PSUBOT Reference Manual: Hardware and Software

Kevin B. Stanton

Portland State University
Spring, 1993

ABSTRACT

The PSUBOT (pronounced *pea-es-you-bought*) is an autonomous wheelchair robot for persons with certain disabilities. It consists of an Everest & Jenning electric wheelchair equipped with a controlling computer, sonar, a compass, odometry feedback, voice recognition, and controlling software. It began as a senior design project in EE406 in the late 1980's and has been the topic of at least three M.S. degrees from the PSU EE department. In this document I describe the various hardware, software, and interface systems installed on the PSUBOT, their operation, a little history, and an occasional improvement or suggestion for future upgrades.

I. Introduction: What is the PSUBOT

The PSUBOT is a voice controlled, computer assisted robot for the disabled. It is intended to be used within known buildings, implying some procedure for entering maps of buildings (currently stored with line segments representing walls).

1. What you need to already know

To fully understand the internal operation of the PSUBOT, the reader should be familiar with the basics of electronics and programming. However, anyone with a sufficient technical background should be able to glean a significant amount from this document as well. It is a good example of integration of many technologies and could be used to demonstrate various design techniques and problems from a hardware, software, and systems approach.

1.1. Digital circuits and systems:

It is assumed that the reader is familiar with basic digital logic circuits such as found in the TTL 74LSXX series. Anyone needing instruction in this area is referred to an excellent text by Douglas V. Hall[Hall89a]. Also, the reader should be familiar with basic analog electronics as they apply to Digital to Analog converters and resistor voltage divider networks. The underequipped reader is again referred to a book by Douglas V. Hall; [Hall92a]. Finally, any further developments and/or modifications to the PSUBOT should be performed by someone with a vast knowledge of shielding and other electromagnetic interactions present on the PSUBOT (particularly near the PWM motor controller mounted to the back-rest of the wheelchair).

1.2. Object oriented design fundamentals/C++:

To understand the software described herein, the reader should have at least a basic understanding of the rationale of the Object Oriented paradigm as well as a familiarity with the C++ language itself. Dynamic memory allocation, inheritance, and virtual base functions are but a few of the facilities of the language exploited in this version of the PSUBOT software system.

Some of the conventions that I used were:
Function names start with lower case with no dashes, underlines etc.
The first part of subsequent words part of the name are in upper case.

EXAMPLE: `whereAreWe()`

User defined types (including classes) begin with a capital letter.
EXAMPLE: Boolean ok;

Class definitions start with ‘‘friend’’ functions, followed by ‘‘public’’ members (constructors first then destructor), followed by ‘‘private:’’ members.

Each header file begins with a commented description and then the rest is inclosed in some c-pre-processor commands which keep the file from being included multiple times in a single compilation. Usually these names are the class name, all in capital letters, followed by two underscores. The last line of the file is a `#endif`.

2. Common Hardware

We now turn our attention to some general purpose hardware used by several systems on the PSUBOT and how these devices are driven by software.

2.1. Input/Output boards:

There are two boards which were purchased for data acquisition and control. First, the digital input/output board from Keithly/Metrabyte (‘‘clones’’ are available from Computer Boards, Inc. at a lower cost). The board is called the DIO/PIO96 and consists of four 82C55 chips each with 24 bits of digital I/O programmable in banks of 8 bit bytes for a total of 96 bits of I/O. Refer to Intel’s ‘‘Peripheral Components’’ for full details of the 8255’s.

I will briefly describe the operation of the 8255 here as review. There are three ports on each 8255, labeled A, B, and C. Each of A and B can be programmed to be either 8 bits of input or 8 bits of output. Port C may be split into two four-bit sections each of which may be configured independently as input or output. It is also possible to independently set and reset bits of the C port.

The 8255 also has three operating modes: 0, 1, and 2. Mode 0 is what is used exclusively for the PSUBOT, and is called ‘‘basic input/output’’. Also available are mode 1 ‘‘strobed input/output’’ and mode 2 ‘‘bi-directional bus’’ where port C is used for handshaking. Interrupts may also be generated when certain conditions are satisfied, though no such facility is currently utilized.

Before sending data to or reading data from an 8255, the chip must first be initialized. Again, the data sheets describe this process. The resulting control-byte is an 8-bit quantity which describes the desired configuration of the chip. This byte is then sent to the Control Port of the 8255. Subsequent INP inputs and OUT outputs use the appropriate A, B, or C port address.

For example, the following is from a file called ports.h

```
#define DEVICE_ADDRESS 0x300

const int   ONE_A  =    DEVICE_ADDRESS+0;
const int   ONE_B  =    DEVICE_ADDRESS+1;
const int   ONE_C  =    DEVICE_ADDRESS+2;
const int   ONE_CTRL   =    DEVICE_ADDRESS+3;
```

Then if one desires to put the byte 0x8D on port A of the first 8255, the function call from ‘‘C’’ would be:

```
outportb(ONE_A, 0x8d);
```

The remaining 8255’s on the PIO96 board are number sequentially so that TWO_A is DEVICE_ADDRESS+4, and so on for the remaining three 8255’s. The base address of 0x300 is set by DIP switches on the board.

TABLE I

PIO96 Board Port Usage

8255 Chip #	Current Use
1	Feedback Board
2	Joystick Emulator Board
3	Sonar/Stepper Motor
4	Unused

PLEASE NOTE that the PIO96 board itself was modified to supply ±12V to the custom control board, so for J2 (chip 2), pins 29 and 31 of the connector are +12V and -12V respectively. This is documented more fully in the report by David Underwood, Spring 1992.

2.2. ANALOG INPUT BOARD.

The analog input board is used to acquire voltage signals from the flux-gate compass. The board is manufactured by Prarie Digital and is called their ‘‘Data Acquisition System’’ and consists of an 8255 as described above, a user configured 12-bit counter, and an 8-channel analog multiplexed A/D converter. It is this A/D converter that we utilized.

The voltage level of the analog input should be between 0 and +5 volts. The procedure for reading an analog voltage is somewhat complex, since data is read out serially and certain handshaking signals must be sent to initiate the A/D conversion. Following we show how an analog to digital conversion takes place, with actual code from the Compass class.

First, $\overline{CS}$ is pulsed low, and then the CLK is raised:

```
outportb(647,129);//CS given low pulse
outportb(646,192);

outportb(646,0);  //Raise the Clock input
outportb(646,64);
```

Next the chip must be cleared of residual bits (8 garbage bits must be clocked out).

```
for (int z = 0; z <= 7; z++) {
     outportb(646,0);  //Clock low
     outportb(646,64); //Clock high
}
```

and finally the data is read one bit at a time and scaled the the appropriate power of two (calculated beforehand and placed in the powerArray[]).

```
int R = 0;
for (int j = 7; j >= 0; j--) {
     for (int p = 0; p < delay; p++);
     unsigned char val = inportb(646);

     R += (val & 8 ) / 8 * powerArray[j];
     outportb(646,0);
     outportb(646,64);
}
```

and R is the 8-bit representation of the result. Note that a minimum delay is required between clock pulses as is true of any successive approximation conversion procedure.

2.3. PC Hardware

The PC now used for both development and operating of the PSUBOT is a VIP 80386-based PC with a 80387 math co-processor, a monochrome monitor, a 20 Megabyte hard drive and a double-density 5.25 inch floppy drive.

The system is currently running DR. DOS version 2. The compiler is BORLAND C++ Version 2.1, which is Kevin's personal version. The school owns version 2.0. It is not possible to install the entire suite of tools with the compiler on the computer due to its limited hard-drive space.

What is needed is a smaller, notebook, 486-based computer to act as the master controller. Communication could be performed through a parallel port to a board which would connect to the multiple peripherals such as sonar, motion control, etc. Another extension to the computing system is the addition of several dedicated controllers which perform real-time operations, such as controlling the wheels and acquiring data from the sonar. This would allow more high-level software to run on the main CPU as additional capabilities (such as path planning) are implemented.

II. Motor Control and Feedback:

1. Joystick Interface

To control the wheelchair by computer, the electro-mechanical interface (the joystick) needed to be replaced by an entirely electrical solution, one which could be interfaced to the computer. This task was solved by Paul Sherman with the help of Kevin Stanton. After several false starts, schematics for the existing electrical system on the wheelchair were obtained from Everest & Jennings. Then a Digital to Analog interface was built and tested by Paul.

1.1. Interface Requirements of Wheelchair

The following paragraph was taken from a report written by Paul Sherman in 1990:

> Wheelchair control characteristics: The wheelchair is an Everest & Jennings D3 wheelchair, whic is a joystick operated, 24vdc motor driven unit. The joystick consists of two 10Kohm potentiometers, one pot controlling each wheel. Since each side is identical in operation, all further references will be to one side of the wheelchair unless otherwise noted. A 6.8vdc reference is provided to the wiper of each pot. In the neutral position an equal voltage drop will occur across half of the pot to the end leads. The ends of the pot lead to the bases of two differential amplifier pairs, one set NPN and the other set PNP. Also at the base of the diff amps is a precision 4.87Kohm resistor. Assuming a negligable current (relatively) through the bases of the transistors, there is a voltage divider network which consists of a portion fo the potentiometer and each precision resistor. If the pot were allowed full travel the voltage swing at the inputs of the diff amps would range from 0 volts on one side and 6.8 volts on the other to the ocnverse, 6.8 volts on one side, 0 on the other. However, due to the physical restrictions of the joystick mechanisms, a maximum voltage of 3.6 to a minimum of 2.8 volts is allowed. The neutral volatege for the system is hence 4.87/9.87 * 6.8 volts, or about 3.20 vdc. Each pair of amps controls a direction, hence as the joystick is moved one direction, one set of amps will be biased on and the other oeft off. Each set of diff amps then drives two following stages each to provide ample power to drive the motors. The motors are driven in a pulse-width-modulcation (PWM) mode via a circuit provided in the wheelchair. For the purposes of this project it is sufficient to assume that this part of the system is working.

1.2. Circuit used to interface to the existing motor driver circuitry.

This section was also taken from a report written by Paul Sherman in 1990.

> Digital to Analog Conversion: To provide digital to analog conversion, the DAC0800 was chosen for its multiple current output capability. The DAC is configured for a maximum total current of 2.5 ma by tying IREF on pins 14 and 16 to _5vdc and ground respectively, thnrough 2Kohm resistors. The logic inputs are set to their respective values through a port of the PIO96 parallel interface expansion board. The currnet outputs are designed within the DAC such that for a 0 input, the IO output is 0ma and the $\overline{IO}$ output is maximum, 255/256*IREF. At 255 input (all ones) the converse applies. The current outputs are tied to voltage conversion circuits which are described below.
>
> Amplification: The amplification consists of current to voltage conversion and voltage range and value setting. To accomplish the conversion, each current output is tied to 1/2 of the LM358 dual op-amp via a 390ohm resistor. For feedback, a 390ohm reistor is also used, providing unity gain, and a voltage range of 390ohm * 2.5ma = 0.971 volts. This provides the range needed to simulate the joystick, plus a small cushion. The other input to each is tied to a voltage source provided by a 3.0 volt Zener diode and a volrate divider network to provide 2.6vdc minimum voltage output to the output at zero current at the DAC output. In this manner, a voltage range of 2.6 to 3.6 volts will be available at the output of each circuit. As with the original joystick, this circuit will provide 2.8 vdc at one output and 3.6vdc at the other for one extreme and the converse for the other extreme. In this manner the joystick is effectively simulated by the computer and is thus software controllable.

1.3. Software Device Driver fundamentals

A device-driver is used to command each wheel to rotate forward or reverse. The procedure is quite simple: one byte is sent to each port, with a 0x00 (0) meaning full reverse, 0xff (255) meaning full forward, and 0x80 (128) meaning stop. There is also a non-ideality inherent in the motor-controller (original equipment on the wheelchair) which defines a null-area about the center of the range (0x80) which does not immediately begin the motors turning. So, for example, any value sent to the controller board between 0x70 and 0x90 would stop the motor. This effect is counteracted in the driver software by essentially skipping the null-region.

1.4. Software Description

Following is the member-function of the Wheel object which initializes the 8255 ports necessary for both the motor control and feedback interfaces.

```
void Wheel::initialize() {

/* set up 8255 ports so that    port 1A -> output
                  port 1B -> output
                  port 1C -> input
*/

     outportb(ONE_CTRL,0x9A);    /* control word to port one */

/* set up 8255 ports so that    port 2A -> output
                  port 2B -> output
                  port 2C -> input */

     outportb(TWO_CTRL,0x89);    // Control word to port-two
     outportb(TWO_A,0x80);  // Stop the Wheel
     outportb(TWO_B,0x80);  // Stop the Wheel

     validInitialization = True;
}
```

Next the function which sends a byte to one of the wheel (left or right) ports (depending on which of the two Wheel objects is calling the function). Notice that the null-region is artificially shrunk to allow better linear response. The parameter passed to the function is absolute, meaning that 0 is stop, positive is forward, and negative is reverse.

```
void Wheel::commandToWheel(int command) {

#define forwardThreshold 160
#define reverseThreshold 90
#define nullWidthOverTwo 5

     command += 128;     // give the offset required by the D to A board.

// Remove the null width (except for the desired NullWidth defined above)

     if (command > 128)
          command += (forwardThreshold - 128) - nullWidthOverTwo;
     if (command < 128)
          command -= (128 - reverseThreshold) - nullWidthOverTwo;

// Make sure the values are valid.
     if (command > 255)
          command = 255;     // Clip control output to no larger than 128
     if (command < 0)
          command = 0;  //Clip control output to no smaller than -128

     outportb(Port,command);     // send values to the D/A converters
}
```

2. Feedback

As with any feedback control system, the PSUBOT has sensors to measure the controlled quantity (speed). This is accomplished by a pair of optical encoders connected to each wheel by a chain and sprocket. The encoders require +5V and ground and output two TTL-level square waves which are proportional to the angular rotation, and are 90° out of phase. From the gear ratios, encoder resolution, and tire size, we calculate the distance traveled for each cycle of the square waves. From their phases we can determine direction. ‘‘A’’ before ‘‘B’’ implies forward and ‘‘B’’ before ‘‘A’’ implies reverse as shown in Figure 1.

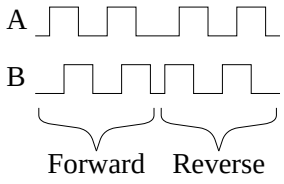

Figure 1. Wheel encoder outputs.

2.1. Shaft Encoders

The shaft-encoders are part number HP9000 from Hewlett Packard. Their outputs are each cleaned up with Schmitt Triggers (74LS14). The original circuit used the signals directly from the encoders but were found to be extremely sensitive to noise. Thus a small move of the wheel (two or three pulses worth) could sometimes result in a count of several hundred. Clearly the situation needed attention. One wheel exhibited worse characteristics than the other. I attributed this to the fact that the code-wheel was warped and slighly off-center. The ’14 cleared up the

problem nicely.

The pinout for the encoders is as follows: When looking at the encoder with the waffled square out, and with the pins sticking straight up, the sequence is [GND, NC, A, V_{CC}, B].

2.2. Feedback Operation

To understand the currently implemented feedback board, a little history of its development are necessary.

2.2.1. Initial Concept

Originally we expected that the feedback board would generate interrupts to the controlling CPU at which time it would read the values from one or both wheels. The question was raised: ''When should the interrupt be generated''. We decided that an interrupt should be generated after a particular distance was traveled, but to allow as much flexability as possible, we wanted to be able to select at runtime how often the encoder counters were serviced. The solution was to tie the high bits of the counter to a multiplexer, the output of which would generate the interrupt. Which bit triggered the event was then selectable by the computer. So, for example, if the robot were moving forward, frequent update of the position would be not as necessary as if the robot were doing some ''tight'' maneuver, such as rotating or turning a corner. In the latter case the bit selected to generate the interrupt would be a less-significant-bit than if the motion was smooth, regular, and less critical (at which time the CPU would then be able to spend more time doing other useful things).

Another idea was to provide one overflow bit so that if the count was greater than 255 the result would not roll over to zero. Thus we extended the counter to 9 bits, or 511 counts.

2.2.2. Shortcomings

First, there are no hardware interrupts generated by any circuitry currently added to the PSUBOT, so the ''interrupt'' output was instead tied to an input bit (bit four) of the 8255 port (ONE_C). This bit is currently not used by any software.

The overflow bit (implemented with a D-Flip-Flop) had a short in the wiring, so during the process of debugging, the device was simply disconnected. I justified this by noting that the computer should look at the counts more often than every 255 counts to ensure stable control.

2.2.3. Hardware Fixes

In addition to the above changes to the board, one other addition was made late in the project (early 1993). It was found that due to a non-ideal mounting of the encoder's code-wheel (foil circle with slots that the encoder detects) and perhaps warpage of the same, the outputs of the encoder was very noisy and prone to high-frequency jitter. This resulted in one instance of a frequency-counter counting up to several hundred counts when the wheelchair was moved bumped. Clearly the situation was a good candidate for hysterisis, which was added to each of the four outputs (two for each wheel) in the form of a 74LS14, Schmitt Trigger Inverter. The results after this modification was made were much superior.

2.3. Feedback Board Schematic

Following, in figure 2, is the schematic for the feedback board as used currently in the PSUBOT. It does not include the multiplexer or the additional D flip-flop as described above, though they are in fact on the board itself. For a diagram of the original intent, see the report by Paul Sherman.

2.4. Software Device Driver fundamentals

To read the distances traveled by the wheels, the latch signal first goes low. Then the two counts are read from the feedback board, and the latch signal goes back high. While the latch is low, the counters are told to load the value on their inputs. This value is binary 10000000, or

0x80. In this way, the actual number of counts may be obtained by subtracting 128 from the input value. If more than 128 counts are detected between reads from the computer, the result will be incorrect.

Also note that while the latch output is low, the counters are disabled, and any pulses occuring during that time will be lost. Therefore it is important that the duration of the low pulse on the latch output be as short as possible.

2.5. Software Description

Since the latch output latches and clears both the left and right counts, both must be read by whichever object (left Wheel or right Wheel) is asking for the distance traveled. The function `Wheel::distanceTraveled()` handles this by maintaining state information.

```
Distance Wheel::distanceTraveled() {

     Distance l,r;

     unsigned char lc, rc;

     // THIS SECTION MUST BE AS QUICKLY EXECUTED AS POSSIBLE
     /* Latch the data (& clear the counters) */
     outportb(ONE_C,0x00); /* Latch it */

     /* Read the data */

     lc = inportb(ONE_B);
     rc = inportb(ONE_A);

     outportb(ONE_C,0x01); /* Allow counting to continue */
     //END OF CRITICAL EXECUTION

     // the minus 128 is because zero travel yields an input of 128.
     l = (lc - 128) / PulsesPerFootLeft;
     r = (rc - 128) / PulsesPerFootRight;

// If the distance variable hasn't been read (used) yet, add it to the value.
// Otherwise move it there.

     if (newLeftDist) {//hasn't been read yet, just accumulate
          leftDist += l;
     }
     else {
          leftDist = l;
     }

     if (newRightDist) {
          rightDist += r;
     }
     else {
          rightDist = r;
     }

//Determine which side called this function, and mark status appropriately.
     if (side == LEFT) {
          newLeftDist = False;
          newRightDist = True;
          return leftDist;
     }
     else if (side == RIGHT) {
          newLeftDist = True;
          newRightDist = False;
          return rightDist;
     }
```

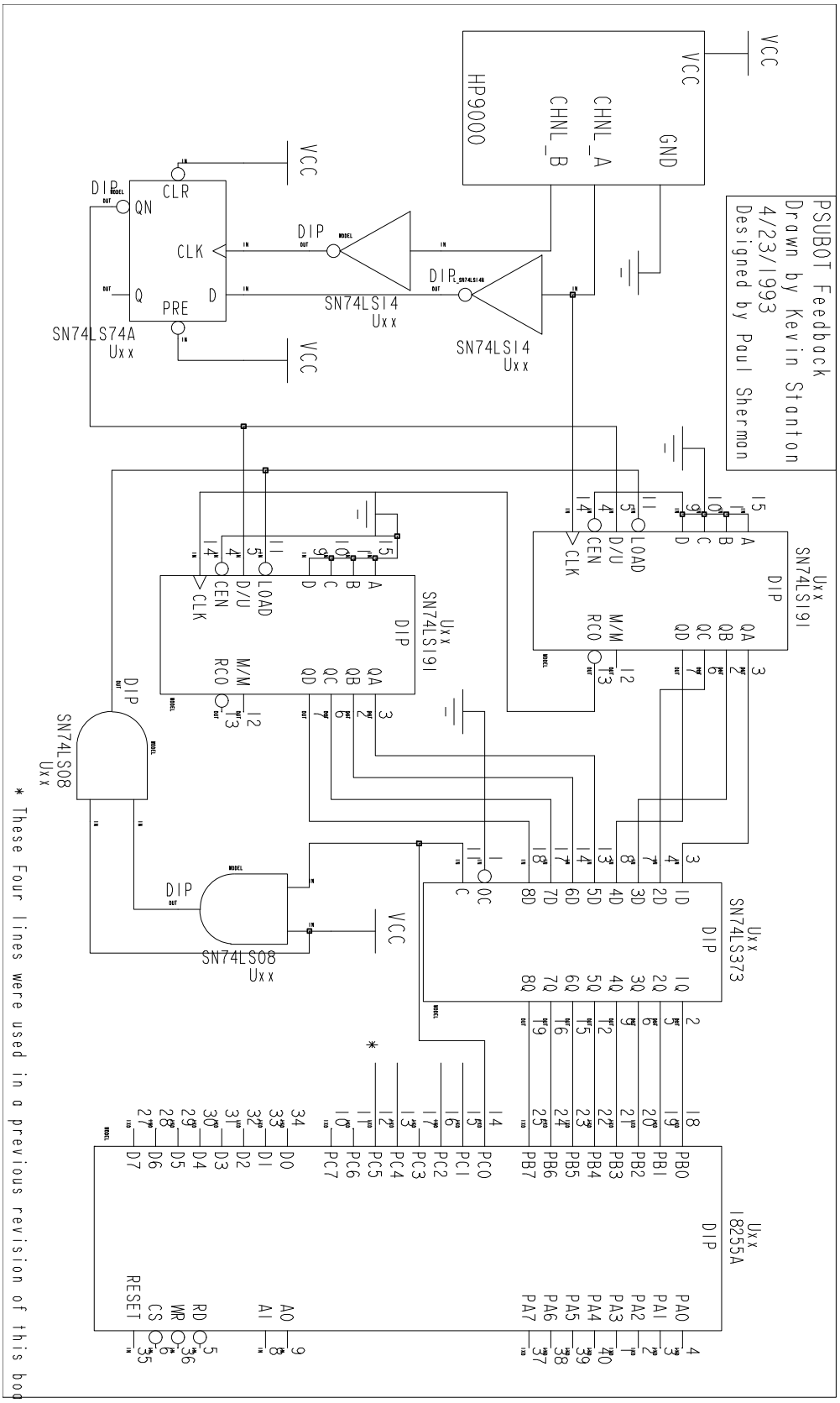

Figure 2. Feedback board schematic

```
        return 0;
}
```

3. Higher Level Software Modules

Figure 3. Wheelchair Motion Object Hierarchy with Inheritance

3.1. PID Controller Object

A PID controller stands for Proportional + Integral + Derivative control algorithm, and indicates how the error (difference between the desired and actual quantity (speed)) are combined to generate the command output). The core of the PID controller was developed by Karl Radestam as a 406 project. The code was written in C and I generated a C++ class with it. It has two forms; velocity and positional. The velocity form (currently the one used) was modified slightly by myself so that the proportional term didn't go away after the first iteration (as would have been the case with the straight PID algorithm). The interface to the object is very straightforward. Upon construction it sets up the arrays used in the calculations, and reads the constants (KP, KI and KD) from the Parameters object. Of course the destructor frees the memory taken up by the arrays, and the two remaining member functions do the actual PID calculation (`work()`), and reset the state of the PID if a new context is desired.

```
class PID {

public:
        PID();
        ~PID();

        float   work(Speed &desired, Speed &current, float& T);
        void    reset();    //Clear the state of the PID

private:

        int     length; //This must be before the K[PID] declarations!!!
        float   KP;
        float   KI;
        float   KD;

        float   *e;     //Error array
        float   *m;     //Command array
};
```

Notice that the order of declaration of private members is critical, since the length of the arrays must be defined before the arrays themselves are defined. The order in which the members are initialized in the constructor is of no consequence; the order of declaration ultimately determines the order of initialization.

3.2. Wheel Object

The Wheel object is responsible for operating a wheel at a particular speed, and maintain a record of what distance the outer surface of the wheel has travled. There are command functions, such as setSpeed, motorOff, there are status functions such as currentSpeed, distanceTraveled, and finally the `work()` function actually polls the object to allow it to do all the controlling and status-update operations.

```
enum SIDE { LEFT, RIGHT };
```

```
class Wheel {

public:

        Wheel(SIDE s);
        ~Wheel();

        void    work();
        void    setSpeed(Speed s);
        Distance travel() {return cumulativeDistance; }
        void    resetTravel();
        Speed   speed() {return currentSpeed; }
        void    motorOff();
        void    markTime();     //Don't control, just read distances.

private:
     •
     •
     •
```

3.3. PathSegment Base-Class (contains two Wheel objects)

The PathSegment class is never instantiated, but rather is the base class of all objects which represent a robot motion, such as Forward, Rotate, Stop, and all extensions which will undoubtedly appear later (Arc, Sideways, etc.). The goal of this class was to encapsulate those data and function members which would be common to all such higher-level classes. The most basic members of the PathSegment class are the left and right Wheel classes. Clearly each of the motions just mentioned require the control of both wheels. Also included is all functions necessary to calculate and maintain the current Posture of the robot based on odometry (distance readings from each wheel).

This class also uses a feature of the C++ language which no other class in the system uses: virtual functions. In this case, the functions are *Pure Virtual Functions* (as can be seen by the ''= 0'' at the end of their declaration) and therefore the class PathSegment is an *Abstract Class*. But this is what we want, since it would not make sense to tell the PathSegment directly to `work()`, since `work()` has no meaning until it is inherited (and thereby defined) in the derived class (Forward, for example). A benefit is that all classes which make PathSegment a base-class, must define a minimum interface which is expected by the Motion class or a compile-time error will result. Thus we can guarantee that the function-call interface for each derived class will be consistant.

```
class PathSegment {

public:

        PathSegment(Posture loc);

virtual void    work() = 0;
virtual void    pause() = 0;
virtual void    continueNow() = 0;
virtual Angle   shieldDirection() = 0;   //Which direction to watch for collisions

        Boolean done()  {return finished;}
        void    setPosture(Posture p)   {current = p;}
        Posture currentPosture()        {return current;}
        void    calculatePosture(Distance l, Distance r);
        void    reset();        // Turn the motors off

protected:    //Derived functions may call these directly, private
         // to everyone else.

        Angle   deltaTheta;
```

```
        Boolean finished, pausing;
        Posture current;
        Time    pauseStart;

virtual void    setSpeeds() = 0;
        Wheel   left, right;
};
```

3.4. Forward, Rotate, Stop objects (inherit PathSegment Object)

Once the abstract base class (PathSegment) had been defined, the derived classes must simply define custom functions to be used when the pure-virtual functions of the base class are called. These include: `work()`, `pause()`, `continueNow()`, `shieldDirection()`, and `setSpeeds()`.

Forward:

The Forward object simply assigns the default maximum-speed to both the left and right wheels as their desired speeds. It returns 0° as the `shieldDirection()` (though 180° would be appropriate for negative speeds (reverse)) and the `work()` function defines when the motion is completed. It would probably be a good idea to include all common commands of the `work()` function of all the derived classes of PathSegment into a common function, which then would call a pure virtual function to detect the termination condition.

```
class Forward: public PathSegment {

public:

        Forward(Distance dist, Posture p);

        void    pause();
        void    continueNow();
        void    work();
        Angle   shieldDirection();

private:

        Speed   ForwardSpeed;   //Desired wheel speeds
        Distance desiredDistance, totalTravel;
        void    setSpeeds();
};
```

Rotate:

The Rotate object returns the correct direction ($\pm 180^\circ$) for the `shieldDirection()` function, and accumulates the total rotation amount to determine when the correct angle is achieved.

```
class Rotate: public PathSegment {

public:

        Rotate(Angle ang, Posture p);

        void    pause();
        void    continueNow();
        void    work();
        Angle   shieldDirection();      //Which way to look out

private:

        Angle   desiredRotation;
```

```
        Angle   initialHeading;
        Angle   rotatedSoFar;
        int     CW;     //This is +-1
        void    setSpeeds();
        Speed   RotateSpeed;//Default maximum rotation speed
};
```

Stop: The Stop object simply maintains the odometry-based position of the robot, and tries to keep both wheels stopped. It also has a termination condition; that of the wheels speeds both being lower than some threshold.

```
class Stop: public PathSegment {

public:

        Stop(Posture loc);

        void    pause();
        void    continueNow();
        void    work();
        Angle   shieldDirection();

private:

        void    setSpeeds();
        Speed   StopSpeed;      //Speed at which we are assumed to have stopped.
};
```

3.5. Motion Object (executes a Forward, Rotate, or Stop)

The Motion object is nothing more than a place to hold a pointer to a PathSegment (which is actually one of the derived classes, but pointers to an objects' base-class are allowed) and functions which pass function-calls to the current motion type. In other words, it provides a convenient interface between the higher-level classes and the low-level motion-types. Thus a higher-level function can tell the motion object that it wants a Forward object to be the current executing pathsegment, and then another higher-level function tell the Motion object to `work()`, at which time it would execute the current segment (which would be a Forward object).

```
class Motion {

public:
        Motion();
        ~Motion();

        Boolean done();

// here are the motions available
        void    forward(Distance dist);
        void    rotate(Angle ang);
        void    stop();

        Angle   shieldDirection();      //Watch for collisions here (theta)
        void    work();
        void    pause();
        void    continueNow();

        Posture currentPosture();
        void    setPosture(Posture& pos);
        void    reset();                // Turn the motors off.

private:
        PathSegment *currentSegment; //The segment currently being executed
```

```
};
```

So, for example, the definition of the member function `::forward(Distance d)` is as follows:

```
void Motion::forward(Distance d) {
        Posture p = currentPosture();   //Find out where we are first
        delete currentSegment;
        currentSegment = new Forward(d,p);      // Give it the current posture
        if (!currentSegment) {
                noMem();
        }
}
```

and a call to `work()` would be simply executing the following one-liner:

```
void Motion::work() {
        currentSegment -> work();
}
```

The other members of this class are very similar. Notice that as each segment is constructed when desired, it is passed the odometry-based, believed posture. Graphically, each segment is given its starting point (Posture), and yields its endpoint (Posture) just before its destructor is called.

III. Sonar Sytem:

1.

Sonar Device

Concepts of operation
Brief overview of hardware (commercially available)
Digital Signal Interface
Software Device Driver

2. Sonar Rotation Device

Since we wanted to be able to look in more than just one direction with the sonar, the ultra-sonic trasnducer was connected (by a piece of toilet-bowl tubing, I might add) to a stepper-motor mounted with the shaft oriented vertically.

2.1. Source for circuit, driver chip

The cheapest stepper motor/driver I found was from a company called Mavin P. Jones Inc. (XXXXX address and phone number) and was sold in the form of a kit. Well, when the kit came, I discovered that the circuit was not very useful. For one, the user had to push a button to make it step each time, or could hook the step-input to a 555 astable timer and then adjust the frequency by turning a potentiometer. Also, the direction was controlled by a switch. What we needed was all these things controlled by logic-level signals from the computer.

Dispite little experience with signal-level conversions, I managed to get the circuit to work by using simple emitter follower (?????) circuits to ground the input when the logic level was high. This worked, except that after more thorough testing I discovered that between times when the line was pulled low by the transistor, the line bascially floated and picked up lots of stray noise (especially when the PWM drive motors were operating), resulting in erratic stepping of the motor. This was solved by putting a pull-up resistor on the line, something that should have been part of the initial circuit.

2.2. Hardware Description (Circuit with modifications)

Basically, the circuit is simply a XXXXYYYY stepper-motor controller connected to +12V. Level tranlsation (as described above) allows control of the chip with logic-level signals. Having learned much and seen numerous power-conversion circuits since the circuit was built, I expect that another chip with direct logic inputs would be a good choice as a replacement.

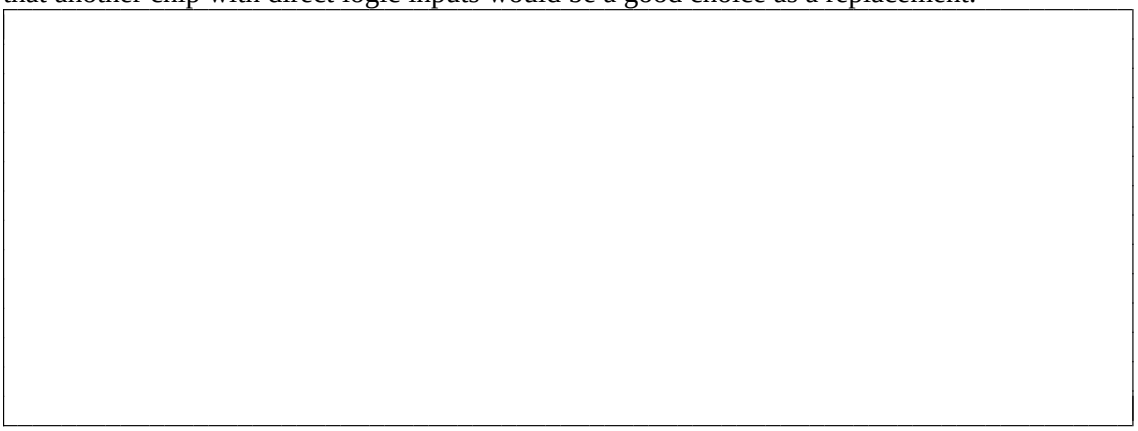

Figure 4. Sonar Rotation Device Circuit

2.3. Software Device Driver

The software for driving the stepper motor involves simply setting a pair of bits in port THREE_C. Bit 0 tells the chip when to step, and bit 1 tells which direction to step (bit 5 indicates the ‘‘home’’ position). The basics of the software was written by Robert Gatlin (who also put the sonar device together).

The result of the logical AND of the value from port THREE_C with 0x10 is zero if the optical detector is blocked (home position) and one otherwise.

Bit 1 of port THREE_C indicates the direction the motor is to step. A one indicates counterclockwise rotation (positive angles) and a zero in that bit position indicates a clocwise rotation is desired. Bit zero of port THREE_C is the step signal. A transition of this bit from 1 to 0 (with bit one set appropriately to indicate direction) steps the motor one step in the appropriate direction. A delay is introduced between steps to put a ceiling on the step rate. Following is a code-segment which steps a specified number of steps in the specified direction:

```
void Sonar::step(int steps) {

        int direction;
        if (steps > 0)
                direction = 0x02;
        else
                direction = 0x00;
        for (int i = 0; i < abs(steps); i++) {

                outportb(THREE_C,(0x01|direction));
                outportb(THREE_C,(0x00|direction));
                delay(StepDelay);
        }
        sonarPosition += steps;
}
```

3. The Sonar Object

The Sonar object is responsible for all interactions with the sonar sub-systems, including reading distances and maintaining the current sonar-heading. Since the sonar device is mounted on a stepper-motor, a particular maximum number of steps are obtainable, thus some rounding may occur, for example, when the object is told to point in the direction of 90° (1.5 radians). Since the distance value returned from the hardware was meant to light a 7-segment display, a conversion function is included (`lookupNum()`).

Everything is in place for the sonar to be able to ‘‘home’’ itself by means of a photo-interruptor module mounted on the sonar pedistal. However, currently there is no linkage which travels around with the sonar transducer to break the light path, so a piece of black electrical tape is place in the slot so that the Sonar object always thinks it is in the home position (whenever it checks).

All functions one would like to have in the Sonar object are present, including the raw distance ahead, the current angle, the number of steps in a complete revolution, the ability to set the angle or directly step a specific number of steps, and finally, an array of points may be returned which represent a full 360° scan of the current room as seen by the sonar.

```
class Sonar {

public:

        Sonar();

        Distance        distance();       //in feet
        Angle           angle();     //in radians
        void            setDirection(Angle ang);//in radians
        void            home();
        void            step(int steps); //+ is clockwise
        PointArray&     scan();
        int             numOfScanPoints();

private:

        int     totalSteps;     //Steps in one revolution
        int     StepDelay;      //How many ms to wait between steps.

        Boolean positionKnown;
        int     sonarPosition;  //in radians

        Boolean atHome();
        void    init();
        void    resetTransducer();
        void    transmit_sonar();    //Chirp it
        void    reset_everything();
        char    lookupNum(int number);   //convert 7-seg to decimal

        float   radiansPerStep;
        Distance MaxDistance;   //Prune any sounding more distant

        PointArray      scanPoints;  //360 degree scan
};
```

IV. Compass:

1. Overview

Radio Shack makes (made) a flux-gate compass for use in automobiles. Since the internals were elctrical signals, it seemed that if we could tap into the right lines, a computer could deduce the heading of the robot. Upon removing the cover and investigating, an MC3403 Quad Differential Input Operational Amplifier was found which amplified two sinusoids. From those two sinusoids we were able to calculate the heading as indicated by the compass.

2. Mathematical Model of the hardware unit (sinusoids)

The two sinusoids were roughly ninety degrees out of phase and each had a range of 2.2V, centered about 3.7V. So, labeling the two signals A and B, I measured A to be highest when pointing East and lowest when pointing West. Signal B was highest when pointing North, and lowest when pointing South. The following table shows the measured values.

TABLE II

Compass Voltage Readings Per Radian Heading

Heading (radians)	A (Volts)	B (Volts)
0	3.75	5.0
$\frac{pii}{2}$	4.9	3.65
pii	3.75	2.3
$\frac{3pii}{2}$	2.6	3.65
$2pii$	3.75	5.0

Figure 5 shows the general form of the signals graphically.

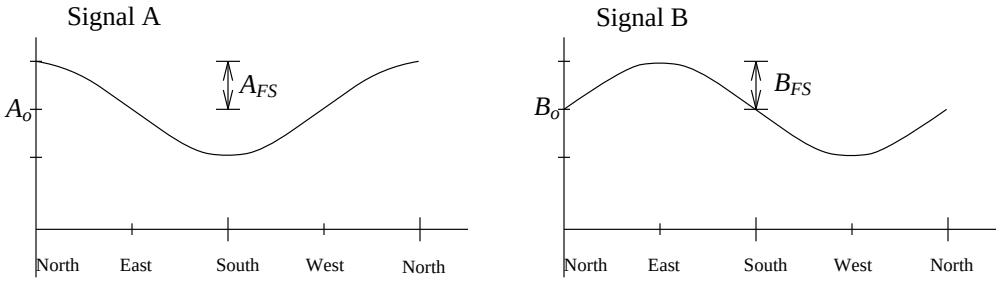

Figure 5. Internal compass signals

One other detail which requires notice is adjusting the compass. To account for differing directions of the magnetic North pole in different regions, and to cancel any effects of the compass being near metal or magnetic sources, there are two adjustments which essentially change the *offset* of the A and B signals. Therefore it is possible for the signals themselves to rise higher than 5 volts; a problem we solve in the next section XXXX.

The signals are modeled as trigonometric functions: $A = A_{FS}\sin(\theta_A) + A_o$ and $B = B_{FS}\cos(\theta_B) + B_o$ and are both needed to determine the heading of the robot since neither function is reflexive ($\sin(\varepsilon) = \sin(\pi - \varepsilon)$). Solving for $\sin(\theta)$ and $\cos(\theta)$ and dividing we have the tangent from which we can then find the heading

$$\theta = \tan^{-1}\left[\frac{\left[\frac{A-A_o}{A_{FS}}\right]}{\left[\frac{B-B_o}{B_{FS}}\right]}\right]$$

3. Analog Input board

As described in section XXXX, the analog input board may be used to convert up to eight, 0-5V signals to an 8-bit quantity. However, as noted above, it would found that to keep the compass signals stable, the signal voltages could rise above 5 volts (in fact, the compass was ultimately adjusted so that the maximum voltages were in the neighborhood of 5.8 volts). We solved this in the simplest (and crudest) manner possible: with a simple set of voltage dividers. Figure 6 shows the adjustable resistor pairs which can cut down the voltage.

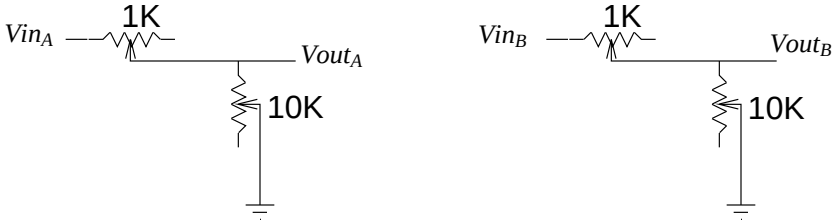

Figure 6. Compass signal voltage dividers

4. Driving Software

The device driver for the compass is a compound object called Compass. Internally, there is another object called CompassInput which actually performs the A/D conversion, does an initial calibration to determine the min and max values to be expected (requires a complete spin of the compass (robot) upon initialization to get this information), and calculates the normalized sinusoid value [-1,1].

4.1. CompassInput Object

This class is nested internal to the Compass class (and private: too). It is necessary to call `initialize()` often while spinning the robot 360 degrees to allow the class to build up a complete and accurate estimate of the minimum and maximum values from the input channel.

```
class CompassInput {

public:
     CompassInput(char which);
     ~CompassInput();
     int raw();
     // initialize is called repeatedly while rotating.
     int     initialize();
     float   sineValue();
     Boolean goodInit();

private:
     unsigned char channel; // N/S or E/W
     int delay;    //Needed to allow the A/D converter to settle
     int min,max;    //Used for calibration
     float   SinusoidValue;
     int     *powerArray;   //Holds powsers of two
};
```

The `sineValue()` function calculates the value of a unit sinusoid (-1 to 1) represented by the current value read from the A/D port. This is accomplished by scaling by the expected full-scale and shifting by the average expected full-scale as seen below:

```
float Compass::CompassInput::sineValue() {

     float theValue;
     float ave, fullscale;

     ave = (max + min) / 2;
     fullscale = max - min;

     float r = raw();  //Read the raw value from the port
     theValue = (r - ave) / (fullscale / 2);
         return val;
}
```

4.2. Compass Object

Since the CompassInput class takes care of much of the ‘‘dirty work’’, the Compass object simply provides a friendly interface to the lower level operations. Of particular interest is the `heading()` command. As seen above from the equations, the heading is simply the $\tan^{-1}$ of the two sinusoids. Following is the `heading()` function of the compass object:

```
Angle Compass::heading() {

        Angle heading = atan2(east.sineValue(), north.sineValue());

        heading = NortherlyAngle - heading;

//Normalize the heading to be in the range [-pi,pi]
        while (heading > Pi)  heading -= TWO_PI;
        while (heading < -Pi) heading += TWO_PI;

        return heading;
}
```

Notice also that any heading may be assigned zero degrees (the NortherlyAnble). This allows the Compass class to adapt to a building which might not have zero degrees assigned as pointing north.

5. Performance/Suggested Improvements

The accuracy of the Compass system is far less than its resolution. Several components may be contributing to this problem of accuracy: The sinusoids may not be precisely 90° out of phase, slight tilts in the flux-gate sensor may cause radical changes in the reported heading, metal near the sensor may skew the results, among others. The first problem (I think the most likely candidate) could be solved with a little better characterization of the signals during initialization, including sinewave-parameter estimation techniques. The second two could possibly be solved with better mounting of the sensor, or moving to a better suited, commercial digital compass.

V. Voice Recognition

1. Introduction

The voice recognition system currently in place on the PSUBOT is called Voice Master Key from COVOX corporation in Eugene, OR. The system was purchased in 1989 for under $200. From the outset I would like it known that clearly there are better and even less expensive

systems available (even from the same company).

The system consists of a headset with a microphone and headphones, a interface board which plugs into the computer, and demonstration and driver software. The demonstration software includes a digital oscilloscope, a crude frequency spectrum. The driver software is a TSR which activates when a key sequence is detected or when something is spoken into the microphone. In either case, the software takes full control of the CPU so that whatever was running previously sleeps.

The system works by the user training it to recognize words. A minimum of three and a maximum of nine trainings are allowed. Up to 256 words may be loaded at one time, though only 16 may be active (chosen from) at any given time. However, once one word is recognized, it may activate a different ‘‘template’’ of 16 words, and so on.

Once a word is recognized, it is inserted into the DOS keyboard-input buffer for processing as though it were typed in. This turned out to be a problem for us since Borland C++ seems to use BIOS routines to read the keyboard, and BIOS knows nothing of DOS. It was therefore necessary to perform a DOS (INT21) interrupt from the program to ask DOS about its keyboard input buffer. This is described in more detail a little later on.

2. Device Driver

The Voice object knows how to ask DOS if there is a character waiting from the keyboard, and then knows how to read one character. This was necessary (rather than just the standard in/out) because Borland C++ seems to read keyboard input directly from the BIOS routines rather than DOS, and the voice recognition system stuffs characters into the DOS buffers, not the BIOS buffers. Following is the (simple) header definition of the Voice class:

```
class Voice {

public:
        Voice();

        Boolean characterReady();
        char    next();              //Read a character

private:
};
```

Both member functions use the `intdos()` function call, which requires some explanation. First, there is a register union, called REGS which stores the a,b,c,d, etc 80X86 register values before and after a system call. So, to test the keyboard input-buffer to see if anything is there, one must simply give the DOS interrupt a value of 0x0B in the AH register, and the interrupt returns with either 0x00 (no) or 0xFF (yes) in the AL register.

Reading a character requires 0x01 in the AH register, the result is then found in the AL register.

```
Boolean Voice::characterReady() {
// A regular C stdio function won't work, since the VMKEY interface stuffs
//  characters into the DOS buffer, but somehow the BIOS routines can't
//  tell they are there (borland uses the BIOS routines)
//
// Otherwise a function such as "int kbhit()" would do the same thing.
//

        union   REGS inregs, outregs;

        inregs.h.ah = 0xb;
        intdos(&inregs,&outregs);       // Perform a DOS interrupt (0x0b)

        return (outregs.h.al);  // 0x00 if none, 0xff if one or more
```

```
}

// ****************
char Voice::next() {

        union   REGS inregs, outregs;

        inregs.h.ah = 0x01;
        intdos(&inregs,&outregs);       // Perform a DOS interrupt (0x01)

        return outregs.h.al;
}
```

3. Caveat:

With any non-preemptive multi-tasking environment, if one process takes control of the CPU and doesn't relinquish it in a timely manner (as in the case of the above described voice recognition system), other real-time dependent processes (such as the PID-based wheel controller) may not be given the processing time they require. This happened during my thesis defense demonstration video. While moving, the wheelchair wheels squeeked loud enough to make the VMKEY software think I was saying something. So, the recognition software took over, and the PID controller suddently had an update period of one or two SECONDS which is relatively quite long. It would be exceedingly nice to have the voice recognition a little more multi-process friendly, or implemented offboard with another processor.

VI. PSUBOT Software System:

1. Overview

Figure 7 shows the object-interaction diagram for the current PSUBOT configuration. Each box with a name represents a major system object, and arrows denote which way messages are passed. (This does not mean that a lower-object cannot communicate with a higher-level object, only that the higher-level must ask the lower-level object for the information.)

2. Data Objects:

Several data elements were encapsulated into classes to improve the readability and reliability of the resulting code, and for ease of programming. They all basically build upon the Vector data structure which is described next.

2.1. Vector (point)

This class was originally a project assigned in an OCATE Object Oriented Design/C++ course. The goal was to define a Vector and Polar class. The Vector class would have been better named a 'Rectangular' class, but an assgnment is an assgnment. A Vector takes an x and y parameter, a Polar object takes an r and θ argument. They can be converted to each other, and all PSUBOT software uses the Vector, though Polar are sometimes used in conversions.

The Vector class overloads many operators, and provides many access functions. Following is a condenced version of the header definition of it.

```
class Vector {

friend Vector   operator+ (const Vector& a, const Vector& b);
friend Vector   operator- (const Vector& a, const Vector& b);
friend Vector   operator* (const Vector& a, const Vector& b);
friend Vector   operator/ (const Vector& a, const Vector& b);

friend Vector   operator/ (const Vector& a, const double   b);
```

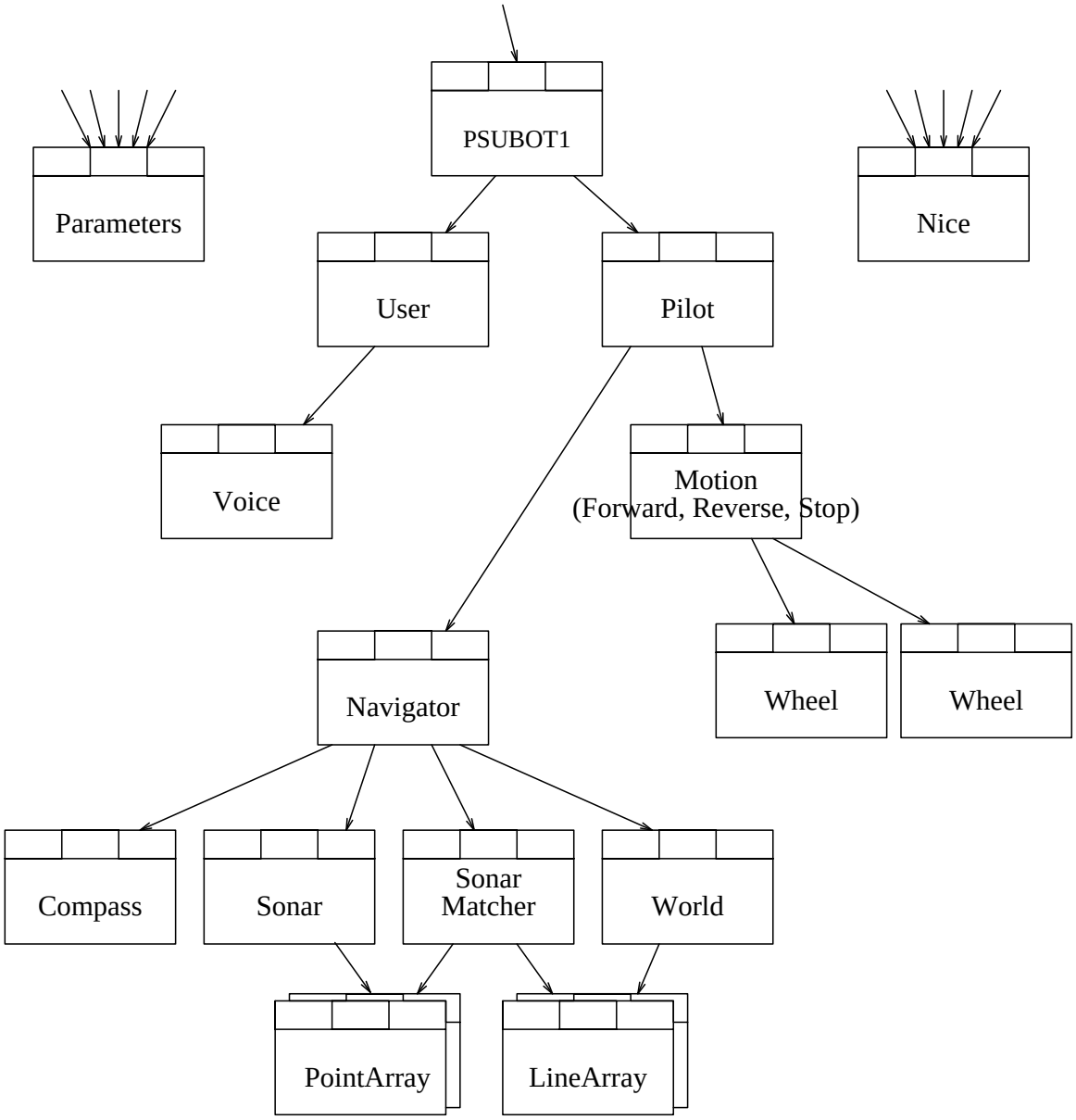

Figure 7. PSUBOT Object Interaction Diagram

```
friend Vector   operator* (const Vector& a, const double   b);
friend Vector   abs(const Vector& a);

// Note: that the above cannot be implemented using reference type
//  return values, since they return the value of a temporary which
//  goes out of scope at the end of the function.

friend Boolean  operator==(const Vector& a, const Vector& b);
friend Boolean  operator!=(const Vector& a, const Vector& b);
friend Boolean  operator< (const Vector& a, const Vector& b);
friend Boolean  operator> (const Vector& a, const Vector& b);
friend ostream& operator<<(ostream&, const Vector&);
friend ostream& operator<<(ostream&, const Vector *);
```

```
friend istream& operator>>(istream&, Vector&);

public:

// CONSTRUCTORS

        Vector() { Real = 0; i = 0; }
        Vector(double real, double imag) { Real = real; i = imag; }
        Vector(const Vector& v) { Real = v.Real; i = v.i; }
        Vector(const Polar& p); //implicit conversion from
                                        //Polar to Vector.

// NON-SYMMETRIC ARITHMETIC OPERATORS

        Vector& operator+=(const Vector& c);

        Vector& operator-=(const Vector& c);

        Vector& operator*=(const Vector& c);

        Vector& operator/=(const Vector& c);

// OTHERS
        double& real() { return Real; }
        double& x() { return Real; }
        double& imag() { return i; }
        double& y() { return i; }
        double radius()  const { return sqrt(Real*Real + i*i); }
        double angle() const;
        char    operator<= (const Vector& a);
        char    operator>= (const Vector& a);

        void    rotate(const double);

private:
// Here is the cartesean data that represents the value of
//  this complex number.

        double Real;
        double i;
};
```

So, using the above class, it would be possible to perform the following actions with two by one vectors (which probably does nothing useful in this case):

```
Vector PointA(-5,2), PointB(3,2), PointC;

PointC = (PointA++ - PointA) * Vector(1, -1);

if (PointC >= Vector(-1,-1))
     cout << PointC.y();
else
     cout << PointC.x();
```

With the addition of the Polar class (not shown here), it is possible to easily tranlate polar coordinates to rectangular, as is done in the Sonar class:

```
Vector sonarDataPoint = Polar(radialDistance, currentSonarHeading);
```

The resulting sonarDataPoint is a regular Vector object, and was converted automatically with a ‘‘user defined conversion’’ (one of the Vector constructors which accepts an instance of the Polar class as its argument).

A Vector may also be created from user input using the standard extraction operator; >>. The exact format of the input expected is defined in the .c files where the operator>> is defined.

2.2. Posture

The robot's location is defined not only in two-dimensional space, but also in terms of a heading. Therefore it was convenient to define another class which would hold the current x, y position as well as the robot's heading θ.

```
class Posture {

friend ostream& operator<< (ostream& os, Posture& p);
friend istream& operator>> (istream& is, Posture& p);
friend Posture  abs(const Posture& p);

public:
        Posture(const Distance x, const Distance y, const Angle heading);
        Posture(const Vector vec, const Angle heading = 0);
        Posture(Posture& p)
         { disp = p.disp; theHeading = normalizeHeading(p.theHeading);}
        Posture(); //Set all data members to zero

        Boolean operator==(Posture &p);
        Boolean operator<(Posture &p);   //Checks both disp and angle
        Boolean operator>(Posture &p);   //Checks both disp and angle
        Distance& x() { return disp.x(); }
        Distance& y() { return disp.y(); }
        Posture operator+(Posture&);
        Posture operator-(Posture&);
        Angle&  heading() { return theHeading; }
        Angle&  theta()   { return theHeading; }
        Angle&  angle()   { return theHeading; }
        void    addDisplacement(Vector vec) { disp += vec; }
        Vector& displacement() { return disp; }
        Vector& vector() { return disp; }
        void    addHeading(Angle theta);
        void    setDisplacement(Vector& displacement)
              { disp = displacement; }
        void    setHeading(Heading newHeading)
              { theHeading = normalizeHeading(newHeading); }

private:
        Angle   normalizeHeading(Angle &a);   //Constrain -PI <= a <= PI
        Vector  disp;
        Angle   theHeading;
};
```

Notice that many of the operations rely on the functions available from the Vector class.

2.3. Line

Since the model of the room used in the matching algorithm developed for my thesis rely on line segments, I thought it fitting to develope a line class which could be optimized for computational efficiency and programming ease. The efficiency was accomplished by calculating all parameters of the line on construction (or when the line was changed) so that subsequent requests for the length of the line, for example, was reduced to a mere return of a reference to that precalculated quantity.

A Line may be constructed from either a pair of points, or another line (the copy constructor). However, member functions are avilable to tell the length of the line, the distance from an arbitrary Point (Vector) to that line (taking into account all three possible cases of orientation), distance from the origin, unit vector from origin which is orthogonal to the line, among others. Follwing is a listing of the header file:

```
class Line {

friend  ostream& operator<< (ostream& s, const Line&);
friend  ostream& operator<< (ostream& s, const Line *);

friend  istream& operator>> (istream& i, Line&);
friend  istream& operator>> (istream& i, Line *);

public:
        Line(Point, Point);
        Line(Line& l);
        Line();

        Angle    normalTheta() { return normalAngle; }
        Distance distanceFromSegment2(const Point& pt) const;
        Distance distanceFromSegment(Point& pt);
        Distance distanceFromInfiniteLine(Point& p);
        Distance radius() const;
        Distance length() const;

        ostream& dump(ostream&);//Output contents

        Point   unitVector() const;
        Point   pointA() const { return pt1; }
        Point   pointB() const { return pt2; }

        void    setPointA(Point p);
        void    setPointB(Point p);

private:
        Angle   normalAngle;
        Distance myRadius;  //distance along u to the line from the origin
        Distance myLength;
        Point   u;//Unit vector orthogonal to the line
        Point   pt1, pt2;   //The endpoints of the line
        void    calculateEverything();
        ostream& internalInsertion(ostream&, const char dump = 'n') const;
};
```

The function `calculateEverything()` is called to calculate the non-settable private values based on the two endpoints.

2.4. PointArray

It proved useful to have a structure which would store a set of points in fixed order (though the order was not critical in this case). Since the same basic class was also desired for Lines, the initial implementation took advantage of a trick using #include's and #define's to custimize the array class to store either Lines or Points. Clearly a Template definition would have been superior, but that extension to the language was not yet available. It was also necessary to have additional functions associated with the different array elements.

So the result is two very similar classes (PointArray and LineArray) which each store their respective data items and provide similar functions, though the PointArray class has a few more functions.

```
class PointArray {

friend ostream& operator<<(ostream&, PointArray &);

public:
        PointArray();
        PointArray(int);
        PointArray(PointArray &);
```

```
        ~PointArray(); //Destructor

        Point& operator[](int index);

        int     size() const {return mySize;}
        int     num() const {return mySize;}
        void    resize(int newSize);
        Point   centroid(); //Geometric centroid of points

        void    rotateAboutPoint(Angle&, Point&);
        void    add(Correction &);  //Rotate and Translate
        void    translate(const Vector &);
        void    resetSize();
private:

        Point *theArray;
        int     mySize;
        int     MaxSize;
};
```

The access member function (`operator[]()`) does bounds checking to ensure access within allocated memory. However, since there is no telling whether the function is used as an lvalue or rvalue, it is not possible to ensure that data is not read where it has not yet been written. As long as access is within the maximum number allocated, simply notes the largest index accessed and reports that as its size.

2.5. LineArray

This class is very simliar to the PointArray as described above. Most notably, it adds the ability to output the lines in the array in grap format (used in connection with troff to generate a picture).

See the program listing for the definition of this class.

3. Utility Objects:

There were some system-wide capabilities desired for both ease of development (Parameters object) and runtime efficiency (Nice). Many of other objects referred to have an `extern` reference to one or both of these classes, and they have both proved to be very very useful in several projects I've worked on.

3.1. Parameters

The Parameters object is basically a runtime custimization class. It allows various objects to read numbers or strings at runtime as specified by the user in a parameters file. For example, the maximum allowed forward speed is not found anywhere in any program file, but is acquired through the Parameters object whenever a Forward object is constructed.

The Parameters object is constructed once globally by some main procedure and then referenced by any number of objects. Here are the member functions of this useful class:

```
class Parameters {

public:

        Parameters(char* filename);

        ~Parameters();

        float   getFloat(  const char* type, const char* param);
        int     getInt(    const char* type, const char* param);
        char*   getString( const char* type, const char* param);
        char    getChar(   const char* type, const char* param);
```

```
private:

        Boolean Initialized;
        char    *first, *second, *string;

};
```

So each member function finds a different type of parameter - float, int, char*, char - and is given two char* parameters. My convention was to pass the name of the object as the first parameter (the ‘‘type’’ parameter), and the parameter name as the second string. So, when the Forward class is constructed, it performs a call to the `getFloat()` function

```
ForwardSpeed = param->getFloat("forward","speed");
```

This runtime resolution of execution parameters allows quick modification of system variables during debugging and performance evaluation.

There is currently no method of changing parameters in the parameters file (commonly called ‘‘param.dat’’), though such a capability could prove useful in the future. It would also be nice to get the compiler to figure out which of the four functions to call based on the type of the variable to which the value is being assigned.

3.2. Nice

The Nice object was used to divide execution time ‘‘fairly’’ between major objects as the nice(1) command in UNIX adjusts priority of a process, though not as well. The concept is this; there is a main execution loop which tells each major object to `work()`. Thus each object is called an equal number of times. However, since some tasks require smaller periods between working, it was necessary to cause the other objects to basically skip a certain number of executions of their `work()` function.

This was accomplished by having a global Nice object which would keep a tally of how many times each object requested execution priviledges (each time their `work()` function was called) and would turn them down a predefined number of times. Thus the Wheel class would be allowed to execute each time its `work()` function was called, while the PathSegment class (which does some numeric computations) would be allowed to execute its routines only after every 15 requests, for example. Following is the header file for the Nice class:

```
//The following is a list of all possible Object Types recognized.
enum ObjectType { _Start,
WHEEL,
MOTION,
MATCHER,
NAVIGATOR,
PILOT,
USER,
_Finish };

class Nice {

public:

        Nice();
        ~Nice();

        Boolean myTurn(ObjectType obj);

private:

        ObjectType      modTotal[12];    //How often each gets to execute
        int             modCount[12];    //# times left to ask before grant
```

```
};
```

Clearly this object is of minimal implementation and better management of the real-time aspects of the PSUBOT will be necessary in the future unless a more multi-tasking or real-time operating system is employed.

4. Other Files:

mdefs.h-
This file is left over from dabbling in Mathematica and converts some of its more readable mathematical outputs to proper C syntax.

object.h-
Here is where I define system-wide constants and typedef some common aliases such as Boolean, Speed, Angle, etc.

types.h-
This is used only for backward compatibility with some other objects of the past.

ports.h-
As described in the section about the PIO96 digital interface board, this header file defines the I/O space addresses used to access the various custom external hardware devices.

file.h-This file is not needed. It is empty and is simply included in the nice.cpp file.

5. Major System Objects

5.1. Voice Object:

The Voice object is described in detail above in the Systems section on the voice recognition system.

5.2. Command Object:

The Command object is nothing more than a portable data structure which can hold all the different commands the PSUBOT recognizes. Basically the commands were broken down into two Types: Action and Goal. Actions included such things as ‘‘stop’’, ‘‘continue’’, ‘‘quit’’, etc. Goals were simply an x, y, θ representation of where the user desired to go. Thus a Command object could be passed from one level in the hierarchy to another with minimal information about the object required of the ‘‘passer’’. Once to its destination the receiver could then interrogate the object for its command type and either its goal or action desired.

Note that additional actions are easily accomodated by simply adding them to the ActionType enumerated list and modifying the User object (described below) to recognize a user input as belonging to that action.

```
enum CommandType { GOAL, ACTION };
enum ActionType  { STOP, CONTINUE, QUIT, SETLOCATION, STARTAGAIN,
                   STARTAGIN, PRINTLOCATION, NONE };

class Command {

public:

        Command();
        Command(CommandType t);

        void            setGoal(Posture p);
        void            setAction(ActionType a);
        void            setType(CommandType t);
        Posture&        goal();
```

```
        ActionType      action();
        CommandType     type() {return theType;}

private:

        Posture         theGoal;
        ActionType      theAction;
        CommandType     theType;//Determines goal or action
        Boolean         valid;  //Am I (command) valid?

};
```

5.3. User Object:

We desired the ability to enter commands by either voice input or from the keyboard while the robot was in motion. We also wanted the class responsible for user input to be easily expandable. While we met the first of these goals, the second one could be improved upon by simply generating a data structure with all the allowed commands from which commands could be simply added by typing them in or deleted by removing the corresponding program line.

The User object is allowed to execute by the object above it using the `work()` function and is polled for a valid command with the `commandReady()` member function. Of course the command it returns is actually a Command object which is described above. Following is the class definition:

```
class User {

public:

        User();
        ~User();

        void    work();
        Boolean commandReady();
        Command readCommand();

private:

        Command currentCommand;
        Boolean validCommand;
        char    buffer[MaxString];
        int     index;  //index into character buffer
        char*   dummy;  // Used to read in don't-care strings
        Voice   voice;  // Tells us what to do.
};
```

This object, when asked to `work()`, checks for keystrokes from the keyboard (via the Voice object) and collects them until a newline is detected. It then attempts to parse the input line and represent it with a Command object.

5.4. World Object:

The World is simply a file-input class: It can read a Room (which is a LineArray) from a file. Here is it's simple class definition:

```
class World {

public:

        World();
```

```
        ~World();

        Room&   getRoom(int roomNum);

private:
        Room    theRoom;
};
```

However, dispite its apparently simple class definition, the implementation of the `getRoom()` function was everything but trivial. This was due to the varying file-manipulation techniques used on the PC and the SUN workstations. Borland C++ 3.1 implements the full ifstream class, while g++ does (did) not. Thus the file is littered with

```
#ifdef __BORLANDC__
     f >> ....
#else
     fscanf(...
#endif
```

As noted in the world.c file, the format for the roomfile, is like so:

```
room <roomnum>
<number of lines>
(<x1a>,<y1a>) (<x1b>,<y1b>)
     •
     •
     •
room <roomnum>
<number of lines>
                •
                •
                •
```

5.5. Matcher Object:

The Matcher is the class which implements the fundamental algorithm on which my thesis was written; matching points to lines. Basically, the algorithm is given a line-segment model of the current room, a set of points obtained from the sonar-device performing a 360° sweep of the room, and an initial guess of the robot's posture (position and heading). The value returned is a correction (x, y, θ) which, when applied to the robot's previously believed posture, corresponds to the location at which the sonar's data-points match the model of the room *the best* (in the mean-squared sense).

Another way of explaining this matching process is to imagine the data-points superimposed over the line-segment room-model. Then move the data-points (as a rigid body) to the lines they represent so that the most points are the closest to their correct line.

The class itself has a rather simple `public:` interface (and a rather complicated `private:` section), and as was seen in the discussion about the Navigator object, this class was intended to be polled, so that other operations could be performed at the same time as the matching was taking place (since it is an interative process). However, testing of the algorithm yielded strange memory errors (diagnostic output strings would started to get scrambled when the program was run). This only happened on the PC (not on the SUN), and I am not sure what the problem was, only that when changed it so that it performed the entire matching session from one function call.

```
class Matcher {

public:
        Matcher();
```

```
        PointArray      match(Posture &, LineArray &, PointArray &);

//      void    work();         //This function removed.
        Boolean done() { return finished; }
        Correction &correction();
        void    setNegligableCorrection(const Correction c) {
                negligableCorrection = c;
        }
        float   calculateS(PointArray &, LineArray &, Vector&, Angle);
        void generateS(Posture &, LineArray &, PointArray &);


private:
     •
     •
     •
```

One public member which was added later, was the `generateS()` function. This was added so that I could make pretty pictures for my thesis, like this:

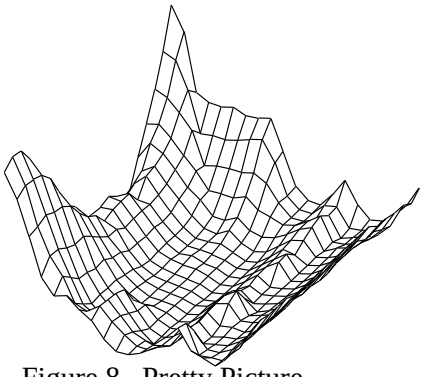

Figure 8. Pretty Picture

5.6. Navigator Object:

The Navigator is responsible for maintaining an estimate the position of the robot. Ultimately this could include the sonar matching algorithm, the compass, the odometry, and perhaps other sources of location information. Currently, however, the Navigator ignores the odometry (except as the initial condition for the matching algorithm) and uses the compass only initially to get a rough estimate of the current heading, and it also uses the compass after each matching session as a litmus test to tell whether the matcher converged to a reasonable location or not. If the compass and matching algorithm disagree by more than some maximum amount, the software terminates. Some more intelligent handling of this condition should be developed in the future.

The Navigator also watches for collisions and may alert the Pilot when an obstacle is detected too close in the direction of motion.

```
class Navigator{

public:

        Navigator();

        Posture ourPosture();
        Boolean done();
        void    work();
                  //initiate the localization
        void    whereAreWe(Posture guess, Boolean wild = True);
```

```
        Boolean inpendingCollision();
        void    watch(Angle direction);
        void    init();
        Boolean goodInit();

private:

        Boolean finished, validLocation;
        Posture location;
        Distance minimumSafeDistance;
        Angle   MaxCompassMatcherDiscrepancy;

        Matcher matcher;        //a matching algorithm
        Compass compass;        //a device
        Sonar   sonar;          //a device
        World   world;          //a world database
        int     currentRoomNumber;
        Room    currentRoom;
        PointArray scan;

};
```

Two additional items need mentioning. First, the Navigator was initially designed to execute on a polled basis; as do the Pilot, Wheel, and User objects. However, due to problems with the Matcher I decided to make the matching operation atomic. So the functions `done()` and `work()` are currently not necessary since `whereAreWe()` does all the work.

The second item to note is that the Navigator requires repetative initialization before using it. This is due to the Compass, and its need to calibrate itself. Therefore, it is the responsibility of some other module to ensure that an entire 360° rotation is achieved, though the Compass can detect if **no** rotation occured.

5.7. Pilot Object: Goals, Abilities Interface (class definition) Code Listing

VII. Custom Demonstration Software to Show Off the PSUBOT

1. PSUBOT1.[hc]

Actually, this is a class which invokes and manages the localization-demonstration software for which all this software was originally written. It's place in the hierarchy can be seen from the Object diagram in Figure 3. It is the class that is called by the primary `main()` function until it (the PSUBOT1 object) says it is finished. Here is the class definition, along with the `go()` function:

```
class PSUBOT1 {

public:

        PSUBOT1();

        void    go();
        Boolean finished();

private:

// These are the two objects it likes to talk to.
        User    user;
        Pilot   pilot;
};
```

and in the .c file,

```
void PSUBOT1::go() {

        Command command;

        do {
                user.work();
                if (user.commandReady()) {
                        command = user.readCommand();
                        pilot.command(command);
                }
                pilot.work();
                variable = 2;
        } while ((pilot.status() != DONE) && (pilot.status() != EXIT));
}

Boolean PSUBOT1::finished() {

        return (pilot.status() == EXIT);
}
```

2. ctest.c

This `main()` function is used to test the compass. It first asks the user to spin the robot around at least once and press a space bar. Then it displays the calculated heading of the robot based on the compass.

3. mtest.c

This `main()` function demonstrates the minimum setup,etc required to test the wheel objects, motion object, etc. This can prove invaluable when debugging lower-level modules when, for example, a new controller algorithm is implemented.

4. stest.c

This `main()` function demonstrates the sonar system. It turns the sonar device to every radian-specified angle until 0 is entered, at which time it begins displaying the distance ahead each time a key other than 'q' is pressed. ('q' quits).

5. demo.c

This program is more involved than those above. When linked with the correct user and command files, it allows the user to specify the commands: forward, rotate_left, rotate_right, stop, and quit. This is useful as a simple demonstration of the current capabilities of the PSU-BOT, though it has nothing to do with localization. It does, however, turn the sonar device in the direction of motion in order to pause when something is seen obstructing the path.

6. graph.c

This `main()` function continually graphs the current room by spinning the sonar device around, and plotting the points it sees on the screen. The robot is indicated with a rectangle. This is another nice demo of one of the PSUBOT subsystems (Sonar).

References

Hall89a.
Douglas V. Hall, *Digital Circuits and Systems*, Glencoe Division of Macmillan/McGraw-Hill, 1989.

Hall92a.
Douglas V. Hall, *Microprocessors and Interfacing; Programming and Hardware*, Glencoe Division of Macmillan/McGraw-Hill, 1992.