# Independent Study: Mobile Robot Guidance System

Casey H. Clark
twist@anakin

Carson McDonald
carson@anakin

Brian L. Naylor
jones@anakin

Dr. Dwight Carpenter
May 21, 1996

# Table of Contents

## 1.0 Introduction

This manual is the culmination of three and a half weeks of work by Casey Clark, Carson McDonald, and Brian Naylor in robot guidance systems. With this manual, one should be able to replicate our research and gain a full understanding of what we have discovered along the way. Since we were the first group of people from Transylvania to work with this robot, a great deal of time was spent learning exactly how the robot worked. In the future, people will (hopefully) use this manual start where we left off.

As is the case with any type of extended project of this type, the robot that we were working with developed a personality and a nickname. To make him seem more "alive", we nicknamed him Hal after the conscious computer in the Stanley Kubrick movie, 2001: A Space Oddessy. We did this for no other reason than to have something to call him other than "the robot." We will be adopting this name throughout the rest of this manual.

Hal was developed by Real World Interfaces, Inc. He is comprised of two separate units that are inter-connected. First, there is a Mobile Base Unit (model # B-12). This unit allows him to move around. The B-12 has three wheels, rigid suspension, synchronous drive, a 40 pound carrying capacity, and batteries rated at 144 watts/hours of power. The second part is the Sonar Unit (model # G96.) This unit gathers information from the robot's surroundings with twelve individual transducers. These two units are controlled separately and communication takes place through two serial ports. Knowing this, it becomes obvious that in order to have the Hal respond to data received from the sonar unit, a computer with two serial ports is needed (one to communicate with the sonar unit and one to communicate with the base unit.)

In order to control Hal and tell him where to go and what to do, an IBM ThinkPad 365 was used. A laptop computer was the obvious choice because it would be small enough to sit on top of Hal when he is moving around. The laptop we used was a 486 with 8 MB of RAM and a 530 MB hard drive. Unfortunately, the ThinkPad only had one external serial port. To remedy this, a PCMCIA serial card was used to allow external access to the second serial port.

After choosing the hardware, the only decision left was what type of operating system we would use. With three of us working together the only obvious choice was a multitasking, mulit-user OS. That way all three of us could work on the same machine at the same time. We chose Linux (a freeware UNIX clone) because, in addition to the above points, it offers a natural programming environment and mature network support.

## 2.0 Communicating with Hal

## 2.1 Using Serial Communications Software

The first step in establishing communications between Hal and the laptop was to confirm that both serial ports on the laptop were operating correctly. We used a

freeware serial communications program called Minicom to test them. (This program is designed for use with a modem, but it works equally well when communicating with any serial device.) We were able to issue commands to the robot and receive its responses, and have Hal do simple things. This way we were able to learn what the various commands and their arguments did.

This method of communication with Hal proved useful during later research when something would go awry in a program we had written. When a program had to to be terminiated prematurely and the sonar device was left on, Minicom always was always available as a last resort for communicatin with Hal.

Note that not everything went smoothly at first- our first attempt to communicate with Hal through Minicom was unsuccessful. After re-reading the manual for Hal, we discovered that his serial hardware does not support hardware flow control. All we had to do was disable the hardware flow control in Minicom and things worked beautifully. In the future, when using communication programs to send commands to Hal, check the program's setup to ensure that all options are properly set.

## 2.2 Serial Communications Programming

After learning that both of the laptop's serial ports functioned properly, we were able to move on to the next phase of our research - writing programs that accessed the serial ports directly. This step was fairly easy to implement due to the fact that serial ports can treated as if they were files under Linux. The following segment of code opens a serial port and assigns it to the variable, port.

```
/* open port */
if ( (port = open("/dev/cua0", O_RDWR|O_NOCTTY)) == -1 ) {
  fprintf(stderr, "Error opening comm port!\n");
  exit(1);
}
```

Note that the file "/dev/cua0" refers to the first serial device on the computer. By opening this file, the first serial port is opened for reading and writing. The file "/dev/cua1" refers to the second serial device on the computer. Many books exist that give detailed information on UNIX and the programming conventions used in UNIX. One book that we found very useful was the Posix Programmers Guide - Writing Portable UNIX Programs by O'Reilly and Associates. This (and many other great books) are available in books stores everywhere.

```
/* write a character to the serial port */
write(port, 'c', 1);

/* read a character from the serial port */
read(port, &ch, 1);
```

Using these two built-in function of C (and a few others), we were able to write a function called toport() that writes a string to a specified comm port and implicitly return Hal's response (if any) to your command. In order for things to make sense to the serial port, we found that a bit of a delay should be included to allow the port to "stabilize" after receiving or sending a character. At first, some commands and return

strings were being truncated without explanation. Our simple remedy was to insert a *for* loop to waste a little time between sending or receiving a character.

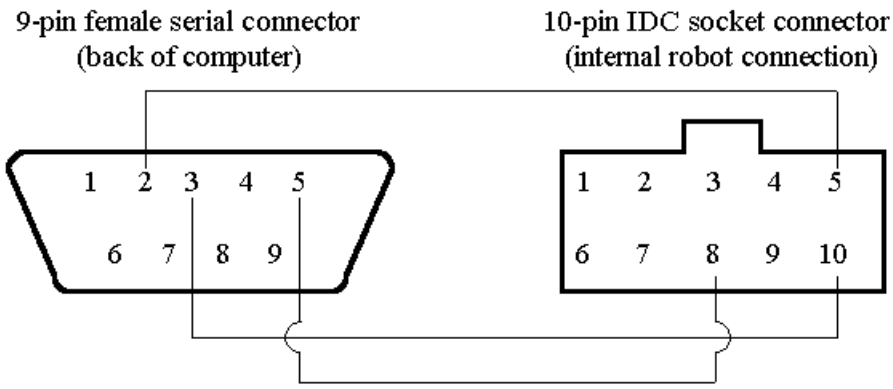
Hal's serial communication hardware contains a 4 KB buffer so that commands can be sent (or removed) from the buffer one character at a time. In order for Hal to actually execute the command stored in the buffer, a carriage return/line feed must be sent to the port. The fact that Hal needs a cr/lf instead of just a cr can be considered one of our small "oversights." We spent many hours wondering why the commands we wrote to the port were being ignored. This was due to the fact that we were writing a cr to the port unaware that a lf was also needed in order to actually execute the command.

## 2.3 Connecting the Computer

The next item of busines to take care of was connecting the computer in a way that would allow Hal to move around freely. The main area of concern was the serial connection to the mobile base unit of Hal. With the computer sitting on top of him, when Hal rotates, the external serial cable connected to the base unit would become extremely twisted and tangled in a short amount time. The manual vaguely referred to a method of connecting to the base unit through the top Hal but the details were VERY nebulous. It talked of a internal connection that was wired in parallel to the external base connection. Knowing that a way existed to make the connection to the base internally, we opened up the robot to see what we could find.

The first problem was the existance of two identical internal connections. Which one was the one we sought? The only thing to do was pull out a multi-meter and test continuity between the external connection and the two internal connectors. This testing yielded the following information:

| 9-pin female serial connector Pin# | Internal connection (10 pin IDC Socket Connector) |
|---|---|
| 1 - Not used | 1 - Not used |
| 2 - Serial Data Out | 2 - Not used |
| 3 - Serial Data In | 3 - Not used |
| 4 - Not used | 4 - Not used |
| 5 - Ground (-) | 5 - Serial data out |
| 6 - Not used | 6 - Not used |
| 7 - Not used | 7 - Not used |
| 8 - Not used | 8 - Ground (-) |
| 9 - Not used | 9 - Not used |
| | 10 - Serial Data In |




The proper internal connection was the one on the very front of the base controller board behind to sonar panels. We then made a serial cable to these specifications, routed it through and access hole in the top of Hal, and to our surprise, our make-shift cable worked perfectly.

Next, we needed to make something that would allow the laptop to sit on top of Hal without having to worry about it falling off when he was in motion. We adjourned to a nearby wood shop and fashioned a sturdy mounting device, which we then affixed it to the top of the robot. The dimensions of the wooden mount are about 12" x 8". This permitted us to place the laptop on Hal when we needed to test some mobililty functions.

## 2.4 Development of Higher-level Functions

After getting the basic serial communications details squared away, we needed to modularize our source code and define some standards to eliminate both the cutting and pasting we were having to do between programs and those hard-to-find errors that crept in whenever a particular step was mistakenly omitted. To this end, we organized our serial communications code snippets into easily callable functions and placed them in an include file. This was to be the model for later code development- controlling functions for both the base and sonar units, navigational functions, etc. For greater ease of use, we also applied a standard naming scheme to these functions.

## 2.5 misc.h

The functions contained in this include file are functions that are needed by more than one component of the robot or they simply do not belong anywhere else. There are also several important #defines declared in misc.h. Here is a list of the #defines and descriptions of the functions.

```
#define TRUE 1
#define FALSE 0
```

These two #defines are just to hold the two log file names.

```
#define SFILE "/tmp/event.log"
#define PFILE "/tmp/port.log"

FILE *porlog, *evnlog;
```
These two files pointers point to the log files.

```
int logging = FALSE;
int nooutput = FALSE;
```
These booleans are used to determine keep track of whether we are logging messages to a log file or the computer screen.

```
typedef char string[64];
```
This typedef allows for easy declaration of strings.

```
typedef struct {
  int xy[700][700];
  float xyf[700][700];
  int max;
  }floorplan;
```
This structure is used in the functions for the mapping Hal's surroundings.

*getcommands();*
```
int getcommands (char *filename, string commands[], int *numcommands);

Returns:
      Whether or not it successfully completed the command.

Arguments:
      char *filename - The filename for the file where the list of
      commands can be found.
      string commands[] - an array of strings that hold each of the
      individual commands read from the file.
      int *numcommands - this is where the number of commands read from
      the file is implicitly returned.

Description:
      This function allows the user to read a list of commands from a
      file.  This way you can send commands to the robot one at a time
      from the array of strings.
```

*dec2hex();*
```
int dec2hex (float decimal, char hex[]);

Returns:
      Success or failure of the function.

Arguments:
      float decimal - the decimal number that you want to the function
      to convert to hexadecimal.
      char hex[] - This array of characters is where the newly converted
      hex number is implicitly returned.

Description:
      This function was written to convert a decimal number to a hex
      number.  This way the user does not have to do it him/herself.

Example:
      char hexreturn[5];
```

```
      float decimal = 234;

      dec2hex (decimal, hexreturn);
```

*hex2dec();*

```
float hex2dec(string hex)

Returns:
      The decimal representation of the hexadecimal string that is
      passed to the function.

Arguments:
      string hex - A string that represents a hex number.  This string
is converted to a decimal number which is then explicitly returned.

Description:
      This function allows for easy conversion from the robot's encoder
      units to decimal numbers.

Example:
      float decimal;

      decimal = hex2dec("7DCA");
```

*stoplogging(); and startlogging();*

```
int stoplogging(void)
int startlogging(void)

Returns:
      The int is returned to tell if this procedure succeeded or not.
      These functions also open or close two file descriptors.

Arguments:
      Nothing

Description:
      These two functions are used to turn on and off logging to files.
      When logging is on, all commands, their messages, and any other
      critical information is written to an event log.  When logging is
      off, all of these messages are sent to the computer screen.

Example:
      startloggin();
      … stuff to log …
      stoploggin();
```

*intersection();*

```
void intersection(int ax, int ay, int bx, int by, int cx, int cy,
                  int dx, int dy, int *ix, int *iy);

Returns:
      (implicitly) int ix and int iy - intersection of the two lines
      passed.

Arguments:
      int ax, int ay - first vertex of first line
      int bx, int by - second vertex of first line
      int cx, int cy - first vertex of second line
      int dx, int dy - second vertex of second line

Description:
```

This function is used to find the intersection of the two lines passed and return that intersection's coordinates as two passed ints. If no intersection is found the 0 is passed back for ix and iy.

Example:
```
intersection(0, 0, 10, 10, 10, 0, 0, 10, &x, &y);
```

*bresenham();*
```
void bresenham(int xa, int ya, int xb, int yb, floorplan *area,
               int color);
```

Returns:
floorplan *area - an array of ints holding a graphic in the XPM format.

Arguments:
int xa, int ya - first vertex of the line
int xb, int yb - second vertex of the line
int color - The color to use to draw the line

Description:
This function draws a line in the floorplan *area using bresenham's line drawing algorithm.

Example:
```
bresenham(10, 10, 20, 20, &floor, 1);
```

*floodfill();*
```
void floodfill(int x, int y, int color, int newcolor, floorplan *area);
```

Returns:
floorplan *area - an array of ints holding a graphic in the XPM format.

Arguments:
int x, int y - start of the flood
int color - color to end at
int newcolor - color to fill with

Description:
This function uses a recursive floodfill algorithm to fill an area given to start in x,y with color newcolor and stopping at color.

Example:
```
floodfill(10, 10, 1, 4, &floor);
```

*makexpm();*
```
void makexpm(floorplan *arr, char *filename);
```

Returns: nothing

Arguments:
floorplan *arr - struct containing 2d mapping information
char *filename - the filename to which the data is written

Description:
This function writes the data contained in a 2d array (in floorplan) to an XPM graphics file. More info on this in section 5.3.1.

## 2.6 portcom.h

The include file, portcom.h, contains the basic serial communications functions, which are listed and described below.

*initport();*
```
int initport(int port)

Returns:
      1 if the functions completed properly.  0 if an error occurred.

Arguments:
      int port - This is an active serial port handle.

Description:
      This function sets up the specified port with the proper
      attributes for communicating with Hal.  First, the port attributes
      are set.  Next, the port's input and output speeds are set.
      Finally, the port is flushed and ready to take commands.

Example:
      initport(baseport);
```

*toport();*
```
int toport(string command, string retstr, int port)

Returns:
      1 if the functions completed properly.  0 if an error occurred.

Arguments:
      string command - The command that should be sent to the robot via
      the specified serial port.
      string retstring - This string contains the port's response to the
      command that was sent, if any.
      int port - The port handle that the command should be sent to.

Description:
      This function send a command to the robot in the form of a string
      through the serial port and implicitly returns the robot's
      response.

Example:
      toport("T> D3E1", response, baseport);
```

*portstatus();*
```
void portstatus(int port)

Returns:
      1 if the functions completed properly.  0 if an error occurred.

Arguments:
      int port - The port handle of the port you wish to see the status
      for.

Description:
      This function writes whether the port is open for reading and
      writing and input and output speed to the screen.

Example:
      portstatus(baseport);
```

## 3.0 Controlling Hal

After modularizing the serial communucations code so that we could call it without worrying over the details, we could then focus on more general things. Next, we needed a similar set of functions to allow easy control over the robot itself. For instance, we wanted to be able to specify that Hal rotate a certain number of degrees and move forward a certain number of centimeters, but the units used in the basic robot commands did not translate easily into human terms. These needs resulted in two more include files, one for the base unit and another for the sonar unit.

### 3.0.1 Converting hexadecimal to decimal and back

The first thing to address was the fact that Hal operated in hexadecimal, which is very convenient for computers, but not for ten-fingered humans. We wrote two functions ( hex2dec() and dec2hex() ) that convert a hexadecimal string to an integer and back, respectively. These functions are relatively simple, but necessary. They wound up in the include file, misc.h.

### 3.0.2 Conversion of encoder units to human-friendly units

After converting the base of the numbers, one has but to do a little math to coax them into human-friendly terms. The distances used by the base are "encoder units," which are determined by the optical sensors on the wheels and the rotation mechanism. Note that the rotational unit conversion listed in the manual is wrong- we determined the magnitude of this measurement by trial and error. In actuality, the hexadecimal number 400 translates into 360 degrees. This translates into about 2.83 encoder units per degree of rotation. Since these functions are used exclusivly by the base unit, they were placed in the basenav.h include file.

The sonar unit operates on an entirely different set of units, called "cycles." These are units of 3.2552 millisecond periods, measured from the initial firing of a sonar transducer until the echo's return. The formula,

```
distance = ((cycles * 3.2552)/1000000)*(speed of sound in air)
```

serves to translate sonar cycles into centimeters. All of these functions can be found in the sonar include file, sonarnav.h.

## 3.1 The Base Unit

### 3.1.1 Waiting to Complete One Command

With the some of the advanced interface functions written, it was clear that we needed to allow the robot to finish on command completely before executing the next command. The creators of the B-12 had thought of this and a set of wait commands existed to take care of just such a situation. There was the possiblity to wait until the robot's accelaration had stopped and a way to wait until the robot's rotation had stopped. Allowances were made in baseRotatefrom(), baseForward(), and baseBackward() functions for the user/programmer to specify the keyword "WAIT" or "NOWAIT" which tells the robot to whether or not it should wait until it is done with a specific command before continuing or to go on to the next command.

One problem we experienced was that sometimes the wait command was being ignored by the robot. Again, the problem was that the robot seemed to receive the commands too quickly and would not receive the command to wait until all motion has stopped to move on to the next command. The addition of a "dummy" *for* loop to take up a little time before sending the wait command cleared up the problem.

### 3.1.2 basenav.h

The high-level base-control functions are found in the include file, basenav.h. Below is a list of the functions found within and a short description of each.

*baseInit();*
```
int baseInit (void)

Returns:
      The integer that acts as a "handle" for the comm port that you
      are initializing.

Arguments:
      None

Description:
      This functions opens up communication with the mobile base unit on
      Hal.  The base unit communicates with com1 on the laptop.  It also
      flushes the port of all garbage and sets the baud rate to 9600.

Example:
      int baseport           /* serial port "handle" */

      baseport = baseInit();
```

*baseBatCheck();*
```
int baseBatCheck (int port)

Returns:
      1 if it successfully wrote the command to the port and the battery
      voltage was returned.  0 if anything along the line failed.
```

Arguments:
The port handle which is open to the base.

Description:
This function prints out the current battery voltage to the screen.

Example:
```
baseBatCheck(baseport);
```

*baseRotateFrom();*
```
int baseRotateFrom (int port, int degrees, int direction, int wait,
                    int speed, int accel)
```

Returns:
1 if the command was completed successfully. 0 if an error occurred.

Arguments:
int port - the port handle
int degrees - how many degrees the robot should rotate
int direction - pass COUNT to rotate counter-clockwise
pass CLOCK to rotate clockwise
int wait - pass WAIT to wait until the rotate command is finished before returning from the function. Pass NOWAIT to the function to send the command and return immediately.
int speed - Velocity of the robot's rotation. This value is in hexadecimal encoder units.
int accel - Accelaration of the robot's rotation. This value is in hexadecimal encoder units.

Description:
This function allows the user to pass how many degrees Hal should turn. This way the user does not have to worry about the conversion to rotational encoder units.

Example:
```
baseRotateFrom(baseport, 134, CLOCK, WAIT, 200, 150);
```

*baseForward();*
```
int baseForward (int port, float distance, int wait, int speed,
                 int accel)
```

Returns:
1 if the command was completed successfully. 0 if an error occurred.

Arguments:
int port - the port handle
int distance - distance in centimeters the robot should move forward.
int wait - pass WAIT to for the robot to wait until the translate command is finished before returning from the function. Pass NOWAIT to the function to send the command and return immediately.
int speed - Velocity of the robot's translation. This value is in hexadecimal encoder units.
int accel - Accelaration of the robot's translation. This value is in hexadecimal encoder units.

Description:

```
      With this function, the user can specify the number of centimeters
      Hal should move forward.  The function converts the centimeter to
      the proper hex encoder units and sends the commands to the base
      port.

Example:
      baseForward(baseport, 43, WAIT, 1700, 1200);


baseBackward();
int baseBackward (int port, float distance, int wait, int speed,
                  int accel)

Returns:
      1 if the command was completed successfully.  0 if an error
      occurred.

Arguments:
      int port - the port handle
      int distance - distance in centimeters the robot should move
                     backward.
      int wait - pass WAIT to for the robot to wait until the translate
                 command is finished before returning from the function.
                 Pass NOWAIT to the function to send the command and
                 return immediately.
      int speed - Velocity of the robot's translation.  This value is in
                  hexadecimal encoder units.
      int accel - Accelaration of the robot's translation.  This value
                  is in hexadecimal encoder units.

Description:
      With this function, the user can specify the number of centimeters
      Hal should move backward.  The function converts the centimeter to
      the proper hex encoder units and sends the commands to the base
      port.

Example:
      baseBackward(baseport, 43, WAIT, 1700, 1200);
```

## 3.2 The Sonar Unit

### 3.2.1 Transducer Location

The transducers are grouped in sets of three, on a "chain." As math would have it, twelve sensors result in four chains labeled 0, 1, 2, and 3. In order to fire a particular transducer, you must be able to designate which one you wish to fire. Each transducer in the chain has a specific number assigned to it. This is where the manual starts to get hazy. We had to do a little experimenting to figure out which number corresponded to which transducer.
Here are the results of our experimentation:

The 4-digit number passed to the robot command specifies which transducers of all four chains of transducers to fire. The least significant nibble refers to chain 0, then next nibble refers to chain 1, the next nibble chain 2, and the most significant nibble refers to chain 3, as illustrated below:

```
RT 3212
   ||||____Transducer 2 on Chain 0
   |||_____Transducer 1 on Chain 1
   ||______Transducer 2 on Chain 2
   |_______Transducer 3 on Chain 3
```

## 3.2.2 Interpreting Sonar Data

Now that we knew how to tell specific transducers to fire, how do we interpret the hexadecimal numbers that were returned to us? When a transducer from all four chains was fired, four hex numbers were returned on four separate lines, when three transducers were fired three hex numbers were returned, etc. Logically this makes sense. The question remained of which of number went with which transducer. From experience (the manual was of NO help at all) we surmised that the first number returned was from the highest numbered chain. For example, if the command that you see above was issue to the sonar port the following would be returned:

```
D4C1            Echo delay from Transducer 3 on Chain 3
3AC             Echo delay from Transducer 2 on Chain 2
6BFE            Echo delay from Transducer 1 on Chain 1
82AF            Echo delay from Transducer 2 on Chain 0
```

Armed with this knowledge, we were able to write functions that fired specific transducers and organized the return echos in a logical fashion.

## 3.3.3 Problems with the Sonar

We ran into several complications with the sonar unit as we used and tested it. First, no single transducer was facing directly to the front with relation to the base unit. Two transducers "split the front." This caused problems with getting a reading of what was directly in front of the robot. This problem also applied to getting reading from the sides and rear of the robot. In order to get a true reading of what was directly to any side, the front, or the back of the robot, Hal must stopped and rotated fifteen degrees. This presented a problem for us as we tried to take sonar readings while the robot was still in motion. We didn't really find an elegant way around it- we just stopped, rotated, read the sonars, rotated back, and continued.

Second, the sonars are not omniscent. The data they return can sometimes be unreliable, especially when there are spaces around it that exceed its range (about 40 feet.) When one of our programs led Hal into the hall, he promptly got lost and headed towards the wall. After extensive testing in the hallway environment, the only conclusion we could come to was that too many echos existed when the surrounding environment became too large. Caution should be exercized when Hal is operated in larger area as he is prone to wander towards walls and other obvious obstacles.

## 3.3.4 sonarnav.h

The include file sonarnav.h contains the functions necessary to communicate with the sonar unit on Hal at a higher level. The following is a description of the functions found in the file sonarnav.h.

*sonarInitSensors();*
int sonarInitSensors()

Returns: filehandle of the serial port (com2)

Arguments: none

Description: initializes serial port and sonar equipment for use

Example: sonarport = sonarInitSensors();

*sonarKillSensors();*
int sonarKillSensors(int port)

Returns: 0

Arguments: none

Description: Turns off sonar transducers and closes serial port. (You need to turn the sonars off if you're not using them constantly- they eat a lot of electricity and kill your battery quickly.)

Usage: sonarKillSensors();

*sonarReadSide();*
int sonarReadSide(float reports[12], int port, char side)

Returns: error flag (TRUE or FALSE)

Arguments:
float reports[12] - array of 12 real numbers. The distances from each transducer are returned here. They go in logical order, 0-1, 0-2, 0-3, 1-1, 1-2…
int port: file descriptor for the sonar serial port.
char side: a one-letter key indicating which side to read from.
'f': front
'r': right
'b': back
'l': left

Description: Reads two sensors from a particular side of the robot.

Usage: sonarReadSide(reports, sonarport, 'f');

*sonarReadAll();*
int sonarReadAll(float reports[12], int port)

Returns: 0

Arguments:

```
        float reports[12]: array of 12 real numbers.  The distances from
        each transducer are returned here.  They go in logical order, 0-1,
        0-2, 0-3, 1-1, 1-2…
        int port: file descriptor for the sonar serial port.

Description:  Reads all sensors, four at a time.

Usage:  sonarReadAll(reports, sonarport);
```

**Summary:**

Essentially, all you have to do to to take a reading from the sonars is do the following:

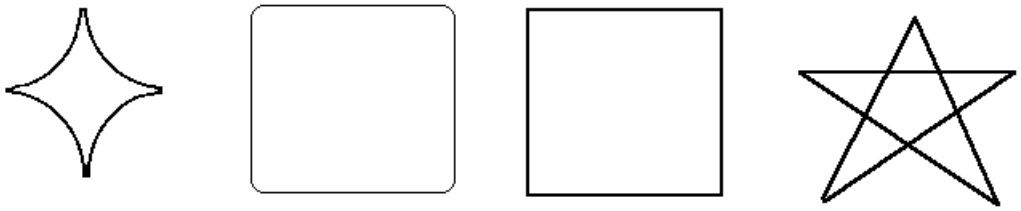
```
        float reports[12];
        int port = sonarInitSensors();
        sonarReadAll(reports, port);
        sonarKillSensors();
```

## 4.0 First Steps

We first used our user functions to write several simple programs that would test and "show off" what we had accomplished. These programs are fairly simple but there results were very satisfying. These first steps were very important because it offered us a great insight on problems that might occur later on in our research.

## 4.1 Executing a Simple Motion Script

Our first successful program was one that takes a file of Hal's low-level robot commands and sends them to the mobile base unit one at a time. This way, by changing which command file we sent to the program, we could change the motion that Hal would take. We developed command files that would make Hal move in the following ways:



Four-Point Star | Rounded Square | Square | Five-Point Star

The following is the command file that make Hal move in a square.

```
TA 1500
TV 1500
RA 150
RV 125
T> 3000
```

```
WE 01
R> FF
WE 81
T> 6000
WE 01
R> FF
WE 81
T> 6000
WE 01
R> FF
WE 81
T> 6000
WE 01
R> FF
WE 81
T> 3000
WE 01
```

The format of this file is very simple. One command should be placed on a single line. Calling the script program with the square file makes Hal move in the shape of a square on the floor.

script square

For help with the low-level robot commands please refer to the manual that comes with Hal. It does a pretty good job in this area explaining how to use the base commands.

## 4.2 Single-Minded Motivation Programs

The next step after making sure that the base movement worked without a hitch was to move to base unit based on what the sonar sensors gathered from the surroundings. This was a large step because up until this point we had explicitly told Hal where he was going to move and in what direction. Now, it depended on what Hal "thought!" He did not do any brilliant deducing - he only knew to do one thing in a program, but he was moving independently.

### 4.2.1 "Move Up to an Object"

The first program that wrote was for Hal to move, on his own, up to an object and stop. To do this a single transducer was fired and after detecting how far he could move forward, he would move that distance. For example, if you put Hal 120 cm from a wall, he would move forward about 100 cm and stop.

To get a true forward reading, Hal must be rotated 15 degrees clockwise. With one transducer facing forward, it is fired six times and the six distances are averaged in the hope to get a more accurate reading. The transducers sometimes return erroneous distances especially when the distance to the object is greater. With averaging, we hoped to be able to take care of some of the errors that naturally occur when using sonar devices. After taking the six readings, the base unit the rotates counter-clockwise

15 degrees and Hal then moves forward until he is just about up to the object that his sonar detected.

### 4.2.2 "Runaway"

This program makes for a very timid robot. It attempts to make Hal go as far away from everything as possible. In other words, he tries to reach the absolute center of the space around him, so that he's equidistant from everything.

The process undertaken is as follows: read all sensors and find the smallest value. Then, check the distance on the opposite side of the robot and go to the spot halfway between them. Take another reading and continue.

Unfortunately, our lab is pretty oddly-shaped and he never really stops moving.

### 4.2.2 "Navigate"

This program is very similar to the Runaway program except that it moves Hal the direction in which nothing was detected. When the program starts, all twelve transducers are fired six times and the distance detected by each transducer each time it is fired is averaged in and stored. After all twelve transducers are fired six times, Hal moves in the direction of the transducer that returned the largest average value. Hal then moves halfway to his destination and the process repeats itself to see if he is still moving in the direction that offers him the greatest distance. This program will keep going forever. There will always be one distance that is greater than the other distances and Hal will move in that direction.

The only problem we experienced with this program is when Hal moves out into a large area, say a hallway. In the hallway, the transducers do not return accurated readings since a lot of the distances are outside of the transducers' operating range. The positive thing is that Hal even found the hallway! When we started the program in our research lab, he moved in directions that appeared to be correct, but after a while he came to the doorway and when he saw that there was a big hallway with nothing to block his way, he proceeded out the door. The source code for all of these programs can be found in the notebook for your reference.

## 5.0 The Next Step

### 5.1 Mapping Hal's Surroundings - why and how

Next we turned our focus to the more advanced topic of mapping Hal's surroundings and displaying that as a picture of what Hal see around him. We discussed various methods of doing this, and what we chose evolved over time. Our approach was to map small sections of a large grid, one-at-a-time. The grid was a

large two-dimensional array of real numbers, which represented objects of varying levels of density. The density is computed using local grids that consist of integers representing where solid objects are and where the robot is at a given reading. The cells of the grid are one centimeter squared and the robot can map at total of 700 square centimeters, and at each local reading will map a 30 centimeter area around itself. These values are not special but were only picked to be easy and to start us out in the correct direction.

The compelling desire behind our intention to map Hal's surroundings was that we wanted to be able to intelligently plot a course from one point to another, and to be able to compare the actual path taken by the robot with its expected path (in real-time), correcting if necessary.

The only way to intelligently plot a course is to know what lies before you, prior to embarking. Similarly, in order to detect an error, to realize that you're not where you expected to be, you have to expect to be somewhere. What we did to accomplish this was keep a current position ourselves (we didn't want to use the robot's functions to keep track of positions because they didn't seem to work that well with the small amount of tests we did with them) and used that to keep track of the robot's position. This position is relative to 0,0 in the matrix which made it easier to implement the graphics routines that we used to draw the graphs. We also always started the robot out at the center of the matrix so that we had the possiblity to move in any direction from the start.

Once an area was mapped, we could conceivably save the data in a file, and the robot would no longer need to map that area again, but would be able to just read the file and go, checking the current surroundings against the map.

## 5.2 The Details

The density of a particular object is, at first, assumed to be 0.0, or completely empty. As the robot maps an area it uses a local area matrix first to exclude far off objects then uses this map to average into the total area map. As the robot returns to a previously-surveyed area, the new readings are super-imposed over the old and again the local area is averaged in. This allowed for some error in the readings and navigation, and if an object were to move, the set of cells it once occupied would eventually be averaged back down to a low number.

This brings us to another idea: density threshold. The robot considers a cell below a certain value (0.1) to be empty (and therefore traversable), so if an object were once there, but had since moved, successive averagings would eventually reduce the stored density value enough that the cell would be reclaimed.

The way we found the local area map at first was just to rotate the robots sonars so that we had the 0th sonar pointing in the forward possition and then just gather data from the four sides of the robot. This scheme worked decently to get us started out but it lacked any real detail and usually gave us bad results. To remedy this we decided to work on using all 12 of the sonar sensors to gather data. The way we accomplished this was to use a little bit of geometry and some simple graphics routines.

The robot is made in such a way as to make it easier to use the sonars that don't point straight but its not this simplest thing to do. Throughout the development of our

strategy to use the data from all 12 sensors we used figure 1. First we could use just the straight data from the four sides like we had before but we needed to bound them so that they didn't overlap into another sonar's field. This was easy because all we had to do was intersect lines from the center of the robot to the edge of our bounding box (which is 65 cm off center). After this we basically just needed to find the fields for the other 8 sonars. Since all of the quadrants are just 90 degree rotations, if we solve one then we just need to rotate 90 degrees to find the other 3. The first difference is that there will be a longer range for this part of the graph, since the bounding box will cut the triangle extending from the center at an unlevel angle. We want this angle to be even with the robot so we can make the assumption that the distance from the sonar is constant for the end of the field. We can start by splitting the corner into two parts by a 45 degree angle which gives us two right triangles. We can then move 15 degrees off of each axis and get our field for the two sonars. Then we can find an end to the fields so that one edge extends past the end of the bounding box and allows the triangle to have a flat base. If we look at this we find that the angle should be 30 degrees off of the bounding box. Now we just need to gather the points at which the fields intersect with the bounding box or where they end past the bounding box due to the extention we allow for.

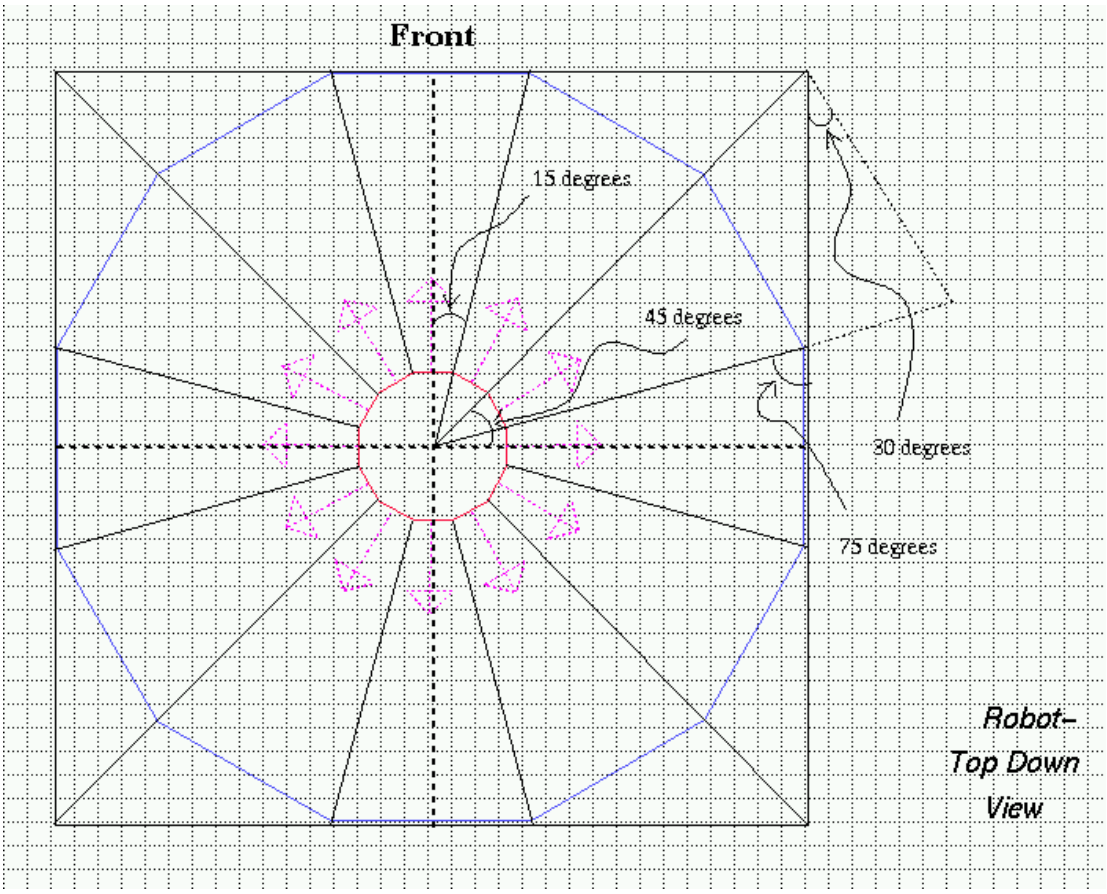



**Figure 1**

Now that we have our fields defined we can use the distances given by our sonars to fill them up. The way we do this is to use intersections based on the slope of

a line parellel with the given sonar's edge and that sonar's field edges. This gives us points to use to draw a section that possibly has something in it. So that does it for getting the local area data.

The way we handle the total area may be odd but it was something that we thought would be a good way to handle a few different problems. The first of these problems is the movement of objects after an initial reading. Along with this goes the error we have from small objects that may be caught in one reading but not in another. We thought that if we used an average to merge the local data into a larger field that this would solve both these problems and also give us a reliable graph given enough merges.

The scheme we used was very simple- all that happens is that the local data is added to the total data given the current robot position and then the whole local area in the total graph is divided by 2. This seems to work somewhat but there are some problems with it such as objects being averaged out too quickly. One solution to this would be to keep a count of how many times each section has been mapped and do an average based on that. From what we found using this though given enough runs through one section a graph might actually start to appear well defined although we didn't test for this.

Another problem we found while testing is that there is a large chance that the robot will give errors in its sonar data that it returns. When this happens it usually gives readings that are either way to close to it or it will return a far number for things that are close. This is taken care of somewhat in our scheme but it would take a few more scans and merges to figure out that this is an error. A possible solution would be to have the robot average a few readings for each scan.

## 5.3 Displaying this map graphically

### 5.3.1 The XPM Graphics Format

We chose to use the XPM graphics format to save our graphical map because of its simplicity and portability. XPMs are widely used in the X Window System and virtually any X graphics application will open it, as well as many Windows applications.

The XPM graphics file format is simply that of a flat ASCII text file, and its contents resemble the declaration of an array of characters in C. You begin with the size of the image in pixels, the number of colors contained in it, and the number of ASCII charcters used to represent one pixel. Then, you define a character's correspondence to an rgb color code, and finally, you simply represent the entire image in the defined ASCII characters. At the end of the file is more bookeeping info. Here's a sample:

```
/* XPM */
static char *test[] = {
/* width height num_colors chars_per_pixel */
"    5     5      4              1",
/* colors */
"0 c #ffffff",
```

```
"1 c #000000",
"2 c #0000ff",
"3 c #00ff00",
/* pixels */
"30303",
"01111",
"30001",
"02201",
"32201",
};
SIMPLE =                T
BITPIX =                8
NAXIS  =                2
NAXIS1 =                5
NAXIS2 =                5
HISTORY Written by makexpm 0.1b
END
```

## 5.3.2 Color-coding Different Objects

The next step in this process is to convert the grid of densities into a 2D array of characters to be output as the map. This was quite simple- we simply color-coded each density according to its value. There are 10 possible colors, so the graphical map essentially only uses the first significant digit of the object's density value. In any case, this is enough accuracy to get an idea of what is around the robot and how often the objects were encountered. This doesn't meen that the robot can only view in these densities though this is only used in the xpm representation of the matricies.

## 5.3.3 Graphics Algorithms

The graphics routines we used are standard ones that should be found in most graphics textbooks. These routines were easily adapted to what we wanted since we are using a matrix and they all gave integer values as positions due to the fact that the screen is basically a large 2d array. We used three routines heavily and they are an intersection routine, a line drawing routine (Bresenham's) and a floodfill routine. Since these routines are easily found I won't go into how to program them but will give a brief discussion about how we used them.

The intersection routine was used to find the points we needed find the section of the field of a sonar to block off as being occupied. The line drawing routine was used to draw the robot along with the polygons that resulted from the blocking off step. The floodfill routine was used to fill in the resulting polygons so that we had a section and not just an outline. The only other thing to add is that we did put in some clipping but it was very primitive due to the fact that some polygons may have been positioned outside of the matrix which could have caused problems, but these are basic if() statements and are not really clipping routines.

## 5.4 Using the Map

Unfortunately, we didn't have enough time to make this work properly. We devised a halfway successful scheme of interactively mapping an area, where the robot

tried to move to the nearest free block in order to map it. This was accomplished using a combination of a comparison between the various density values and the application of the distance formula (to find the nearest of the low cells.)

Other than that, we were unable to produce any working schemes to navigate an area using the map we'd created.

## 5.5 mapping.h

The routines in thie include file are used to generate the maps that we have described above. First are the #defines followed by the descriptions of the functions.

This define is used to define the value of the robot in the graph.

```
#define ROBOT 9
```

This define is used to convert radians into degrees.

```
#define RADS 0.01745329
```

*getlocalarea();*

```
void getlocalarea(floorplan *local, float *distances);

Returns:
      floorplan *local - a graphics structure containing a description
                         of the local area

Arguments:
      float *distances - an array of floats containing distances

Description:
      This function uses the array distances to figure out what local
      object are around the robot.

Example:
      sonarReadAll(port, distances);
      getlocalarea(&localarea, distances);
```

*mergearea();*

```
void mergearea(floorplan *local, floorplan *allareas, int x, int y);

Returns:
      floorplan *allareas - a graphics structure containing a
                            description of the total area

Arguments:
      floorplan *local - a graphics structure containing a view of the
                         local area
      int x, int y - current possition of robot

Description:
      Merges the local area with the total area and skips the robot
      given the values passed in x,y.

Example:
      mergearea(&local, &total, 350, 350);
```

*converttoxpm();*

```
void converttoxpm(floorplan *allareas, int x, int y);
```

```
Returns:
      floorplan *allareas - a graphics file containing a view of the
                            total viewed space

Arguments:
      int x, int y - current possition of the robot

Description:
      This function converts the values in allareas from floats into
      shades of color depending on how high or low they are. It then
      adds in the robot.

Example:
      converttoxpm(&local, 350, 350);
```

## 6.0 The Future

Obviously, any future work might attempt to pick up where we left off and come up with a method for plotting a path through an area intelligently. There are several additional considerations here- it should be able to plot an efficient and safe course through an array of obstacles, as well as testing to see whether it obeyed the course it had plotted while still in motion, hopefully correcting those mistakes as well.

That is an ambitious goal, and to add to it further, the next step might be some sort of object recognition (doorway, wall, etc.) Whether the sonars on the robot are precise and consistent enough to achieve this, we do not know.

There are many directions future research could conceivably take, and undoubtedly many of them we haven't even considered. We feel, however, that whomever takes up this challenge stands to save a great deal of time by building on what we've learned so far. We've hopefully built a strong base upon which more work can be based, allowing others to make advances without re-discovering things we've already solved.