Theseus: A Maze-Solving Robot

C. Scott Ananian and
Greg Humphreys

Independent Work
Presented to the
Department of Electrical Engineering
at Princeton University

Advisors:
Steve Lyon and Doug Clark

May 23, 1997

This paper represents my own work in accordance with University regulations.

Contents

1 Introduction

Autonomous robotics is a field with wide-reaching applications. From bomb-sniffing robots to autonomous devices for finding humans in wreckage to home automation, many people are interested in low-power, high speed, reliable solutions. There are many problems facing such designs: unfamiliar terrain and inaccurate sensing continue to plague designers.

Robotics competitions have interested the engineering community for many years. Robots compete on an international scale hundreds of times a year, in events as varied as the annual MIT robot competition and the BEAM robotics games in Singapore. One of the competitions with the richest history is the MicroMouse competition. Micromice are small, autonomous devices designed to solve mazes quickly, and efficiently. The MicroMouse competition has existed for almost 20 years in the United States, and even longer in Japan. It has changed little since its inception.

The goal of the contest is simple: the robot must navigate from a corner of a maze to the center as quickly as possible. The actual final score for a robot is primarily a function of the total time in the maze and the time of the fastest run. MicroMouse mazes are typically designed as a lattice of square posts with slots in the sides to accomodate modular wall units. The specifications for the MicroMouse contest are presented in appendix A (See [3]).

Our design incorporates many of the advances in small scale robotics that have been explored in 20 years of MicroMouse competitions. We also introduce some features never before seen in a working design. We therefore believe that our design is the most advanced MicroMouse ever conceived. We incorporate sophisticated long and short range sensing mechanisms, accurate motion and distance measurements, adaptability, high speeds, and a sophisticated, physically based algorithm for maze solving.

2 MicroMouse Mechanical Design

The MicroMouse rules are remarkably flexible with respect to mouse design; we did considerable research into current and past international-class MicroMouse designs in order to formulate a philosophy of design.

The most important recent advance in micromouse performance has been Dave Otten's development of "diagonal-capable" mice. Instead of adhering to 90-degree turns and a manhattan path, Otten's MITEE mouse looked for op-

portunities to cut corners and run path diagonals. This imposes more strigent width requirements on the mouse. We collected an exhaustive library of maze layouts, dating from the 1^{st} All-Japan Micromouse Contest in 1980, an in our analysis discovered the increasing prevalence of what we dubbed "30-degree" paths. In constrast to a diagonal path, which is constructed from a sequence of alternating orthogonal paths (for example, "one-up, one-over"), a "30-degree" path contains a two-to-one ratio of orthogonal segements (for example, "two-up, one-over"). Our calculations revealed that the width requirements of a "30-degree"-capable mouse was required to satisfy extremely tight width requirements, which our completed mechanical design could not meet. We fell back on traditional 45-degree diagonal capability, which dictated the 9cm width of our mouse.

According the Dave Otten, the most pressing challenge to mouse design today is traction control. High-power DC motors can easily outstrip the traction of the mouse tires, leading to wheel slippage and poor performance. Robin Hartley of New Zealand was the first to attempt to utilize a ground-effects fan to achieve better wheel traction: the downforce generated by the fan increases the amount of acceleration force obtainable from wheel traction. However, Hartley designed his mouse on a large scale: a 15cm mouse diameter mouse, weighing 1.6kg, with a fan capable of providing 2kg down force. As a result the traction benefits did not compensate for his mouse's poor cornering and sensing capabilities. We performed an analysis along the lines laid out in [7], using fans from a variety of manufacturers. For a typical example, we discovered we could increase our traction by about 20%, if we could keep the total weight of the mouse below 200 grams. Using miniature DC brushless fans, this was possible, and we were able to draft a mechanical design meeting our width requirements incorporating the ground-effect fan.[1] Appendix C details the mathematics supporting our claim of efficacy.

There is no strong agreement on the advantages of two-wheeled over four-wheeled micromice, but it was felt that a two-wheeled mouse was mechanically simpler, and thus more likely to allow us to meet our weight requirements. Servo inaccuracies, increased parts count, and other factors made us shy away from four-wheeled designs.

Sensing devices have traditionally been classified as "over-the-wall" or "under-

[1]The fan used in the final design was a Panasonic FBK series brushless DC fan from Digikey electronics, measuring 40mm square and 20mm deep. It develops 11.5mm H_2O static pressure.

the-wall." The original micromice used the red-painted wall tops to determine orientation, with long wing-like sensor arrays extending over the walls. More recent designs have avoided the large moments of inertia that these extended sensor assemblies create, and have opted instead for compact low-riding mice which measure distances from the insides of the walls. This latter approach was markedly superior, and permitted extremely compact designs. Sensor design will be discussed further in section 3; from the mechanical design aspect it is only important that we decided no use optical, rather than ground-contact (rolling) distance sensors.

The mechanical design of the mouse was completed using Pro/Engineer CAD tools, and fabricated using computer-controlled milling machines. Appendix D details the design. The use of computer-controlled milling machines enabled a high degree of precision in the design; clearances as tight as one-quarter millimeter were successfully utilized. Most of the components are plastic, to save weight; the motor and bearing assemblies are aluminum for strength and rigidity, and the top surface is made of aluminum to allow it to act as a heat-sink for the power and control electronics.

3 Sensors

In order to execute these algorithms (and keep from crashing into obstacles), the MicroMouse must be able to "see" the environment around it. There are some very serious technical challenges associated with this task. First, we need not only to be able to detect obstacles, but also to measure our distance from them at very close proximities. When our mouse traverses a diagonal path, the maximum spacing between posts is exactly $\frac{16.8}{\sqrt{2}} = 8.4\sqrt{2} \approx 11.76$cm. Because of space constraints imposed by the sizes of our motors and batteries, the mouse is exactly 9cm wide, which leaves 1.38cm of clearance on either side of the mouse. At top speeds of >15 ft/sec, it is clear that we must have accurate distance measurements at short distances in order to avoid hitting the posts.

Problems with traditional short range sensors make this difficult. In order to get the resolution we need to keep our control system stable at high speeds, we need to limit the maximum visible distance of our short range sensors to a maximum of $\approx$ 8cm. Now consider a mouse accelerating at 1g to a maximum velocity of 10 ft/sec. Physically, the wheels *must* slip when accelerating; figure 1 is what automotive engineers call a "grip-slip" curve for a typical rubber tire. It

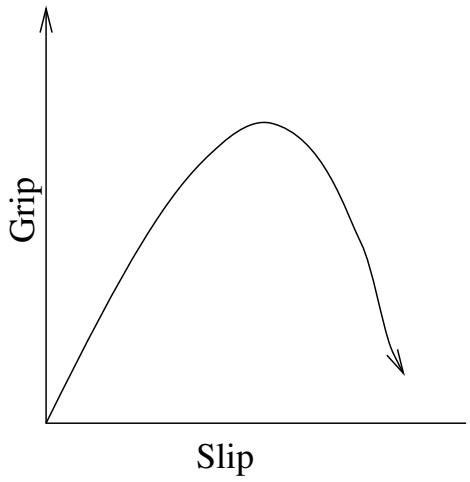


Figure 1: Grip-versus-slip curve for typical tire

is obvious that some amount of wheel slip is necessary to exert the acceleration force. Worse, the actual grip-slip curve is dependent on floor material and thus unknown to us—all we know about the floor is that it absorbs light. However, we must have a reliable way to measure the distance we have traveled. It is clear, then, that any distance measure we make can not totally rely on the motion of the wheels or the motors. Typical advanced mouse designs get around this problem by mounting "idler" wheels on the ground that are not powered by the motors, but merely roll on the ground. While this solution is attractive, it uses precious space under the mouse, and, in our design, this space is at a premium. It was therefore necessary to design a solution on top of the mouse. This led to the design of our long range sensing mechanisms.

3.1 Short range sensors

As discussed in the introduction to this section, our short range sensors need to have a dynamic range of 1-8cm. To accomplish this, we use a fairly traditional design for distance measurement. Our short range sensors use one infrared LED and two photo-sensitive detectors, spaced a known distance apart. This configuration is shown in figure 2.

Note that the angle of acceptance of the two detectors is very narrow, while the angle of light given off by the LED is large. Therefore, because of the $\frac{1}{r^2}$

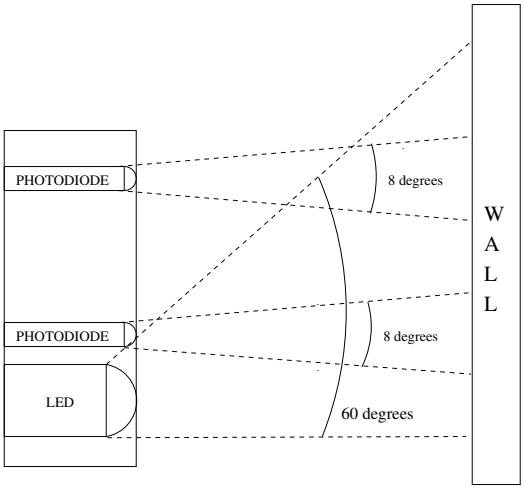


Figure 2: Design for short range sensors

falloff nature of light, the quotient of the voltages produced by the two phototransistors will be a monotonic function that is directly related to distance. An empirically determined lookup table will allow us to compute the actual distance if necessary.

Another important property of this design is the fact that it is completely insensitive to the operating environment. At startup time, we calibrate the sensors by taking measurements of voltage readings in the contest environment. This allows us to correct for the ambient light levels at run-time. In addition, our two detector design has a significant advantage over the more conventional one detector method: it is totally insensitive to the reflectivity of the surface.

Recall that MicroMouse mazes are typically designed as a lattice of square posts with slots in the sides to accommodate modular wall units. It is well known in the MicroMouse community that the reflectance of the walls of the maze may not be the same as the reflectance of the posts, especially in Japanese mazes. Many American mice have failed in Japan because their sensors got confused when they were shining on a post. Because we are taking a quotient of two voltages measured with a vertical displacement, as long as the reflectivity is constant *in the vertical dimension*, our sensors will behave properly.

It is important to note that the above quotient is in fact proportional to the square of the distance, which means we will get much higher resolution measurements as the distance decreases. This is exactly the behavior we desire,

since it is very important to remain stable at close proximities to walls, especially in the diagonal path case.

3.2 Long range sensors

Our long range sensor mechanisms are much more sophisticated. In order to understand the design, it is first important to discuss how these sensors are to be used. The long range sensors measure distance to an object that is much farther away than can be measured by our short range sensors. However, we expect the accuracy to be much less. The only requirements we place on our short range sensors are that they can measure our position to within half a maze block, or 9cm. The reason for this comes from the worst case scenario for wheel slippage, shown in figure 3.

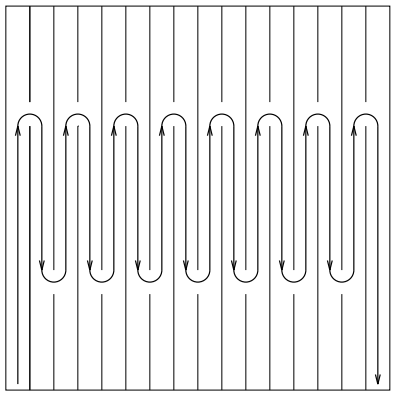


Figure 3: Worst case scenario for wheel slippage

Every time we pass an opening in a wall, our short range sensors will detect this. As long as we know what block we're in, we can update our current absolute position with high accuracy. The worst case situation shown in figure 3 shows a maze configuration where we must speed down a long straightaway and then make a very sharp turn somewhere in the middle. For motor control reasons, the turn must begin before the front of the mouse crosses the opening in the wall. Therefore, we must know where we are to within half a block, or two things might go wrong. First, we might begin the turn at the wrong time, hitting the wall mid-turn. Second, we might not begin braking early enough to make the turn at all without slipping. Wheels slipping in place is a recoverable error, but skidding around a turn poses a much harder problem.

To design a sensor to meet these requirements, we created a special purpose laser range finder. Our design uses a linear triangulation method that exploits properties of similar triangles in a way similar to the grade school method of computing the height of a building based on its shadow. Our design is shown in figure 4.

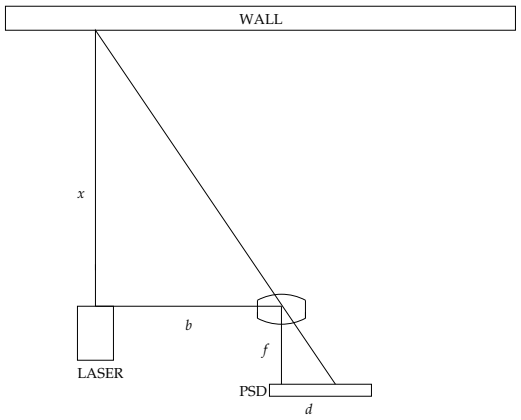


Figure 4: A long range sensor

The spacing b between the laser and the lens is known, as is the distance f between the lens and the rear surface (in fact, it is the focal length of the lens). Because of this, we can compute the distance from the laser to the object x if we can measure the displacement d of the image of the laser beam on the rear surface. This turns out to be relatively easy; we simply use a "Position Sensitive Detector", or PSD. PSD's are devices typically found in auto focus cameras, where they are used for approximating distances to objects. This application is very similar to what we are trying to accomplish with our long range sensors. By focusing the image of a laser beam on the PSD, we can obtain a fairly accurate measurement of our distance up to 5-6 feet away.

3.3 Electronics for long range sensing

The long-range sensors use sophisticated techniques to achieve maximum possible resolution. The raw sensor output is a micro-amp level current swamped by light noise, especially 60Hz flourescent "buzz." We modulate the laser in the frequency-domain, and then demodulate the sensor output phase-locked to the laser to recover the desired signal while achieving very high noise reduction. The

modulation frequency is 10kHz, which is as far from our 60Hz dominant noise source as possible without running afoul of the limited high-frequency response of the position-sensitive detector sensing element.

The first step in the signal chain is a pre-amplifier to convert the sensor current to a voltage, amplified one million times. The output of this stage is a nominally one-volt signal.

This signal is fed into a switched-capacitor bandpass filter. This device melds digital and analog technology to allow filter design with excellent frequency precision. The switched-capacitor device creates a bandpass filter using the same frequency source as the laser modulation, so the center frequency of the filter is guaranteed to coincide exactly with the laser modulation frequency. The output of this stage is a small-amplitude 10kHz signal, which is then AC-coupled to eliminate any DC bias. The bandpass filter is second order, so it reduces the 60Hz noise by more than 80dB.

This signal is fed into a demodulator, which multiplies it by a phase-corrected version of the laser modulation to recover the signal amplitude. This signal is low-level and contains some high-frequency noise at the modulation frequency and higher.[2]

The next stage is a 3rd-order low-pass filter to remove this high-frequency noise. This filter also amplifies the signal to the correct level for input to analog-to-digital converter. The filter cut-off frequency is 1kHz, so the maximum signal bandwidth is 1 sample/millisecond after passing through the filter. The 3rd order filter attenuates the 10kHz carrier by 60dB.

The analog-to-digital conversion maintains higher than 60dB accuracy, so the 10kHz ripple is still significant. To eliminate this effect, the two signals from each sensor are fed into a sample-and-hold amplifier. This allows us to sample both signals at exactly the same instant in time. Since the desired information (distance) is a function of the ratio of the two signals, sampling synchronously minimizes the effect of the high-frequency ripple.

The last stage allows an amplifier with gain of 4 to be inserted into the signal chain, to enhance resolution at low signal amplitudes.

The use of a single frequency reference for the laser modulation, bandpass filter, and demodulator allows high precision with simple circuitry. Furthermore, the frequency generation is accomplished with a reprogrammable logic element, so that the different signals can be phase-corrected through software.

[2]The switched capacitor filter also adds some 200kHz noise which will be eliminated along with the 10kHz modulation noise.

4 Hardware Platform

The electronics design centers around a Motorola MC68334 processor [4]. The MC68334 has integrated motion-control functions which make it ideal for our project and it's high integration level helps keeps parts count low. It is interfaced to 2MB of EEPROM for in-circuit reprogrammable non-volatile storage and 1M of SRAM for data structures and stack. The BDM (Background Debugging Mode) interface of the processor allows debugging and reprogramming [2].

The processor is interfaced with a complex analog signal chain described in section 3.3 through the integrated A/D. Short-range and long-range sensors are fed through analog multiplexors, sample-and-hold and variable gain amplifier stages before being input to the A/D; battery-level and temperature signals use the remaining channels of the integrated A/D.

Pulse-width-modulated outputs from the 68334 TPU (Time Processor Unit) drive a pair of PALs programmed to do brushless-motor commutation. They also generate an output signal whose frequency is proportional to commutation speed, in order to provide a secondary velocity sensing mechanism. The commutator output is fed through opto-isolators before input to an array of power transistors, cooled by direct mounting to the aluminum mouse chassis.

A total of four reprogrammable logic devices (ispGALs) are used in the design. This allows flexible modification to the electronics without necessitating socketed parts or desoldering. Two are used for the motor commutation, one is used for random processor support logic, and the last is used to generate the laser modulation and demodulation clocks. The reprogrammable device allows us to coalesce a long clock-divider chain into one device, as well as letting us flexibly change the phase relationship of the modulation and demodulation clocks to compensate for analog processing phase delays.

Power for the electronics is carefully conditions to isolate battery voltage swings and motor transients from the sensitive digital and analog electronics. The 12V, 2A battery pack (8 Lithium rechargable AA cells made by Tadrian electronics) is rectified and filtered to isolate transient negative-going voltage spikes, then down-converted to 8.0V by a pair of 1A regulators. The main processor electronics are powered by a 5V 1A regulator from one of the 8V 1A supplies. The other 8V 1A supply powers a second 5V 0.5A regulator feeding the ispGALs, and also feeds the analog electronics. The 8V supply is mirrored with a switched-capacitor voltage converted to -8V, and then the nominally $\pm$8V supplies are down-converted to $\pm$5V using low-noise linear regulators. The

output is a ±5V 100mA. Finally, a precision 3V reference (generated either by an Analog Devices AD780AR part, or a National Semiconductor LP2950ACZ-3.3 part) is generated from the stable analog +5V supply for use by the A/D converter.

5 Algorithms

Traditional maze solving algorithms, while appropriate for purely software solutions, break down when faced with real world constraints. When designing software for a maze solving robot, it is important to consider many factors not traditionally addressed in maze solving algorithms.

5.1 Traditional Algorithms

Our initial attempt at a maze solving algorithm was a traditional depth first search. We were mainly interested in testing features of our simulator, and wanted merely to find a path from the start to the center. Unmodified depth first search is not applicable to a physical device, because it must backtrack when it encounters a dead end in its search to reach its last decision point. Except in the high school level competitions, MicroMouse mazes are almost always constructed so that there is more than one path from the start of the maze to the center. Therefore, depth first search fails miserably for use in a MicroMouse competition. Although it will always find a path from the start to the center, it will not even be the shortest path, much less the fastest.

Kruskal's shortest path algorithm for graphs [6] can be easily modified to solve mazes; the maze is simply represented by a graph with a node at each cell, and links of equal weight connecting adjacent cells without a wall between them. However, Kruskal's algorithm suffers drawbacks, as well.

First, MicroMice do not have immediate random access to the maze configuration, as the mouse must explore unseen portions of the maze, which takes time. In fact, most maze solving algorithms assume that the entire maze configuration is known *a priori*, and that we have random access to the complete maze configuration. Obviously, this is not the case.[3] Our information about the maze changes as we explore, so we must make certain assumptions about unseen portions of the maze at certain steps in most algorithms.

[3] Actually, a team from Motorola created a mouse that extended a CCD camera overhead and performed image processing to determine the configuration of the maze before running. The mechanical overhead of the scheme ensured its failure.

Second, the path computed may not necessarily be the best path to choose. Although it is true that Kruskal's algorithm will return the *shortest* path, this is not necessarily the fastest path because our mouse can accelerate and also has cornering behavior. Therefore, we would like to favor paths that have long straightaways, and few tight turns. Even if the maze configuration was known *a priori*, the shortest path from the start to the finish may not be the fastest. We must factor in things such as maximum acceleration, turning speed, variable floor conditions, etc. This requires a sophisticated, physically based, path planner, with the appropriate assumptions about unseen areas of the maze.

5.2 Maze solving

First, we present pseudocode for the main body of our maze solving algorithm.

```
procedure AlgorithmStep(path_t best_path)
{
   if already in center
      Take best_path back to start square
   else if current position is on best_path
   {
      Follow best_path towards destination until in center or best_path blocked
      if best_path blocked
         best_path = ComputeBestPath(best_path.start, best_path.end);
      AlgorithmStep(best_path)
   }
   else
   {
      new_path = ComputeBestPath(current_position, best_path);
      AlgorithmStep(new_path); /* Now we know we're on best_path */
      AlgorithmStep(best_path);
   }
}
```

Notice that the `ComputeBestPath` function can either compute the best path between two points, or from a point to a path. The latter computation is simply the minimum of all point-to-point path costs from the given point to every point on the given path.

This algorithm not only finds the best path from the start to the center, but it also deals nicely and efficiently with the fact that we do not have random access to the maze.

In fact, this pseudocode is slightly simplified. The main code actually calls this routine twice; once to search for the center, and once to search for the start. When we are searching for the *center* of the maze, the `best_path` variable is recomputed at each step to go from the current position to the center, rather than from the start to the center. This is to ensure that we find the center as quickly as possible, in case we are unable to make a high speed run. Once the center is found, the true best path is searched for. Another slight difference involves the assumptions about the unseen portions of the maze. When computing the best path between center and start, we assume that unseen portions of the maze are open and can be traversed at high speeds. However, when computing best paths from the current position, we make the more conservative assumption that the space is open, but we must traverse that space at searching speed.

This is because if we are computing a best path that originates at our current position, we intend to traverse it immediately, and we must therefore account for entering unseen territories. However, for start-to-finish paths, we will ensure that all blocks along the path are open, so once the path is computed and verified as traversable, there is no longer a need to assume anything about it, and we can traverse it at the highest speed possible. This distinction does not matter for the simple flooding algorithm, but will come into play later once the algorithms become more physically based.

With this algorithm, the bulk of the problem has been isolated to one routine, called *ComputeBestPath*. This routine takes a start point and and end point (and some parameters for assumptions to make), and computes the best path between those points. The question remains then, how can we implement this algorithm to fully encapsulate all the physical characteristics of our maze solving device?

5.3 Naive shortest paths through flooding

Our first attempt was to use a slight modification of a shortest paths algorithm. We call this a "flooding" algorithm. This is a simple implementation of Kruskal's shortest path algorithm made specific to mazes. Imagine a hose being placed in a cell in the maze, and water being allowed to flow from a cell to a neighbor in one unit time. A simple recursive algorithm will yield the amount of time

it takes for water to reach any point in the maze from any other point in the maze, and an appropriate traversal of the matrix of flood values will yield the shortest path.

Code for this algorithm is shown below. The flood values are represented as a 16x16 array of integers (`maze`), and the wall configurations are represented with four 16x16 boolean arrays called `north`, `south`, `east`, and `west`.

```
void Flood(int x, int y, int flood_val)
{
   if (maze[y][x] > flood_val)
   {
      maze[y][x] = flood_val;
      if (!east[y][x])
         Flood(x+1,y,flood_val+1);
      if (!west[y][x])
         Flood(x-1,y,flood_val+1);
      if (!north[y][x])
         Flood(x,y+1,flood_val+1);
      if (!south[y][x])
         Flood(x,y-1,flood_val+1);
   }
}
```

Once we have a flooding routine, it is easy to answer the question "what is the best path from point a to point b". We simply flood the maze starting at b with a flood value of zero, and then traverse the maze starting at a and following decreasing flood values until we reach b. This works because once flooding is done, the flood value at a is exactly the length of the path from a to b. Therefore, if we follow the flood values in a monotonic decreasing order, we are guaranteed to reach b in exactly the minimum amount of steps. Note that the path generated by this algorithm is not unique, as there may be several paths from a to b with identical lengths.

Also, because we know that the center of the maze is open, the maximum flood value for any flood in any square is 254, so we can use 255 to initialize the flood array, and can therefore represent the entire array of flood values in 256 bytes of memory. This is because the worst case for flooding is a spiral in which the flow has to touch every square. If our flood starts at one corner of the maze, and spirals towards the center, it will have length 252 when it enters the center

of the maze. Because we know that the center is an open 2x2 square, it will take 254 steps to get to the corner of the center, not 255 in the closed-center worst case. Therefore, if all flood values are initialized to 255, a flood value of 255 represents an unreachable portion of the maze. MicroMouse maze designers will often put these in the maze (i.e., a fully walled off square) to test the robustness of the MicroMouse algorithms.

Although the straightforward flooding algorithm works well for a shortest path computation, it does not allow our mouse to take full advantage of its acceleration capabilities, and its ability to traverse diagonal paths without turning. That is, because of how thin our mouse is, and the precise spacing of the sensors, if we intend to traverse a "stairstep" pattern in the maze, we can simply turn 45 degrees and go straight without turning at all. This is a huge advantage, and is yet another thing that MicroMouse maze designers will intentionally put in the maze. The maze flooding algorithm is the one typically used by student mouse projects when they are simply interested in getting something that works. It is not a world class implementation of a physical maze solving algorithm, and we clearly needed something better.

5.4 Better flooding

Our first attempt to account for the acceleration values was a modification to the flooding algorithm to "weight" the flood values based on a record of how long the water had been traveling without turning. This algorithm turned out to be ugly to implement, but when fully functional, yielded promising results. By creating a table lookup function based on measured acceleration values, we were able to get a fairly accurate model of the characteristics of our mouse. This simply required retaining state information at each cell in the graph. The state information would store the current cost of the path, as well as the direction of flow. This way, as the water flowed along a straight path, the added penalties would be a function of the length of the flow.

Again, this algorithm does not deal with diagonal paths. Although our path optimizer was able to collapse stair-step patterns in a path into a long diagonal, it was not clear how a modification of the flooding algorithm could use the knowledge of diagonal paths to allow water to "flow" diagonally.

Another severe disadvantage of both flooding algorithms is their large stack requirements. Like depth first search, maze flooding is highly recursive. On a maze that has 256 blocks, the stack requirements can quickly become unrea-

sonable, especially for an embedded system. The standard Computer Science paradigm of "memory is cheap" simply doesn't apply here. It was clear we needed to come up with something new.

5.5 "Oracle" path planner

A software module named `oracle` implements our algorithm for computing the fastest path while accounting for physical characteristics of our mouse. The basic strategy is a textbook breadth-first search, extended with look-ahead and pruning. The problem is that in order to compute the sequence of turns (and thus the time required) to run a path, we need a certain amount of look-ahead. For example, the cost to travel one square north is different if we're describing a straight-line path, or the top of a turn in which the next square to be visited is to the east. It turns out that it is fairly simple to make a canonical collection of such "turn patterns," and efficiently search through them to discover path cost.

This requires that we keep a certain amount of path context available. For paths which include 45-degree turns, we need 7 unit-travel elements of context. But the costs computed necessarily lag behind the extent of path "discovery"—we can't tell the cost of a turn until we've seen 4 blocks after the turn. Conceptually, we need to see enough blocks before the turn to determine the entrance angle and enough blocks after the turn to determine the exit angle before we can determine the angle subtended by the turn. The net result is that our computed costs reflect the time needed to reach the mid-point of the 7 units of context. This requires look-ahead in the standard breadth-first search algorithm; which impacts the amount we can prune the path. We use some heuristics to eliminate contexts which self-intersect or do other obviously incorrect things.

The only thing left is to efficiently map 7-unit "contexts" to their proper turn costs. We have implemented a simple lexer to perform this mapping in no more than 7 array references, with a compact 177-entry table. The base patterns to be recognized are:

`N-N-E-N`	45-degree turn from orthogonal
`N-N-E-E`	90-degree turn from orthogonal
`N-N-E-S-E`	135-degree turn from orthogonal
`N-N-E-S-S`	180-degree turn from orthogonal
`N-W-N-N`	45-degree turn from diagonal
`N-W-N-E-N`	90-degree turn from diagonal
`N-W-N-E-E`	135-degree turn from diagonal

The patterns have been normalized so that the first direction is north. Orthogonal directions are north, east, south, and west; diagonal directions are northeast, southeast, southwest, and northwest. The lexer-generator we wrote automatically generates the mirror-image and rotated versions of these patterns from the normalized patterns. Further pattern expansion is necessary to smoothly integrate the turn cost as the path is extended. See appendix B for the complete lexer pattern input.

The lexer-generator creates the lexical analyzer from turn patterns. It is extendible to accomodate the much larger library of turn patterns necessary to properly evaluate 30-degree paths. Our lexer is provided with a set of predetermined path costs for all types of patterns.

The search algorithm is optimized to grow the search tree from both the start and goal simultaneously, to reduce the polynomial growth of cells searched. The algorithm as implemented suffers from some search instabilities at the path midpoint because of this optimization, as well as occasionally overly-agressive search pruning. These trade-offs are acceptable for the performace gains they allow.

6 A Simulator

Considering the amount of time required for the construction of the MicroMouse, it was important that we understand the behavior of our algorithms and visualize the behavior of the proposed system before the finalization of the hardware and electronics.

To accomplish this, we created a MicroMouse simulator that accurately simulated both our algorithms and the physical characteristics of our mouse. This simulator is highly cross-platform, and allows us to develop and debug our algorithms in almost any environment.

6.1 Graphics

We wanted our simulations to be as graphical as possible, and to allow us to visualize as many aspects of our system as possible. There are many issues involved in implementing a visual simulation of a moving system, which we discuss in this section.

6.1.1 Animated vs. static

Since we are simulating the motion of a robot in real time, obviously it would be beneficial to have our simulation run in a graphical environment in real time as well, to visualize the behavior of the mouse. However, since real time graphics are notoriously system dependent, we also created a static version of our simulator that merely dumps results to disk for later analysis.

Both methods have their advantages and disadvantages. The real-time simulator allows us to see exactly how the mouse moves while it is searching the maze, which lets us optimize certain cases until we are satisfied that the mouse searches in what we consider to be an intelligent way. In addition, the real-time version can be compatible with telemetry information relayed back to the computer by the mouse. This would allow us to debug high level software problems specific to the embedded version of the algorithm. On the other hand, a complete run on the real-time simulator takes considerable time, since the mouse must animate slowly enough for the programmer to comprehend its motion paths.

The static version allows us to make runs much more quickly, and merely look at the path chosen by the mouse at the end of the search phase. This method clearly has the advantage that the results can be viewed at a later time, and that it allows more detailed analysis, since the path is not constantly moving. However, the static version gives no insight into how the mouse arrived at its decision of what path to take.

6.1.2 Static simulation

First, we look at the simpler case; the static version, named Frankie. In this case, we simply want to be able to see the path that the mouse chose as the best path through the maze. This information is not as useful as the full spectrum of information afforded by the graphical version, but it served very well when we were improving the algorithm for deciding which path through the maze is

"best".

Another goal for Frankie was that it be completely portable to any system that we had not written a real-time interface for, to allow us to work in any environment that we encountered. To accomplish these goals, we chose to have the simulator output TCL/TK code, which could be saved to a file and viewed at a later time.

TCL is an interpreted language developed by John Ousterhout [5]. It has been ported to many platforms, and offers a system-independent set of "widgets" called TK, which allow the creation of graphical user interfaces in an extremely simple, portable way. It is the perfect tool for our static graphical output.

Frankie outputs code to create a window, draw a picture of the maze in question, and draw a long line representing the chosen path through the maze. This output can then be saved to a file and interpreted by the TCL/TK interpreter, which then displays our graphics in a window, regardless of the system we were working on. X Windows on a UNIX platform, Windows 3.1, Windows 95/NT, or Macintosh platforms are all supported by TCL/TK, thus achieving our portability goal. We are even able to simulate our mouse's behavior as a web page using the TCL/TK plug in for Netscape Navigator.

6.1.3 Real-time simulation

To visualize the full behavior of our system, we needed a graphical simulator that could animate the behavior of our mouse. In addition, we wanted a system that would be compatible with a debugging telemetry board we designed for the mouse. The real-time simulator, called Benji,[4] runs with multiple *threads* to allow this behavior.

A *thread* is a sequence of execution of instructions; one program can create multiple threads, which are executed independently and either share a single processor or can take advantage of multiple processors if they are available in the simulator machine. Threads support the notion of "shared memory". That is, certain portions of memory are readable and writable by all threads within one program. It is this shared memory feature that allowed us to create Benji.

Benji consists of two threads: one thread controlling the graphics, and one thread controlling the motion of the mouse. The interaction between these two threads is crucial to our design. The graphics thread is, in a sense, "dumb". It

[4]Benji and Frankie were the two white mice in Douglas Adams' *The Hitchhiker's Guide To The Galaxy* who oversaw the construction of the Earth [1].

doesn't know anything about mice or physical properties or motion; all it knows how to do is draw a picture of the world. To do this, it first draws a picture of the maze in the window, and then looks in shared memory to find out where to draw the mouse, and how to plot path tracking information. The structure that holds the mouse position information is constantly being updated by the other thread, which is simulating the motion of the mouse.

This behavior is very convenient for a number of reasons. First, it allows us to separate the graphical user interface almost completely from the code doing the computation. In fact, Benji and Frankie share most of their code. In addition, it makes Benji more portable since it is very clear which portions of the code need to be re-written for a new platform. In addition, since the graphics thread is merely drawing whatever it finds in a shared structure, it has no knowledge of what is making the values in that structure change. In particular, the second thread of the application does *not* have to be our simulator. It can instead be code that reads debugging information directly from the telemetry board on the mouse, giving us a graphical representation of the mouse's perspective on the world. This feature allows us to determine quickly when the mouse's internal model of the world becomes inconsistent with the actual world.

Our first implementation of a real-time graphical engine for our simulator was done for Silicon Graphics machines, using OpenGL. Although this version was easy to implement and worked very well, it was not practical to cart an SGI machine to the field where we wanted to take telemetry measurements from a real mouse. Therefore, a second version for the Power Macintosh was written. Versions of our simulator have also existed in various forms for the BeOS Operating System, as well as Windows NT. We found that our simulator and our separation of algorithm and visualization allowed us to critically evaluate our choice of algorithm, as well as debug the actual code that would later run on the embedded controller of the mouse.

7 Conclusion

MicroMouse is a prime example of an engineering challenge in which the shortcomings of theory are exposed. Many real-world constraints conspire to render standard techniques for problem solving inappropriate. We have presented our solutions to some of these problems.

Solving mazes is a problem that has been explored in great detail in Com-

puter Science, but none of the standard extant solutions were appropriate for our problem. We have created a new maze solving technique appropriate for use in a physical device that fully accounts for its physical characteristics. In addition, we have created a cross platform MicroMouse development environment that facilitates investigation of new MicroMouse algorithms. Our system shows great promise for physically correct simulation and solutions of real-world problems that are not fully addressed by standard theoretical results.

In addition, we have created a hardware platform that is able to support international-level MicroMouse contest competition. The cross-disciplinary nature of the project has required us to learn elements of mechanical, control, signal, and computer engineering. We are confident that future work will validate our design decisions.

References

[1] Douglas Adams. *The Hitchhiker's Guide to the Galaxy.* Ballantine Books, 1981.

[2] Scott Howard. A background debugging mode driver package for modular microcontrollers. Application Note AN1230/D, Motorola Semiconductor, 1996.

[3] The MicroMouse competition. World Wide Web. `http://sst.lanl.gov/robot/micromouse_rules.html` .

[4] Motorola Semiconductor. *MC68334 Technical Summary*, 1996. MC68334TS/D.

[5] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[6] Robert Sedgewick. *Algorithms in C.* Addison-Wesley, 1990.

[7] J. Y. Wong. Introduction to air-cushion vehicles. In *Theory of Ground Vehicles*, chapter 8. J. Wiley, 2nd edition, 1993.

A MicroMouse contest specifications

These are the specifications for the MicroMouse Competition held at Princeton University in April, 1997. This competition was part of the IEEE Region I student conference.

A.1 Maze Specifications

1. The maze shall comprise 16 x 16 multiples of an 18 cm x 18 cm unit square. The walls constituting the maze shall be 5 cm high and 1.2 cm thick. Passageways between the walls shall be 16.8 cm wide. The outside wall shall enclose the entire maze.

2. The sides of the maze shall be white, and the top of the walls shall be red. The floor of the maze shall be made of wood and finished with a non-gloss black paint. The coating on the top and sides of the walls shall be selected to reflect infra-red light and the coating on the floor shall absorb it.

3. The start of the maze shall be located at one of the four corners. The starting square shall have walls on three sides. The starting square orientation shall be such that when the open wall is to the "north," outside maze walls are on the "west" and "south." At the center of the maze shall be a large opening which is composed of 4 unit squares. This central square shall be the destination. A red post, 20 cm high, and 2.5 cm on each side, may be placed at the center of the large destination square if requested by the handler.

4. Small square posts, each 1.2 cm x 1.2 cm x 5 cm high, at the four corners of each unit are called lattice points. The maze shall be constituted such that there is at least one wall touching each lattice point, except for the destination square.

5. The dimensions of the maze shall be accurate to within 5cm, whichever is less. Assembly joints on the maze floor shall not involve steps greater than 0.5 mm. The change of slope at an assembly joint shall not be greater than 4. Gaps between the walls of adjacent squares shall not be greater than 1 mm.

A.2 Mouse Specifications

1. MicroMouse shall be self contained. It shall not use an energy source employing a combustion process.

2. The length and width of a MicroMouse shall be restricted to a square region of 25 cm x 25 cm. The dimensions of a Micro Mouse which changes its geometry during a run shall never be greater than 25 cm x 25 cm. The height of a MicroMouse is unrestricted.

3. A MicroMouse shall not leave anything behind while negotiating the maze.

4. A MicroMouse shall not jump over, climb, scratch, damage, or destroy the walls of the maze.

A.3 Contest Rules

The basic function of a MicroMouse is to travel from the start square to the destination square. This is called a run. The time it takes is called the run time. Traveling from the destination square back to the start square is not considered a run. The total time from the first activation of the MicroMouse until the start of each run is also measured. This is called the maze time. If a mouse requires manual assistance at any time during the contest it is considered touched. By using these three parameters the scoring of the contest is designed to reward speed, efficiency of maze solving, and self-reliance of the MicroMouse.

1. The scoring of a MicroMouse shall be done by computing a handicapped time for each run. This shall be calculated by adding the time for each run to 30 of the maze time associated with that run and subtracting a 10 second bonus if the MicroMouse has not been touched yet1. 1 For example assume a MicroMouse, after being on the maze for 4 minutes without being touched, starts a run which takes 20 seconds; the run will have a handicapped time of: $20 + 4300 - 10 = 18$ seconds. The run with the fastest handicapped time for each MicroMouse shall be the official time of that MicroMouse.

2. Each contesting MicroMouse shall be subject to a time limit of 15 minutes on the maze. Within this time limit, the MicroMouse may make as many runs as possible.

3. When the MicroMouse reaches the maze center it may be manually lifted out and restarted or it may make its own way back to the start square. Manually lifting it out shall be considered touching the MicroMouse and will cause it to lose the 10 second bonus on all further runs.

4. The time for each run shall be measured from the moment the Micro Mouse leaves the start square until it enters the finish square. The total time on the maze shall be measured from the time the MicroMouse is first activated. The mouse does not have to move when it is first activated but it must be positioned in the start square ready to run.

5. The time taken to negotiate the maze shall be measured either manually by the contest officials or by infra-red sensors set at the start and destination. If infra-red sensors are used, the start sensor shall be positioned at the boundary between the start square and the next unit square. The destination sensor shall be placed at the entrance to the destination square. The infra-red beam of each sensor shall be horizontal and positioned approximately 1 cm above the floor.

6. The starting procedure of the MicroMouse shall not offer a choice of strategies to the handler.

7. Once the maze configuration for the contest is disclosed, the operator shall not feed the MicroMouse with any maze information.

8. The illumination, temperature, and humidity of the room in which the maze is located shall be those of an ambient environment. Requests to adjust the illumination will not be accepted.

9. If a MicroMouse appears to be malfunctioning, the handlers may ask the judges for permission to abandon the run and restart the MicroMouse at the beginning. A MicroMouse shall not be re-started merely because it has taken a wrong turn.

10. If a MicroMouse team elects to stop because of technical problems, the judges may, at their discretion, permit the team to run again later in the contest with a 3 minute maze time penalty. For example, assume a MicroMouse is stopped after 4 minutes; it must be restarted as if it had already run for 7 minutes, and will have only 8 more minutes to run.

11. If any part of a MicroMouse is replaced during its performance, such as batteries or EPROMS, or if any significant adjustment is made, the memory of the maze within the MicroMouse shall be erased before restarting. Slight adjustments, such as to the sensors may be allowed at the discretion of the judges, but operation of speed or strategy controls is expressly forbidden without a memory erasure.

12. No part of the MicroMouse (with the possible exception of batteries) shall be transferred to another MicroMouse. For example if one chassis is used with two alternative controllers, then they are the same MicroMouse and must perform within a single 15 minute allocation. The memory must be cleared with the change of a controller.

13. The contest officials shall reserve the right to stop a run, or disqualify a MicroMouse, if they believe its continued operation is endangering the condition of the maze.

B Lexer-generator Input

```
/* Pattern matching for 45 and 90 degree turns */

#ifndef PAT_C_INC
#define PAT_C_INC

/* The magical context parameter definitions */
#define CONTEXT_REQUIRED 7
#define CONTEXT_OFFSET   4

/* Set up yylex() prototypes & etc */

#define YY_DECL int context_match (char *yytext,
                                   long *cost,
                                   short *run_type,
                                   short *straightaway)

/* prototype */
YY_DECL;

#ifndef HEADERS_ONLY

/* local definitions */
#include "patcost45.h"  /* Pattern costs */

/* Default rule: assign straightaway costs */
#define YY_DEFAULT  { *cost += straight_cost(*run_type, *straightaway); }
%%
..nnen.         { *cost += turn_cost(PATH90, TURN45)/2; }
.nnen..         {
                  *cost += turn_cost(PATH90, TURN45)/2;
                  *straightaway = 0;
                  *run_type = PATH45;
                }
..nnee.         { *cost += turn_cost(PATH90, TURN90)/2; }
.nnee..         {
                  *cost += turn_cost(PATH90, TURN90)/2;
                  *straightaway = 0;
                }
```

```
..nnese         { *cost += turn_cost(PATH90, TURN135)/2; }
.nnese.         {
                  *cost += turn_cost(PATH90, TURN135)/2;
                  *straightaway = 0;
                  *run_type = PATH45;
                }
..nness         { *cost += turn_cost(PATH90, TURN180)/3; }
.nness.         { *cost += turn_cost(PATH90, TURN180)/3; }
nness..         {
                  *cost += turn_cost(PATH90, TURN180)/3;
                  *straightaway = 0;
                }
..nwnn.         { *cost += turn_cost(PATH45, TURN45)/2; }
.nwnn..         {
                  *cost += turn_cost(PATH45, TURN45)/2;
                  *straightaway = 0;
                  *run_type = PATH90;
                }
..nwnen         { *cost += turn_cost(PATH45, TURN90)/3; }
.nwnen.         { *cost += turn_cost(PATH45, TURN90)/3; }
nwnen..         {
                  *cost += turn_cost(PATH45, TURN90)/3;
                  *straightaway = 0;
                }
..nwnee         { *cost += turn_cost(PATH45, TURN135)/2; }
.nwnee.         {
                  *cost += turn_cost(PATH45, TURN135)/2;
                  *straightaway = 0;
                  *run_type = PATH90;
                }
%%

#endif /* !HEADERS_ONLY */

#endif /* !PAT_C_INC */
```

C Ground Effect Calculations

Analysis of Ground Effect Devices for a Micromouse.
C. Scott Ananian

Some unit and constant definitions:

$CFM \equiv ft^3 \cdot min^{-1}$ $gmf \equiv 0.001 \cdot kgf$ $in\_H20 \equiv 2.54 \cdot gmf \cdot cm^{-2}$

Density of Air: $\rho := 0.002378 \cdot slug \cdot ft^{-3}$ $\rho = 1.226 \cdot 10^{-3} \cdot gm \cdot cm^{-3}$ (At sea level)

Micromouse dimensions:

Mouse width: $w_{mouse} := 6 \cdot cm$ Mouse length: $l_{mouse} := 12 \cdot cm$

Mouse mass: $M_{mouse} := 200 \cdot gm$ (without fans)

Fan description:

Fan type is "Comair/Rotron FS12B3", Newark Electronics pg 1284, Stock no. 50F9151.
12VDC brushless DC fan, 60mm square x 25mm deep; 1.7W, 142 mA running current.

Specified flow rate: $Q_{spec} := 18 \cdot CFM$

Mass of fan: $M_{fan} := 3.9 \cdot oz$

These fans derate their flow rate almost linearly with static pressure.
We define Q as a linear function of p, with a slope of:

$k := 0.010 \cdot in\_H20 \cdot CFM^{-1}$

$Q_{fan}(p) := Q_{spec} - \frac{1}{k} \cdot p$

At some maximum pressure p, there is zero flow rate:

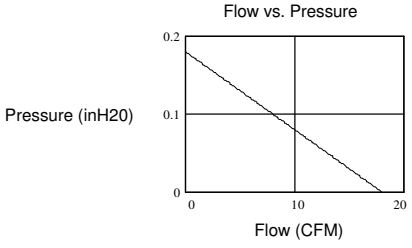
$p_{zero} := 1 \cdot gmf \cdot cm^{-2}$ Initial "guess" value.

$p_{zero} := root\left(Q_{fan}\left(p_{zero}\right), p_{zero}\right)$ Solve

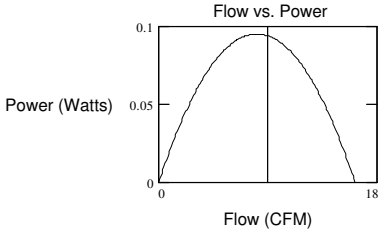
$p_{zero} = 0.18 \cdot in\_H20$ Zero flow-rate pressure.

We graph the relationship between flow rate and pressure:

$Q_{graph} := 0 \cdot CFM, .1 \cdot CFM .. Q_{spec}$ $x := 0 \cdot psi$



Since power is defined as the product of pressure and flow rate, there is the following relation of power to flow rate:



Air cushion system.

Now we define the parameters of the air-cushion system. The mouse is in the shape of an ellipse, and the cushion wall is the outside edge of the mouse.

Number of fans: $n := 1$

Cushion Perimeter: $l_{cu} := \pi \cdot \sqrt{\frac{1}{2} \cdot \left(w_{mouse}^{2} + l_{mouse}^{2} \right)}$ $l_{cu} = 29.804 \cdot cm$

Effective Cushion Area: $A_c := \pi \cdot w_{mouse} \cdot l_{mouse}$

Clearance height: $h_c := 1 \cdot mm$ Discharge coefficient: $D_c := 0.6$

The fans pump air out from underneath the mouse to form a low-pressure area, generating a downward force to aid traction. Under steady-state conditions for this negative ground-effect vehicle, the air being pumped out of the space beneath the mouse is just sufficient to counteract the air leaking in through the gap beneath the chamber skirt. The force generated is obviously:

$$F_{cu}(p_{cu}) := p_{cu} \cdot A_c$$

Assuming that the air inside the chamber is initially at rest, the velocity of air escaping under the periperal gap is given by:

$$V_c(p_{cu}) := \sqrt{\frac{2 \cdot p_{cu}}{\rho}}$$

The total volume flow of air into the chamber is thus:

$$Q_{gap}(p_{cu}) := h_c \cdot l_{cu} \cdot D_c \cdot V_c(p_{cu})$$

We solve for the steady state case where flow in equals flow out of the chamber:

$p_{cu} := p_{zero}$ Initial "guess" value.

Given

$$n \cdot Q_{fan}(p_{cu}) = h_c \cdot l_{cu} \cdot D_c \cdot \sqrt{\frac{2 \cdot p_{cu}}{\rho}}$$

$p_{cu} := \text{Find}(p_{cu})$

The steady-state pressure and corresponding fan flow rate is:

$p_{cu} = 0.15 \cdot in\_H20$ $Q_{fan}(p_{cu}) = 2.962 \cdot CFM$

For comparison, the maximum (zero-flow rate) pressure specified for the fan is:

$p_{zero} = 0.18 \cdot in\_H20$

Using the formula previously derived for generated force, we obtain:

$F_{cu} := p_{cu} \cdot A_c$

$F_{cu} = 86.396 \cdot gmf$ Force generated by air cushion system

The augmentation factor is a measure of the effectiveness of air cushion as lift generating device:

$K_a := \frac{A_c}{2 \cdot h_c \cdot l_{cu} \cdot D_c}$ $K_a = 63.246$

Now we analyze the weight/acceleration penalties of adding the ground effect device.

$M_{ge} := M_{mouse} + n \cdot M_{fan}$ Mouse mass with ground effect fans

$\frac{F_{cu}}{M_{ge} \cdot g} = 0.278$ Added acceleration (in g's) due to ground effect fan.

For a "typical" one cell run in a maze (no maximum velocity restrictions), we might observe the following time improvement:

$a_{before} := 0.5 \cdot g$ Best case for MITEE mouse III

$a_{after} := a_{before} + \frac{F_{cu}}{M_{ge}}$ $a_{after} = 0.778 \cdot g$

$t := 0 \cdot sec$

$t_{before} := 2 \cdot root\left(\frac{1}{2} \cdot a_{before} \cdot t^2 - 9 \cdot cm, t\right)$ $t_{before} = 0.383 \cdot sec$

$t_{after} := 2 \cdot root\left(\frac{1}{2} \cdot a_{after} \cdot t^2 - 9 \cdot cm, t\right)$ $t_{after} = 0.307 \cdot sec$

$\frac{t_{before} - t_{after}}{t_{before}} = 19.843 \cdot \%$

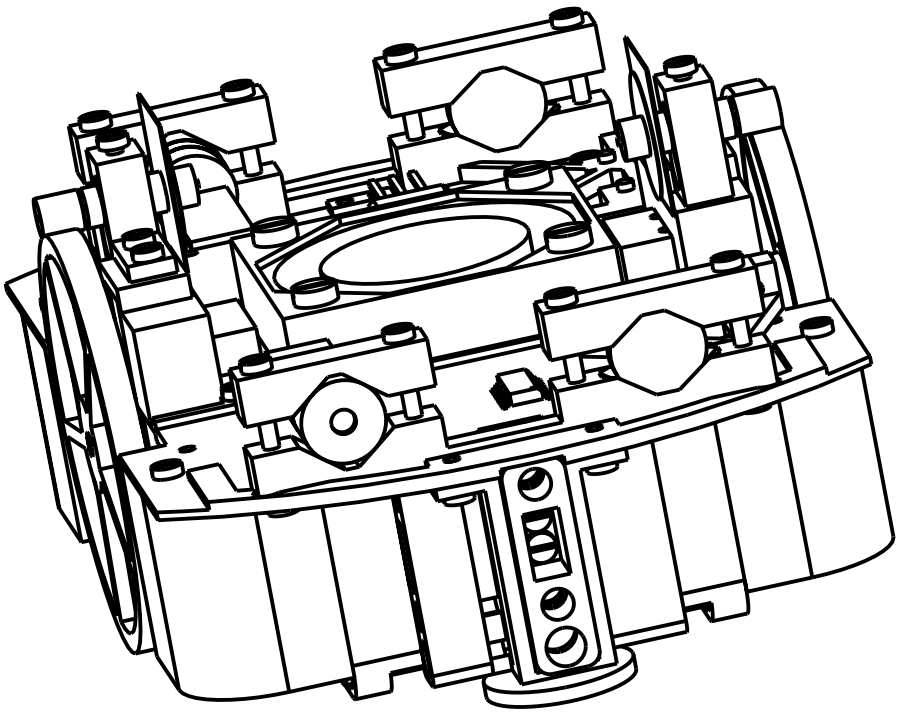
In conclusion, this analysis indicates an almost **20%** speed improvement using the ground effect devices.

References:

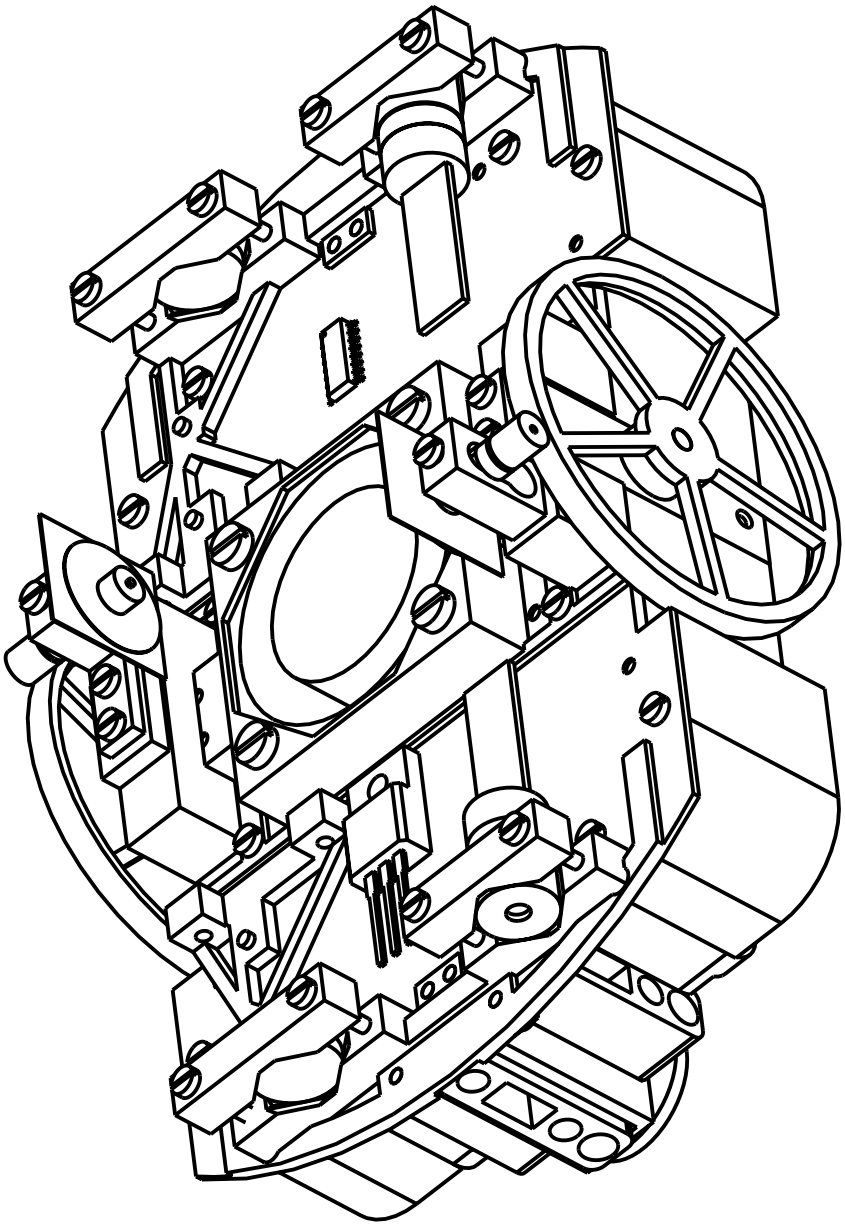
J. Y. Wong, "Introduction to Air-cushion vehicles," in Theory of Ground Vehicles.

D Mechanical Drawings

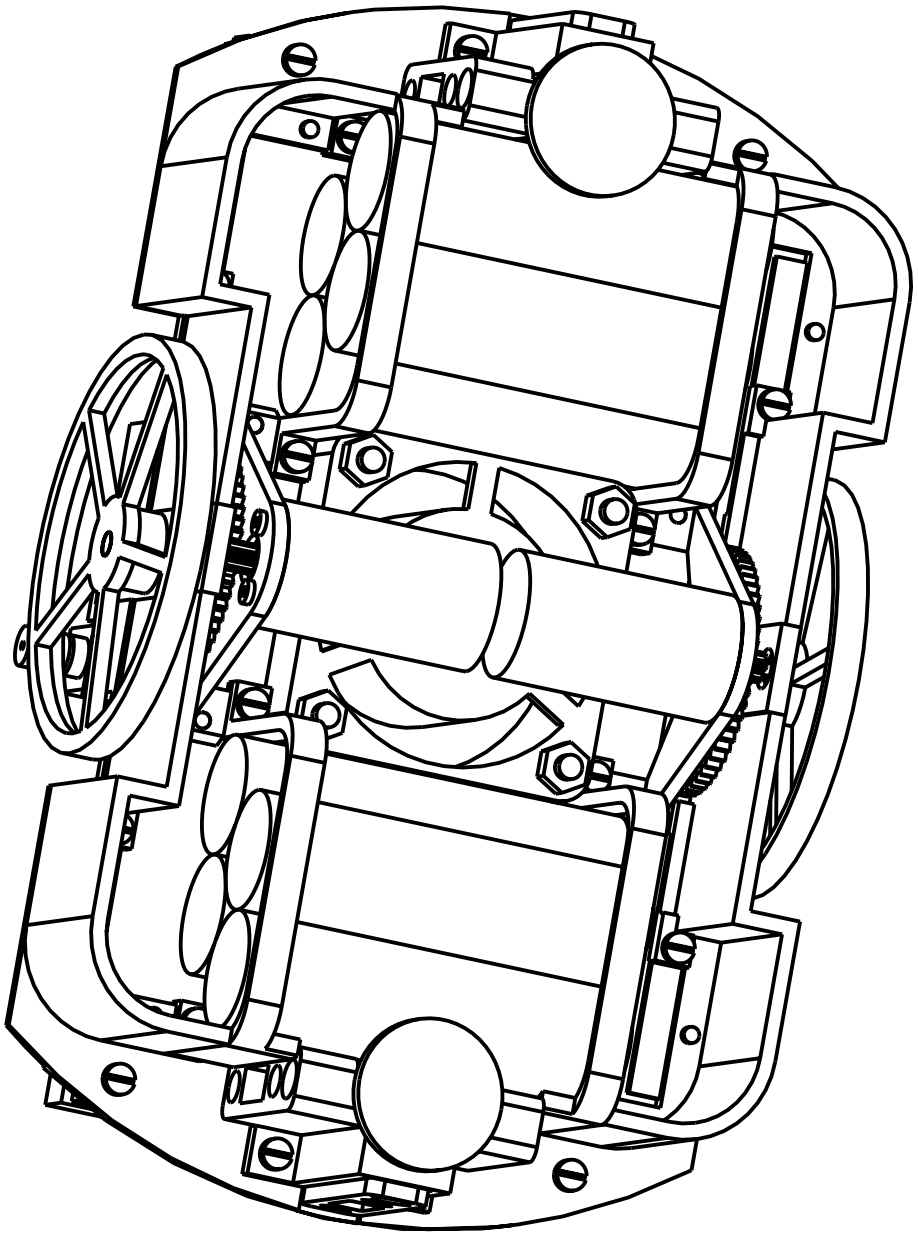
D.1 Mouse front view



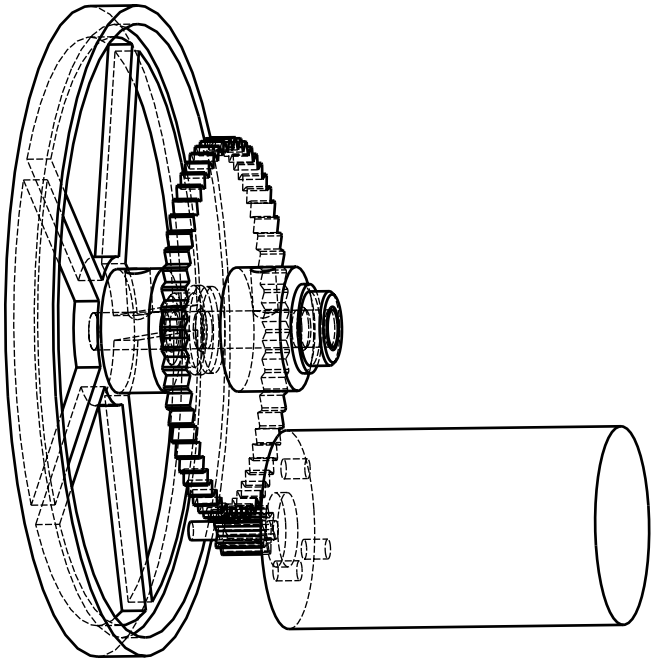
D.2 Mouse top view



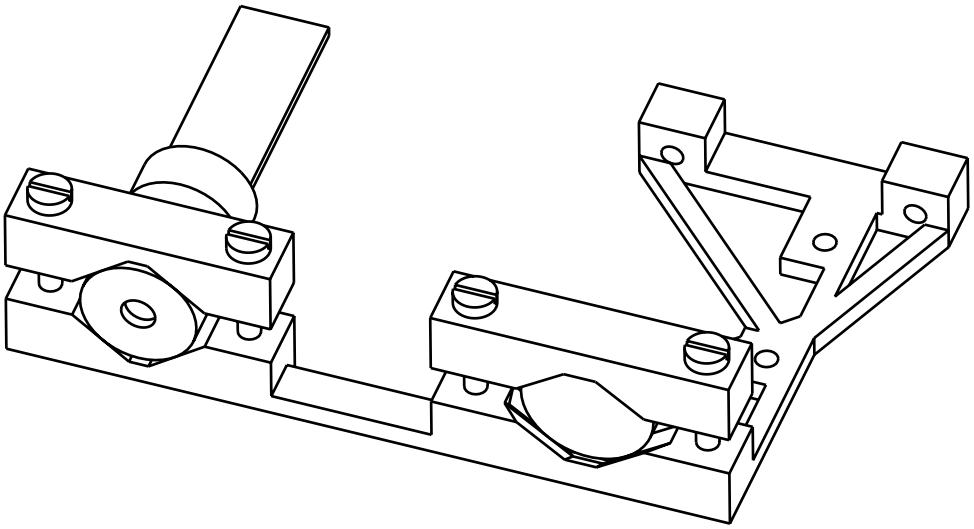
D.3 Mouse bottom view



D.4 Gear train

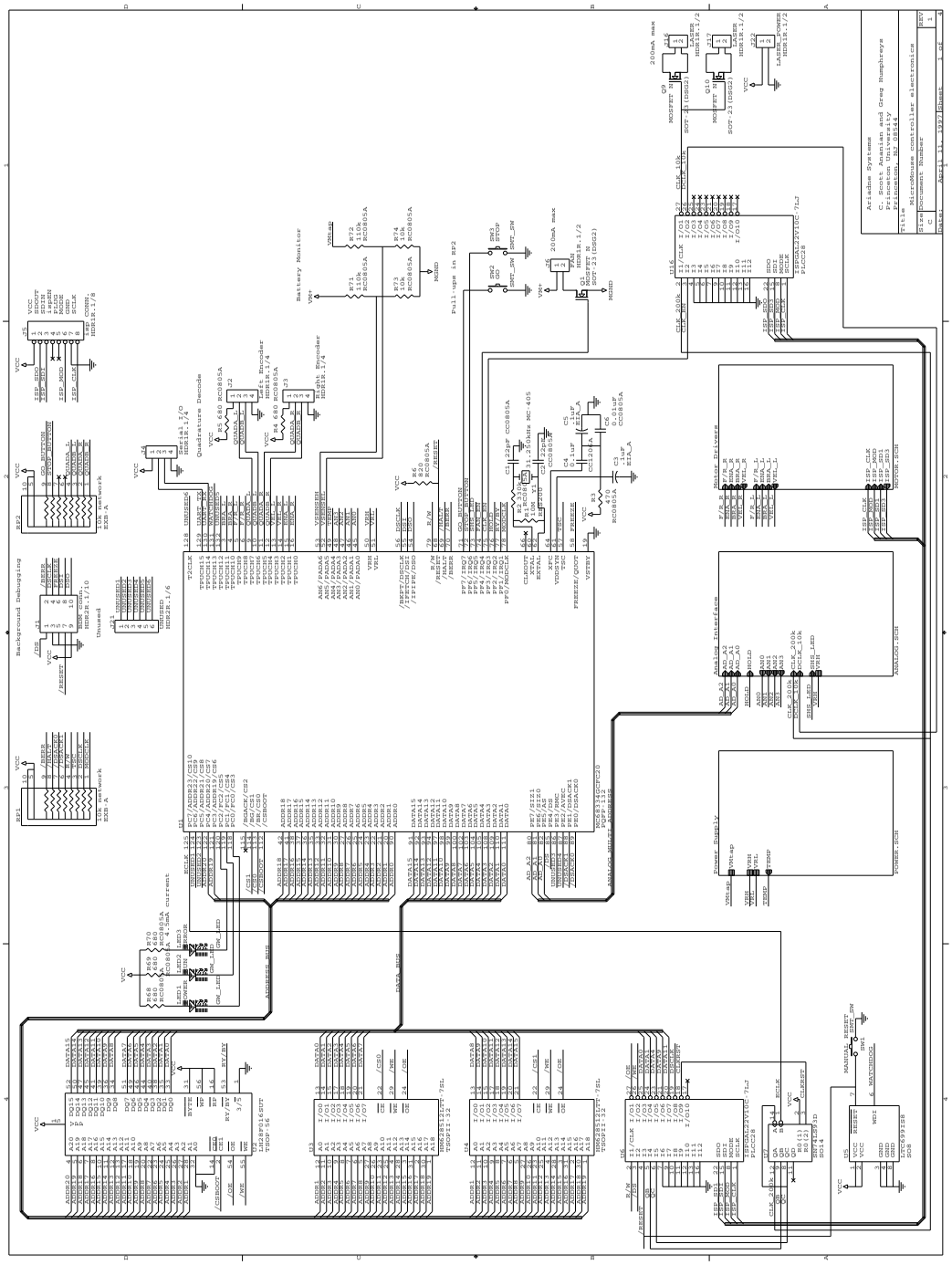


D.5 Long-range sensor mount

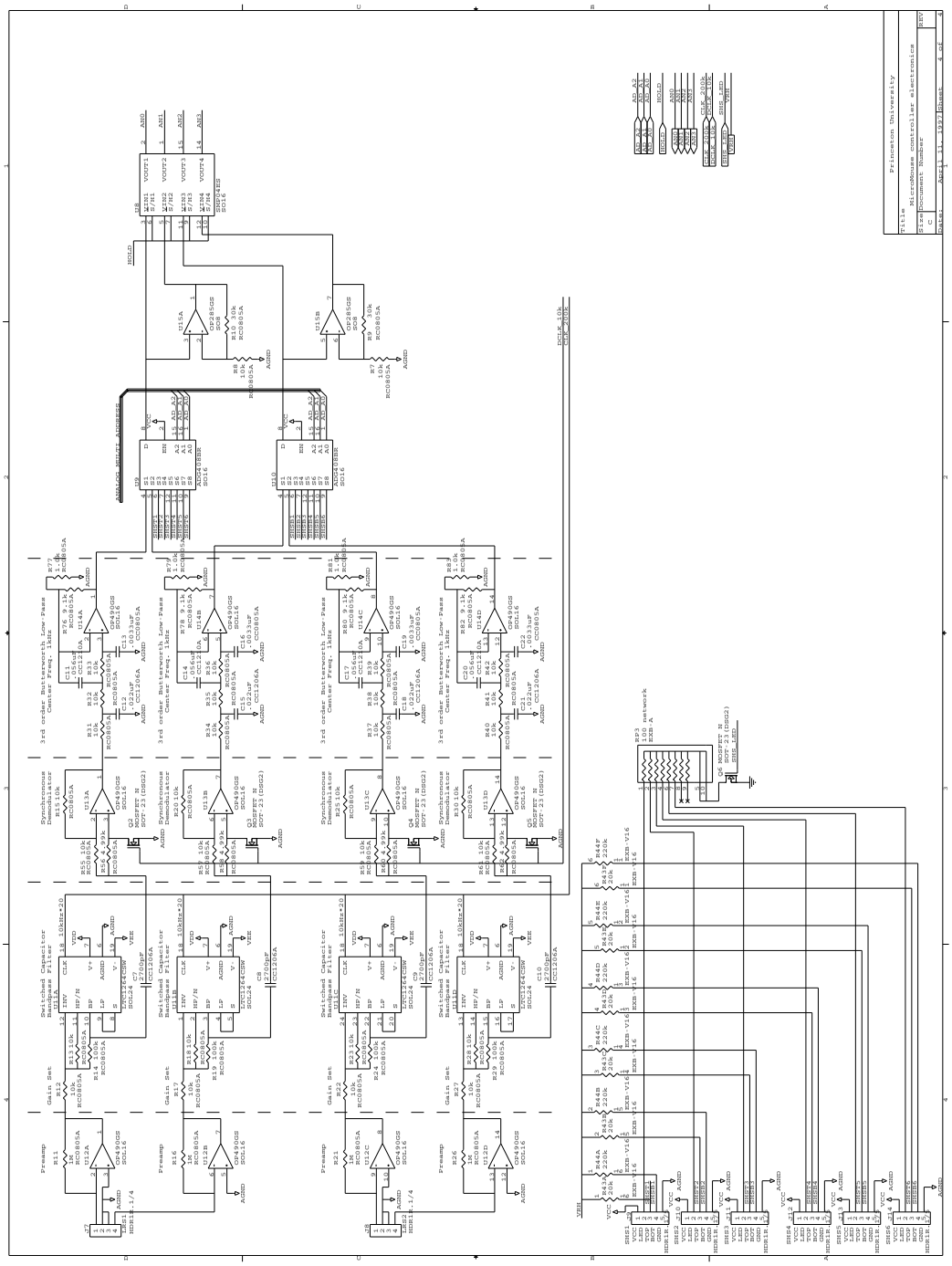


E Hardware Schematics

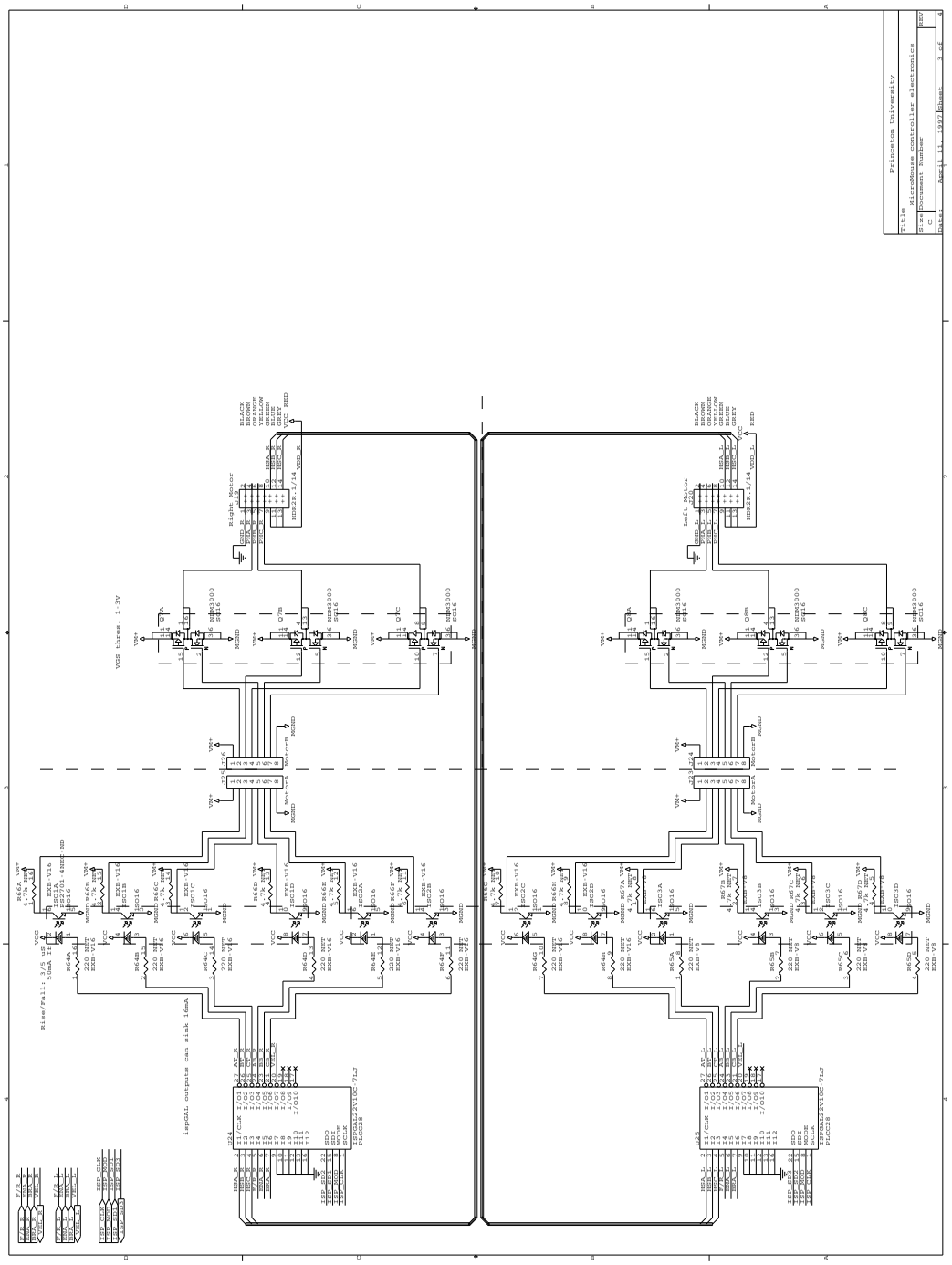
E.1 Processor interface



E.2 Analog electronics



E.3 Motor drivers



E.4 Power supply