```
#use "cmucamlib.ic"

#define MIDDLE_SERVO_POS 2000

#define LEFT_ENC 0
#define RIGHT_ENC 1
#define L_MOTOR 3
#define R_MOTOR 0
#define ENC_PER_FT 335
#define CONFIDENCE 20
#define TURN_SPEED 60
#define TURN_90_SPEED 60
#define TURN_90_CLICKS_RIGHT 95
#define TURN_90_CLICKS_LEFT 90
#define TURN_90_CLICKS 90
#define tick 0.5
#define BLACK_TAPE_VAL 170

int black_flag;
int dir_state = 4;
int hopper_full=0;
int flag,pid,mpid,exe,exe_done,checking,false_pos;


float CURRENT[]={4.0,5.5};
float TARGETS[]={7.0,3.0,2.0,3.5,6.0,4.5,6.0,6.0,1.0,7.0,1.0,9.5,6.5,10.0,7.5,10.0,-1.0,-1.0,-1.0,-
1.0,-1.0,-1.0,-1.0,-1.0,-1.0,-1.0,-1.0,-1.0,-1.0,-1.0,-1.0,-1.0};
float GOALS[]={1.0,1.0,6.0,3.0,2.0,11.0,-1.0,-1.0};

//first two elements are coordinates of waypoint, second two are directions to travel in order
float vertex[4];

//array that keeps track of targets visited/picked up, associates probabilities with unknown target
positions
float verified_targets[16];

float verified_goals[4];

//array of coordinates.  ARRAY[0]=x1, ARRAY[1]=y1, ARRAY[2]=x2, ARRAY[3]=y2, and so on....
//x1,y1 thru x16,y16 are targets.  x17,y17 thru x20,y20 are goals.  x21,y21 is origin of robot
float ARRAY[]={};

//converts the distance in feet to the distance in encoder clicks
float DistToEncoderCount(float dist)
{
    return (dist*(float) ENC_PER_FT);
}

//given 2 sets of coordinates and determines the euclidean distance
float distance(float x1,float y1,float x2,float y2)
{
    return sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
}

float abs(float x)
{
    if (x > 0.0)
      return x;
    else
      return -x;
}

void jolt()
{
    motor(R_MOTOR, -30);
    motor(L_MOTOR, -30);
    sleep(0.2);
    ao();
}

void backup()
```

```
{
    motor(R_MOTOR, -50);
    motor(L_MOTOR, -50);
    sleep(1.0);
    ao();
}

//aligns the robot with the black tape
void align()
{
    int rIR, lIR;

    rIR = analog(3);
    lIR = analog(2);

    printf("in align\n");
    while((rIR < BLACK_TAPE_VAL) || (lIR < BLACK_TAPE_VAL))
      {
        rIR = analog(2);
        lIR = analog(3);

        if (rIR >= BLACK_TAPE_VAL)
          {
            jolt();
            motor(R_MOTOR, -20);
            motor(L_MOTOR, 40);
            sleep(0.05);
            printf("right sees tape\n");
            beep();
            ao();
        }
        else
          //of the left sees black and the right one doesnt....wiggle left
          if (lIR >= BLACK_TAPE_VAL)
            {
              jolt();
              motor(L_MOTOR, -20);
              motor(R_MOTOR, 40);
              sleep(0.05);
              printf("left sees tape\n");
              beep();
              beep();
              ao();
          }
          //if neither one sees black, go forward slowly
          else
            {
              motor(R_MOTOR, 25);
              motor(L_MOTOR, 25);
          }

        sleep(0.08);
    }
    ao();
}
//this process looks for tape while the robot gathers targets
void lookForTape()
{
    int lIR, rIR;
    int i;

    float min=1000.0;

    while(1)
      {
        lIR = analog(3);
        rIR = analog(2);
        //printf("l=%d, r=%d\n", lIR, rIR);
        if ((lIR > BLACK_TAPE_VAL) || (rIR > BLACK_TAPE_VAL))
          {
            beep();
```

```
            //robot does not have a target
            if (!hopper_full)
              {
                //check to see if robot is in vicinity of a known goal
                for (i=0; i<4; i++)
                  {
                    if ((GOALS[2*i]!= -1.0) && (distance(CURRENT[0], CURRENT[1], GOALS[2*i],
GOALS[2*i+1]) < min) && (distance(CURRENT[0], CURRENT[1], GOALS[2*i], GOALS[2*i+1]) < 1.5))
                      {
                        CURRENT[0]=GOALS[2*i];
                        CURRENT[1]=GOALS[2*i+1];
                        min = distance(CURRENT[0], CURRENT[1], GOALS[2*i], GOALS[2*i+1]);
                    }
                }
                if (min > 2.0)
                  {
                    for (i=0; i<4; i++)
                      {
                        if (GOALS[2*i] == -1.0)
                          {
                            verified_goals[i]=1.0;
                            GOALS[2*i]=CURRENT[0];
                            GOALS[2*i+1]=CURRENT[1];
                            break;
                        }
                    }

                }
            }
            //hopper is full, kill executer process and drop off target in the goal
            else
              {
                kill_process(exe);
                ao();
                sleep(0.5);
                hopper_full=0;
                goStraight(.75, -40);
                tone(400.0, 0.5);
                tone(500.0, 0.5);
                tone(600.0, 0.5);
                printf("take out block and press start!\n");
                while(!start_button())
                  {}
                //check to see if you are around a known goal, of not update goal list
                for (i=0; i<4; i++)
                  {
                    if ((GOALS[2*i]!= -1.0) && (distance(CURRENT[0], CURRENT[1], GOALS[2*i],
GOALS[2*i+1]) < min) && (distance(CURRENT[0], CURRENT[1], GOALS[2*i], GOALS[2*i+1]) < 1.5))
                      {
                        CURRENT[0]=GOALS[2*i];
                        CURRENT[1]=GOALS[2*i+1];
                        min = distance(CURRENT[0], CURRENT[1], GOALS[2*i], GOALS[2*i+1]);
                    }
                }
                if (min > 2.0)
                  {
                    for (i=0; i<4; i++)
                      {
                        if (GOALS[2*i] == -1.0)
                          {
                            verified_goals[i]=1.0;
                            GOALS[2*i]=CURRENT[0];
                            GOALS[2*i+1]=CURRENT[1];
                            break;
                        }
                    }
                }
                goStraight(0.2, 40);
                min = 1000.0;
                exe_done=1;
                black_flag=1;
```

```
                sleep(1.0);
            }
        }
        min=1000.0;
        sleep(0.05);
    }

}


/*determines the nearest known goal from the current location
by updating the global var closest_goal_x and closest_goal_y */
float closest_goal_x;
float closest_goal_y;

void find_nearest_goal(float x,float y)
{
    float min, temp, fallBackX, fallBackY;
    int n, changed;

    changed = 0;
    min = 1000.0;

    for(n = 0; n < 4; n++)
      {
        if(verified_goals[n] <= 0.5)
          temp = 1000.0;
        else
          {
            temp = distance(GOALS[2*n], GOALS[2*n+1], x, y);
            printf("dist: %f\n", temp);
            fallBackX = GOALS[2*n];
            fallBackY = GOALS[2*n+1];
        }

        if(temp < min)
          {
            min = temp;
            closest_goal_x = GOALS[2*n];
            closest_goal_y = GOALS[2*n+1];
            changed = 1;
        }

        printf("min: %f\n", min);
        sleep(tick);

        // If the minimum value has not been changed, an error
        // has occurred, and the closest goal values will be
        // set to the last valid goal location.
        // This should not happen.
    }

    if(!changed)
      {
        closest_goal_x = fallBackX;
        closest_goal_y = fallBackY;
    }
}



/*determines the nearest known target from the current location
by updating the global var closest_target_x and closest_target_y */
float closest_target_x;
float closest_target_y;

void find_nearest_target(float x,float y)
{
    float min, temp, fallBackX, fallBackY;
    int n, changed;
```

```
    changed = 0;
    min = 1000.0;

    for(n = 0; n < 16; n++)
      {
        if(verified_targets[n] <= 0.5)
          temp = 1000.0;
        else
          {
            temp = distance(TARGETS[2*n], TARGETS[2*n+1], x, y);
            fallBackX = TARGETS[2*n];
            fallBackY = TARGETS[2*n+1];
        }

        if(temp < min)
          {
            min = temp;
            closest_target_x = TARGETS[2*n];
            closest_target_y = TARGETS[2*n+1];
            changed = 1;
        }

        // If the minimum value has not been changed, an error
        // has occurred, and the closest goal values will be
        // set to the last valid goal location.
        // This should not happen.
    }

    for (n = 0; n < 16; n++)
      {
        if (TARGETS[2*n]==closest_target_x)
          {
            verified_targets[n]=0.0;
        }
    }

    if(!changed)
      {
        closest_target_x = fallBackX;
        closest_target_y = fallBackY;
    }
}



//goes straight for a specified distance (in feet)
void goStraight(float distance, int speed)
{

    int rCount, lCount;
    int rSpeed, lSpeed;

    ao();
    sleep(tick);

    printf("going straight\n");
    sleep(tick);

    enable_encoder(0);
    enable_encoder(1);

    reset_encoder(0);
    reset_encoder(1);

    rSpeed = lSpeed = speed;

    motor(R_MOTOR, rSpeed);
    motor(L_MOTOR, lSpeed);

    rCount = lCount = 0;
```

```
    //while distance travelled is less than desired distance
    while(((float)rCount < DistToEncoderCount(distance)) && ((float)lCount <
DistToEncoderCount(distance)))
      {
        rCount = read_encoder(1);
        lCount = read_encoder(0);

        printf("l=%d %d, r=%d %d\n", lCount, lSpeed, rCount, rSpeed);

        //right encoder going faster than left
        if (rCount > lCount+5)
          {
            rSpeed -= 3;
            motor(R_MOTOR, rSpeed);
            lSpeed += 3;
            motor(L_MOTOR, lSpeed);
        }
        //left encoder going faster
        else
          if(lCount > rCount+5)
            {
              lSpeed -= 3;
              motor(L_MOTOR, lSpeed);
              rSpeed += 3;
              motor(R_MOTOR, rSpeed);
          }
          if ((lSpeed < 50) || (rSpeed < 50))
          {
            lSpeed += 30;
            rSpeed += 30;
        }

        sleep(0.1);
    }
    printf("finished straight\n");
    ao();
    sleep(tick);

}

//turns the robot south based on the current direction
void turnSouth()
{
    ao();
    sleep(tick);
    printf("turning south\n");
    sleep(tick);
    if(dir_state == 1)
      {
        turnRight();
        turnRight();
    }
    else if(dir_state == 2)
        turnRight();
      else if(dir_state == 3)
          {  }
        else if(dir_state == 4)
            turnLeft();

          dir_state = 3;
}

//turns the robot north based on the current direction
void turnNorth()
{
    ao();
    sleep(tick);
    printf("turning north\n");
    sleep(tick);

    if(dir_state == 1)
```

```
      {  }
    else if(dir_state == 2)
        turnLeft();
      else if(dir_state == 3)
          {
            turnLeft();
            turnLeft();
        }
        else if(dir_state == 4)
            turnRight();
          dir_state = 1;
}

//turns the robot west based on the current direction
void turnWest()
{
    ao();
    sleep(tick);
    printf("turning west\n");
    sleep(tick);
    if(dir_state == 1)
      {
        turnLeft();
    }
    else if(dir_state == 2)
        {
          turnRight();
          turnRight();
      }
      else if(dir_state == 3)
          turnRight();
        else if(dir_state == 4)
            { }
          dir_state = 4;
}

//turns the robot east based on the current direction
void turnEast()
{
    ao();
    sleep(tick);
    printf("turning east\n");
    sleep(tick);
    if(dir_state == 1)
      turnRight();
    else if(dir_state == 2)
        { }
      else if(dir_state == 3)
          turnLeft();
        else if(dir_state == 4)
            {
              turnLeft();
              turnLeft();
          }
          dir_state = 2;
}

// Goes to the vertices (waypoint, goal) that the planner generates
void execute_plan()
{
    float deltaX, deltaY;
    int n=0;

    while (n < 4)
      {
        deltaX = vertex[n] - CURRENT[0];
        deltaY = vertex[n+1] - CURRENT[1];

        if(deltaX < 0.0)
          turnNorth();
        else if(deltaX > 0.0)
```

```
            turnSouth();

          if (abs(deltaX) > 0.0)
          {
            goStraight(abs(deltaX), 70);
            brake();
            CURRENT[0] = CURRENT[0] + deltaX;
        }

        if(deltaY < 0.0)
          turnEast();
        else if( deltaY > 0.0)
            turnWest();

          if (abs(deltaY) > 0.0)
          {
            goStraight(abs(deltaY), 70);
            brake();
            CURRENT[1] = CURRENT[1] + deltaY;
        }
        n=n+2;
    }
    exe_done=1;

}

//generates the waypoint to the next segment
void planner(float cur_x,float cur_y, float tar_x, float tar_y)
{
    float tol = 0.5;
    int i;

    int path1=1;
    int path2=1;
    int count1=0;
    int count2=0;

    for (i=0; i<16; i++)
      {
        if (TARGETS[2*i]==cur_x)
          i++;
        else if (verified_targets[i] > 0.5)
            {
              //verifies that first leg of path1 is clear, if not increment object count
              if ((TARGETS[2*i+1] < (cur_y + tol)) && (TARGETS[2*i+1] > (cur_y - tol)) &&
(TARGETS[2*i] > (cur_x - tol)) && (TARGETS[2*i] < (tar_x + tol)))
                {
                  path1=0;
                  count1++;
              }
              //verifies that second leg of path1 is clear, if not increment object count
              if ((TARGETS[2*i] < (tar_x + tol)) && (TARGETS[2*i] > (tar_x - tol)) &&
(TARGETS[2*i+1] < (cur_y + tol)) && (TARGETS[2*i+1] > (tar_y - tol)))
                {
                  path1=0;
                  count1++;
              }
              //verifies that first leg of path2 is clear, if not increment object count
              if ((TARGETS[2*i+1] < (tar_y + tol)) && (TARGETS[2*i+1] > (cur_y - tol)) &&
(TARGETS[2*i] < (tar_x + tol)) && (TARGETS[2*i] > (tar_x - tol)))
                {
                  path2=0;
                  count2++;
              }
              //verifies that second leg of path2 is clear, if not increment object count
              if ((TARGETS[2*i] < (cur_x + tol)) && (TARGETS[2*i] > (tar_x - tol)) &&
(TARGETS[2*i+1] < (cur_y + tol)) && (TARGETS[2*i+1] > (cur_y - tol)))
                {
                  path2=0;
                  count2++;
              }
```

```
          }
      } //end for


      //path one is clear, take it
      if (path1==1)
        {
          vertex[0]=tar_x;
          vertex[1]=cur_y;
          vertex[2]=tar_x;
          vertex[3]=tar_y;
      }

      //path two is clear take it
      else if (path2==1)
          {
            vertex[0]=cur_x;
            vertex[1]=tar_y;
            vertex[2]=tar_x;
            vertex[3]=tar_y;
        }

        //both paths are blocked, take the path with the least number of obstacles
        else if (count1 < count2)
            {
              vertex[0]=tar_x;
              vertex[1]=cur_y;
              vertex[2]=tar_x;
              vertex[3]=tar_y;
          }
          else
            {
              vertex[0]=cur_x;
              vertex[1]=tar_y;
              vertex[2]=tar_x;
              vertex[3]=tar_y;
          }
}

//turns right 90 degrees based on the number of encoder clicks
void turnRight()
{
    int leftClicks = 0;
    int rightClicks = 0;

    ao();
    sleep(tick);

    motor(R_MOTOR, -TURN_90_SPEED);
    motor(L_MOTOR, TURN_90_SPEED);

    enable_encoder(0);
    enable_encoder(1);

    reset_encoder(0);

    while(leftClicks <= TURN_90_CLICKS_RIGHT && rightClicks <= TURN_90_CLICKS_RIGHT)
      {
        leftClicks = read_encoder(0);
        rightClicks = read_encoder(1);
        sleep(0.05);
    }

    ao();
    sleep(tick);

}

//turns left by 90 degrees based on the number of encoder clicks
void turnLeft()
```

```
{
    int leftClicks = 0;
    int rightClicks = 0;

    ao();
    sleep(tick);

    motor(R_MOTOR, TURN_90_SPEED);
    motor(L_MOTOR, -TURN_90_SPEED);

    enable_encoder(0);
    enable_encoder(1);

    reset_encoder(0);

    while(leftClicks <= TURN_90_CLICKS_LEFT && rightClicks <= TURN_90_CLICKS_LEFT)
      {
        leftClicks = read_encoder(0);
        rightClicks = read_encoder(1);
        sleep(0.05);
    }
    ao();
    sleep(tick);

}

void brake()
{
    ao();
    motor(R_MOTOR, -50);
    motor(L_MOTOR, -50);
    sleep(0.1);
    ao();
}

//aligns the robot with the cube, verifies that cube is in the hopper
void align_cube()
{
    int SumClicksTurned = 0;
    int leftClicksTurned = 0;
    int rightClicksTurned = 0;
    int ClicksTravelled =0;
    int SumClicksTravelled=0;
    float radians, deltaX, deltaY;
    int leftClicks = 0;
    int rightClicks = 0;
    int i,pos;
    int max=0;
    float theta;

    enable_encoder(0);
    enable_encoder(1);

    reset_encoder(0);
    reset_encoder(1);

    motor(R_MOTOR, TURN_SPEED);
    motor(L_MOTOR, -TURN_SPEED);

    //turn 45 degrees to the left
    while((leftClicks <= (int)((float)TURN_90_CLICKS/2.0)) && (rightClicks <=
(int)((float)TURN_90_CLICKS/2.0)))
      {
        leftClicks = read_encoder(0);
        rightClicks = read_encoder(1);
        sleep(0.05);
    }

    ao();
    sleep(0.1);
```

```
    //get initial camera reading
    track_orange();

    if ((track_confidence > 30) && (track_confidence > max))
      {
        max = track_confidence;
    }


    for (i=1; i<4; i++)
      {
        reset_encoder(0);
        reset_encoder(1);

        motor(R_MOTOR, -TURN_SPEED);
        motor(L_MOTOR, TURN_SPEED);

        leftClicks = 0;
        rightClicks = 0;

        //divides the search area into 4 slices, each with a 22.5 degree arc
        while((leftClicks <= (int)((float)TURN_90_CLICKS/4.0)) && (rightClicks <=
(int)((float)TURN_90_CLICKS/4.0)))
          {
            leftClicks = read_encoder(0);
            rightClicks = read_encoder(1);
            sleep(0.05);
        }
        ao();
        sleep(0.1);
        track_orange();

        if ((track_confidence > 30) && (track_confidence > max))
          {
            max = track_confidence;
        }
    }

    //take initial camera reading
    if (track_orange() < (max-20))
      {

        reset_encoder(0);
        reset_encoder(1);

        motor(R_MOTOR, TURN_SPEED);
        motor(L_MOTOR, -TURN_SPEED);

        leftClicks = 0;
        rightClicks = 0;

    }

    reset_encoder(0);
    reset_encoder(1);

    //turn until max confidence is seen with a certain tolerance
    while (track_confidence < max - 20)
      {
        leftClicks = read_encoder(0);
        rightClicks = read_encoder(1);
        motor(R_MOTOR, TURN_SPEED);
        motor(L_MOTOR, -TURN_SPEED);
        sleep(0.05);
        ao();
        track_orange();
    }

    ao();

    SumClicksTurned = rightClicks;
```

```
//aligns the robot with the cube and verifies cube is in hopper
while(!hopper_full)
  {
    reset_encoder(0);
    reset_encoder(1);

    track_orange();


    if ((track_x < -4) && (track_confidence > 30) && (track_y > 0)) //turns left
      {
        rightClicksTurned = read_encoder(1);
        motor(R_MOTOR, TURN_SPEED);
        motor(L_MOTOR, -TURN_SPEED);
        sleep(0.05);
        ao();
    }
    else
      if ((track_x > 4) && (track_confidence > 30) && (track_y > 0))//turns right
        {
          leftClicksTurned = read_encoder(0);
          motor(R_MOTOR, -TURN_SPEED);
          motor(L_MOTOR, TURN_SPEED);
          sleep(0.05);
          ao();
      }
      //cube is in the center, pick it up
      else if (track_confidence > 30)
          {
            ClicksTravelled = read_encoder(0);
            if (track_y < 20)
              {
                goStraight(0.55, 70);
                ClicksTravelled=(int)DistToEncoderCount(0.55);
                hopper_full = 1;
                printf("hopper full!!!\n");
            }
            else
              {
                goStraight(0.3, 70);
                ClicksTravelled=(int)DistToEncoderCount(0.3);
            }
        }
        //false positive
        else
          {
            hopper_full=0;
            false_pos=1;
            for (i=0; i<16; i++)
              {
                if ((TARGETS[2*i]==closest_target_x) && (TARGETS[2*i+1]==closest_target_y))
                  verified_targets[i]=0.0;
            }
        }

    SumClicksTravelled = SumClicksTravelled + ClicksTravelled;

}


reset_encoder(0);
reset_encoder(1);
leftClicksTurned=0;
rightClicksTurned=0;
//turn back to original direction
if ((SumClicksTurned-(int)((float)TURN_90_CLICKS/2.0)) > 0)
  {
    motor(R_MOTOR, -TURN_SPEED);
    motor(L_MOTOR, TURN_SPEED);
```

```
        while (leftClicksTurned <= SumClicksTurned-(int)((float)TURN_90_CLICKS/2.0))
          {
            leftClicksTurned = read_encoder(0);
            sleep(0.05);
        }
        ao();
    }
    else if ((SumClicksTurned-(int)((float)TURN_90_CLICKS/2.0)) < 0)
        {
          motor(R_MOTOR, TURN_SPEED);
          motor(L_MOTOR, -TURN_SPEED);

          while ((float)rightClicksTurned <= abs((float)(SumClicksTurned-
(int)((float)TURN_90_CLICKS/2.0))))
            {
              rightClicksTurned = read_encoder(1);
              sleep(0.05);
          }
          ao();
      }

      //update current position based on the degrees turned and the distance travelled.
      //The current x and current y is updated by deltaX and deltaY.  A positive theta
      //represents a right turn, and a negative theta a left turn.

      theta = (float)SumClicksTurned/(float)TURN_90_CLICKS * 90.0;
    radians =  abs(theta) * (3.141592/180.0);

    printf("theta: %f", theta);
    sleep(1.0);

    //robot is facing north
    if (dir_state == 1)
      {
        deltaY=sin(radians)*(float)SumClicksTravelled/(float)ENC_PER_FT;
        deltaX=cos(radians)*(float)SumClicksTravelled/(float)ENC_PER_FT;

        if (theta > 0.0)
          {
            CURRENT[0]=CURRENT[0]-deltaX;
            CURRENT[1]=CURRENT[1]-deltaY;
        }
        else if (theta < 0.0)
            {
              CURRENT[0]=CURRENT[0]-deltaX;
              CURRENT[1]=CURRENT[1]+deltaY;
          }
    }
    //robot is facing east
    else if (dir_state == 2)
        {
          deltaX=sin(radians)*(float)SumClicksTravelled/(float)ENC_PER_FT;
          deltaY=cos(radians)*(float)SumClicksTravelled/(float)ENC_PER_FT;

          if (theta > 0.0)
            {
              CURRENT[0]=CURRENT[0]+deltaX;
              CURRENT[1]=CURRENT[1]-deltaY;
          }
          else if (theta < 0.0)
              {
                CURRENT[0]=CURRENT[0]-deltaX;
                CURRENT[1]=CURRENT[1]-deltaY;
            }
      }
      //robot is facing south
      else if (dir_state == 3)
          {
            deltaY=sin(radians)*(float)SumClicksTravelled/(float)ENC_PER_FT;
            deltaX=cos(radians)*(float)SumClicksTravelled/(float)ENC_PER_FT;
```

```
            if (theta > 0.0)
              {
                CURRENT[0]=CURRENT[0]+deltaX;
                CURRENT[1]=CURRENT[1]+deltaY;
            }
            else if (theta < 0.0)
                {
                  CURRENT[0]=CURRENT[0]+deltaX;
                  CURRENT[1]=CURRENT[1]-deltaY;
              }
        }
        //robot is facing west
        else if (dir_state == 4)
            {
              deltaX=sin(radians)*(float)SumClicksTravelled/(float)ENC_PER_FT;
              deltaY=cos(radians)*(float)SumClicksTravelled/(float)ENC_PER_FT;
              if (theta > 0.0)
                {
                  CURRENT[0]=CURRENT[0]-deltaX;
                  CURRENT[1]=CURRENT[1]+deltaY;
              }
              else if (theta < 0.0)
                  {
                    CURRENT[0]=CURRENT[0]+deltaX;
                    CURRENT[1]=CURRENT[1]+deltaY;
                }
          }
          printf("%f, %f\n", CURRENT[0], CURRENT[1]);
}

//main process that combines the planner, executer, and the goal seeker
void running()
{
    int i,state_flag;
    //    int target=1;
    //    int goal=0;
    float points=1.0;

    flag=1;
    checking=1;
    false_pos=0;


    while(flag)
      {


        //find closest target and pass to planner
        while(!false_pos)
          {
            find_nearest_target(CURRENT[0], CURRENT[1]);
            planner(CURRENT[0], CURRENT[1], closest_target_x, closest_target_y);
            execute_plan();
            align_cube();
        }

        find_nearest_goal(CURRENT[0], CURRENT[1]);
        planner(CURRENT[0], CURRENT[1], closest_goal_x, closest_goal_y);

        exe = start_process(execute_plan());

        //while a goal is not found
        while (exe_done && !black_flag)
          {
            goStraight(0.5,60);
        }

        ao();
        flag=0;
        //check to see if any targets remain
```

```
        for (i=0; i<16; i++)
          {
            if (verified_targets[i] > 0.0)
              flag=1;
        }

    }

    //all known targets have been checked, search for unknown targets and try to tag blue bot
    state_flag=0;
    while (points <= 7.0)
      {

        if (!state_flag)
          {
            planner(CURRENT[0], CURRENT[1], points, 1.0);
            execute_plan();
            planner(CURRENT[0], CURRENT[1], points, 15.0);
            execute_plan();
            state_flag=1;
        }

        else if (state_flag)
            {
              planner(CURRENT[0], CURRENT[1], points, 15.0);
              execute_plan();
              planner(CURRENT[0], CURRENT[1], points, 1.0);
              execute_plan();
              state_flag=0;
          }

          points=points + 1.2;
    }
    printf("done!!!");

}

void main()
{

    int i;

    //initialize verified array
    for (i=0; i<16; i++)
      {
        if (TARGETS[2*i] == -1.0)
          verified_targets[i]=0.5;
        else
          verified_targets[i]=1.0;
    }

    for (i=0; i<4; i++)
      {
        if (GOALS[2*i] == -1.0)
          verified_goals[i]=0.5;
        else
          verified_goals[i]=1.0;
    }

    init_camera();// initialize the camera in YUV mode
    setWin(1,60,80,143);
    clamp_camera_yuv();
    beep();

    while(!start_button())
      {}


    pid = start_process(lookForTape());

    //go to nearest goal first
```

```
    find_nearest_goal(CURRENT[0], CURRENT[1]);
    planner(CURRENT[0], CURRENT[1], closest_goal_x, closest_goal_y);
    execute_plan();

    mpid = start_process(running());

    sleep(1.0);

    while(flag || checking)
      {}

    kill_process(pid);
    kill_process(mpid);

}
```