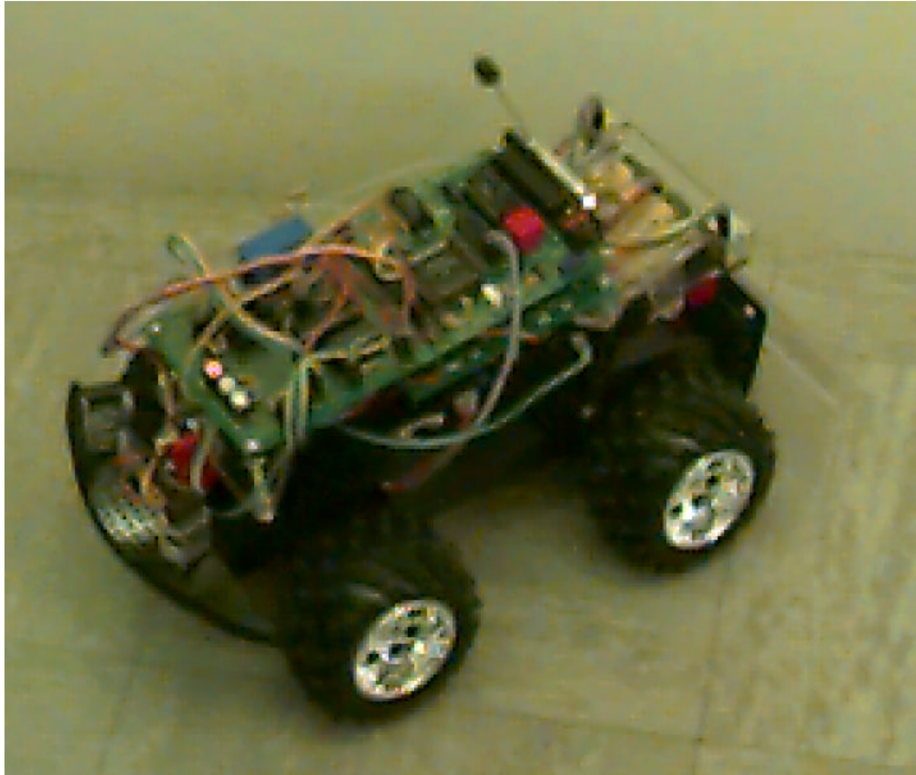

The Path Tracer:
A Path Recording Robot

Robert J. McGee
University of Florida
EEL 5666: Intelligent Machines Design Laboratory
Professor Keith L. Doty
April 25, 1997

Table of Contents

Abstract 3

Executive Summary 3

Introduction 4

Encoder Design... 4

Using the Encoder... 5

Direction System 8

Braking 10

Sensors 11

Behaviors 12

Recording a Path 12

Playing Back a Path 13

Integrated System.. 14

Platform 15

Actuation 15

Increasing Accuracy.. 16

Experimental Layout.. 17

Considerations 19

Results and Conclusions 19

Appendices 21

Abstract

The purpose of this project is to demonstrate a simple method of recording and playing back the trajectory of a robot using direction and distance segments. This can easily be done due to the fact that the radio control car used was controlled by full-throw drive and steering. Therefore, the control system can be treated digitally with an encoder to tell how far to go in each direction. Although this system is not as accurate as most of the systems being used to do this type of autonomous navigation, it is much simpler and does not require expensive positioning and orientation sensing systems. This project shows that recording a path in a blind fashion can be done with good accuracy in the play back of that path depending on and limited to the robustness of the robot's mechanical design, senses, and its knowledge of the errors at hand.

Executive Summary

This project presents a simple and inexpensive way to record and playback the path of a robot. It is done in a blind fashion, meaning that both global and local position and orientation are unknown at all times. This is accomplished by using an optical shaft encoder to measure distance and the outputs (of which there are four, which are digital) of a radio control receiver (which is controlled by an operator) as inputs to a microprocessor. The foundation of the robot is that of a small radio control car. Because the direction signals are digital, there are only 6 possible directions the robot can travel in. Three other directions are used to distinguish types of coasting. The robot is equipped with four infrared proximity sensors as well as a front and rear bumper for avoidance and collision detection behaviors. It uses these behaviors to override the operator if the situation becomes critical. Velocity of the robot is evaluated and kept at a constant level to allow for successful reaction of all behaviors. By making the robot aware and able to detect the types of error that occur in completing this task, they can be compensated for as best as possible.

Introduction

Several methods of recording and playing back the motion of a robot have been introduced in the past, however most are expensive and very complex. This project presents a simple way to accomplish this task by using a radio controlled vehicle. The robot can record the path it drives while being controlled by an operator. Using the receiver's motor leads as direction inputs and the encoder as a distance sensor, the robot's motion can be broken into separate line segments. These line segments each have a direction and distance and can be recorded as such. Playing back these commands is where the problems arise. Slippage between the recorded and playback paths must be accounted for and minimized if the robot is to play back the recorded path with fairly good accuracy.

Encoder Design

The drive system of the robot that was used is like that of a conventional rear-wheel drive automobile with the drive shaft in the rear and a four-bar steering mechanism in the front. Because the robot does not have a differential drive system, only one encoder is needed on the drive shaft for distance calculations. The encoder chosen is a pass through interference encoder with a notched wheel. There are four notches in the shaft giving it four ticks per revolution. Since the wheel diameter is 3.125 inches, there is a standard deviation of 2.454 inches due to this encoder wheel. There is already a hexagonal piece on the drive shaft. This made mounting of the encoder firmly to the wheel easy, however, because the wheels are secured to the drive shaft by press-fit nuts, they cannot be removed and reassembled easily. For this reason, the encoder wheel was mounted in another way than simply sliding the end of the shaft through the center hexagonal hole of the encoder wheel. To accomplish this the encoder was made into two separate pieces (see Figure 1) where one

piece could be slid over the hexagonal piece on the shaft. Then, the second piece could pop into the first locking the encoder wheel into place. The encoder wheel was then kept from sliding coaxially on the shaft by applying hot-glue on the both sides of the encoder wheel where it meets the shaft.

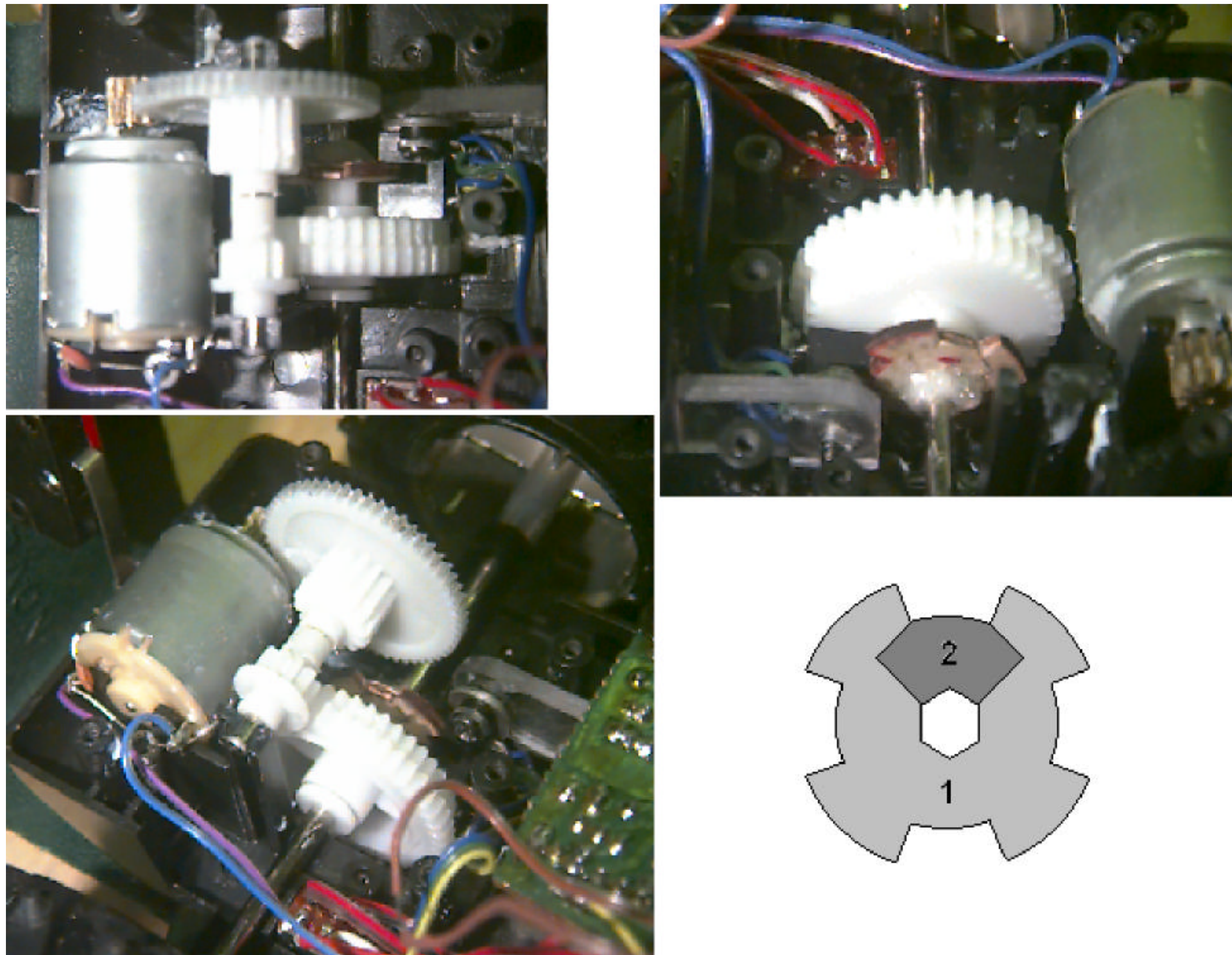

Figure 1. Encoder design for mounting inside existing gear box.

Using the Encoder

The reason for using an encoder is to sense the distance that the robot moves. In this case,

it is used to sense the distance for each new direction the robot moves in. The encoder gives this information by simply providing so many signals, or "ticks," per revolution. By knowing the wheel diameter and the number of ticks per revolution, the distance can easily be found by:

$$D = \pi * d * t / N,$$

where 'd' is the wheel diameter, 't' is the number of ticks the encoder provides, and 'N' is the number of ticks per revolution. However, the system presented in this paper has no concept of distance, nor does it care. This system simply records the ticks for each direction and plays them back. The distance the robot travels is, therefore, irrelevant except from a standpoint of accuracy to the resulting endpoints from the recorded and the played back paths.

One method of finding a tick of the encoder is by using three variables. One, measures the status of the starting position of the encoder (high or low); another measures the current status of the encoder; and the third variable indicates that the encoder is armed and ready to be triggered. When the current status is opposite to the starting status the encoder is considered armed, which is signified by setting the armed variable high (instead of low, which is what it is initialized to). Now that the encoder is armed, any time the current status is equal to the starting status, the encoder is considered triggered, meaning that one tick has just occurred. Now the armed bit must be set to low.

Through experimentation, it was observed that overshoot is one of the biggest problems associated with the accuracy of repeatability. For this reason, velocity control of the robot is critical in maintaining and accounting for overshoot. The pulse width modulation, or pwm, could just be adjusted to slow down the robot in its overall actions and limit the effects of its inertia. However, by decreasing the pwm, the robot may end up stopping or slowing more than expected in certain situations, such as when it is turning or is low on batteries. When this occurs, the robot may

overcompensate in its deceleration and stopping process, and undershoot the intended distance. By maintaining a constant speed, the robot can use the same decelerating and braking algorithms no matter what it is doing.

This can be done by calculating the difference in time between each encoder tick. By using Interactive C's mseconds command, an almost instantaneous representation of speed can be calculated. It is a representation because it is a proportionality between speed and time. The following equation shows the relationship of velocity and time:

$$V = r*(d\theta/dt),$$

where 'V' is the velocity of the robot, 'r' is the radius of the wheel, '$d\theta$' is the change in angle of the wheel, and 'dt' is the change in time. It can be seen that $d\theta$ is constant, as it is the change in the angle of the wheel between two ticks of the encoder, and each of the encoder ticks should be equally spaced radially. The wheel radius is also constant, therefore, it can be seen as dt increases, velocity is decreasing. The relationship between pwm and dt is not, however, linear. A complex equation could be used to convert dt to a pwm value, however a much simpler method is to just use conditions to obtain the desired velocity curve. Because slippage due to "peeling out" is not an issue (as was found by experimentation), the curve desired is to ramp up as quickly as possible to the desired speed and level off at that speed (as can be seen in Figure 2). This can be accomplished by finding the difference in time, call it twant, between encoder ticks at a pwm, and velocity, that is desirable to the decelerating and braking algorithms and is still slow enough to let the obstacle avoidance systems have ample time to avoid obstacles. Conditions are then based on the difference between dt and twant (dt-twant). If this difference is positive, the robot is moving slower than desired. In this case, the pwm would be set faster than the desired pwm. As the difference between dt and twant

decreases, the amount that pwm is over the desired pwm decreases, leveling the velocity off to the desired velocity. If the robot is ever traveling faster than the desired velocity, the difference between dt and twant will be negative. In this case the pwm should be set low, but still in the same direction. Although changing the direction of the motor would decelerate the robot quicker, it will be too quick and make the system out of control moving the robot in a very jerky motion. Figure 2 shows an

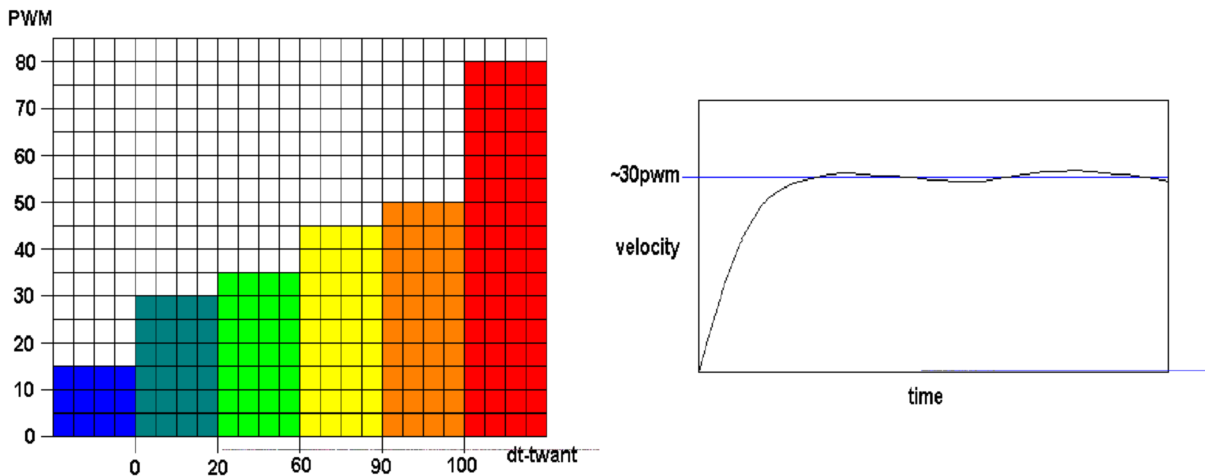

Figure 2. Algorithm chart (left) to obtain velocity curve (right).

algorithm for setting the pwm that works well for a desired pwm of between 30 and 35.

Direction System

Sensing the direction of the robot can be easily done using the receiver board. (The control of the receiver board to the motors is full-throw, meaning that the steering is either turned fully to the right or left or it is straight. The same is true for the drive motor for forward and reverse.) The circuit shown in Figure 3 was suggested by Kevin Harrelson. By using the motor leads from the receiver board as inputs, the receiver board can be treated as a black box. To tell which direction the

vehicle is going, the inputs are read. The drive motor and steering motor are read separately, each using a pair of if-else conditions as such:

```
if (analog(3) > threshold) {         /* where port 3 is the forward input */
        forward = 1;
        reverse = 0;
}
else if (analog(4) > threshold) {  /* where port 4 is the reverse input */
        reverse = 1;
        forward = 0;
}
if (analog(5) > threshold) {         /* where port 5 is the left input */
        left = 1;
        right = 0;
}
else if (analog(6) > threshold) {  /* where port 6 is the right input */
        right = 1;
        left = 0;
},
```

where analog(3) corresponds to the input port of the forward drive motor lead, analog(4) corresponds to the input port of the reverse drive motor lead, and so on. Direction is recorded as a value of 1-6 (see Figure 3) where:

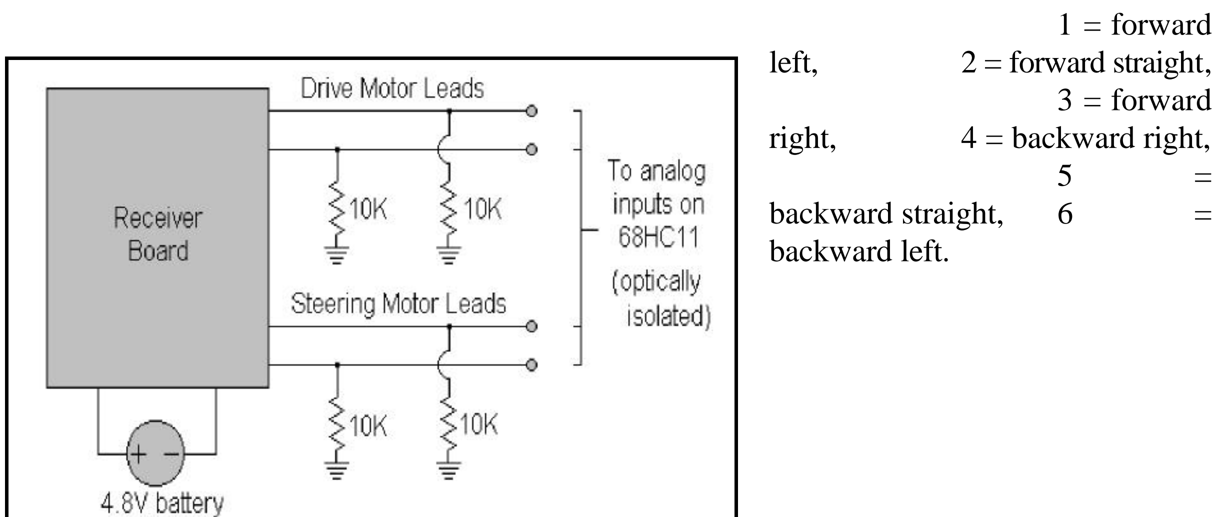

Figure 3. Reading directions as inputs.

1 = forward left, 2 = forward straight, 3 = forward right, 4 = backward right, 5 = backward straight, 6 = backward left.

However, there will be times when the robot coasts to a rest before it starts moving again. In this case the car is not being propelled by the motor, but rather decelerating to a stop due to its own inertia. This must be accounted for. After all, if the robot goes forward for a distance and then reverses, it will be more likely to skid, thereby changing orientation. The robot will behave totally different if it is driven forward for a distance, coasted to a stop, and then driven in reverse. Coasting can be detected by noticing that the robot's encoder is still ticking while there is no forward or reverse input. In this case, the last direction it moved will be the decelerating motion. Coasting is signified with a value of 0 for straight coasting, 11 for coasting to the left, and 12 for coasting to the right. When this direction is read during playback mode, the robot will go in the last direction it traveled in (direction[index-1]) very slowly for the amount of ticks that it had recorded to have come to rest and stop on that tick.

Braking

The easiest way to brake the robot is to drive the wheels in the opposite direction in which they are rolling due to the robot's inertia. If this is done too hard the wheels could lock, causing the robot to skid and change orientation without knowing it. If braking is done for too long, the robot will overcompensate its inertial momentum and go in the opposite direction for a short time. Braking can be done in many ways, depending on the type of braking (hard or soft) that is needed and the limits that can be

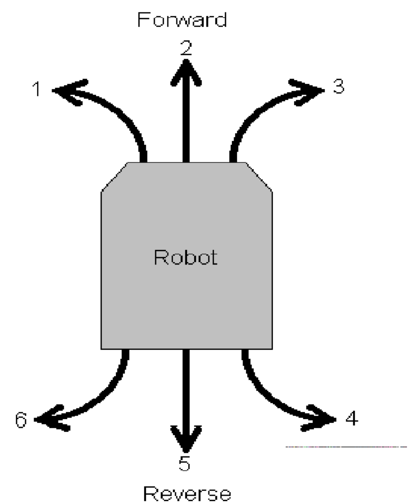

Figure 4. Direction possibilities.
0 = coasting
11 = coasting left
12 = coasting right

performed without causing slippage and/or overcompensation. One way to stop the robot is to just use a slight opposite pwm and then hold it for a short time. Anti-lock brakes can be simulated by pulsing the pwm (applying an opposite pwm for a very slight time and then setting the pwm to zero over and over). Another way is to apply a small opposite pwm and increasing it as the robot slows down. The code that was used to do this is listed in the program in the appendix of this paper.

Sensors

Many sensors are used to accomplish the tasks presented by this project. For obstacle avoidance there are two infrared proximity sensors in the front of the robot (left and right front). There will also be two infrared proximity sensors in the back of the robot (left and right rear). These sensors will be used for obstacle avoidance when backing up. For obstacle detection there is a front and a back bumper. Font and back collision will be detected separately, however, each bumper does not need to be able to tell on which side of itself (left, right or center) contact was made. If a collision occurs, repeating the path recorded will not be very precise. This is because collision with an obstacle creates slippage that is very difficult to account for. All of these collision avoidance and collision detection sensors will be used to override the operator as he drives the robot by radio control. To record the path, the robot will be equipped with an optical shaft encoder. Because the receiver's

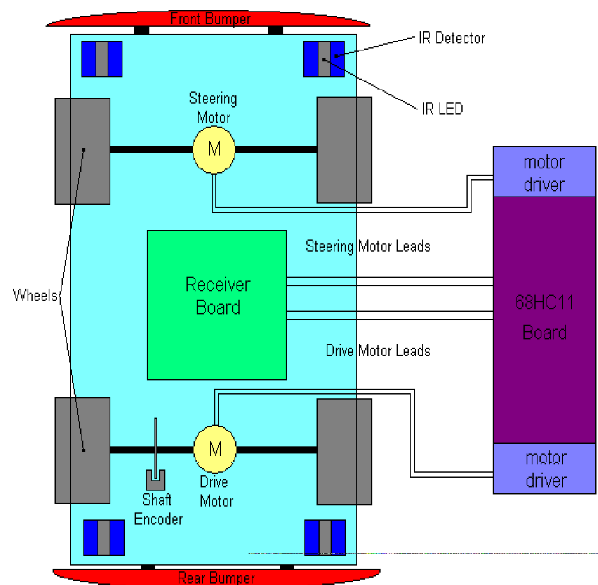

Figure 5. Sensor and design layout.

motor leads are used to sense direction, the encoder will be used as a counter for the number of ticks the robot traveled in a certain direction. All sensors and other inputs are shown in Figure 5.

Behaviors

All forms of obstacle avoidance override the operator's commands. When the one of the front IR sensors detects an obstacle, the robot turns in that corresponding direction (left or right) and backs up for so many ticks of the encoder. This keeps the robot from colliding with the obstacle and skidding undetected. If the robot happens not to see an obstacle and collides with it with its front bumper, it does the same thing as it would with the front IR sensors in a random direction. If either of the rear IR sensors or the rear bumper detects an obstacle, the robot simply goes straight forward for so many ticks of the encoder. The reason for turning if an obstacle is detected in the front but not in the back is to avoid getting trapped in a corner.

The robot uses its shaft encoder to measure distances. It also measures velocity using the encoder so that it may maintain a constant speed. The robot has the ability to both record and playback the path of an operator, while using its other behaviors to override the operator if need be. The robot may be operated by radio control, RC. At starting time, the robot uses an algorithm to drive while taking readings from its IR sensors. This is done to calculate good threshold values for each IR sensor separately while experiencing the vibrations it will during driving. Self-calibration allows the robot to be easily adjustable for different environments.

Recording a Path

The path is recorded as direction and distance segments. This means that when a certain direction is chosen, the distance that the direction is driven is recorded along with that corresponding distance. Distance is measured by encoder ticks. Direction is recorded as a value of 0-6 as described above. These values are stored in two separate one-dimensional matrices (distance[index] and direction[index]), where each cell of one matrix corresponds to the same number cell in the other matrix. For example, in record mode, the operator turns the car left while driving forward for 40 ticks and then drives straight for 30 ticks. Therefore, this path is recorded as:

direction[0] = 1 distance[0] = 40,
direction[1] = 2 distance[1] = 30.

Recording a path is done by reading the encoder. Each time the encoder ticks, the path needs to be updated. To do this the current direction is looked at. If the current direction is the same as the previous direction, nothing needs to be done unless the index is zero and the direction is zero, in which case, the direction need to be set to the current direction. After this initialization process has taken place, the path only needs to be updated when either the current direction is not equal to direction[index] or the distance cell reaches its limit (65,535 for unsigned integer). When this occurs, the distance set equal to ticks, ticks is then reset to zero, the index is incremented when either or the direction changes, and direction[index] is set equal to the new current direction. The matrices are created at the beginning of the program, therefore there length (maximum number of cells) must be long to accommodate a long driving path.

Playing Back the Recorded Path

Now that the path is stored in the two matrices, playing it back is just a matter of reading the

matrices and displaying the information through the motors of the robot correctly. First, the index and the variable ticks are set to zero. The first direction is now read. The robot then moves in that direction using the constant velocity algorithm described earlier. This persists until ticks is equal to the corresponding distance of this index. When this occurs, ticks is set to zero again and the index is incremented. Now the next direction is read. Whenever the direction reads 0, 11, or 12, the robot must try to simulate coasting. To do this the robot applies light brakes to slow down, then it moves in the last ongoing direction read (forward if direction is less than 4 and reverse otherwise) slowly while turning in the appropriate direction if direction[index] reads 11 or 12. Direction and distance are read and performed this way until either distance[index] is equal to zero or the index is equal to the maximum number minus one. This is because the distance of any segment will never be zero unless a point is reached where the path recording has stopped, since direction and distance are initialized to zero for every index as an initialization of record mode.

Integrated System

Now that the path recording and playback algorithms have been defined, they must be incorporated with the other fundamental behaviors of the robot. In other words, behaviors must be prioritized. For instance, obstacle avoidance and detection should take priority over any driven path, recorded or played back. It was assumed that some error had occurred in the path tracing if an obstacle was detected by either touch or sight of the IR sensors. For this reason, the robot avoids the obstacle and path playback is terminated. If an obstacle is detected during record mode, the operator is overridden by the obstacle avoidance routine, shutting off control of the RC transmitter while avoiding the obstacle. However, the path is still recorded while the robot avoids the obstacle.

During playback mode, there is no need for the RC transmitter, therefore, its control is shut off during the entire course of playback.

Platform

The radio controlled vehicle being used is a $50 Asahi 4.8 volt "Road Warrior". The vehicle's motors cannot be directly controlled by using both the remote control and the microprocessor's motor drivers at the same time. Doing so could cause a short circuit. For instance, if the operator tells the robot to turn left and the robot's obstacle avoidance system tells it to turn right a short circuit could occur. To control this, the motor leads from the receiver are read as digital inputs into the microprocessor. These inputs can be read by the microprocessor and used to control the motors via the microprocessor's motor drivers (see Figure 3).

Actuation

The motors being used are those that were put on the vehicle when it was manufactured. The control for both the steering and the turning motor is full-throw (meaning that it is either fully on or fully off). The steering motor drives a servo type mechanism that controls the 4-bar steering

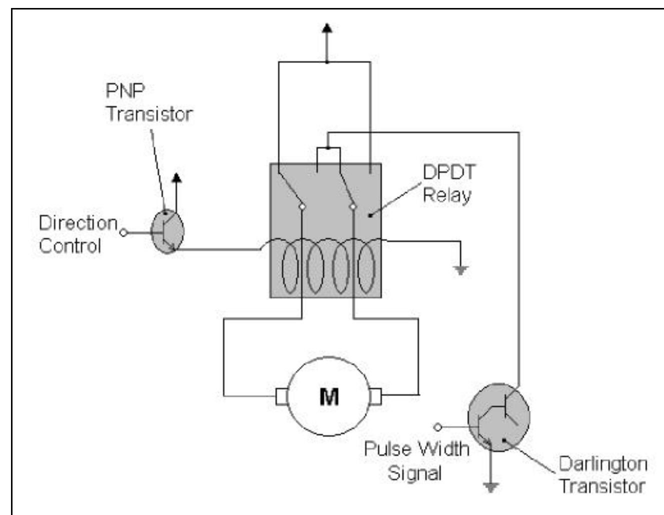

Figure 6. Circuitry to drive higher current motors.

mechanism through a series of gears. This motor stalls when the steering mechanism reaches full extension. The stall current for this motor is about 1 ampere. The drive motor controls the rear shaft of the vehicle through a series of gears (there is no differential drive system). Because the stall current of this motor is approximately 2 amperes, some additional circuitry must be added in order to use the pulse width modulated command (motor(1, 50);). Controlling this motor directly by one of the microprocessor's motor drivers would blow up the motor driver. One way to fix this problem is to use a Darlington transistor and a double pull - double throw relay. The Darlington transistor can handle much more current than the motor drivers. The relay is used to switch the direction of the current through the motor (and therefore, the direction of the motor). This circuit, which was designed by Scott Jantz, is shown in Figure 6. This circuit is connected to the point where the motor driver is, thereby bypassing the motor driver.

Increasing Accuracy

The hardest aspect of this project is accuracy between a recorded and a played back path. To increase accuracy many mechanical considerations must be taken into account. Most of the error between the two paths can be mainly attributed to some form of slippage. The main types of error that may occur are due to:

* Slippage due to collision with an obstacle.
* Slippage due to peeling out.
* Slippage due to turning at high speeds.
* Overshoot.
* Inaccuracies of steering mechanism.

* Inaccuracies of placement at starting point.

If it is found to occur, each type of error must be dealt with in a different way. Collision with obstacles must be avoided at all costs because this type of slippage is extremely difficult to calculate and account for. Peeling out can be eliminated by accelerating instead of just going at 100% pulse width modulation, pwm. Slippage due to turning at high speeds is difficult to compensate for since there will need to be some sort of delay in the operator's control during record mode so that the car may slow down before it turns. This type of error can also be eliminated by the speed control algorithm described earlier in this paper. Overshoot during playback may be accounted for by simply decelerating and/or braking. Inaccuracies of the steering mechanism may cause the robot to veer in opposite directions while trying to go straight on two separate occasions. By tweaking the steering mechanism this can be minimized, but not totally eliminated. Differences in the point of placement at the starting point during record and playback modes could be minimized by mounting a laser pointer on the robot. Before recording is begun, the laser could be used to make a dot on some wall. The dot is marked so that the same procedure can be done for the starting point placement of playback mode, and the dot will line up with the mark. Some of the slippage that occurs during the recording of a path can be neglected as it will closely repeat itself during the playback of the path.

Experimental Layout

The main concern of this project is to be able to repeat a recorded path with fairly good precision. To display the concerns, a test was performed to try to make the robot be able to travel for only 120 ticks of the encoder. The robot was run at 100% pwm for 120 ticks of the encoder. Then, the distance that it came to rest was compared to other tests that accelerated and decelerated

the robot by changing the PWM, also for 120 ticks of the encoder. The following tables list the data recorded by the experiments.

Trial	Distance (inches)
1	461
2	439
3	439.5

Table 1. Robot's drive motor ran at 100% PWM for 120 ticks of encoder.

Trial	Distance (inches)
1	421
2	422
3	422

Table 2. Drive motor accelerated to 100% by starting at 50% and rising 5% every 5 ticks, then decelerating to 50% the last 30 ticks.

Trial	Distance (inches)
1	329
2	318
3	329.5

Table 3. Drive motor accelerated to 60% by starting at 35% and rising 5% every 5 ticks, then turning the motor off for ticks 80-100, then running at 25% for the last 20 ticks.

It was observed that peeling out was not a concern even when starting at 100% PWM. However, as can be seen by the tables, overshoot is a major concern. As the entire system is slowed down and

deceleration is increased with more time for deceleration, the accuracy will increase and overshoot can be minimized. The perimeter of the wheel is 9.817477 inches. Therefore, 120 ticks of the encoder should be equal to 294.524 inches. It should be noted that the robot did veer to the left for all of the trials (at slightly different arcs). This is due to imprecision of the steering mechanism. By using braking and the constant velocity algorithm described earlier, the robot performed these tests with no overshoot.

Considerations

Two separate battery supplies (a 9.6V and a 4.8V supply) were used by this robot. Due to some sort of battery noise between the common grounds or current draining problems, the range of the receiver is drastically reduced (to about 3-4 feet) when the 68HC11 board is powered. It is not electrical noise as the signals produced by the receiver board and read by the inputs of the 68HC11 board are clean and distinct. Several methods were tried in order to increase the range and eliminate the phenomenon that was occurring. Connecting the common grounds with a resistor or an inductor did not work. Not even the use of an optical isolator solved the problem. This shows that it has something to do with the circuitry of the receiver board. Due to time constraints, the problem was not eliminated successfully. Before the RC car was hacked, it had a range of about 30-50 feet.

Results and Conclusions

Errors are obviously summed during playback, therefore, the more complex the path is, the less the repeatibility. Through observation, stops and starts, as well as changes in ongoing direction (forward and reverse) tend to increase the overall error of the playback. For a forward path around

the lab in a circle with s-curves in it, the robot was repeatible up to 1 foot accuracy. This error could be minimized as the playback could be made to better resemble what is occurring during the recording process.

Although this method is not to be used for extreme precision, it can be used as a quick and inexpensive way to record and playback a robot's trajectory. Through more precise design and further optimization of the program, the robot could be made more robust and further minimize the errors common to this type of project. The robot cannot account for every situation it encounters, but through optimization it could be programmed to adapt and repeat a path as accurately as possible using this sort of method.

Appendices

Final program for record and playback of path

```
/*  motor 0 --> turning motor -->  right = negative */
/*  motor 1 --> drive motor ---> forward = negative */

/*  GLOBALS */
int ethresh=90;
int ticks=0;
int enc_current, enc_armed, enc_start;
long twant=45L;    /* dt/dticks -> the greater the slower */
long told;
long tcurr=0L;
long dt;
float pwm=0.0;
int sensor1;       /* left-front IR sensor */
int sensor2;       /* right-front IR sensor */
int sensor3;       /* right-rear IR sensor */
int sensor4;       /* left-rear IR sensor */
int s1thresh=100;
int s2thresh=100;  /* corresponding IR sensor threshold */
int s3thresh=100;  /* set to 100, but calibrated for    */
int s4thresh=100;
int forward, reverse, left, right;
int roll;          /* roll<4 = rolling forward,  roll>3 = rolling reverse */
int obstacle=0;
int fbthresh=50;
int rbthresh=100;
int bump;
int mode=0;        /* mode0=drive  mode1=record  mode2=playback */
int max=300;
int index;
int direction[300];
int length[300];
int cdir;
int cswitch=1;
int waiting=0;
int calibration=0;

void init_variables() {
  for(index=0; index<max; ++index) {    /* initialize some of the globals */
    direction[index] = 0;
    length[index] = 0;
```

```
 }
 index = 0;
 forward = 0;
 reverse = 0;
 left = 0;
 right = 0;
}

void slow_down(int d) {
 if (d) motor(1, -20.0);
 else motor(1, 20.0);
}

void record_path() {
 if (forward && left) cdir = 1;
 else if (forward && right) cdir = 3;
 else if (forward) cdir = 2;
 else if (reverse && right) cdir = 4;
 else if (reverse && left) cdir = 6;
 else if (reverse) cdir = 5;
 else if (left) cdir = 11;
 else if (right) cdir = 12;
 else cdir = 0;
 if (index == 0 && direction[0]==0) {
  direction[0] = cdir;
  ticks=0;
 }
 else if (index < max) {
  if (cdir!=direction[index] || ticks>=1000) {
   length[index] = ticks;
   ticks = 0;
   index++;
   direction[index] = cdir;
  }
 }
 if ((cdir==0 || cdir==11 || cdir==12) && index)  roll = direction[index-1];
 else roll=0;   /* not rolling */
}

void play_back() {
 index=0;
 while (length[index]) {
  if (obstacle) {       /* if obstacle detected, stop playback */
```

```
  while (length[index]) ++index;
 }
 if (direction[index] == 1) {
  forward = 1;
  left = 1;
  reverse = right = 0;
 }
 else if (direction[index] == 2) {
  forward = 1;
  left = 0;
  reverse = right = 0;
 }
 else if (direction[index] == 3) {
  forward = 1;
  right = 1;
  reverse = left = 0;
 }
 else if (direction[index] == 4) {
  reverse = 1;
  right = 1;
  forward = left = 0;
 }
 else if (direction[index] == 5) {
  reverse = 1;
  left = 0;
  forward = right = 0;
 }
 else if (direction[index] == 6) {
  reverse = 1;
  left = 1;
  forward = right = 0;
 }
 else if (direction[index] == 11) {
  forward = reverse = right = 0;
  left = 1;
  if (direction[index-1]<4) slow_down(1);
  else slow_down(0);
 }
 else if (direction[index] == 12) {
  forward = reverse = left = 0;
  right = 1;
  if (direction[index-1]<4) slow_down(1);
  else slow_down(0);
```

```
    }
    else if (direction[index] == 0) {
      forward = reverse = right = 0;
      left = 0;
      if (direction[index-1]<4) slow_down(1);
      else slow_down(0);
    }
    if (ticks >= length[index]) {
      ++index;   /* if end of segment reached, read next one */
      ticks=0;
      if (direction[index]==0 || direction[index]==11 || direction[index]==12)
            brake();
            wait(300);
    }
  }
  forward = reverse = left = right = 0;
  forward = reverse = left = right = 0;
  brake();
}

void check_mode() {      /* check mode switches */
  while(1) {
    hog_processor();
    poke(0x6000, 0x05);
    if (analog(0) > ethresh)  {
      if (mode==1) length[index] = ticks;
      mode=0;
      roll=0;
      pwm=0.0;
      cswitch=1;
    }
    poke(0x6000, 0x06);
    if (analog(0) > ethresh)  {
      mode=1;
      init_variables();
      roll=0;
      ticks=0;
      pwm=0.0;
      cswitch=1;
    }
    poke(0x6000, 0x07);
    if (analog(0) > ethresh) {
      if (mode==1) length[index] = ticks;
```

```
      mode=2;
      roll=0;
      pwm=0.0;
      cswitch=0;
      ticks=0;
      waiting=1;
      sleep(3.0);
      waiting=0;
      play_back();
    }
    defer();
  }
}

void control() {              /* read radio transmitter */
  int tf, trev, tl, trt;
  while(1) {
   while(mode != 2 && cswitch) {    /* shut off radio control if either */
    hog_processor();               /* in playback mode or if obstacle  */
    poke(0x6000, 0x00);
    tf = analog(0);
    poke(0x6000, 0x01);
    trev = analog(0);
    poke(0x6000, 0x02);
    tl = analog(0);
    poke(0x6000, 0x03);
    trt = analog(0);
    if (tf > ethresh) forward = 1;
    else forward = 0;
    if (trev > ethresh) reverse = 1;
    else reverse = 0;
    if (tl > ethresh) left = 1;
    else left = 0;
    if (trt > ethresh) right = 1;
    else right = 0;
    defer();
   }
  }
}

void brake() {
  hog_processor();
  pwm = 0.0;
```

```
  motor(0, 0.0);
  motor(0, 0.0);
  if (forward || roll<4) {
    motor(1, 50.0);
    wait(50);
    motor(1, 80.0);
    wait(50);
  }
  else {
    motor(1, -50.0);
    wait(50);
    motor(1, -80.0);
    wait(50);
  }
    motor(1, 0.0);
    motor(1, 0.0);
  defer();
}

void brake_hard() {
  obstacle=1;
  pwm = 0.0;
  motor(0, 0.0);
  motor(0, 0.0);
  hog_processor();
  if (forward || roll<4) {
    motor(1, 30.0);
    wait(30);
    motor(1, 50.0);
    wait(30);
    motor(1, 70.0);
    wait(30);
    motor(1, 100.0);
    wait(30);
    motor(1, 0.0);
  }
  else {
    motor(1, -30.0);
    wait(30);
    motor(1, -50.0);
    wait(30);
    motor(1, -70.0);
    wait(30);
```

```
    motor(1, -100.0);
    wait(30);
    motor(1, 0.0);
  }
  motor(1, 0.0);
  motor(1, 0.0);
  defer();
}

void avoid_obstacle(int d) {     /* see where obstacle is, then turn */
  int go_until;                  /* and drive away from it for a bit */
  brake_hard();
  cswitch = 0;
  if (d == 0) {
    reverse = 1;
    forward = 0;
    left = 1;
    right = 0;
    defer();
    go_until = ticks + 20;
    motor(0, 100.0);
    wait(50);
    motor(1, 40.0);
    while(ticks<go_until)
      defer();
    brake();
    reverse = 0;
    forward = 0;
  }
  else if (d==1) {
    reverse = 1;
    forward = 0;
    left = 0;
    right = 1;
    defer();
    go_until = ticks + 20;
    motor(0, -100.0);
    wait(50);
    motor(1, 40.0);
    while(ticks<go_until)
      defer();
    brake();
    reverse = 0;
```

```
    forward = 0;
  }
  if (sensor3 >= s3thresh || sensor4 >= s4thresh || bump >= rbthresh) {
    forward = 1;
    reverse = 0;
    left = 0;
    right = 0;
    defer();
    go_until = ticks + 10;
    motor(1, -40.0);
    while(ticks<go_until)
      defer();
    brake();
    reverse = 0;
    forward = 0;
  }
  obstacle = 0;
  forward = 0;
  cswitch = 1;
}

void avoidance() {        /* use IRs and bumpers to check for obstacles */
  poke(0x7000, 0xFF);
  while(1) {
    hog_processor();
    sensor1 = analog(1);
    sensor2 = analog(2);
    sensor3 = analog(4);
    sensor4 = analog(5);
    poke(0x6000, 0x04);
    bump = analog(0);

    if (sensor1>s1thresh && sensor1<150 &&
          sensor1>(sensor2+s1thresh-s2thresh) && !obstacle)
            avoid_obstacle(0);
    else if (sensor2>s2thresh && sensor2<150 && !obstacle)
      avoid_obstacle(1);
    else if (bump > fbthresh && bump < rbthresh)
      avoid_obstacle(0);
    else if (sensor3>s3thresh || sensor4>s4thresh || bump>rbthresh)
      {
      if (sensor3<150 && sensor4<150)
          avoid_obstacle(2);
```

```
      }
    defer();
  }
}

void wait(int m_second)
{
     long stop_time;
     waiting=1;
     waiting=1;
     stop_time = mseconds() + (long)m_second;
     while(stop_time > mseconds())
             defer();
     waiting=0;
     waiting=0;
}

void encoder()       /* read encoder ticks */
{
  if (analog(3) > ethresh) enc_start = 1;
  else enc_start = 0;
  enc_armed = 0;
  while(1)
  {
    if (analog(3) > ethresh) enc_current = 1;
    else enc_current = 0;
    if (enc_current == enc_start && enc_armed == 1)
    {
       ticks++;
       enc_armed = 0;
       told = 0L;
       tcurr = mseconds();
       if (!waiting) reset_system_time();
       dt = tcurr - told;
       if (mode == 1) record_path();
    }
    else if (enc_current != enc_start) enc_armed = 1;
  }
}

void drive_motors()     /* motor driver routine */
{
  while(1) {
```

```
if (!obstacle) {
 if (forward) {
   if (pwm==0.0) {
         pwm = -80.0;
         dt = twant + 200L;
   }
   if ((int)(dt-twant) > 100)
    pwm = -80.0;
   else if ((int)(dt-twant) > 90)
         pwm = -50.0;
   else if ((int)(dt-twant) > 60)
         pwm = -45.0;
   else if ((int)(dt-twant) > 20)
         pwm = -35.0;
   else if ((int)(dt-twant) < 0)
         pwm = -15.0;
   else
         pwm = -30.0;
 }
 else if (reverse) {
   if (pwm==0.0) {
         pwm = 80.0;
         dt = twant + 200L;
   }
   if ((int)(dt-twant) > 100)
    pwm = 80.0;
   else if ((int)(dt-twant) > 90)
         pwm = 50.0;
   else if ((int)(dt-twant) > 60)
         pwm = 45.0;
   else if ((int)(dt-twant) > 20)
         pwm = 35.0;
   else if ((int)(dt-twant) < 0)
         pwm = 15.0;
   else
         pwm = 30.0;
 }
 else pwm = 0.0;
 motor(1, pwm);
 if (left) motor(0, 100.0);
 else if (right) motor(0, -100.0);
 else motor(0, 0.0);
}
```

```
  }
}

void calibrate_sensors() {      /* routine performed first to calibrate */
  s1thresh=analog(1);           /* IR thresholds individually while the */
  s2thresh=analog(2);           /* robot experiences the vibrations it  */
  s3thresh=analog(4);           /* will during run time by driving      */
  s4thresh=analog(5);
  ticks=0;
  motor(1, -40.0);
  while (ticks < 10) {
    calibration=analog(1);
    if (calibration > s1thresh && calibration < 120) s1thresh = calibration;
    calibration=analog(2);
    if (calibration > s2thresh && calibration < 120) s2thresh = calibration;
    calibration=analog(4);
    if (calibration > s3thresh && calibration < 120) s3thresh = calibration;
    calibration=analog(5);
    if (calibration > s4thresh && calibration < 120) s4thresh = calibration;
  }
  ticks=0;
  motor(0, -100.0);
  while (ticks < 10) {
    calibration=analog(1);
    if (calibration > s1thresh && calibration < 120) s1thresh = calibration;
    calibration=analog(2);
    if (calibration > s2thresh && calibration < 120) s2thresh = calibration;
    calibration=analog(4);
    if (calibration > s3thresh && calibration < 120) s3thresh = calibration;
    calibration=analog(5);
    if (calibration > s4thresh && calibration < 120) s4thresh = calibration;
  }
  ticks=0;
  motor(0, 100.0);
  while (ticks < 10) {
    calibration=analog(1);
    if (calibration > s1thresh && calibration < 120) s1thresh = calibration;
    calibration=analog(2);
    if (calibration > s2thresh && calibration < 120) s2thresh = calibration;
    calibration=analog(4);
    if (calibration > s3thresh && calibration < 120) s3thresh = calibration;
    calibration=analog(5);
    if (calibration > s4thresh && calibration < 120) s4thresh = calibration;
```

```
  }
  motor(1, 0.0);
  motor(0, 0.0);
  ticks=0;
  s1thresh += 5;
  s2thresh += 5;
  s3thresh += 7;
  s4thresh += 7;
  motor(1, 0.0);
  motor(0, 0.0);
}

void main()
{
  init_variables();
  start_process(encoder());
  calibrate_sensors();
  wait(1000);
  start_process(avoidance());
  start_process(check_mode());
  start_process(drive_motors());
  start_process(control());
}
```