University of Florida

Department of Electrical And Computer Engineering

EEL 5666C
Intelligent Machine Design Labratory

Slappy
(The Mappy Bot)

Final Report
Louis Brandy
Tuesday, April 23, 2002

Table of Contents

Abstract

Slappy is an autonomous mobile robot that will roam around a maze, mapping as he goes. He will align and correct himself on the walls of the maze to keep his position within the maze where he expects it. He will produce his map at the user's request.

Executive Summary

Slappy (named by my little cousin) will wander a maze attempting to map out the maze as he goes. He does this using 5 infrared sensors with split duties between wall-detection and error correction. He also uses a shaft-encoder to measure distances traveled (and angles turned).

Slappy moves using two modified servos to act as gearhead motors. These motors, along with all electronics are controlled by an HC11 on the Axiom EVBU board which is mounted to his moving platform.

The maze must be set up of walls corresponding to a grid. This means all walls are orthogonal and of a specific distance. With these constraints, Slappy is able to navigate the maze, without touching a wall, and produce a map of the areas he traveled.

Drift errors are corrected as he moves by using the walls of a maze to orientate himself. He corrects both linear and rotational errors by using the walls as reference points.

Introduction

Slappy is a circular robot of about 1 foot across that stands about a foot high. He wanders around a maze, negotiating it and mapping it to memory to be shown on a terminal later. He should also do all of this without touching any walls.

The maze that Slappy maps has certain restrictions. Most importantly, the maze is laid upon a grid with the walls taking up an entire segment. There are no partial walls, nor anything that is not orthogonal. The actual maze was constructed of 2x6s of wood that were cut to size, with the 6-inch faces being the actual walls that Slappy deals with. The minimum wall length is 16.5 inches (three 2x6 widths – 2x6s aren't actually 2x6, but instead 1.5 inches by 5.5 inches. 5.5 x 3 = 16.5). Therefore the entire maze can be decomposed into a grid of square cells with each side as 16.5 inches.

Slappy's software was written entirely in HC11 Assembly language and therefore many routines that would otherwise be simple turn out to be rather difficult or long. Assembly, however, provided extremely good control over certain aspects of the system (like when interrupts occurred). It was also an excellent learning experience.

He uses a shaft-encoder to do ninety percent of the work. His forward movements are always a specific distance determined by a certain number of clicks of the encoder. Likewise, both his left and right turn are tuned to a specific value of clicks. However, he will never be able to turn exactly 90-percent, nor will he be able to move exactly one cell forward. Every time he moves, or turns, these small errors will add up until Slappy is nowhere near the center of a cell, or worse, cause him to run into a wall. Solving these incremental errors is the heart of the problem.

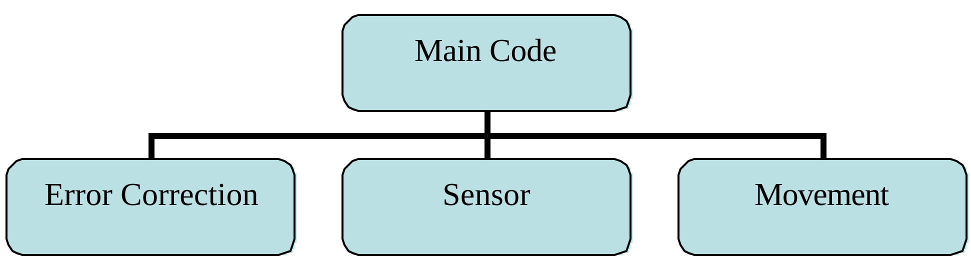
After creating a program for Slappy to navigate the maze without any error correction, he was observed to see the most common types of mistakes he makes and thus different schemes were created to correct each of them. In the end, three different schemes are used at different times to help Slappy keep his bearings.

Integrated System

Slappy's software is written entirely in assembly language for the HC11. It, in total, is about 1100 lines of assembly that is broken down into sensor routines, movement routines, error-correction routines, and the main program which uses the routines to solve the problem.

The software was written in a modular style. For instance, a routine to sample an indicated sensor (passed to the function in a register) was written and tested thoroughly using a set of test programs. Then, a routine to turn both motors on so the robot was moved forward, left, and right was written and tested thoroughly. The final step was to combine all these into one program that was able to call these functions individually and use them to solve the problem. Slappy's routines are rather robust and form, in their own right, a "higher-level" language to use to solve this problem.

As stated before, the code is broken down as follows:



In the following sections, each of these individual sections will be outlined in detail, and the final algorithm is described in the behaviors section.

Mobile Platform

Slappy's platform is only 4 wooden pieces that fit easily onto two sheets of the wood provided as part of the class. The main platform component is an 8 inch circular disk consisting of holes for the board mount and the servo mounts.

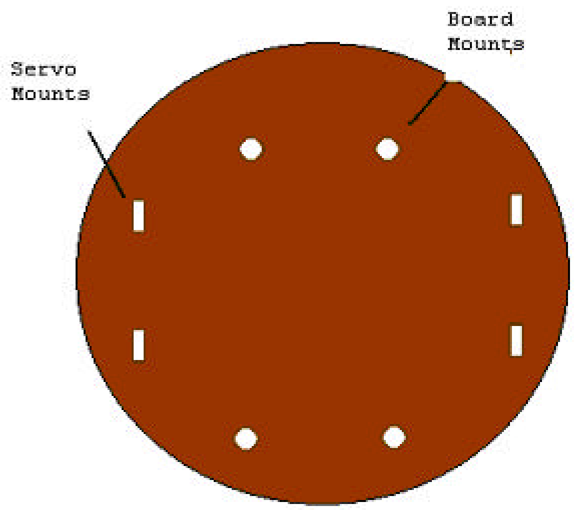
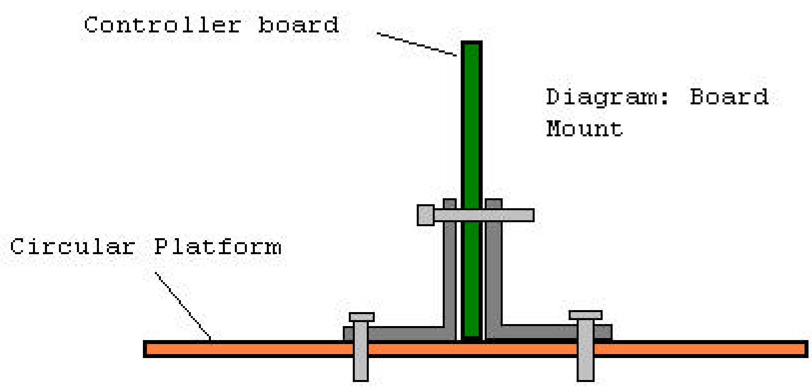


Diagram of main platform

The board is mounted vertically, with two steel L-brackets serving as the support mechanism. Four of them are used, in total, two on each side. There is a screw going through the board, to an L-bracket on both sides of the board, and each bracket is screwed into the main platform.



The servos are mounted on the underside of the board using three more pieces of wood for support. The slots in the circular board for the servo mounts are the exact width of the board, so a smaller board with tabs the right size can be pushed into place and secured with wood glue. A quick diagram of the remaining pieces is shown below:

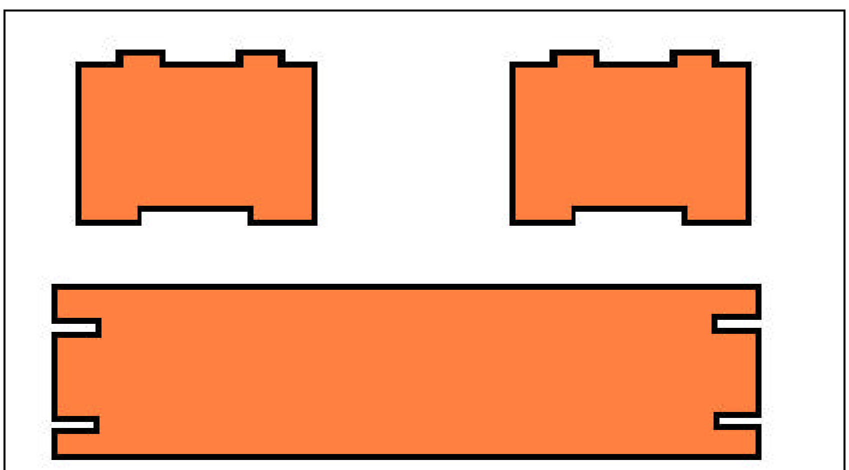
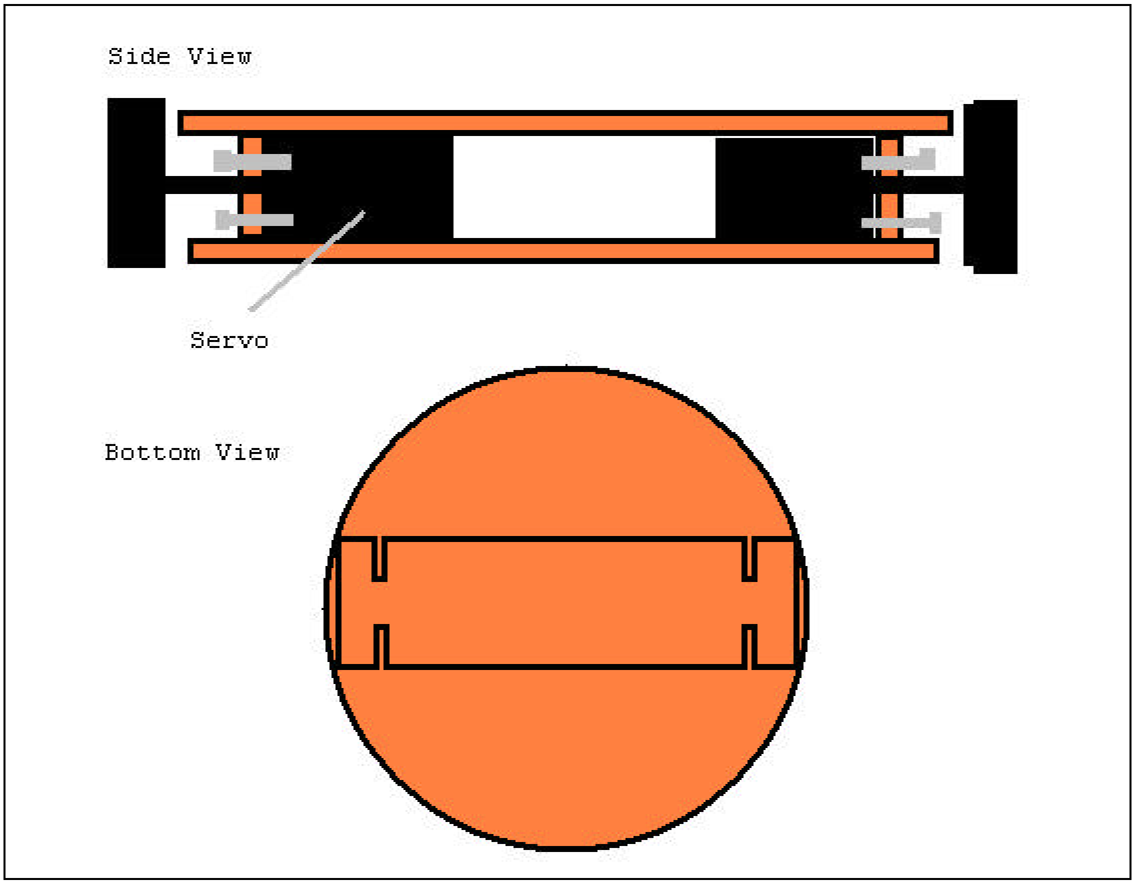


Diagram of the servo mount peices

The two top pieces are inserted vertically into the circular body piece. The bottom piece using the same tab concept, bridges the two on the bottom, parallel to the circular board. It is diagramed below:


Side View
Servo
Bottom View

Actuation

The only actuation on the robot consists of two hacked servo motors. The potentiometer was removed and replaced by two equal resistors so that the servo always thinks it is at the same position. The gears were also altered so that there was no mechanical stopping mechanism. This turned the servos into gearhead motors that, with different pulse-widths, can control both speed and direction.

The software to control these two motors took quite a while, though it was quite simple. Two output compare routines were used to create pulse-widths with a total period of 32ms. In order to keep them from running over each other, one was set to go off at 0x0000 and the other at 0x8000. On these interrupts, it would set the mask for the next interrupt at its current counter, plus the specified pulse-width. In this way, the two memory locations that hold the current pulse-width can be altered at any time from the program and the motors will be updated accordingly.

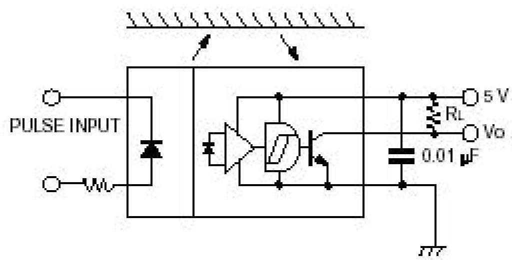
The actual routines to go forward, and turn consist solely of updating the pulse-widths variables in memory with specified constants, and returning to the caller. Tuning those values is what took up the brunt of the time spent. The experiment consisted of setting them manually, running the robot down the hall, and making adjustments. Once decent values for going straight and turning in place were determined, the routines were finished.

Sensors

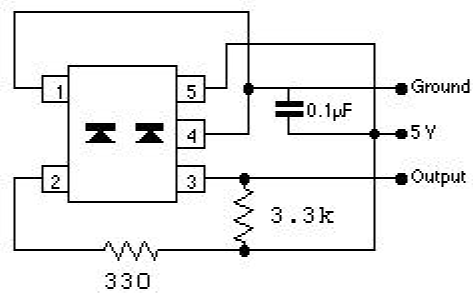
Shaft Encoder

The design of a shaft-encoder was vital to the proper working of the robot, so a little more work was put into making a shaft-encoder and proving it worked correctly.

The part chosen for the encoder was the Hamatsu P5587 photoreflector. The device consists of an IR emitter and a phototransistor pair. It is a 5-pin device with the follow layout:

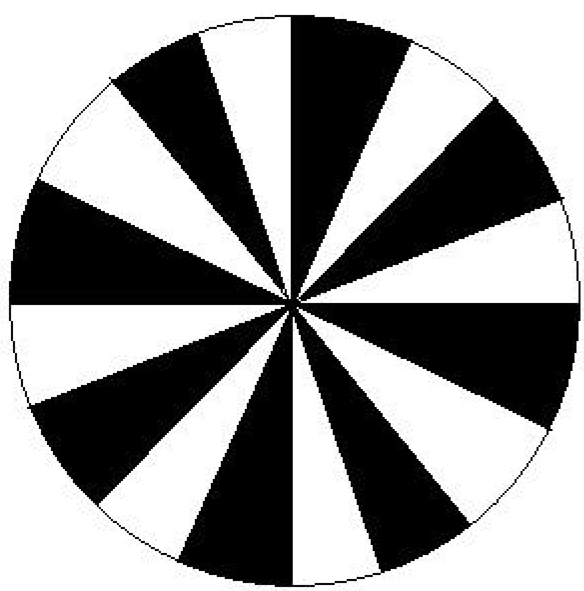


Using a pull-up resistor, the device will be at 5-volts if black paper is in front of it, and 0V if white paper is in front of it. Using this simple property, a shaft encoder can be constructed. The following is the actual design implemented on the circuit board that was tested:



The 330 ohm resistor simply regulates the current that flows through the diode. The 3.3k resistor is simply a pull-up resistor and it's value has very little effect on performance of the digital output.

The second part of the encoder is the encoder disk. These are circles divided in equal slices of alternating black and white. Mounting these disks to the wheel, with the photoreflector fixed in place and facing the wheel creates the encoder. Below is a reproduction of one:



The actual encoder disks were made in auto-cad. Three resolutions were made for experimenting: a 16 section (above), a 32 section, and a 64 section.

Testing

The first test done on the shaft-encoder was to hook it up to an oscilloscope and see what kind of signal it gave. The signal appeared quite crisp, but it was near impossible to tell what types of noise or bounces were occurring at the extremely slow frequency. This, at the very least, verified the encoder was at least wired correctly.

Again, there is no easy way to test the frequency of the signal and compare that with the angular speed of the wheel. The best way to measure if the shaft-encoder is doing it's job is to actually write the software to make the robot move a certain amount of ticks and measure the distance traveled. Over the course of 20 trials, the distance the robot traveled after two wheel-rotations was measured. The distance was chosen after a bit of trial and error. Having the distance too long would lead to errors due to motors not being the same speed, and having the distance too short amplifies the simple measurement errors. These numbers were compiled and the mean, the standard deviation, and the range were calculated. This test was done for the 16, 32, and 64 sectioned disks.

What follows is the tabulated data of the three tests done (All numbers in inches):

16 section		**32 section**		**64 section**
26.1875		26.25		26.375
26.375		26.25		26.375
26.375		26.25		26.4375
26.5		26.375		26.4375

26.5		26.4375		26.4375
26.5		26.4375		26.4375
26.5		26.4375		26.4375
26.5		26.5		26.5
26.5		26.5		26.5
26.5		26.5		26.5
26.5625		26.5		26.5
26.5625		26.5		26.5
26.5625		26.5		26.5
26.5625		26.5		26.5625
26.5625		26.5625		26.5625
26.625		26.5625		26.5625
26.6875		26.5625		26.625
26.75		26.5625		26.625
26.8125		26.625		26.625
26.875		26.6875		26.75

What follows is some secondary calculations about the datasets:

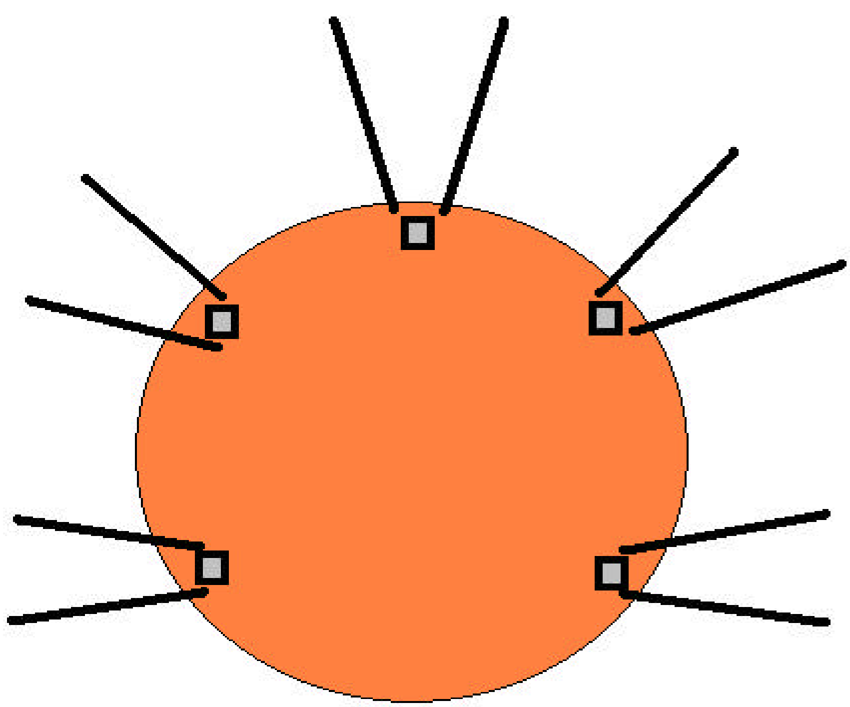
		16 section		**32 section**		**64 section**
Mean:		26.55		26.475		26.5125
Std.Dev		0.153897		0.118932		0.094242
Range		0.6875		0.4375		0.375
Theoretical		0.834461		0.41723		0.208615

Much as expected, as the resolution increases, the standard deviation and total range go down. The theoretical values are obtained by dividing the circumference by the number of sections. In theory, the encoder knows nothing about anything smaller then this number and it's provided as an ideal number for comparison. Two other sources of error keep the estimated numbers from being much closer to the actual values observed. First, there is measurement error because this is a difficult measurement to do to a high degree of precision. Secondly, the starting and stopping of the robot is very suddenly and jerk and often causes slight shifts and overshoots.

The final robot used the 32 section version because the 64 version proved to be too error-prone in various noisy conditions and especially on reflective floors (ie, tile).

Infrared Sensors

The robot consisted of 5 infrared sensors, situated as follows:



The two rear sensors are, in theory, orthogonal with the maze walls at all times. Therefore, these two sensors and the fore sensor make up the wall-finding group. These three sensors are used to tell if a wall is in either of the three main directions. The side pointing front two sensors are used in error correction routines to detect when the robot is not facing orthogonally and heading towards a wall.

A routine was written to sample a specified (by passing it through a register) analog port. This routine includes the wait times to allow for a stable output of the LED (not very long), and then a stable analog value (much longer). Waiting for an analog output value takes quite a bit of time and consequently determines the overall sampling rate of the robot. Since no other system of the robot waits nearly as long, this one value, effectively, determines the speed at which the robot makes decision. It is referred to as SAMPLETIME in the attached code.

At this point, a few of the problems with my sensors should be discussed. The distances involved in these infrared sensors are essentially from zero range to about two cells worth, or about 3 feet. The most important readings are point-blank walls abutting the robot, walls about one half cell away (a nearby wall), a wall about 1.5 cells away (a wall one cell over), and an infinite reading. Three of the sensors performed extremely well within these ranges, but the other two were not very good. In one case, very close readings were very easy to confuse with infinity. In the other case, it was very difficult to tell the difference between a wall nearby, and a wall one cell over (as the "peak" of the analog output existed between the two).

After quite a bit of fiddling with the robot, it was decided that the best sensor should be used in the front, and the other two good ones should be the front-side sensors that are used to correct errors. The back sensors only job was to have a threshold and say 'Yes a wall is here' or 'No, a wall is not here'. Since these back sensors were rather poor, they were limited to solely this role. As is seen later, had these back sensors been better at these ranges, they could also be used for some error-correction routines that would greatly add to the stability of the robot in a maze.

Behaviors

The robot has a host of behaviors that will be described here, as well as a main program which transfers from behavior to behavior based on certain stimuli. All of these behaviors are programmed for the HC11 in straight Assembly code using the old DOS tool Edit. The board used is from Axiom (www.axman.com). It is the Axiom EVBU board for the HC11, revision D.

Map Negotiating

This was the first behavior to get working, and essentially set the basis for what else was required. It was assumed that if the robot was able to navigate the maze, randomly, without hitting a wall, then mapping each cell around it was a trivial next step. Therefore, the brunt of design was centered around successfully map negotiating. The first algorithm went like this:

```
Sample Front
Is there a Wall? (Front Reading > Front Threshold)
        Go Forward
else
        Sample Right
        Is there a Wall?
                Turn Right
        else
                Turn Left
```

This basic map-negotiation routine was the basis of the final main program, and it's influence is still extremely prevalent. All of the routines stated above will be described below, in their inner workings, and this will be returned to in a moment

Wall-Detection

All wall-detection is done in a very simple way. Each of the there wall-finding sensors (fore, left rear, right rear) has a threshold value. These, along with many other constants, are at the beginning of the ASM file and can be changed quickly for various conditions.

Slappy assumes a wall is present in that direction if and only if the sensor reading from the appropriate analog port is above the threshold.

Go Forward/Turn Left/Turn Right

All three of these routines incorporate the motor driver routines and the shaft encoder. The go-forward routine is a close relative of the following:

```
Clear PulseCount
Set Motor to Forward
Is Pulses equal to FORWARD_PULSE_CONSTANT, yet?
Set Motor to STOP
```

The routine to turn right and left is essentially identical except the motor is set to turn in those directions. All three have their own constant that, again, is tuned through large amounts of trial-and error. The left and right routines were tuned by having the robot perform 8 turns, and then go forward. In this way, any small error was easily recognized and the number of pulses adjusted. Going forward was tuned later, when error-correction was the concern.

Map Negotiating, Revisited

Now that all of the pieces of the following have been described, it can be shown how this formed the basis of the eventual solution.

```
Sample Front
Is there a Wall? (Front Reading > Front Threshold)
        Go Forward
else
        Sample Right
        Is there a Wall?
                Turn Right
        else
                Turn Left
```

At this point, after hours and hours of tuning three IR thresholds, one IR timing delay, four motor speed values (2 forward, 2 reverse), and three numbers of pulse-lengths for movement, Slappy was ready to be run in an actual maze. The first time Slappy ran, with the above decision scheme, he actually negotiated the maze fairly well, and the finely tuned values served him well through the first 5 or 6 cells. At this point, the small errors, mostly linear in nature caught up with him and he made the first, of many, collisions with a wall.

Error

There are three kinds of error that Slappy incurs over time. The most obvious two are linear errors and rotational errors. Rotational error occurs, over time, as Slappy's attempts at 90-degree turns have small errors that collect until he is off the 0-90-180-270 axis. The linear error is split into two kinds, one is situated in the forward-rear direction, as opposed to side-to-side errors. The reason this differentiation is made is because Slappy can easily correct front-linear errors since he can move in that direction. It's much harder for him to correct side-linear errors.

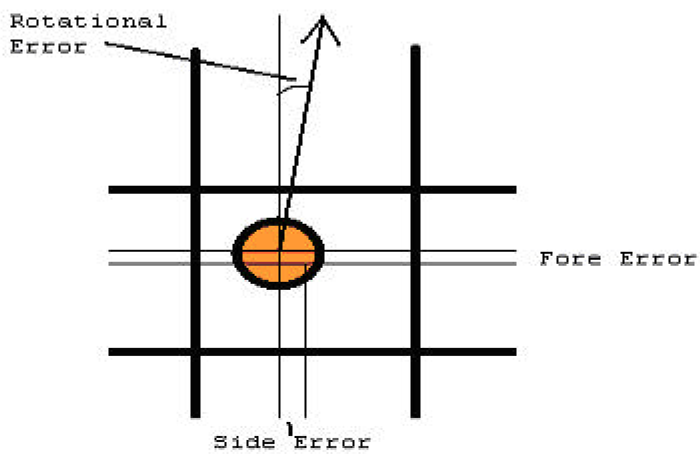


Diagram of the robot in a cell, with the types of error labeled. The robot faces in the direction of the arrow

Fore-Linear Correction

Again, this was a fairly straightforward and powerful technique to correct error. Essentially, two more routines for the motor were written with two new, slower speeds – one for forward, one for backward. Using these very slow movements, Slappy could scoot up, or back off of a wall until his FORE sensor read a certain value, another constant, called STDDIST. This value was chosen very carefully for later reasons discussed.

It should be noted that since Slappy's wall-detecting sensors are in the rear of the platform, Slappy is much better to be closer to a fore wall, to maximize the amount of wall in the view of these sensors. This also meant his FWDPULSE should be increased a little as to allow Slappy to use those sensors to detect things, and consequently back off the wall after the detection took place. In the end, Slappy works a bit better with a few more pulses then is required to go 16.5 inches. The obvious flaw to this design, is that in a long straightaway, Slappy will be creating large amounts of forward error and when he finally reaches a wall, he may be too far forward to stop in time.

To correct this lesser issue, Slappy's GoForward routine was changed to contain an emergency stop value. If, when going forward, Slappy's Fore Sensor hits an emergency threshold, that means he needs to stop immediately, and return out of the subroutine. Going full speed, and stopping at this threshold will stop Slappy about an inch from the wall he was about to hit. Slappy assumes, always, that although he has not moved an entire set of pulses forward, he did move one cell forward. This assumption is based on the fact that he tends to create forward error (ie, moving too far) as opposed to rear error (not moving far enough). After an emergency stop, Slappy will do a fore-error correction to back off this wall to his STDDIST.

Rotational-Correction

The rotational correction algorithm used is quite simple in theory, and quite impressive in practice. Two new behaviors were created to do this: AlignL and AlignR. These two turned in the specified direction until the Fore Sensor decreased at all. By doing AlignR and then AlignL, you could, in theory, end up near orthogonal with any wall in front of you.

This pair of behaviors, in conjunction, work fairly well but often the FORE sensor takes a quick dip that it shouldn't and this results in a false stopping. To correct this, in the actual implementation, the right and left alignment is repeated 3 times each, in an alternating fashion. In this way, he does a fairly good job of fixing his rotational errors.

The STDDIST that Slappy corrects himself to is governed by this routine. Remember, STDDIST is the distance that Slappy moves to when there is a wall in front of him that is supposed to represent the center of that cell. As a separate consideration, the most sensitive distance region on the fore sensor should be used for the Align routines. Therefore, STDDIST doesn't quite work out to the center of a square. It is, instead, near the center, where the FORE sensor is the most sensitive.

It may seem like a coincidence that the distance from the center of a square to a wall in front is very near the most sensitive region of a particular IR sensor. The IR sensor to be used in the front was chosen for its sensitivity in that range, and positioned on the robot accordingly to maximize this effect.

Side-Linear-Correction

At this point, there were two main ways of correcting error that, together, did a pretty decent job of keeping Slappy from hitting walls. At this point, it would be wise to discuss a little bit of the relationship between the errors. Linear-Forward error is easily correctable and corrected the most accurately. Once the robot turns, his Linear-Side error becomes his Linear-Forward error, which is corrected the next time he hits a wall. Therefore, given no rotational error, over time he will correct all of his linear errors extremely well.

Rotational Error, however, complicates these things. Rotational error creates side-error over time, as the robot moves forward. With Slappys' decent rotational error correction, major side-error isn't really a concern. He can have moderate side-error that will be corrected as long as there is a wall to turn him soon. Without a wall to turn him, and move his side error into the correctable forward region, he will eventually fail.

What all this means, is that our two schemes are ineffective in cases where Slappy does not turn for long periods of time – say three squares. In this case, Slappy's success rate goes down considerably. Therefore, a third scheme must be developed to correct side-errors when in long corridors.

The technique used is simply to poll the front-right and front-left sensors using a threshold while moving forward. When the threshold kicks in, the opposite wheel is slowed down. For instance, if the front-right sensor is over the threshold, he must turn left, and thus slow down his left motor. The question is, how much should it be slowed down?

Several techniques were tried:

1) Slow down the motor by a constant amount

2) Slow down the motor equal to the difference in threshold

3) Slow down the motor twice the difference

By ‘slowing down’ the motor, I mean subtracting (or adding, in the case of the servo going in reverse) a number from the pulse-width used.

Slowing down the motor by a constant amount caused a serious overcorrection problem if that constant was too high, and a serious under-correction problem when it was too low. Numbers in the middle ground tended to overcorrect small errors, and under-correct big ones. This scheme was clearly ineffective.

The second and third were identical in implementation other then a SHIFT-LEFT in the multiplied version. Again, I ran into similar control issues as before. The un-multiplied one ran into under-correction problems with large errors, and the multiplied one ran into overcorrection problems with smaller errors.

Since most of the errors he encountered were small, the un-multiplied version seemed to be the logical choice. To aid in his under-correction problem, the threshold was lowered a bit from desired.

However, this system is not perfect. The errors that can cause him problems are discussed in the conclusion.

Final Negotiating Routine

Below is a pseudo-code representation of what the final main routine looks like, and how it puts together the error correction and movement routines together to form the main brains of the robot.

Wall in Front?

No -> Go Forward while correcting side errors

Go Back to Top

Yes ->

Correct Linear Distance in Front of us

Orient to wall in front (AlignL/AlignR x3)

Re-Correct Linear (new orientation, new reading of fore)

Wall to the Right?

Yes -> Turn Left

Go Back to Top

No -> Turn Right

Go Back to Top

Mapping

The mapping aspect was a close conceptual extension of the previous decision making algorithm, but the implementation was quite tricky. He has three sensors that already tell him if a wall is in a specific spot. This means he already knows, for certain, of the three walls around him. Do not make mistake of thinking he also knows the fourth, the wall behind him, because he came from that direction. That is only true after a forward motion, and not after a turn.

At the very beginning of the decision routine, he jumps to a mapping routine which takes the three binary values of the three walls around him and has the job of writing those three bits to the correct place in memory.

Each cell is given it's own place in memory, with the lesser 4 bits denoting the walls as shown:

1

8 2

4

For the rest of this text, 1 will be referred to as North, 2 and East, etc. While these terms are technically incorrect, they make it a bit easier to both explain and understand. Whatever direction the robot is started in is his North. The most significant bit is used to set whether a cell has been visited at all or not. The other three bits are unused.

The robot must also keep track of his direction, as he turns. Knowing his fore-sensor reads a wall is not of any value if he doesn't know if that represents a North or East. To do this, he has a direction variable that is updated with every turn and is defaulted to 1 (North).

At this point in the design phase, it is wise to note that a pair of lower four-bit circular rotations was created, for left and right. It will be used extensively as we will see. It zeros the top four bits, and rotates the bottom four in the specified direction.

If the data is passed in a poor or arbitrary format, then the amount of bit manipulation required is a bit staggering to update the three corresponding wall bits in memory. To correct this problem, the data format was carefully chosen. First, the sensors are read and a bit-field is created with the following format 0000 P0SF. P stands for Port (left), and S for starboard (right). It is formatted so oddly as to correspond to the bit assignments of North, South, East, and West. When the robot is facing North, then this bit-field is correct and ready to be ORed with memory. However, when he is not facing north, the lower four bits needs to be rotated as many times as the direction indicates before it is ORed with the current cell in memory.

The scheme of storage and rotation was designed after a painstaking thought process as to doing this in an efficient way. The final solution is far more elegant then using a random assignment system would have been. It's also a lot easier to code and thus less prone to error.

Map Output

The map, in memory, was now complete. However, there needed to be a way to output it to the terminal. The simplest way to do this was to create a display program at another address, far from the original code and run that when you wish to display the map.

The program to output the map in memory is essentially two for-loops to traverse each memory location, and then three sequential for-loops nested inside to run through an entire line on the screen and print the top, middle, and bottom of each cell respectively.

The actual code for the map display is extremely long and a bit convoluted, but if the basic premise is understood, it tends to modularize into repeated segments pretty easily.

Deficiencies

Using the three correction algorithms described and the decision making process above, Slappy is fairly successful in negotiating and mapping a maze. Slappy has a few pitfalls. As described, two of the hacked IR sensors were poor. If five good IR sensors were available, the error-correction routines could be a lot better (ie, the AlignR and AlignL could be applied after turning, instead of before) and Slappy's error-correction ability could be vastly improved.

The truth is, there are times when Slappy will still hit a wall. Since Slappy is prone to severe rotational or side-error while moving forward, these are essentially the main causes of his collision with walls. His correction algorithm, as is, doesn't have enough control to really swerve him out of the way of these collisions. These types of error almost always result from a poor turn, since no rotational error is corrected after the turn, only before. This means that if the shaft-encoder malfunctions he can turn too far or too short, and the results are often catastrophic.

For this reason, Slappy works best on very short carpet or a mat ofr some sort with minimal reflectivity. Sunlight onto tile-floor will really harm his ability to use the shaft-encoder and often cause unpredictable errors in his motions.

Furthermore, Slappy absolutely requires obstacles to alleviate his errors. Without the feedback of obstacles, Slappy has no way of correcting his errors and will find himself in a completely different location than he thinks, over time. In order to improve this, the dead-reckoning aspect of Slappy would have to be improved. This would require a more precise shaft-encoder, and much more finely tuned motor speeds for his movement. The better the dead-reckoning of the robot, the less obstacles he requires to fix his position.

Conclusion

Overall, I'd have to rate Slappy's final performance to be a great success. He works in a large variety of mazes and works with pretty good results. He has accomplished all of the initial goals I'd set out for, and although his success rate isn't perfect – there are several plausible solutions I've come up with. I think that, much like several other things, I ran out of time, more then I ran out of ideas, to solve his problems.

If Slappy had 5 good IR sensors, and a slight rewrite in his error-correction scheme, his ability to negotiate a maze in a stable fashion for long periods of time would greatly improve. Unfortunately, the downside of the current system is it has no way of detecting a major error and likely the results would leave large unexplored areas corrupted with bad data. Given enough time, Slappy would recover from most errors, and remap the entire thing correctly – just with extra unvisited squares corrupted by old data. Perhaps with a method of detecting major errors (bump switch?) and a new routine, those obsolete areas of the maps could be erased as a finishing routine.

Also, given more time with Slappy, after I got two better sensors, I would add in some computer science to his decision making to make him decide to go to new and exciting places rather then wandering around in his current arbitrary fashion. He lacks any explorative instinct that could make his feats all the more impressive.

Documentation

Credits

In case you were wondering, my robot is officially entitled "Slappy the Mappy bot". It was given by my little cousin (who is 8) who began laughing hysterically at her own joke when she said it, that none of us have had the courage to change it.

I'd like to thank Dr. Arroyo, Dr. Schwartz, Aamir, Tae, and Uriel for all their help in the lab, and in the administrative details that has given me this experience.

The following websites are excellent reads and extremely informative on how to do certain things.

(Shaft Encoder)
http://www.gorobotics.net/servoencoder.shtml
Also a very good site for all things robotics! Lots of great ideas.

(A similar project)
http://web.sbu.edu/cs/roboticsLab/mapperI/index.html
This site details an extremely similar project done at St. Bonaventure University. I found this late in the semester when I was getting a bit depressed that this might not even be possible with the hardware I had. Our sensor setups worked out to be somewhat similar, though our error correction routines came out pretty different. They mainly provided psychological support for me, that I could get this working.

Parts List

Infrared Sensors – Sharp GPIUX sensors. These are essentially your basic IR cans. At some point during the semester, www.radioshack.com stopped carrying them. They were my supplier.

Shaft Encoder – Hamatsu P5587 photoreflector
www.acroname.com

Servos – Standard S3003 servos – hacked
www.servocity.com

HC11 Board – Axiom EVBU-D board
www.axman.com

Appendix
Final Code

```
TCTL1   EQU     $1020
TMSK1   EQU     $1022
TMSK2   EQU     $1024
TFLG1   EQU     $1023
TFLG2   EQU     $1025
PACTL   EQU     $1026
PACNT   EQU     $1027
TOC2    EQU     $1018
TOC3    EQU     $101A
PORTA   EQU     $1000
BAUD    EQU     $102B
SCCR1   EQU     $102C
SCCR2   EQU     $102D
OPTION  EQU     $1039
ADCTL   EQU     $1030
ADDATA  EQU     $1031
PORTD   EQU     $1008
DDRD    EQU     $1009
SPCR    EQU     $1028


PE1     EQU     %00000001
FORE    EQU     %00000010
RFORE   EQU     %00000011
PE4     EQU     %00000100
LREAR   EQU     %00000101
RREAR   EQU     %00000110
LFORE   EQU     %00000111

******** ASCII Constants **************
SPACE   EQU     $20
DASH    EQU     $2D
USCORE  EQU     $5F
PIPE    EQU     $7C
AST     EQU     $2A

**** Number of pulses of encoder disk until stop

FWDPULSES  EQU  42      * 42 (other numbers for testing)
LEFTPULSES EQU  18      * 17
RIGHTPULSES EQU 20      * 20

**** IR Thresholds
LREART   EQU     $60    * Wall here?
RREART   EQU     $5D    * Wall here?
FORET    EQU     $59    * Wall here?
STDDIST  EQU     $6C    * Standard distance (FORE READING)
```

```
LFORET   EQU     $71     * 6A
RFORET   EQU     $73     * 73
EMERG    EQU     $72     * Fore reading meaning to STOP


**** IR Delay
SAMPLETIME EQU  $3F

FORW    EQU     $0AF0   * Standard movement (and stop)
STOPR   EQU     $0A40
STOPL   EQU     $09C0
BACK    EQU     $0910
SL2     EQU     $0A10   * Right Motor (SLOW to LEFT)
SL1     EQU     $0998   * Left  Motor (SLOW TO LEFT)
SR2     EQU     $0A70   * Right Motor (SLOW TO RIGHT)
SR1     EQU     $09F8   * Left  Motor (SLOW TO RIGHT)
SLOWF1  EQU     $09F0   * L
SLOWF2  EQU     $0A10   * R
SLOWB1  EQU     $0998   * L ** SlowF/B PWs
SLOWB2  EQU     $0A6A   * R


***Stop: RIGHT Motor : OC2.asm : OC3 : 0A40 : BACK
***Stop: LEFT Motor  : OC.asm  : OC2 : 09C0 : FORW

        ORG     $00D9
        JMP     OC3_ISR
        JMP     OC2_ISR

        ORG     $00CD
        JMP     PAO_ISR

        ORG     $0100
PW1     FDB     $07D0           ; Current Pulse Width: Oc2 (Left)
PW2     FDB     $0FA0           ; Current Pulse Width: Oc3 (Right)
PCOUNT  RMB     1

        ORG     $3000
MAP     RMB     256
XX      RMB     1
YY      RMB     1
DIR     RMB     1

        ORG     $4000
        JSR     SHOWMAP
INFIN4  BRA     INFIN4

        ORG     $4100
        JSR     CLRMAP
        JMP     INFIN4

        ORG     $2000
        LDS     #$1FF
        LDAA    #$10

        JSR     INIT_SCI
        JSR     INIT_MOTOR
```

```
        JSR     INIT_SHAFT
        JSR     INIT_SENSOR
        JSR     INIT_AD

        LDAA    #4
        STAA    XX              * Start position set to 4,4
        STAA    YY
        LDAA    #1
        STAA    DIR             * Direction set to NORTH

        JSR     STOP
        CLI


HERE
        JSR     DECIDE
        PSHA
        SUBA    #1
        BNE     NOTS      * If we go straight
        JSR     WAIT
        JSR     UPDATEXY
        JSR     GOFCOR
        JSR     MAPIT     * After all going forwards *
        JMP     BOT

NOTS    PULA
        PSHA
        CMPA    #2
        BNE     NOTL      * If we turn left
        JSR     LINDIST
        JSR     WAIT
        JSR     ALIGNR
        JSR     ALIGNL
        JSR     ALIGNR
        JSR     ALIGNL
        JSR     ALIGNL
        JSR     ALIGNR
        JSR     LINDIST
        JSR     WAIT

************
** Re-decide to ensure accuracy?
************
        JSR     UPDATEL
        JSR     TURNL
        BRA     BOT

NOTL    PULA
        PSHA
        CMPA    #3
        BNE     NOTR      * If we turn right
        JSR     LINDIST
        JSR     WAIT
        JSR     ALIGNR
        JSR     ALIGNL
        JSR     ALIGNR
        JSR     ALIGNL
```

```
        JSR     ALIGNL
        JSR     ALIGNR
        JSR     LINDIST
        JSR     WAIT
        JSR     UPDATER
        JSR     TURNR
        BRA     BOT


NOTR    JSR     LINDIST
        JSR     WAIT
        JSR     ALIGNL
        JSR     ALIGNR
        JSR     ALIGNL
        JSR     ALIGNL
        JSR     ALIGNR
        JSR     LINDIST
        JSR     WAIT
        JSR     UPDATER
        JSR     TURNR    * If we are at a deadend

BOT     JMP     HERE


***************************************
***** MAPPING ROUTINES ****************
***************************************
*********
**** Lower 4 bit circular shift routines
**** Using register A
*********
CLRMAP  LDX     #MAP
        LDAA    #$FF
        LDAB    #$00

CLRMAP2 STAA    0,X
        INCB
        INX
        CMPB    #0
        BNE     CLRMAP2

        RTS

ROTTMP  RMB     1

ROTAL   LSLA
        STAA    ROTTMP
        ANDA    #%00010000
        BEQ     ROTAL2
        LDAA    ROTTMP
        ORA     #%00000001
        ANDA    #%11101111
        RTS
ROTAL2  LDAA    ROTTMP
        RTS

ROTAR   STAA    ROTTMP
```

```
        ANDA    #%00000001
        BEQ     ROTAR2

        LDAA    #%00010000
        ORA     ROTTMP
        LSRA
        RTS
ROTAR2  LDAA    ROTTMP
        LSRA
        RTS


********
**This subroutine examines the area around
** around it, and updates the map
********

MAPIT   LDAB    #$0

        LDAA    #FORE
        JSR     SAMPLE
        JSR     OutA
        SUBA    #FORET
        BLO     MAP3
        ORAB    #%00001000

MAP3    LDAA    #LREAR
        JSR     SAMPLE
        JSR     OutA
        SUBA    #LREART
        BLO     MAP5
        ORAB    #%00000001


MAP5    LDAA    #RREAR
        JSR     SAMPLE
        JSR     OutA
        SUBA    #RREART
        BLO     MAP7
        ORAB    #%00000100

MAP7    TBA
        JSR     OutA
        JSR UPDATEMAP
        RTS


**********
*** UPDATEMAP: Given the wall locations in the form
***            0000 FR0L    -- In register B
***            It should update the 3 bits effected
***            in the map at X,Y, taking into account
***            direction
*** *** THIS IS ONLY RUN AFTER A GOFORWARD
*** *** BECAUSE IT ASSUMED THAT BEHIND IS A 0
**********
```

```
UPDATEMAP LDAA  DIR

UPDMAP3 LSRA
        BCS     UPDMAP2
        PSHA
        TBA
        JSR     ROTAR
        TAB
        PULA
        BRA     UPDMAP3
UPDMAP2 JSR     LDXAD
        STAB    0,X
        RTS
****
* Turning routines for updating direction
****

UPDATEL LDAA    DIR
        CMPA    #%00000001
        BEQ     UPLONE
        LSRA
        BRA     UPLOUT
UPLONE  LDAA    #%00001000
UPLOUT  STAA    DIR
        RTS

UPDATER LDAA    DIR
        CMPA    #%00001000
        BEQ     UPRONE
        LSLA
        BRA     UPROUT
UPRONE  LDAA    #%00000001
UPROUT  STAA    DIR
        RTS

**************************
** Used when going FORW **
**************************
UPDATEXY LDAA   DIR
        CMPA    #%00000001
        BNE     UPD2
        INC     YY
        BRA     UBOT

UPD2    LDAA    DIR
        CMPA    #%00000010
        BNE     UPD3
        INC     XX
        BRA     UBOT

UPD3    LDAA    DIR
        CMPA    #%00000100
        BNE     UPD4
        DEC     YY
        BRA     UBOT

UPD4    LDAA    DIR
```

```
        CMPA    #%00001000
        BNE     UBOT
        DEC     XX
UBOT     RTS


***************************************
***** WAIT ROUTINES *******************
***************************************

WAIT    PSHA
        PSHB
        LDAB    #$FF
WAIT2   LDAA    #$FF
WAIT3   DECA
        BNE     WAIT3
        DECB
        BNE     WAIT2
        PULB
        PULA
        RTS

***************************************
***** ERROR CORRECTION ****************
***************************************
** Linear Correction ******************
***************************************

LINDIST LDAA    #FORE
        JSR     SAMPLE
        SUBA    #STDDIST
        BLO     LINLOW
        BEQ     LINDONE

** Too Close **
LINHI   JSR     SLOWB
        LDAA    #FORE
        JSR     SAMPLE
        SUBA    #STDDIST
        BEQ     LINDONE
        BLO     LINDONE
        BRA     LINHI

** Too Far **
LINLOW  JSR     SLOWF
        LDAA    #FORE
        JSR     SAMPLE
        SUBA    #STDDIST
        BLO     LINLOW
        BRA     LINDONE

LINDONE JSR     STOP
        RTS

***************************************
***** ALIGNR    ***********************
***************************************
```

```
FREQ    RMB     1
VALR     RMB     1

ALIGNR   LDAA    #$0
        STAA    FREQ
        STAA    VALR
        LDAA    #FORE
        JSR     SAMPLE
        STAA    VALR
        JSR     SLOWR

AL2     LDAA    #FORE
        JSR     SAMPLE

        SUBA    VALR
        BEQ     ALEQ
        BLO     ALLO

*** Greater Than **
        ADDA    VALR
        STAA    VALR
        LDAA    #$00
        STAA    FREQ
        BRA     AL2

*** Equal To      **
ALEQ    LDAA    FREQ
        INCA
        STAA    FREQ
        BRA     AL2
*** Less Than     **
ALLO    JSR     STOP
        RTS

******************************
***********AlignL*************
******************************
VALL     RMB     1

ALIGNL   LDAA    #$0
        STAA    FREQ
        STAA    VALL
        LDAA    #FORE
        JSR     SAMPLE
        STAA    VALL
        JSR     SLOWL

AL2     LDAA    #FORE
        JSR     SAMPLE

        SUBA    VALL
        BEQ     ALEQ
        BLO     ALLO

*** Greater Than **
        ADDA    VALL
```

```
        STAA    VALL
        LDAA    #$00
        STAA    FREQ
        BRA     AL2

*** Equal To     **
ALEQ    LDAA    FREQ
        INCA
        STAA    FREQ
        BRA     AL2
*** Less Than    **
ALLO    JSR     STOP

        RTS

***************************************
***** DECISION MAKING *****************
***************************************
******
****** Return:
****** 0 -> Dead-End
****** 1 -> Go Forward
****** 2 -> Turn Left
****** 3 -> Turn Right
***************************************
DECIDE  LDAA    #FORE
        JSR     SAMPLE
        SUBA    #FORET
        BLO     DEC2
        BRA     DEC3
DEC2    LDAA    #1
        RTS

DEC3    LDAA    #LREAR
        JSR     SAMPLE
        SUBA    #LREART
        BLO     DEC4
        BRA     DEC5
DEC4    LDAA    #2
        RTS

DEC5    LDAA    #RREAR
        JSR     SAMPLE
        SUBA    #RREART
        BLO     DEC6
        BRA     DEC7
DEC6    LDAA    #3
        RTS

DEC7    LDAA    #0
        RTS


***************************************
***** MOTOR SUBROUTINES ***************
***************************************
GOF     PSHA
```

```
        PSHB
        LDAA    #$0
        STAA    PCOUNT

        LDD    #FORW
        STD    PW1
        LDD    #BACK
        STD    PW2

GOF2    LDAA    PCOUNT
        CMPA    #FWDPULSES
        BEQ     GOF3
        BRA     GOF2

GOF3    JSR     STOP
        PULB
        PULA
        RTS

*****************************
*****************************
TEMP    RMB     2

GOFCOR  PSHA
        PSHB
        LDAA    #$0
        STAA    PCOUNT

        LDD    #FORW
        STD    PW1
        LDD    #BACK
        STD    PW2

GOFC2   LDAA    #LFORE
        JSR     SAMPLE
        SUBA    #LFORET
        BLO     GOFC5
        TAB
        LDAA    #$0
*        LSLD
        ADDD    PW2
        ADDD    #10
        STD     PW2
        BRA     GOFC7

GOFC5   LDAA    #RFORE
        JSR     SAMPLE
        SUBA    #RFORET
        BLO     GOFC6
        TAB
        LDAA    #$0
*        LSLD
        STD     TEMP
        LDD     PW1
        SUBD    TEMP
        SUBD    #10
        STD     PW1
```

```
        BRA     GOFC7

GOFC6   LDD     #FORW
        STD     PW1
        LDD     #BACK
        STD     PW2
GOFC7   LDAA    #FORE
        JSR     SAMPLE
        SUBA    #EMERG
        BLO     GOFC4
        JSR     STOP
        BRA     GOFC3


GOFC4   LDAA    PCOUNT
        CMPA    #FWDPULSES
        BEQ     GOFC3
        BLO     GOFC2
        BRA     GOFC3

GOFC3   JSR     STOP
        PULB
        PULA
        RTS


*****************

*****************
SLOWF   PSHA
        PSHB
        LDD     #SLOWF1
        STD     PW1
        LDD     #SLOWF2
        STD     PW2
        PULB
        PULA
        RTS

SLOWB   PSHA
        PSHB
        LDD     #SLOWB1
        STD     PW1
        LDD     #SLOWB2
        STD     PW2
        PULB
        PULA
        RTS

TURNL   PSHA
        PSHB
        LDAA    #$0
        STAA    PCOUNT

        LDD     #BACK
        STD    PW1
        STD    PW2
```

```
TURNL2  LDAA    PCOUNT
        CMPA    #LEFTPULSES
        BEQ     TURNL3
        BRA     TURNL2
TURNL3  JSR     STOP

        PULB
        PULA
        RTS

TURNR   PSHA
        PSHB
        LDAA    #$0
        STAA    PCOUNT

        LDD     #FORW
        STD     PW1
        STD     PW2

TURNR2  LDAA    PCOUNT
        CMPA    #RIGHTPULSES
        BEQ     TURNR3
        BRA     TURNR2
TURNR3  JSR     STOP


        PULB
        PULA
        RTS

SLOWL   PSHA
        PSHB
        LDD     #SL1
        STD     PW1
        LDD     #SL2
        STD     PW2
        PULB
        PULA
        RTS

SLOWR   PSHA

        PSHB
        LDD     #SR1
        STD     PW1
        LDD     #SR2
        STD     PW2
        PULB
        PULA
        RTS


STOP    PSHA
        PSHB
        LDD     #STOPL
        STD     PW1
```

```
        LDD     #STOPR
        STD     PW2
        PULB
        PULA
        RTS

OC2_ISR LDAA    #%01000000
        STAA    TFLG1

        LDAA    TCTL1
        ANDA    #%01000000
        BNE     LASTHI

        LDAA    TCTL1
        ORA     #%11000000
        STAA    TCTL1


        LDD     #$0000
        STD     TOC2
        JMP     OC2_OUT

LASTHI  LDAA    TCTL1    ; or TCTL in version 2
        ORA     #%10000000
        ANDA    #%10111111
        STAA    TCTL1

        LDD     PW2
        STD     TOC2
OC2_OUT RTI


OC3_ISR LDAA    #%00100000
        STAA    TFLG1

        LDAA    TCTL1
        ANDA    #%00010000
        BNE     LASTHI2

        LDAA    TCTL1
        ORA     #%00110000
        STAA    TCTL1

        LDD     #$8000
        STD     TOC3
        JMP     OC3_OUT

LASTHI2 LDAA    TCTL1
        ORA     #%00100000
        ANDA    #%11101111
        STAA    TCTL1

        LDD     PW1
        ADDD    #$8000
        STD     TOC3
OC3_OUT RTI
```

```
**********************************
****** SHAFT ENCODER CONTROL ******
**********************************

PAO_ISR LDAA    #%00100000
        STAA    TFLG2

        LDAA    PCOUNT
        INCA
        STAA    PCOUNT

        LDAA    PACTL
        ANDA    #%00010000
        BEQ     PAO1

*Put 0 in here
        LDAA    PACTL
        ANDA    #%11101111
        STAA    PACTL
        BRA     PAO2

PAO1
*Put 1 in here
        LDAA    PACTL
        ORA     #%00010000
        STAA    PACTL

PAO2    LDAA    #$FF
        STAA    PACNT

        RTI

***********************************
***** SENSOR SAMPLING ROUTINE ******
***********************************
*Usage, LOAD A with defined port to sample
*JSR here, read sample in A
***********************************

SAMPLE  PSHB
        PSHX
        PSHA
        LDX     #$1000

        SUBA    #FORE
        BNE     SNext1
        BSET    0,X #%00010000
        JMP     Wout

SNext1  ADDA    #FORE
        SUBA    #LFORE
        BNE     SNext2
        BSET    8,X #%00000100     **Port D, Pin 2
        JMP     Wout

SNext2  ADDA    #LFORE
        SUBA    #RFORE
```

```
        BNE     SNext3
        BSET    8,X #%00000100
        JMP     Wout

SNext3  ADDA    #RFORE
        BSET    8,X #%00001000

Wout    LDAB    #SAMPLETIME     **Wait 300us for analog to stablize
WSamp4  LDAA    #$FF
WSamp2  DECA
        BNE     WSamp2
        DECB
        BNE     WSamp4

**** Pull stored A and read that port
        PULA
        STAA    ADCTL
        LDAA    #10
WSamp   DECA
        BNE     WSamp


        LDAA    #$0
        STAA    PORTD
        BCLR    0,X #%00010000
        LDAA    ADDATA

        PULX
        PULB
        RTS




*********************************
*********** INITIALIZATIONS *****
*********************************

INIT_AD LDAA    #%10000000
        STAA    OPTION
        LDAA    #40
WADINT  DECA
        BNE     WADINT
        RTS

INIT_SENSOR     LDAA    #%00000100
        STAA    SPCR
        LDAA    #%00111100
        STAA    DDRD
        RTS

INIT_MOTOR        LDD     #$0000
        STD     TOC2
        LDD     #$8000
        STD     TOC3

        LDAA    #%01100000
```

```
        STAA    TMSK1
        LDAA    #%10100000
        STAA    TCTL1
        RTS

INIT_SHAFT      LDAA    #$00
        STAA    PCOUNT
        LDAA    PACTL
        ORA     #%01010000
        ANDA    #%01011111
        STAA    PACTL

        LDAA    TMSK2
        ORA     #%00100000
        STAA    TMSK2

        LDAA    #$FF
        STAA    PACNT
        RTS

OutA    PSHA
        PSHA
        JSR     $E4DE
        PULA
        JSR     $E4E2
        JSR     $E508
        PULA
        RTS

OutD    PSHB
        PSHA

        PSHA
        JSR     $E4DE
        PULA
        JSR     $E4E2

        TBA

        PSHA
        JSR     $E4DE
        PULA
        JSR     $E4E2
        JSR     $E508

        PULA
        PULB
        RTS

INIT_SCI PSHA
        LDAA    #$30
        STAA    BAUD
        LDAA    #%00000000
        STAA    SCCR1
        LDAA    #%00001100
        STAA    SCCR2
```

```
        PULA
        RTS

********************************
*** Subroutin: LDXAD - Load X with Address of current
***                       X Y data.
***************************

LDXAD   PSHA
        PSHB
        LDAB    YY
        LSLB
        LSLB
        LSLB
        LSLB
        ADDB    XX
        LDAA    #$30
        XGDX

        PULB
        PULA
        RTS


SHOWMAP PSHX
        PSHA
        PSHB

        LDAA    #0
        STAA    YY
SMAP3    LDAA    #0
        STAA    XX



***********First Pass (TOP)***********
SMAP2    JSR     LDXAD

        LDAB    0,X
        ANDB    #%10000000
        BEQ     SMAP222

        BRA     SMAP22

SMAP222 LDAB    0,X
        ANDB    #%00000010   *Top mask
        BEQ     SMAP22

        LDAA    #DASH
        JSR     $E4EC
        LDAA    #DASH
        JSR     $E4EC
        LDAA    #DASH
        JSR     $E4EC
        BRA     SMAP23
SMAP22   LDAA    #SPACE
        JSR     $E4EC
```

```
        LDAA    #SPACE
        JSR     $E4EC
        LDAA    #SPACE
        JSR     $E4EC

SMAP23   INC     XX
        LDAA    XX
        CMPA    #16
        BNE     SMAP2

        LDAA    #0
        STAA    XX

        PSHA
        JSR     $E508
        PULA


***********SECOND PASS (middle)***************
SMAP4    JSR     LDXAD
*        JSR     OutD

        LDAB    0,X
        ANDB    #%10000000
        BEQ     SMAP444
        LDAA    #SPACE
        JSR     $E4EC
        LDAA    #AST
        JSR     $E4EC
        LDAA    #SPACE
        BRA     SMAP45

SMAP444 LDAB    0,X
        ANDB    #%00000001
        BEQ     SMAP42
        LDAA    #PIPE
        BRA     SMAP43
SMAP42   LDAA    #SPACE
SMAP43   JSR     $E4EC
        LDAA    #SPACE
        JSR     $E4EC

        LDAB    0,X
        ANDB    #%00000100
        BEQ     SMAP44
        LDAA    #PIPE
        BRA     SMAP45
SMAP44   LDAA    #SPACE
SMAP45   JSR     $E4EC


        INC     XX
        LDAA    XX
        CMPA     #16
        BNE     SMAP4

        LDAA    #0
```

```
        STAA    XX

        PSHA
        JSR     $E508
        PULA

***********Third Pass (bottom)***************
SMAP5    JSR     LDXAD

        LDAB    0,X
        ANDB    #%10000000
        BEQ     SMAP555
        BRA     SMAP52

SMAP555 LDAB    0,X
        ANDB    #%00001000   *Bot MASK
        BEQ     SMAP52

        LDAA    #DASH
        JSR     $E4EC
        LDAA    #DASH
        JSR     $E4EC
        LDAA    #DASH
        JSR     $E4EC

        BRA     SMAP53

SMAP52   LDAA    #SPACE
        JSR     $E4EC
        LDAA    #SPACE
        JSR     $E4EC
        LDAA    #SPACE
        JSR     $E4EC

SMAP53   INC     XX
        LDAA    XX
        CMPA     #16
        BNE     SMAP5


        PSHA
        JSR     $E508
        PULA


        INC     YY
        LDAA    YY
        CMPA    #10
        BEQ     SMAPEND
        JMP     SMAP3
SMAPEND  PULB
        PULA
        PULX
        RTS
```