].visitedFlag = TRUE;
442               currentState = STATE_FIND_DESTINATION;
443
444               currentRetrievalSubState = SUB_STATE_LOCATE_TARGET;
445           }
446       }
447
448       // Do we need to travel to a target?
449       if( currentRetrievalSubState == SUB_STATE_TRAVEL_TO_TARGET )
450         {
451           executePath();
452           beep();
453           printf("\nTarget  %d retrieved!!!", targetIDX);
454           msleep(1000L);
455           align();
456           targets[targetIDX].retrievedFlag = TRUE;
457           currentRetrievalSubState = SUB_STATE_LOCATE_TARGET;
458
459           // If the target bump sensor is on, we have the target
460           // and we need to return it
461           //setPath(destinations[destOfInterestIdx].position.x,
              destinations[destOfInterestIdx].position.x, FIND_DESTINATION_PATH);
```

```
462           //currentRetrievalSubState = SUB_STATE_RETURN_TARGET;
463       }
464   }
465
466   //Set the path the robot should follow
467   void setPath(float x, float y, int pathType)
468   {
469       int xDirection;
470       int yDirection;
471       int tempHeading = currentHeading;
472
473       float xDistance = x - lastKnownPosition.x;
474       float yDistance = y - lastKnownPosition.y;
475
476       //if the desired position has the same x or y, we do not plan a step
477       if (xDistance == 0.0)
478         xDirection = NONE;
479       if (yDistance == 0.0)
480         yDirection = NONE;
481
482       //relative directions
483       if (xDistance < 0.0)
484         {
485           xDirection = WEST;
486           xDistance *= -1.0;
487       }
488       if (xDistance > 0.0)
489         xDirection = EAST;
490       if (yDistance < 0.0)
491         {
492           yDirection = NORTH;
493           yDistance *= -1.0;
494       }
495       if (yDistance > 0.0)
496         yDirection = SOUTH;
497
498       //if the robot is not facing towards the cube, we insert a step to turn towards
          it
499       if (tempHeading != xDirection && tempHeading != yDirection)
500         {
501           if (tempHeading == xDirection + 1 || tempHeading == xDirection - 1)
502             {
503               insertPlanElement(xDirection, 0.0);
504               tempHeading = xDirection;
505           }
506           else
507             {
508               insertPlanElement(yDirection, 0.0);
509               tempHeading = yDirection;
510           }
511       }
512       //we begin with the step in the direction the robot is now facing
513       if (tempHeading == xDirection)
514         {
515           insertPlanElement(xDirection, xDistance);
516           insertPlanElement(yDirection, yDistance);
517           if (pathType == FIND_TARGET_PATH)
518           {
```

```
519             insertPlanElement((xDirection + 2) % 4, xDistance);
520             insertPlanElement((yDirection + 2) % 4, yDistance);
521         }
522     }
523     if (tempHeading == yDirection)
524       {
525         insertPlanElement(yDirection, yDistance);
526         insertPlanElement(xDirection, xDistance);
527         if (pathType == FIND_TARGET_PATH)
528         {
529             insertPlanElement((yDirection + 2) % 4, yDistance);
530             insertPlanElement((xDirection + 2) % 4, xDistance);
531         }
532     }
533
534     //our work here is done
535 }
536
537 //insert a path segment to be executed
538 void insertPlanElement(int pathDirection, float pathDistance)
539 {
540     pathPlan[planSize].direction = pathDirection;
541     pathPlan[planSize].distance = pathDistance;
542     planSize++;
543 }
544
545 //motor command:  execute the path
546 void executePath()
547 {
548     while(planSize > 0)
549       {
550         //turn the direction
551         turnDirection(pathPlan[0].direction);
552         //go the distance
553         travelDistance(pathPlan[0].distance);
554         //remove the step when it has been done
555         removePathElement();
556     }
557 }
558
559
560 //deletes the first item in the plan array
561 void removePathElement()
562 {
563     int i;
564
565     for(i = 0; i < 8; i++)
566       {
567         pathPlan[i].direction = pathPlan[i+1].direction;
568         pathPlan[i].distance = pathPlan[i+1].distance;
569     }
570
571     planSize--;
572
573     //initialize the last step to empty
574     pathPlan[9].direction = NONE;
575     pathPlan[9].distance = EMPTY;
576
```

```
577 }
578
579 //Turns the robot so that it is facing the desired direction
580 void turnDirection(int pathHeading)
581 {
582     int dirDiff = pathHeading - currentHeading;
583
584     //uses the directions as numbers
585     if (dirDiff == -3 || dirDiff == 1)
586       {
587         turn(RIGHT);
588     }
589     if (dirDiff == -2 || dirDiff == 2)
590       {
591         turn(RIGHT);
592         turn(RIGHT);
593     }
594     if (dirDiff == -1 || dirDiff == 3)
595       {
596         turn(LEFT);
597     }
598 }
599
600 //move forward, the desired distance
601 void travelDistance(float feet)
602 {
603     int eventCount = (int) ( EVENTS_PER_FOOT * feet );
604     int leftEncoder = 0;
605     int rightEncoder = 0;
606
607
608 //    printf("\nTraveling %f feet", feet);
609 //    msleep(1000L);
610
611
612     reset_encoder( ENCODER_LEFT_ID );
613     reset_encoder( ENCODER_RIGHT_ID );
614
615     while( leftEncoder < eventCount - EVENT_TOLERANCE )
616       {
617         leftEncoder = read_encoder( ENCODER_LEFT_ID );
618         rightEncoder = read_encoder( ENCODER_RIGHT_ID );
619
620         if( leftEncoder - rightEncoder > ENCODER_TOLERANCE )
621           {
622             leftPower -= MOTOR_PWR_CHANGE;
623             rightPower += MOTOR_PWR_CHANGE;
624         }
625         else
626           {
627             if( rightEncoder - leftEncoder > ENCODER_TOLERANCE )
628               {
629                 leftPower += MOTOR_PWR_CHANGE;
630                 rightPower -= MOTOR_PWR_CHANGE;
631             }
632         }
633
634         motor( MOTOR_LEFT_ID, leftPower );
```

```
635         motor( MOTOR_RIGHT_ID, rightPower );
636         msleep( 100L );
637     }
638 
639     motor( MOTOR_LEFT_ID, -30 );
640     motor( MOTOR_RIGHT_ID, -30 );
641     msleep( 100L );
642     ao();
643 
644 
645     if (currentHeading == 0)
646       lastKnownPosition.y -= feet;
647     if (currentHeading == 1)
648       lastKnownPosition.x += feet;
649     if (currentHeading == 2)
650       lastKnownPosition.y += feet;
651     if (currentHeading == 3)
652       lastKnownPosition.x -= feet;
653 
654 }
655 
656 //make sure the cube exists
657 int verifyCube()
658 {
659     if (track_orange() > 100 && track_x >= -5 && track_x <= 5)
660       return TRUE;
661     else
662       return FALSE;
663 }
664 
665 
666 /* ------------------------------------------------------------*
667  * Motor functions
668  * ------------------------------------------------------------*
669  */
670 
671 /*
672  * Turn the robot the specified number of degrees
673  *   degreesToTurn - The # of degrees to turn
674  */
675 void turn( int turnType )
676 {
677 
678     int startEncoder = read_encoder(ENCODER_LEFT_ID);
679     int leftMotorPower = MOTOR_LEFT_TURN_PWR * -1;
680     int rightMotorPower = MOTOR_RIGHT_TURN_PWR;
681 
682     if (turnType == RIGHT)
683       {
684         leftMotorPower *= -1;
685         rightMotorPower *= -1;
686     }
687 
688     while( read_encoder(ENCODER_LEFT_ID) - startEncoder < TURN_90_TICKS )
689       {
690         printf("\nLE: %d", read_encoder(ENCODER_LEFT_ID));
691         pulseMotor(leftMotorPower, rightMotorPower);
692     }
```

```
693
694
695     //modify the new heading based on right or left turn
696     if (turnType == RIGHT)
697       currentHeading = (currentHeading + 1) % 4;
698     else
699       currentHeading = (currentHeading + 3) % 4;
700 }
701
702 void pulseMotor(int motorLeftPwr, int motorRightPwr)
703 {
704
705     int j;
706     for(j=1; j<=8; j++)
707       {
708         sleep(0.1);
709         motor(MOTOR_LEFT_ID, (int)(motorLeftPwr*j)/8);
710         motor(MOTOR_RIGHT_ID, (int)(motorRightPwr*j)/8);
711     }
712
713     msleep( MOTOR_ON_SLEEP );
714
715     motor(MOTOR_LEFT_ID, 0);
716     motor(MOTOR_RIGHT_ID, 0);
717
718     msleep( MOTOR_OFF_SLEEP );
719
720 }
721
722
723 //turnCamera
724 void turnCamera(float newBearing)
725 {
726     /*
727         int position = 0;
728         float TICKS_PER_DEGREE = 4000.0 / 180.0;
729
730         //Orient the robot to the North
731         turnDirection(NORTH);
732
733         //If the bearing is between 90 and 270, we cannot turn the camera to see the
          object
734         if (newBearing >= 90.0 && newBearing <= 180.0)
735           {
736             printf("\nTurning Right...");
737             msleep(1000L);
738             turn(RIGHT);
739             newBearing -= 90.0;
740         }
741         else
742           {
743             if (newBearing > 180.0 && newBearing <= 270.0)
744               {
745                 printf("\nTurning Left...");
746                 msleep(1000L);
747                 turn(LEFT);
748                 newBearing -= 270.0;
749             }
```

```
750         }
751
752
753         //At this point, the newBearing should be between -90 and 90,
754         //which is easy to accomplish on the servo
755     position = SERVO_CENTER + (int) (newBearing * TICKS_PER_DEGREE);
756     if (position > SERVO_RIGHT)
757       position = SERVO_RIGHT;
758         servo0 = position;
759 */
760 }
761
762     /* ------------------------------------------------------------*
763  * Utility functions
764  * ------------------------------------------------------------*/
765
766     /*
767  * Method to calculate the closest destination for a target
768  *   idx - The index of the target
769  *   Return - The index of the closest destination
770  */
771 int findClosestDestinationIdxForTargetIdx( int idx )
772 {
773     return getClosestDestinationIndexToPositior( targets[idx].position.x,
774                                                  targets[idx].position.y);
775 }
776
777 /*
778  * Method to add a newly found destination
779  *   x - The x value of the coordinate
780  *   y - The y value of the coordinate
781  */
782 void addFoundDestination( float x, float y )
783 {
784     // Find the index to use
785     int index = findFirstEmptyDestinationSlotIdx();
786     int i = 0;
787
788     destinations[index].position.x = x;
789     destinations[index].position.y = y;
790     destinations[index].confidence = CONFIDENCE_SENSED;
791
792     //we now attempt to set the closest destination index for each variable
793     for ( i = 0; i < 16; i++ )
794     {
795         if (targets[i].position.x != -1.0)
796           targets[i].closestDestIDX = findClosestDestinationIdxForTargetIdx(i);
797     }
798 }
799
800
801 /*
802  * Method to add a newly found target
803  *   x - The x value of the coordinate
804  *   y - The y value of the coordinate
805  */
806 void addFoundTarget( float x, float y )
807 {
```

```
808      // Find the index to use
809      int index = findFirstEmptyTargetSlotIdx();
810
811      targets[index].position.x = x;
812      targets[index].position.y = y;
813      targets[index].confidence = CONFIDENCE_SENSED;
814      targets[index].closestDestIDX = findClosestDestinationIdxForTargetIdx(index);
815  }
816
817
818  /*
819   * Clears the indicated destination.  This is done when a destination is not found
820   * (within the error) of the specified location.
821   * index - The index of the destination to clear
822   */
823  void clearDestinationAtIdx( int index )
824  {
825      destinations[index].position.x = EMPTY;
826      destinations[index].position.y = EMPTY;
827      destinations[index].confidence = CONFIDENCE_UNKNOWN;
828      destinations[index].visitedFlag = FALSE;
829  }
830
831
832  /*
833   * Clears the indicated target.  This is done when a target is not found
834   * (within the error) of the specified location.
835   * index - The index of the target to clear
836   */
837  void clearTargetAtIdx( int index )
838  {
839      targets[index].position.x = EMPTY;
840      targets[index].position.y = EMPTY;
841      targets[index].confidence = CONFIDENCE_UNKNOWN;
842      targets[index].closestDestIDX = -1;
843      targets[index].retrievedFlag = FALSE;
844  }
845
846
847  /*
848   * Get the index of the closest target to a destination
849   *   Return - The index
850   */
851  int getClosestUnretrievedTargetIdx()
852  {
853      int i;
854      int closestIdx = -1;
855      int currTarget = -1;
856
857      //Get the closest indices
858      getTargetIndicesSortedByDistance( lastKnownPosition.x,
859                                        lastKnownPosition.y);
860
861      // Determine which is the closest unretrieved one
862      for( i = 0; i < targetCount; i++ )
863      {
864          currTarget = indices16[i];
865          if( targets[currTarget].retrievedFlag == FALSE
```

```
866             && targets[currTarget].position.x > 0.0
867             && targets[currTarget].position.y > 0.0
868             && targets[currTarget].closestDestIDX == destOfInterestIdx)
869         {
870             closestIdx = currTarget;
871             break;
872         }
873     }
874
875     return closestIdx;
876
877 }
878
879
880 /*
881  * Get the index of the closest destination to the last known position
882  *   Return - The index
883  */
884 int getClosestDestinationIdx()
885 {
886     return getClosestDestinationIndexToPositior( lastKnownPosition.x,
887                                                  lastKnownPosition.y);
888 }
889
890
891 /*
892  * Get the index of the closest destination that hasn't been visited.
893  *  Return - The index
894  */
895 int getClosestUnvisitedDestinationIdx()
896 {
897     int i;
898     int closestIdx = -1;
899     int currDest = -1;
900
901     // Get the closest indices
902     getDestinationIndicesSortedByDistance( lastKnownPosition.x,
903                                            lastKnownPosition.y );
904
905     // Determine which is the closest unvisited one
906     for( i = 0; i < destinationCount; i++ )
907     {
908         currDest = indices8[i];
909         if( destinations[currDest].visitedFlag == FALSE
910            && destinations[currDest].position.x > 0.0
911            && destinations[currDest].position.y > 0.0)
912         {
913             closestIdx = currDest;
914             break;
915         }
916     }
917
918     return closestIdx;
919 }
920
921
922 /*
923  * Get a list of the destination indices sorted by distance from the
```

```
924   * last known position.
925   */
926  void getClosestDestinationIndices()
927  {
928      getDestinationIndicesSortedByDistance( lastKnownPosition.x,
929                                             lastKnownPosition.y );
930  }
931
932
933      /*
934   * Get the index of the closest destination to a specific position
935   *    x - The x value of the position
936   *    y - The y value of the position
937   *  Return - The index
938   */
939  int getClosestDestinationIndexToPositior( float x, float y )
940  {
941
942      getDestinationIndicesSortedByDistance( x, y );
943
944      return indices8[0];
945  }
946
947
948      /*
949   * Get a list of target indices sorted by distance from the
950   * specified coordinate.
951   *  x - The x value of the coordinate
952   *  y - The y value of the coordinate
953   *  indices - The list to put the indices into
954   */
955  void getTargetIndicesSortedByDistance( float x, float y )
956  {
957          struct targetDistance distances[16];
958          int i, j, tempIdx, sortedFlag;
959          float tempDistance;
960
961          // Get the distances to each of the destinations
962          for( i = 0; i < targetCount; i++ )
963            {
964              // Make sure it isn't empty
965              if( (targets[i].position.x < 0.0) && (targets[i].position.y < 0.0) )
966                {
967                  // It is empty, give it the max distance
968                  distances[i].targetIdx = i;
969                  distances[i].distance = MAX_DISTANCE;
970              }
971              else
972                {
973                  // Compute the distance
974                  distances[i].targetIdx = i;
975                  distances[i].distance = getDistance( x, y,
976                                                       targets[i].position.x,
                                                       targets[i].position.y );
977              }
978          }
979
980          // Sort the list
```

```
981           // Just do a simple bubble sort
982           for( i = 0; (i < targetCount) && !sortedFlag; i++ )
983             {
984               // Default to the list being sorted
985               sortedFlag = TRUE;
986               for( j = 0; j < targetCount - i - 1; j++ )
987                 {
988                   // Is the current distance greater than the next?
989                   if( distances[j].distance > distances[j + 1].distance )
990                     {
991                       // Save the current distance
992                       tempIdx = distances[j].targetIdx;
993                       tempDistance = distances[j].distance;
994 
995                       // Move the next distance
996                       distances[j].targetIdx = distances[j + 1].targetIdx;
997                       distances[j].distance = distances[j + 1].distance;
998 
999                       // Move the current distance
1000                      distances[j + 1].targetIdx = tempIdx;
1001                      distances[j + 1].distance = tempDistance;
1002
1003                      sortedFlag = FALSE;
1004                    }
1005                }
1006          }
1007          //Put the indices into the array
1008          for( i = 0; i < targetCount; i++ )
1009          {
1010               indices16[i] = distances[i].targetIdx;
1011          }
1012
1013  }
1014
1015  /*
1016   * Get a list of destination indices sorted by distance from the
1017   * specified coordinate.
1018   *  x - The x value of the coordinate
1019   *  y - The y value of the coordinate
1020   *  indices - The list to put the indices into
1021   */
1022   void getDestinationIndicesSortedByDistance( float x, float y )
1023        {
1024          struct destDistance distances[8];
1025          int i, j, tempIdx;
1026          float tempDistance;
1027          int sortedFlag = FALSE;
1028
1029          // Get the distances to each of the destinations
1030          for( i = 0; i < destinationCount; i++ )
1031            {
1032              // Make sure it isn't empty
1033              if( (destinations[i].position.x < 0.0) && (destinations[i].position.y <
                  0.0) )
1034                {
1035                  // It is empty, give it the max distance
1036                  distances[i].destIdx = i;
1037                  distances[i].distance = MAX_DISTANCE;
```

```
1038              }
1039              else
1040                {
1041                  // Compute the distance
1042                  distances[i].destIdx = i;
1043                  distances[i].distance = getDistance( x, y,
1044                                                       destinations[i].position.x,
                                                         destinations[i].position.y );
1045              }
1046          }
1047
1048          // Sort the list
1049          // Just do a simple bubble sort
1050          for( i = 0; (i < destinationCount) && !sortedFlag; i++ )
1051            {
1052              // Default to the list being sorted
1053              sortedFlag = TRUE;
1054
1055              for( j = 0; j < destinationCount - i - 1; j++ )
1056              {
1057                  // Is the current distance greater than the next?
1058                  if( distances[j].distance > distances[j + 1].distance )
1059                    {
1060                      // Save the current distance
1061                      tempIdx = distances[j].destIdx;
1062                      tempDistance = distances[j].distance;
1063
1064                      // Move the next distance
1065                      distances[j].destIdx = distances[j + 1].destIdx;
1066                      distances[j].distance = distances[j + 1].distance;
1067
1068                      // Move the current distance
1069                      distances[j + 1].destIdx = tempIdx;
1070                      distances[j + 1].distance = tempDistance;
1071                      sortedFlag = FALSE;
1072                  }
1073               }
1074          }
1075
1076          for( i = 0; i < destinationCount; i++ )
1077            {
1078              indices8[i] = distances[i].destIdx;
1079          }
1080      }
1081
1082      /*
1083    * Find the first empty target slot in the array
1084    * Return - The index of the first empty target slot
1085    */
1086      int findFirstEmptyTargetSlotIdx()
1087        {
1088          int index = -1;
1089          int i;
1090
1091          // Loop through the array
1092          for( i = 0; i < targetCount; i++ )
1093            {
1094              // Test the slot
```

```
1095            if( (targets[i].position.x < 0.0) && (targets[i].position.y < 0.0) )
1096              {
1097                // Bingo!
1098                index = i;
1099                break;
1100            }
1101        }
1102
1103        return index;
1104    }
1105
1106
1107    /*
1108     * Find the first empty destination slot in the array
1109     * Return - The index of the first empty destination slot
1110     */
1111    int findFirstEmptyDestinationSlotIdx()
1112    {
1113        int index = -1;
1114        int i;
1115
1116        // Loop through the array
1117        for( i = 0; i < destinationCount; i++ )
1118        {
1119            // Test the slot
1120            if( (destinations[i].position.x < 0.0) && (destinations[i].position.y < 0.0)
                )
1121            {
1122                // Bingo!
1123                index = i;
1124                break;
1125            }
1126        }
1127
1128        return index;
1129    }
1130
1131
1132    /*
1133     * Calculate the distance from one set of coordinates to the other
1134     * x1 - X value of first coordinate
1135     *  y1 - Y value of first coordinate
1136     *  x2 - X value of second coordinate
1137     *  y2 - Y value of second coordinate
1138     *  Return - The distance
1139     */
1140    float getDistance( float x1, float y1, float x2, float y2)
1141    {
1142        return sqrt( (x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2) );
1143    }
1144
1145
1146    /*
1147     * Get a bearing from the last calculated position and heading to
1148     * the specified coordinate.
1149     * x - The x value of the coordinate
1150     *  y - The y value of the coordinate
1151     */
```

```
1152  float getBearingToCoordinate( float x, float y )
1153  {
1154      //North is 0 degree bearing
1155
1156      float bearing = 0.0;
1157
1158      if ((x - lastKnownPosition.x)==0.0)
1159      {
1160          if ((y - lastKnownPosition.y) > 0.0)
1161            return 180.0;
1162          if ((y - lastKnownPosition.y) < 0.0)
1163            return 0.0;
1164      }
1165
1166      bearing = atan((y - lastKnownPosition.y)/(x - lastKnownPosition.x));
1167      if ((x - lastKnownPosition.x) < 0.0)
1168      {
1169          bearing += 180.0;
1170      }
1171
1172      return bearing;
1173  }
1174
1175  // The Align, right_ir, and left_ir functions were taken from Group 3's code for
1176  //       Project 1.
1177
1178  void align()  /* The robot must be staged on the edge of a box to ensure
1179                   perpendicluar allignment to drive to the next box.*/
1180  {
1181      int l,r,bump;
1182      l=0;
1183      r=0;
1184      bump=0;
1185
1186      while (Rhat < threshold || Lhat < threshold)
1187        pulseMotor(20, 20);
1188
1189
1190      /* the robot reads the left and right top hat sensor values and will try
1191   to stop the robot where they are both reading very similar values. */
1192
1193      while(bump<1000)
1194        {
1195          if(l==1 && r==1)
1196            {
1197              ao();
1198              reset_encoder( ENCODER_LEFT_ID );
1199              reset_encoder( ENCODER_RIGHT_ID );
1200              break;
1201          }
1202          l=left_ir();
1203          r=right_ir();
1204          bump=bump+1;     /* an counter was added to avoid being stuck in an
1205       infinite loop */
1206      }
1207
1208  }
1209
```

```
1210  int right_ir()  /* function for the right top hat sensor used in the stage
1211       function to control the right side of the robot and align
1212       the right sensor */
1213  {
1214      if(Rhat > threshold)
1215          return(1);
1216      else if(Rhat < threshold)
1217      {
1218          pulseMotor(0,20);
1219          return(0);
1220      }
1221
1222  }
1223
1224  int left_ir() /* same function as above except for the left side */
1225  {
1226      if(Lhat > threshold)
1227          return(1);
1228      else if(Lhat < threshold)
1229      {
1230          pulseMotor(20,0);
1231          return(0);
1232      }
1233  }
1234
```