BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-92-M6

"Blank Map Orienteering For an Autonomous Mobile Robot Using Certainty Grids"

by

Timothy Todd Good

Blank Map Orienteering For an Autonomous Mobile Robot Using Certainty Grids

by

Timothy Todd Good

B.S., Brown University, 1981

Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University.

April 1992

This thesis by Timothy Todd Good

is accepted in its present form by the Department of Computer Science

as satisfying the thesis requirement for the degree of Master of Science

Date 4/17/92 Thom L. Dean

Thomas L. Dean

Blank-Map Orienteering For an Autonomous Mobile Robot Using Certainty Grids

Timothy T. Good

April 8, 1992

Abstract

Certainty grids have been shown to be an effective method of generating accurate map data from incomplete sensor data. Using a robot based certainty grid to maintain map information generated from eight fixed sonars, we compare three robot navigators. A modular approach is used to keep the navigator and mapper independent of each other. The navigator has access only to the certainty grid when making its decisions. Each navigator uses a weighting function to determine a potential for each grid element and navigates by minimizing the potential over the robot's immediate surroundings. Local route selection is done in real time while traveling as the navigator continuously re-evaluates the path with new information from the certainty grid. The navigators differ in their methods of global route selection. One uses intermediate destinations and backtracking to handle dead ends. The others incorporate dead end information directly into local route selection, one with intermediate destinations and the other without them. The certainty grid includes a variety of averaging and weighting techniques to improve sonar accuracy and reduce noise, in addition to a decay procedure which fuzzes old map information as the robot moves.

Contents

1 Introduction

In the sport of Orienteering[1] the object is to follow a course through unknown (usually wooded) terrain using only a map and compass. The course is determined by a set of control points which are placed in the woods and whose locations are marked on the map. They must be visited in order and the winner is the one who traverses the course in the least amount of time. Four abilities are useful to be a good orienteer:

- Speed
- Route selection (path planning)
- Route following
- Error recovery[2]

Similarly an autonomous mobile robot has need of the same abilities. It must be able to efficiently traverse the type of terrain where it is expected to operate: whether it is a smooth factory floor for industrial robots, swamps and forests for amphibious tanks, or dry rocky ground for lunar exploration. It must be able to select a route to its destination based on what it knows of the area and its own abilities. It must be able to follow the selected route by matching what it sees (senses) with what it already knows or expects to see. Finally it should be able to recover from mistakes and select new or corrected routes if previous ones fail or become outdated by new information.

A special type of Orienteering is *Blank-Map Orienteering.* Instead of a highly detailed orienteering map (which is both expensive and time-consuming to construct) a competitor is given a map with nothing but the control points, start location, and a north arrow. He or she is expected to plan the route based only on the relative location of the destination to the start and what can be seen along the way. Our robot is in an analogous situation. When it starts, it has no knowledge of its surroundings beyond what it can

[1]For more on Orienteering contact the US Orienteering Federation, Box 1444, Forest Park, GA 30051

[2]It should be noted that *very* good orienteers have little need for error recovery. They do not make mistakes.

sense. Given a destination, it must plan a route using only the limited map it can construct, and it must modify both the map and the planned route along the way.

This project implements a blank-map navigator, which in addition to the standard goal of arriving collision free at a destination, aims to be both domain independent and reasonably efficient. Of the four abilities listed earlier, the most important for blank-map orienteering are route following and speed. This is due to the relative difficulty people have going in a straight line and the simplicity of the route selection. The generally open terrain usually suggests a route of directly toward the destination. Error recovery is necessary only if the route following fails. With little time spent in path planning, and essentially the same path for everyone, speed at following it becomes very important.

For this project, however; the priorities are different. Route following is still very important, but route selection and error recovery also receive consideration. The first ability, speed, has to do with the physical capabilities of the robot which are fixed for the scope of this project. Our robot will not be doing any lunar exploration and is restricted to smooth, level surfaces in range of its radio link. The speed of the robot is limited not by the drive motors but by the communication and processing time needed to keep the map and route accurate. (its route following ability) We move at the best speed possible. The second ability, route selection, is initially simple due to the limited knowledge of the surroundings. Instead of planning an entire route at once, it is sufficient to incrementally plan a route using only the immediate surroundings. While initially simple, route selection becomes more important as a map is generated. As described below, better route selection means less error recovery. One assumption we make during route selection is that the area is traversable[3]; an incremental planner such as this one may fail to recognize when something is unreachable while a global planner could determine this. Having make this assumption, however; we proceed to do as much global planning and checking as possible. The third ability, route following is still very important. Since the robot has no easy way of determining its position if lost, it must be very careful to always know where it is. Obstacle avoidance is a large part of route following since unlike

[3]By *traversable* we mean that there is sufficient space between most obstacles for the robot to pass, a route to the destination exists, and maze solving will not be required.

people, the robot cannot simply identify the object and determine how to go around it based on what it is. The final ability, error recovery, can play as large a role as the designer wishes. The need for error recovery varies inversely with the quality of route selection. Our first attempt at a local navigator (section 3.1) had neither good route selection nor good error recovery and its global performance showed this. As global navigation becomes possible, so do global navigation errors and recognition of local navigation errors. One navigator uses a system of intermediate destinations, checkpoints, and backtracking to handle errors, another depends on intermediate destinations and good route selection, while the last depends only on good route selection to prevent errors. At the local level, most of the effort is spent on avoiding errors since, due to hardware limitations, collisions (the most obvious type of error) tend to overload the robot's drive motors or disable the communication lines effectively killing the robot.

The project breaks down into three major areas: accurate sensing of the world, position tracking and route selection. Sensing is done using eight fixed wide-angle sonars and the data maintained in a certainty grid. The certainty grid approach [2] allows incomplete sonar data to be combined with other sonar readings to increase accuracy while avoiding the brittleness or need for domain specific knowledge common to many geometric paradigms. Geometric information can still be extracted from the certainty grid, although we do not do so. Sensor data is filtered at the front end using averaging and thresholding to determine probabilities and *Regions of Constant Depth* [6, 12] applied to each sonar to set confidence levels. Additional confidence factors are associated with each grid element. Position tracking is accomplished through the robot base's built-in distance and angle measuring procedures. While this is sufficient for maintaining the certainty grid and the robot's location within it, positional errors can accumulate, making it difficult to accurately reach a distant destination. Ideally, some sort of map matching locator would be included. For both route selection and route following we use a weighting function based primarily on the occupied probabilities of nearby areas and direction to the destination to determine their potential *badness*. The robot simply heads in the least objectionable direction by going to cells of lower potential. [4]

The project uses the robot Luey in the Brown University AI lab and consists of three modules and a driver. A separate simulator is also provided

to replace the robot routines. The certainty grid routines are contained in the *Cgrid* module and described in section 2. It contains procedures for building and maintaining the certainty grid as well as the sonar routines. The navigator and location routines are contained in the *Oman* module and described in section 3. Subroutine calls are made to the Cgrid module to maintain and update the grid while the grid itself is available through special access functions. Section 4 describes the remaining module, *Display*, as well as the driver and the simulator. The display module handles the terminal output using the Curses package and an X window for the certainty and badness grids. The driver ties the modules together and provides a user interface to the robot, while the simulator mimics robot responses for testing purposes.

The project was originally written in C but was converted to C++ when the robot libraries were changed.

2 Certainty Grids

A certainty grid [7] is a probabilistic tesselated representation of spatial information. In its simplest form, it is merely an array in which each element holds a value representing the probability that something is true at the portion of space represented by the element. In our case each element of the certainty grid ($C(x,y)$) holds a value reflecting the probability that a space is occupied[4].($P(OCC)$)

$$C(x,y) = P(OCC) \tag{1}$$

Earlier versions [8] used two grids, one with occupied probabilities and one with empty probabilities ($P(EMP)$), but since the occupied probability can be related to the empty probability by

$$P(OCC) + P(EMP) = 1 \tag{2}$$

only one array is needed. Some information is lost, however; since two grids allowed a measure of confidence to be associated with the probabilities. If

[4]Certainty grids for spatial representation are also know as *Occupancy Grids* for this reason.

the two grids were independent, (equation 2 is not satisfied), then the confidence in the probability values would be the difference between $P(OCC)$ and $P(EMP)$. We use a second array to associate a confidence with each grid probability.

Another related spatial representation is described in Slack and Miller [10] where an array holds traversability information and links to neighbor cells instead of occupational probabilities. Elfes [2, p.37] has also proposed that each cell could have a vector of probabilities. This *Inference Grid* could hold factors for traversability, terrain, etc. in addition to those for spatial occupancy. Although the information is stored in separate arrays, this is technically what Luey does. As described in section 3.1.1, he maintains factors for occupancy, destination, and previous route. A complete description of occupancy grids can be seen in Elfes. [2, 3]

2.1 Grid and Background Values

Within the certainty grid we assign a value of 0 to all cells known to be empty, and a value of 1 to all cells known to be occupied. All other cells are assigned a value between 0 and 1 representing the probability that that area is occupied. Initially all cells are assigned the background value. Luey has no knowledge about the cells with the background value. An obvious choice for the background value would be 0.5, midway between the values for an empty and occupied cell. Since the certainty value is a probability, Luey would then think that all unknown cells have an equal chance of being open or filled. A better choice would be to have the background value reflect the actual density of the surroundings. An average of all the cells in an area would yield a background value which reflects the object density of that area. The area surveyed should be sufficiently large to provide a representative sample. Fig. 1 shows the floor plan of the Brown University AI lab.[5] Overlaying a scaled 20cm grid on the plan and taking the average over the entire area gives a background value of 0.457 which is surprisingly close to the value suggested earlier. When the walls are included, the background value rises to 0.499. Taking the average over just a portion of the area (left half) gives background values of 0.48 and 0.52 which are close approximations of

[5]Appendix A shows Luey's view of the same area.

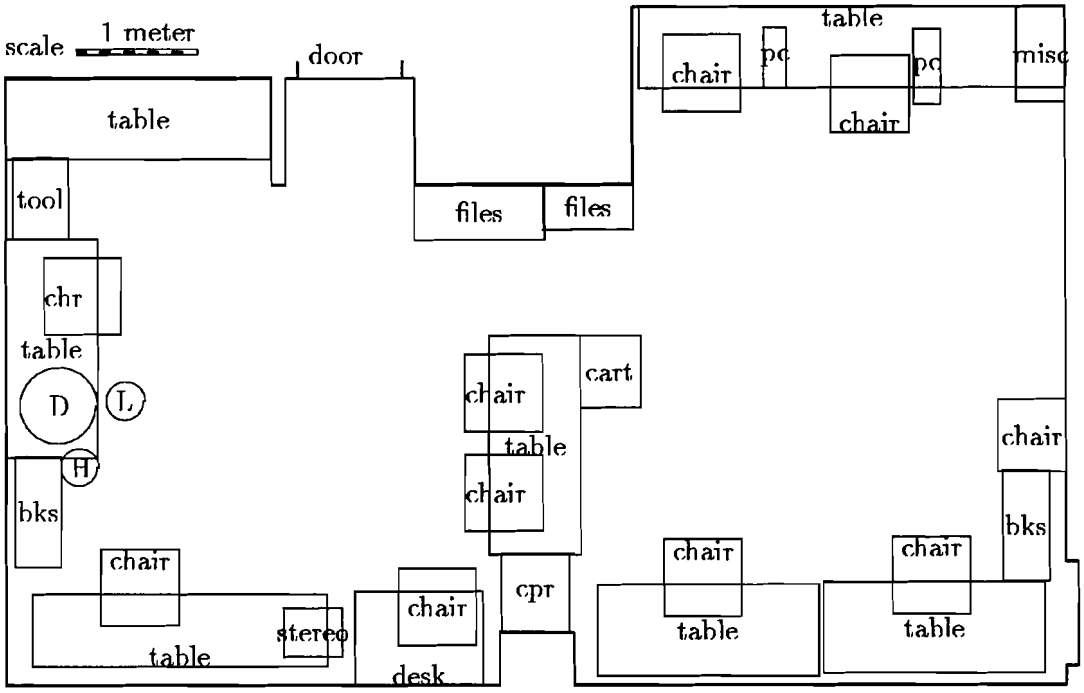


Figure 1: Floor plan of Brown AI lab

the actual background but require less calculation. In order to do this we must assume some foreknowledge of Luey's working environment. At the current time this is not a problem since his environment is well defined. It does however violate our assumption that Luey is operating in an unknown environment and is domain independent. Instead Luey can generate his own background values by doing a preliminary survey. He will scan as much as he can see in all directions. The occupied and empty cells are put into his certainty grid and an accelerated decay done on the certainty grid with no additional scans. With proper handling of the grid boundaries, the decay will eventually reach a steady state with all cells at a new background value. The process can be repeated using the new value until the starting and ending values are the same. Even simpler would be to just average the cells after the survey. Since Luey doesn't know the size of the area he can see, he uses both methods to find the background. After looking around, he will first decay the grid a few times to even out how much is visible, then take the average of some nearby cells. He averages all cells within 3/4 of his sonar range. Be-

cause scanning from one position does not give much depth to objects, and it is unlikely that Luey will be in a congested area, it is expected that the procedure will yield values lower than the actual background. Running this procedure in the left side of the AI lab yields a background value of 0.299. When run in the simulator the background varies from 0.319 when only the circle scan is used to 0.528 when the entire map was preloaded. This turns out to be largely irrelevant since any place Luey is likely to go will be also be scanned by his sonars. It becomes significant only when the sonars are turned off.

For most operations on the certainty grid, the full range of probabilities between 0 and 1 is used, but some operations require less information. Occasionally, the navigator needs to know only whether a cell is occupied or not. Luey applies two thresholds to a grid value to determine this. If the certainty value of a cell exceeds the occupied threshold ($THO = 0.75$) then the cell is considered occupied, while if it falls below the empty threshold ($THE = 0.25$) then the cell is considered empty. A third threshold is the path threshold $PATH\_VALUE$ used to determine paths through the grid. It is usually set to either THE or the background value. The application of these thresholds is described in section 3. These three thresholds are changeable and can be set by either the navigator or the user. Other fixed probability values which may be needed to reflect sonar uncertainties are listed below:

HIGHO	High occupied probability	1.00
MEDO	Medium occupied probability	0.75
LOWO	Low occupied probability	0.62
LOWE	Low empty probability	0.38
MEDE	Medium empty probability	0.25
HIGHE	High empty probability	0.00

These values are independent of the background value and are determined solely from the choice of 1 for occupied and 0 for empty. Note that with our terminology a low occupied probability is not the same as a high empty probability. A low occupied probability is still more likely to be occupied than empty.

2.2 Grid and Cell Sizes

There are several issues to consider when selecting both the size of a grid cell and the number of cells in the grid. Large cell sizes allow the mapping of larger areas but at the expense of accuracy. An area containing small objects but mapped using a large cell grid may cause uncertainty in the grid. Different sensings of adjacent areas may be covered by the same grid cell. If one sensor sees the object while another does not, then the certainty grid will remain uncertain. Large cell sizes can also lead to object boundary uncertainties. Once an object has been sensed and put into the certainty grid, its actual size is lost as it is consumed by the cell size. An object of less than one cell in area could be mapped as covering 1, 2 or 4 cells depending on its location relative to the grid as shown in figure 2. In practice, however; it can be expected to tend toward the larger numbers due to the spatial uncertainty of the sonars.

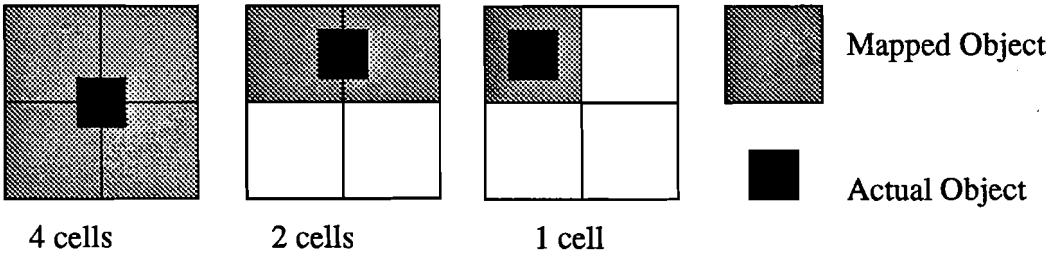


Figure 2: Possible mappings of a small object

On the other hand, some sensors are inaccurate enough that a smaller cell size gains little accuracy while increasing the computational costs. Having an object grow in size to the next cell boundary can be useful for avoiding near collisions, and the loss of accuracy may be insignificant if the grid is decayed. Proper selection of the cell size may even provide a simple approximation of ellipsoidal estimations of spatial uncertainty discussed by Smith and Cheeseman [11]. A single sonar on Luey can have a 30 degree spread which at its full range of 6 meters spans an arc of 3.2 meters. A small cell size of 10cm means 32 grid cells might be updated just for the object found by

a single sonar reading. Increasing the cell size to 50cm reduces the number to 7. For a robot taking almost continuous sensor readings as well as doing cell calculations for navigation this could become computationally expensive and strain a small on-board computer. Additionally a cell size smaller than the robot can add another level to the navigator. If the robot cannot fit into a single open cell then the path planner must expand its search in some way or take its chances.

One method used to gain advantages of both large and small cell sizes is a hierarchy of maps with different resolutions [7]. Another method with interesting possibilities would be to use a polar coordinate based certainty grid. Cells at a greater distance from the center would naturally cover a larger area for the same angle. Either method would provide greater detail near the robot and a larger coarser map for long range planning.

The two most important considerations we used to select the cell size were the size of the spaces between obstacles[6] and the size of Luey. For our grid we use a cell size of 20cm. While smaller than Luey's diameter of 33cm, a two cell opening is large enough for him to pass through. The average boundary positioning error ($1/2 * \text{cell size}$) is less than 1/3 of Luey's size and full cell positioning errors are still less than his width. Our assumption of traversable terrain allows positioning errors up to 1 cell while still allowing Luey to pass. Similar to using a point robot and enlarging obstacles to match, we trust in our navigator fuzzing routine to close off openings too small for Luey to pass through. Also, since the cell size exceeds half of Luey's width, he can enter a cell without projecting into (and hitting an obstacle in) the cell one away.

The size of our grid is based on the following:

Robot's scanning range The grid should be large enough to hold everything Luey can see plus have enough extra for some movement. Luey's sensors have a maximum range in one direction of 6m, so our minimum grid should cover 12m.

Grid decay rate As the grid decays with time or movement, earlier map data is lost. An overly large map will be filled with meaningless data.

[6]While remaining domain independent, our assumption of traversability means we expect a traversable path to be at least two robot widths. This is not unreasonable when the relative sizes of hallways and doors to people are considered.

as Luey moves away. A checkerboard pattern of empty and filled cells decays to low probability in about 15 decay cycles (3 meters of travel) if not updated.

Navigation lookahead The grid should be large enough to hold the minimum data required by the navigator. Luey's local navigator looks no more than 5 cells away from the current one. The global portions of the navigator use all of the grid regardless of its size.

Purpose of the map Map making or long range planning requires a larger map to be kept, although the decay problem must be overcome.

Robot centered map Our grid is robot based but only centered when Luey approaches an edge. This saves time on copying the map and makes the display look nicer, but does require enough extra grid size for him to move without approaching the edge.

The grid size used is 100 by 100 for a map size of 20 meters. This gives Luey 4 meters of travel before his sonar range is off the map. Since few obstacles will be detected at the maximum range, (and if they are the navigation problem becomes the uninteresting one of traversing open space) Luey is allowed to come within 3 meters of the grid edge before being recentered. This easily meets the 1 meter requirement of the navigator. Testing to date has shown that if anything, the current grid is too large. The decay procedure and display of the grid are both time consuming operations due to the large number of cells, while Luey could easily get by with a smaller grid size of 60 by 60 (12 meters).

2.3 Updating the Grid

Among the virtues of a certainty grid is its ability to combine data from several sources into a single world map. The more input devices used, and the more different types, the more accurate the resulting map. Luey unfortunately has only one type of sensor; an eight element sonar array, but the certainty grid still allows readings from different sonars to be easily merged. As described in Sec. 6, the problem of a single type of sensor device is more serious than originally suspected. The only other source of grid updates is

from robot movement. As Luey leaves a cell, the map is updated to mark that area as empty with high probability. This can be a problem if the sonars are turned off and the background value is high.

2.3.1 Sonar Updates

The simplest way to update the map would be to fire each sonar independently and using the distance returned and the sonar's orientation convert to the rectangular coordinates of the grid. The indicated cell would be marked occupied ($C(x, y) = 1$) and all intervening cells marked empty ($C(x, y) = 0$). Unfortunately; the sonars aren't nearly so accurate. Uncertainties can arise from any of the following as shown in Fig. 3:

- A sonar can receive echoes from a 30 degree cone with side lobes, not just from the single direction it faces. (Spatial uncertainty)
- A valid echo may not reflect back to the sonar. (Specular reflections)
- An echo may have bounced from 3 or more surfaces before being received. (High-order specular reflections)
- One sonar may interfere with another. (Interference)

We use a variety of methods to deal with these uncertainties.

Figure 4 shows the possible sonar arrangements that Luey has available. In both the *square* and *front* arrays each sonar is paired with another facing in the same direction but offset by 24cm. If the two returned readings are within a cell of each other, then they are probably sensing the same object located in the intersection of their cones as in Fig. 5. Luey will update the grid with a high probability of an object on the centerline between the sensors and lessor probabilities on an 8 degree arc on each side. A 16 degree cone of emptiness is also added with high probability. The smaller (than 30 degrees) cone is used to avoid erroneous empty readings: The echoes least likely to be reflected back due to specular reflection are those with a low angle of incidence, and it is these which are eliminated by narrowing the cone[7].

[7]Experience has shown that Luey has a blind spot to obstacle faces nearly parallel to his line of travel at $\pm$ 45 degrees from his direction. It is these false negatives we wish to avoid.

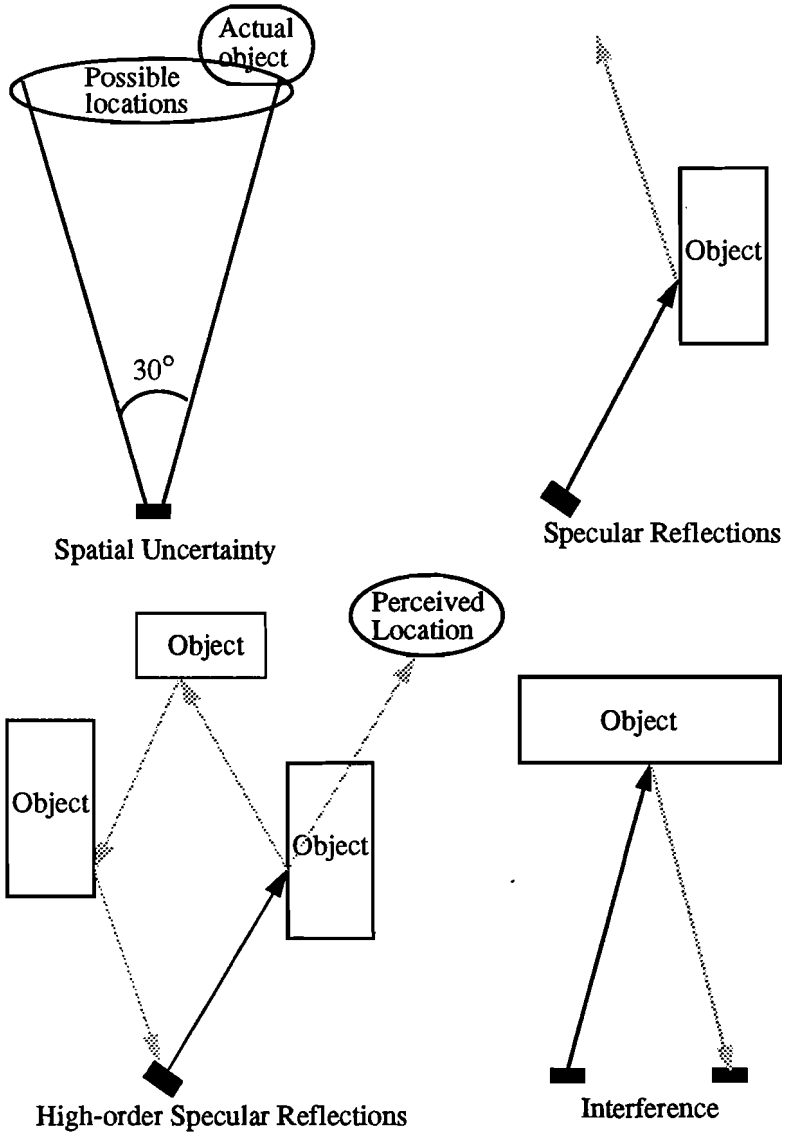


Figure 3: Potential Sonar Uncertainties

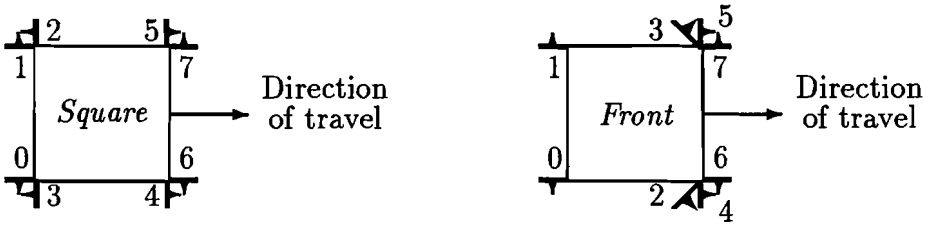


Figure 4: Sonar arrangements on Luey. Sonars located on the same corner are separated vertically so that they do not physically interfere with each other.

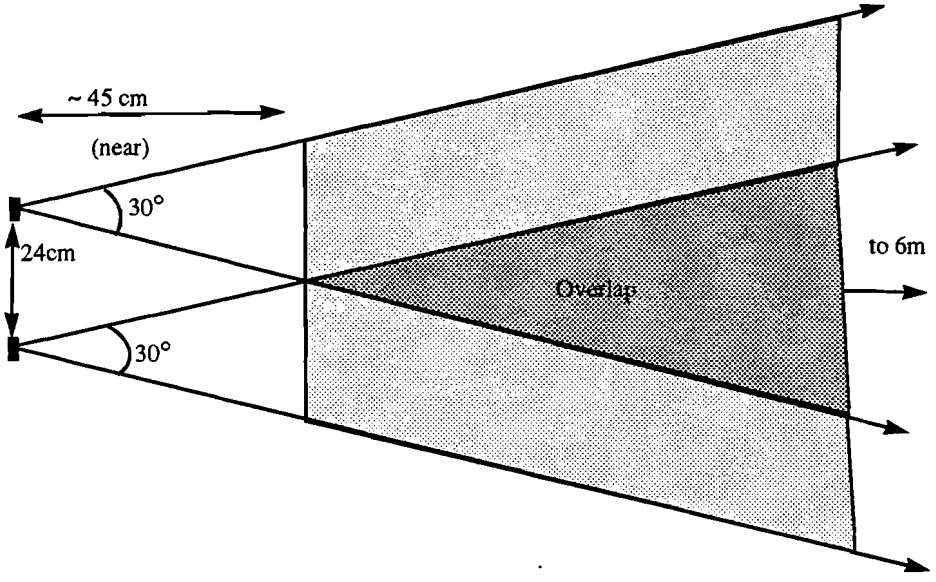


Figure 5: Paired sonar coverage

The potential loss of information from the side sonars is negligible since forward movement will change the sonar's relative angle allowing later sensing. In the forward direction, however; the loss of information is more critical so the *front* sonar array has the two rear sonars from the *square* array moved to the two forward corners to check Luey's blind spots. Testing has shown that this still often fails to correct Luey's faulty vision as these sonars are particularly vulnerable to specular reflections when Luey is following a wall.

While preventing collisions caused by Luey turning into a wall, the false readings from the angle sonars are usually the cause of the turn in the first place.

If the two readings are not within a cell of each other, then the spatial uncertainty of the object's location actually decreases with accurate sonars. For distant objects, each sonar is probably picking up a distinct object somewhere in its own cone of spread. The nearer object is not in the overlap area or it would have registered on both sonars, so it must be in the small region to one side. Luey's sonars, however, are not particularly accurate, and a more common reason for disagreement is a spurious reading. The exact object locations are not crucial since they are far away so Luey will update the grid with only medium occupied probability. Each object is marked as 8 degrees of arc centered on the line 4 degrees off the perpendicular. The empty space is marked with high probability as in the case where the sonars agreed. For near[8] objects, proper placement is important since they will affect the navigator and collision avoidance may be necessary. Near objects are placed on a 16 degree arc of high probability centered on the sonar which recorded it.

The two unpaired sonars in the *front* sonar array are there to check for objects which may be in Luey's path but are slightly off his course. Their range reading is used directly with no cone of spread. Because these sonars are primarily for collision avoidance, close objects are given high probability so they will be noticed. To help avoid the problem of false negatives from specular reflection, only objects closer than two meters are mapped and no empty space is marked.

Interference between sonars is avoided by specifying a firing order which separates nearby sonars. The current firing order is 05274163 with a 10msec delay between firings. Fig. 4 shows the relative positions of the sonars.

2.3.2 Regions of Constant Depth

Leonard & Durrant-Whyte [6] developed the idea of *Regions of Constant Depth* (RCDs) as a unified framework for obstacle avoidance, position estimation and map building. An RCD is an area where sensor readings taken

[8]Objects closer than the cone overlap (45cm) are considered *near*.

over a sequence of headings have a constant range value. Large RCDs (the useful kind) can be observed over large angles of sensor rotation or during extended robot movement. Leonard & Durrant-Whyte used them to find *beacons* for position tracking, to classify and identify sonar targets, and to eliminate high-order sonar reflections. Their methods, however; required some domain-specific knowledge about the expected targets and worked best when the sonars tracked the targets. Tsai [12] used RCDs with a certainty grid in an attempt to eliminate high-order reflections. Luey uses Tsai's method for grouping RCDs to apply a *confidence* factor to each sonar reading. This confidence factor is combined with the confidence factors associated with certainty grid elements to determine the relative weights of new sonar readings verses old readings from the grid. Large RCDs have a high confidence so are given greater weight when updating the certainty grid while small RCDs have a small confidence and therefore less influence on the map.

Tsai's system of grouping range readings into RCDs employs three statistical factors for each sonar sensor: mean (μ), variance (σ^2), and number (δ). For a new range reading $r_{\delta+1}$ to be placed in the current RCD for sonar S it must meet the following criteria [12, eqns. 2 and 3]:

$$0.84 \times \mu_{\{S\}\delta} \leq r_{\delta+1} \leq 1.16 \times \mu_{\{S\}\delta} \tag{3}$$

$$\sigma^2{}_{\{S\}\delta} \leq 1000 \tag{4}$$

That is it must be near the mean of the range readings in the RCD and the variance must be small. If the new range reading satisfies both equations 3 & 4 then it is included in the current RCD and the three factors updated as follows:

$$\begin{aligned} \mu_{\{S\}\delta+1} &= \frac{\delta \times \mu_{\{S\}\delta} + r_{\delta+1}}{\delta+1} \\ \sigma^2{}_{\{S\}\delta+1} &= \frac{(\sigma^2{}_{\{S\}\delta} \times \delta) + (\mu_{\{S\}\delta} - r_{\delta+1})^2}{\delta+1} \\ \delta_{\{S\}\delta+1} &= \delta + 1 \end{aligned} \tag{5}$$

Equation 5 merely calculates the new mean, variance and number using the previous values and the new range reading. If the new reading r_1 does not satisfy both equations then a new RCD is started for that sonar and initialized as follows:

$$\begin{aligned} \mu_{\{S\}1} &= r_1 \\ \sigma^2{}_{\{S\}1} &= 0 \\ \delta_{\{S\}1} &= 1 \end{aligned}$$

2.3.3 Confidence Values

The sonar confidence value is taken from the number (δ) of readings in the current RCD. In [12], Tsai assumes all RCDs with $\delta \leq 5$ are high-order reflections and either ignores them or uses a weak sensory model to update the map only slightly. Luey does not explicitly eliminate high-order reflections, choosing instead to use all sonar readings to update the map. Small RCDs (such as those coming from high-order reflections) will however, have low confidences and be easily overwritten in the map.

In addition to confidence values for each sonar reading, confidence values are also associated with each grid element. The grid confidence values are a measure of how many times a particular cell has been marked as occupied or empty. They are stored as a positive or negative number with the absolute value indicating the confidence and the sign indicating occupied or empty. Each time a grid cell is updated with medium or high occupancy, its confidence value is increased by 1. When a cell is updated with medium or high empty probability, then its confidence is decreased by 1. When an update is performed, the sonar and grid confidence values are used to weight their respective certainty values as they are averaged. This serves two purposes: spurious sonar readings will not replace valid map information, and doubtful map values will be quickly replaced by accurate sonar readings.

2.4 Decaying the Grid

As Luey moves, his positional uncertainty relative to earlier mappings grows until he is not sure of the location of objects previously mapped[9]. Luey simulates this by decaying the certainty grid. The probability that an object located at (x, y) in the certainty grid is still there decreases slightly while the probability of the same object at any of (x, y)'s immediate neighbors increases slightly. Luey does this by taking a weighted average of each cell with a weighed average of its neighbors as in equation 6. Modified grid confidence values ($F(x, y) = ABS(Confidence(x, y)) + 1$) are used to weight the neighbor cells when they are averaged. The remaining weight (W) is based

[9]He is probably unsure of the location of his destination also, a fact we conveniently ignore.

on how much Luey has moved and is currently in the range 0.95 when Luey has moved an entire cell.

$$C'(x,y) = W \times C(x,y) + (1-W) \times \frac{\sum_{(a,b)\in N1} (C(a,b) \times F(a,b))}{\sum_{(a,b)\in N1} F(a,b)} \quad (6)$$

There are several side effects of the decay process. The fuzzing of objects makes them appear larger to a potential based navigator. We use this effect in a separate local fuzzing routine in the navigator to keep Luey from approaching too closely to obstacles and to close off small openings through which he won't fit. If a large map is maintained or Luey were involved in building a permanent map, then earlier information is lost as he moves out of the area. Luey's decay procedure can be turned off to prevent this. The remaining problem is with the decay of the confidence values. Each grid value has an associated confidence value which must also be decayed. Luey decreases the confidence value by an amount proportional to the difference between the occupancy values of the current cell and the weighted average of its neighbors. (Equation 7)

$$F'(x,y) = F(x,y) - ABS(C(x,y) - \frac{\sum_{(a,b)\in N1} (C(a,b) \times F(a,b))}{\sum_{(a,b)\in N1} F(a,b)}) \quad (7)$$

By linking the decay of the confidence values to the decay of the occupancy values, the confidence value of a cell should approach 0 at the same time the occupancy value approaches the local background.

2.5 Transient Objects

Transient objects are those which are moving and therefore seem to disappear from one spot and appear in another. Luey handles them the same way as stationary objects; that is, no special effort is made to track or distinguish them. Since Luey has no way of distinguishing one object from another except by location, in order to properly track moving objects, he would have to constantly follow them with his sonars. This isn't feasible because his

sonars have fixed orientations, and tracking with them would prevent him from navigating to his destination. The same limitation applies to stationary objects as well. Luey can not easily use objects as beacons to determine his location since it would disrupt his navigation.

Moving objects should not be a major navigation problem however; since an object which appears in a previously mapped empty area will have an occupied probability exceeding 0.5 in four or fewer sonar readings. The actual number of readings needed depends on the mapped certainty, mapped confidence values and the size of the RCD of the new object. Even a single sonar reading should be sufficient for Luey to prefer a neighbor cell allowing time for the object to fully materialize in the map. While the occupied probabilities can change rapidly, the confidence factors will tend to lag behind. Twelve sonar cycles are required to go from one maximum to the other. This means that as an object continues to move and subsequently disappears after a short time, the map will be corrected more quickly than it was perturbed. Because the map is updated by averaging, there is a tendency for a transient object to never completely disappear, and for Luey to avoid empty space because he once saw an object there. This is alleviated by reducing the probability to high empty probability once the probability has dropped to medium empty probability and the confidence below negative three. The probability values are described in section 2.1.

Transient objects which are out of Luey's sonar range will gradually decay to the local background probability as described in section 2.4. The decay also applies to the area near Luey but is overshadowed by the sonar updates.

3 Navigator

Given no information about the surrounding terrain beyond the relative position of the destination, only one navigation option is available: head in the direction of the destination and hope for the best. (Fig. 6a) With a little information about the route, obstacle avoiding routines could be added to bypass simple objects as in Fig. 6b&6c. More timely or complete information would allow an efficient route to be chosen at the start as in Fig. 6d&6e. While initially Luey is in situation (a) , he quickly builds a map of the surrounding area allowing navigation as in b-d while attempting e.

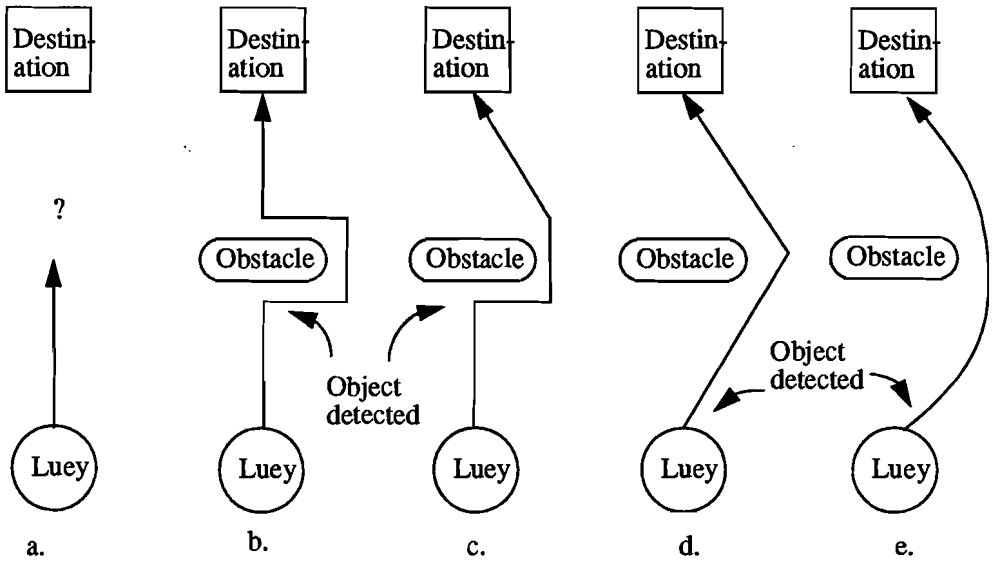


Figure 6: Navigation Options Based on Knowledge

The navigators are presented in historical order. The first developed was a strictly local navigator. While unable to do navigation of any difficulty it serves as the route follower for both the *Backtrack* and *Path* navigators which followed. The backtrack navigator added global route selection in the form of subdestinations and error recovery using checkpoints and backtracking. The path navigator replaced the checkpoints and backtracking with an improved subdestination generator. Finally the improved subdestination generator was applied to the original local navigator to get the *Incremental* navigator.

Within the navigator descriptions the following definitions apply.

Empty cell - A cell with a certainty value below or equal to the empty threshold THE.

Occupied cell - A cell with a certainty value above or equal to the occupied threshold THO.

Path cell - A cell with a certainty value below or equal to the path threshold $PATH\_VALUE$.

Adjacent cell - A cell which shares a common side with another.

Reachable cell - A cell next to a start cell or any cell which is adjacent to an empty reachable cell.

3.1 Local Navigator

3.1.1 Badness Potential

Luey's local navigator uses three factors when selecting a route:

- Direction to the destination, $D(x, y)$
- Obstacles in the way, $C(x, y)$
- Previous route, $T(x, y)$

Each factor is encoded into its own grid and maintained separately. Direction factors are stored in the *direction* grid, $D(x, y)$, obstacles are stored in the certainty grid, $C(x, y)$, and the previous route is stored in the *travel* grid, $T(x, y)$. The three factors are combined into a single potential value which represents the *badness potential*, $B(x, y)$, of a particular location.

$$B(x, y) = D(x, y) + C(x, y) + T(x, y)$$

Luey chooses his route by selecting a direction which takes him to a cell of lower badness. The destination is eventually reached because cells close to it have lower badness and are favored over those far away. Obstacles are avoided because high badness is given to certainty grid cells with high occupied probabilities. Luey doesn't go in circles because cells he has already traveled through are given a higher badness. Instead of repeating a previous path, he will tend to avoid cells he has already traversed and select a new route.

3.1.2 Direction Grid

The direction grid assigns a potential value to each location based on how far it is from Luey's destination. The cells which are far from the destination have high potential values while cells near the destination have low potentials.

Luey simply travels toward the area of lowest potential in order to reach the destination. The potential is calculated by dividing the distance from each cell to the destination by a normalizing value to keep it in the range of (0-1)[10]. The normalizing value is the larger of the starting distance or 10. For a large starting distance from the goal the difference in potential will be small from cell to cell allowing greater leeway in avoiding obstacles while when closer the potential 'push' toward the goal is stronger. Luey recalculates the directional potentials whenever he recenters himself in order to take advantage of this.

3.1.3 Certainty Grid

The certainty grid is described in detail in section 2. When constructing the badness grid, the local certainty values are fuzzed slightly before being used. The fuzzing occurs in two decay cycles and is used to blur object boundaries. This increases the badness potential of cells near objects so that small openings will be closed off, and Luey will avoid approaching closely. The fuzzing is limited in the direction of decreasing potential so that empty spaces can have their potential increased, but an occupied cell will not have its potential lowered below the occupied threshold. After the fuzzing procedure any certainty below 0.12 (half the empty threshold) is set to 0 in the badness grid. This gives the directional potentials greater weight and prevents the local navigator from being excessively picky.

3.1.4 Travel Grid

The travel grid assigns higher potential values to areas which have been traversed by Luey. This discourages him from retracing his path but still allows him to backtrack when necessary. As Luey leaves a cell the travel potential is incremented by a predefined value. ($T(x,y) = T(x,y) + TRAVEL\_VALUE$) If the travel grid indicates that Luey is stuck, (same cell entered over three times) then he will give up and stop where he is. The travel grid is reset to zero each time Luey starts to a new destination. This is an area which could use some further refinement. It should be possible to retrace a route using the travel grid.

[10]Only cells closer than the start will be in this range. Others may exceed it.

3.1.5 Neighborhoods

Given our potential function, all that remains is to decide how far Luey should look while attempting to minimize it. We define the *Neighborhood*, Nn of a cell (x,y) to be all cells reachable in a minimum of n cell moves from (x,y). For the neighborhood $N1$ there are 8 cells one away and 8 possible directions to them. In $N2$ there are 16 cells and 16 straight line directions. Larger neighborhoods have a correspondingly larger number of possible directions to the outer most cells. For Nn there are $8n$ cells in the neighborhood.

Unfortunately; for larger neighborhoods $Nn : n > 1$ the issue is complicated by obstacles occurring in enclosed neighborhoods $Nm : m < n$. In $N2$ there are an additional 24 possible paths due to obstacles in $N1$. In $N3$ the problem is even more complex as both $N1$ & $N2$ obstacles must be considered yielding at least 96 additional paths.

Luey would like to look at as large a neighborhood as possible in order to obtain a good route but not spend a long time selecting the route. In addition to the real time needed to determine an optimal route in a large neighborhood, we must also consider the question of accuracy. Since Luey does not have a map in advance, he cannot know what is ahead unless he can see it. Planning a route based on no knowledge is wasteful and if he can see far ahead then the path must be clear so detailed planning isn't needed. For these reasons, Luey confines his potential search to $N2$. This avoids both the jerky piecewise linear motion from using only $N1$ and the computational complexity for limited results from larger neighborhoods.

3.1.6 Position Tracking

The robot itself keeps track of two values used in determining its location: the current angle and the total distance traveled. When Luey is first powered up these are read to determine the adjustments needed to start each at zero. Subsequent queries include the adjustment automatically. There are three types of robot movement which may require a new position to be calculated.

- Rotation only: Luey changes direction with no distance change.
- Displacement only: Luey moves along a straight line with no direction change.

- Rotation & Displacement: Luey moves and rotates at the same time.

In the first case Luey's position should not change[11] so no new position needs to be found. His current angle is merely updated. In the case of displacement only movement, the new position is easy to determine. Since the distance traveled is known, and the direction remains constant, *sin* and *cos* can be used as in Fig. 7 to determine X and Y components of the distance which can in turn be used to locate Luey in a rectangular certainty grid. These two methods would be sufficient for a navigator which restricted Luey to only 1 type of movement at a time, and in fact would probably be more accurate in determining position changes.

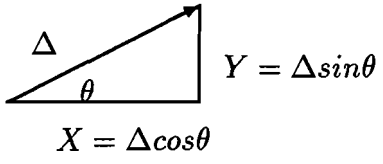


Figure 7: X & Y components of distance Δ along constant direction θ

The remaining case, however; is more interesting and a necessity for a navigator which attempts to do route selection on the fly (ie. without stopping Luey). If Luey travels at a constant speed while also changing his direction at a constant rate then he will trace a circular arc. Knowing the distance traveled and the change in orientation the X and Y components of the movement can again be calculated as in Fig. 8. In order to do this, however; some assumptions about Luey's acceleration must be made. When Luey is starting from rest he will have an acceleration phase where neither the speed nor the turn rate will be constant. We make three assumptions about the potential errors generated.

1. The acceleration phase is short enough that the errors are small. In support of this Luey does not make large changes in speed at once and turns at a conservative value.

[11]This is not always a valid assumption. Testing has shown that Luey often moves a centimeter while rotating.

2. The turning and speed accelerations will tend to occur at the same time which will decrease the errors.

3. The errors will be random in direction so will tend to cancel out over time instead of accumulating. This assumes that Luey's route will have turns in both directions.

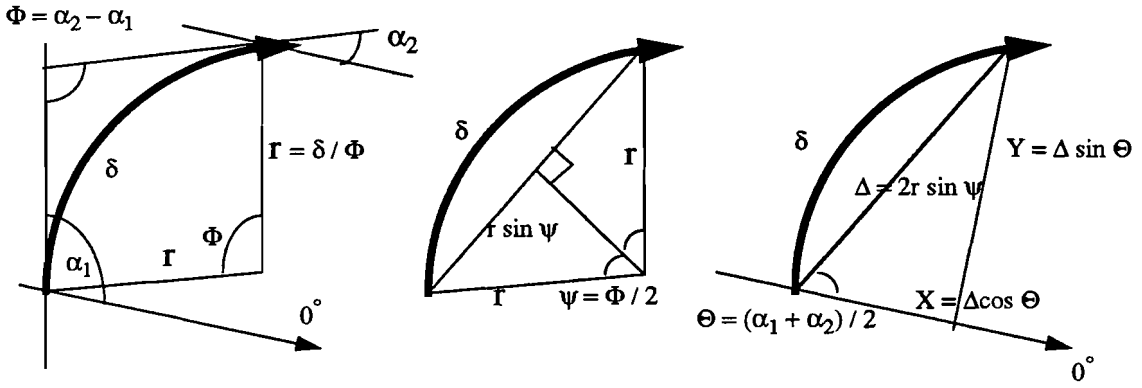


Figure 8: X & Y Components of distance δ along circular arc

An additional source of error is caused by the communication delay when querying Luey. His distance and angle cannot both be queried at the same time. The interval (due to the relatively slow radio link) between the two queries can allow small discrepancies in Luey's position calculations if he is both rotating and translating. These are minimized by changing the query order so that the type of movement with the largest change is queried first. The same problem also occurs when changing Luey's speed and direction. Where possible, the navigator attempts to change only one factor at a time to avoid the additional errors.

Any positional errors which do occur are simulated by decaying the certainty grid map as described in section 2.4.

3.1.7 Elimination Arrays

Each time Luey makes a selection of a cell to head towards, he implicitly eliminates all the other cells in N1 and N2. As he moves to another cell he

must carry this knowledge with him or risk selecting a cell he just bypassed. One possible method is to increase the travel potential of all cells not selected. This would have the advantage of sweeping out a wider traveled path which would aid in covering dead end routes. At the time keeping Luey moving in a reasonably constant path was the concern, not global performance, so instead of a permanent elimination in the travel grid, the cells are only temporarily eliminated in the *Elimination arrays*. The elimination arrays are two 5x5 arrays (large enough to hold Luey's cell and all its N1 and N2 neighbors). One holds the cells previously eliminated for his current location, while the other holds cells eliminated from his current location. As he moves to a new cell, the former array is renewed by transforming the latter's indices to conform to his new position, and the latter reinitialized.

While the elimination arrays did achieve the desired result of keeping Luey moving without unnecessary direction changes, his later performance indicated that perhaps more consideration should have been given to using the travel grid.

3.1.8 Extensions

The local navigator makes several exceptions to its normal procedure of heading toward the cell with the lowest potential. If Luey is within two cells of the destination, he will head directly toward it as long as the way is clear. If the destination itself is occupied then he will stop at a cell next to it. While Luey selects the cell with the lowest potential as his immediate destination, he is not constrained to only go to cells of lower potential. In order to escape a local minimum, he must be able to go to a cell of higher potential. Since the travel grid will increase the potential of the cell just left, Luey should be prevented from returning and becoming stuck in a local minimum.

Once a direction has been selected, Luey checks the map to see that the way is clear in that direction. Based on his current speed and the open distance Luey will speed up or slow down. Ideally, long straight runs will be taken at higher speeds while congested areas requiring turns will be traveled slowly. He also checks the map for obstacles to the immediate left or right. Because of his blind spot at ± 45 degrees off the line of travel, side obstacles which have been getting closer are treated as if they extend in the direction of travel. Luey will adjust his direction slightly to turn away from a possible

glancing collision. This is similar to using the side sonars as a wall follower. Instead of two separate sonars taking readings at the same time, the changing distance to a side object is taken from the map at different times.

Finally, when selecting a new direction, Luey bases his decision not on where he is, but on where he expects to be when the new direction is implemented. Due to the calculation time and communication delay, Luey has often moved significantly in the time between his position determination and the execution of any resulting changes.

3.1.9 Performance

With the local navigator Luey's global performance was dismal. As long as the route to the destination was strictly decreasing in potential he was successful and performed as predicted. If however, any potential increase was required, he got into trouble. He would explore every nook and cranny of a dead end before eventually making his way out and trying a new route. Since this required a close exploration of obstacles, he rarely lived to make it to the destination. Instead of forcing him to try new and successful routes, the increasing travel potential forced him closer to obstacles until he hit something. The inaccuracy of the sonars further increased the problems. False negative readings created non-existant openings for Luey to explore which led to more collisions or slowed progress toward the destination. The problems with the navigator could be summed up as follows:

- Could not identify dead end routes without traveling over them.
- Did not take advantage of what map he had.
- Increasing the badness of routes which lead nowhere forced him into cells with high badness due to occupancy instead of high badness due to directional potential.
- Sonar inaccuracy practically guaranteed that something would be hit before reaching the destination.
- Took forever to accomplish anything difficult.

3.2 Backtrack Navigator

Since the local navigator was successful only at traversing a route which always decreased in potential, what was needed was a global navigator which parsed the route into reachable *subdestinations* and passed them to the local navigator. The *Backtrack* navigator was developed to serve this purpose. It picks intermediate destinations using the portion of the map it has and stores them on a stack. If upon reaching a subdestination, it is found to be a dead end then Luey uses the stack to go back to a previous subdestination and try again.

3.2.1 Subdestinations

Selecting a new subdestination is a relatively simple matter. Luey looks at all the empty cells reachable from his location and selects the one with the lowest direction potential which has not already been eliminated as a dead end as in section 3.2.3. If none is found (or the current cell is the one selected) then he is in a dead end and backtracking is required. For the search he uses a breadth first traverse through reachable the empty cells starting at his current position.

3.2.2 Destination Stack

The destination stack is a LIFO stack used to store subdestinations which have been reached as checkpoints in case later backtracking is required. The top of the stack always holds the current subdestination, while the remainder is made up of checkpoints. Each time a new subdestination is chosen, it is pushed onto the stack, and the previous subdestination on top becomes a checkpoint. If the navigator cannot find a new subdestination, the top of the stack is popped to reveal the last checkpoint. The navigator can then backtrack to the checkpoint and search for a new subdestination from that point. When the ultimate destination is reached, the stack will have all the checkpoints along the way and none of the dead ends. This could be used to repeat the same route later or reverse directions[12].

[12]Unfortunately, due to the new start procedure in section 3.2.4, the eliminated dead ends could include the original start cell also.

Because the subdestinations selected are often at the limit of Luey's sonar range, it is not surprising that often the location selected is later found to be occupied. In this case Luey simply moves it to the neighbor cell with the lowest certainty. If for some reason the local navigator is unable to reach the subdestination, then a new one is selected. This is described in section 3.2.5

3.2.3 Dead End Grid

Whenever backtracking is required, some method of eliminating the dead end must be used or Luey would simply select the same subdestination again. The previous method of eliminating traversed cells from consideration was to use the travel grid. If a cell had a non-zero travel value then Luey had been there before. Since the travel grid is still used by the local navigator, it is reset for each subdestination and not available for global travel information. A new global travel grid designated the *Dead End* grid was created instead.

One of the problems with the old travel grid is the need for Luey to physically visit each cell in order to mark it as traveled. This was necessary because of the small lookahead. Unless the cell was visited, Luey risked missing possible routes which began there. Clearly this is not necessary now. The subdestinations are determined from as far away as possible, so limiting the cells eliminated to those actually traveled would just repeat the same problem on a larger scale. Instead of eliminating cells one at a time as he traversed from one cell to another, he would eliminate them one at a time as he went from a cell to the checkpoint and then back to a neighbor cell. What was needed was a way to sweep along his line of travel eliminating all the nearby cells. Two opposing factors must be taken into consideration. We wish to mark everything which is part of the dead end but not mark anything which isn't. The first criterion eliminates the use of a fixed size marked area such as an extension of the elimination arrays of section 3.1.7, while the second criterion means Luey can't use a complete marking algorithm until he is confident that no hidden route exists. Luey meets these two criteria by using two separate dead end marking algorithms. Which is chosen depends on the degree of backtracking. The first uses the end points of his route to create a fixed width box over the route and eliminates the cells in the box. Since this doesn't necessarily eliminate all cells, it is applied once for each partial backtrack known as a *reverse.* (section 3.2.6) When a full backtrack

to the previous checkpoint is required, the second algorithm is used. After putting a limiting line on the far side of the dead end from the checkpoint, it marks all reachable cells which are closer to the dead end than the checkpoint is. Both algorithms are stopped by cells which are not empty so only empty cells and those immediately adjacent become dead ends.

3.2.4 New Start Cell

When the navigator must backtrack from the start point, it has no previous checkpoint to backtrack to. This is solved by selecting a new start point but instead of pushing it onto the top of the stack like a normal subdestination, it is slipped onto the bottom. The new point can now be backtracked to just like any other checkpoint. A new start point is selected in much the same way as a new subdestination. A search is made of all cells reachable from Luey. The non dead end cells are checked against the three categories of candidates for a new start. In order of preference Luey will use: the cell with certainty and confidence near the background which has the lowest direction potential, the empty cell with the lowest direction potential, and the cell with the lowest certainty value below the background. The first category yields cells which may represent unmapped paths. The second category is the same used to select subdestinations except that the restriction that the direction potential be below that of the current cell is removed. It can lead to a new path or merely move the robot into a position to find a cell in the first category. The last category yields cells which may be blocking a path which will be visible upon closer inspection. This often occurs due to sonar inaccuracies. If no cells of any category are available then Luey is completely enclosed, and no path to the destination exists.

3.2.5 Quitting

The navigator can already determine when the robot is boxed[13] in but what about when the destination is boxed and unreachable? As currently described the navigator would proceed to eliminate all the points around it and then backtrack farther and farther away while looking for a route which doesn't

[13]As long as the enclosing box fits on the grid.

exist. To avoid this, a special check is made periodically to see if a possible path to the destination exists. If either of these situations occurs then the navigator quits immediately.

The local navigator can quit also and does whenever changing conditions have made a subdestination unreachable. If the local travel grid indicates that a cell has been entered over three times on a single leg of the journey, then the local navigator assumes there is a problem and quits. The local quit is caught by the backtrack navigator and a new subdestination provided. The unreachable subdestination is replaced on the stack by the current location and treated as if it had just been reached. A user interrupt is handled the same way. The local navigator exits and the current subdestination is replaced by the current location. Note that this can cause problems if a backtrack was under way and a checkpoint is lost.

3.2.6 Extensions

As with the local navigator some extensions to the basic algorithm are used. When backtracking it is often not necessary to actually go all the way back to the checkpoint before selecting a new subdestination. By marking the current dead end and selecting a new subdestination as if he were at the checkpoint, Luey can often avoid the physical movement. If the next subdestination after a dead end is closer to the dead end than the checkpoint is, then Luey will head directly for the subdestination and bypass the checkpoint. This is known as a *reverse* and uses only the limited dead end marking. The dead end marking is done prior to the reverse in order to allow selection of the next subdestination. When the reverse cell is close[14] it is not necessary to physically travel there either. The dead ends can be marked as if the move were made and new subdestinations picked until a far reverse is found or backtracking to a checkpoint is needed. This keeps Luey from wasting his time making small movements without gaining additions information about the route.

When adding new subdestinations we would like to be able to again skip ones which are close to the current location. In this case, however; the additional sonar data can be significant so the physical move must be made. To

[14]In this case, close means within 4 cells.

prevent the proliferation of close together checkpoints, we simply overwrite the current subdestination instead of putting a new one on the stack.

Knowing that a subdestination was reachable did not guarantee that the local navigator would not have trouble reaching it. In most cases the subdestinations are on a direct line from the last checkpoint, (a natural occurrence since the subdestinations follow to expanding map) but when backtracking and reversing, more complicated local routes may be needed. If a path to the destination is known to exist why not use this information when constructing the path? The construction of the direction grid was modified to include this path information. Instead of the distance based potentials used previously, if the area adjacent to the destination is empty, a new method is used. Starting from the destination a breadth first traverse of all reachable cells is used. Each cell is given a direction potential slightly higher than its previously marked adjacent cell. Cells which are reachable but not empty are saved for a second pass. After all empty reachable cells have potentials set, a second traverse starting with the saved list is used to determine potentials for all remaining cells. The potential starts at 0.0 in the destination with the per cell increase determined by the distance from the start to the destination. It is normalized to keep it between 0 and 1 for cells between the start and destination. Since each subdestination is both empty and reachable, the direction grid now provides a direct path to it. The local navigator can be just a route follower of a well marked route. Even in cases where backtracking through small openings which have been accidentally closed by sonar inaccuracies, the direction potentials unerringly lead Luey to the former opening. The change to the direction grid only impacts upon the local navigator. When determining subdestinations the ultimate destination is used as the destination point. Since it is normally unmapped, no path to it can be found and the distance based technique of section 3.1.2 is still used.

With the improvements to the destination grid, the elimination arrays were no longer needed to keep Luey moving so were eliminated themselves. Because error recovery is done using backtracking, the travel potentials are no longer needed in the badness potential. The travel grid is still used to determine if Luey is making no progress at the local level.

3.2.7 Performance

The backtrack navigator performed very well in the simulator[15]. It easily did every thing the original local navigator could do, plus found doorways out of rooms, traversed long halls, and successfully backtracked out of a zigzag corridor. It's real world performance; however, was again overshadowed by the sonars. False readings would encourage Luey to approach too closely to objects. Once there the sonar's low accuracy of very close objects often lead to a collision. The only navigation problems were a tendency to reenter areas which had supposedly been completely marked as dead ends, to quit prematurely, and trouble when the starting cell and best route were equidistant from the destination.

The first two usually occurred on long and complicated routes and were traced to the sonar inaccuracies. After an area was marked as a dead end, additional sonar readings would slightly change the object boundaries. Even though dead end marks went 1 deep into objects, often a new unoccupied cell appeared causing Luey to return to check it out. The premature quitting was not completely unexpected. In the test data used, the path to the destination often led down a long hallway or through narrow doorways. The combination of the grid decay and the spatial uncertainty of the sonars sometimes caused these to be blocked off. If this happened just as Luey was checking for a path to the destination, he would find none and quit.

Moving the dead end marking after the move to the checkpoint instead of before cleared up a lot of the problems and turning the decay procedure off fixed some more but the sonars still occasionly close off small openings causing Luey to give up.

The remaining problem is more serious. When the start cell is the same distance from the destination as the correct path, Luey tends to oscillate between the two. On each cycle he gets a little closer to the destination or eliminates a few more cells, but he spends a lot of unnecessary time going between the two locations. The same problem occurs with two apparently equal route choices. If none is closed off as an obvious dead end, he will try to follow both at once.

[15]Most testing was done with the simulator to be sure the navigator was being tested and not the sonar performance.

3.3 Path Navigator

The routine to determine if the destination was blocked was very similar to the one used to encode empty paths into the direction grid. If the former could find possible paths, (or their lack) why couldn't the direction grid be reconfigured to include not only known paths but also possible paths. This would help determine which of two apparently equal routes were better, the major problem with the backtrack navigator. A new navigator parameter, $PATH\_VALUE$, was introduced and used instead of THE to determine paths ending at the destination. This led to the *Path* navigator.

While originally considered as an improvement to the backtrack navigator, the changes to the direction potential made the checkpoint and backtrack system redundant. Since each new subdestination is always on a possible path to the destination, the criteria for backtracking were never met. Dead ends were eliminated automatically, negating the need for checkpoints, backtracking, and dead end marking. The path navigator takes this into account and gets rid of all the error recovery used by the backtrack navigator. At the basic level they are still the same as can be seen by setting the $PATH\_VALUE$ in the backtrack navigator to the value used by the path navigator. The only differences are that the backtrack navigator will occasionally backtrack through an accidentally closed off opening and continue while the path navigator will quit immediately, and the path navigator oscillates much less.

3.3.1 Direction Grid

The direction grid had to be further modified to allow for destinations off of the initial grid. Previously a destination off of the grid caused the distance based potentials to be used. Since the path navigator required the paths to be present this had to be changed. Any destination off of the grid is assumed to have some path leading to it from the closest grid edge. To implement this, the same breadth first traverse is used but instead of starting at the destination, all path cells along the grid edge are used as starting points. The adjacent cell potential increases are calculated the same way except that a maximum distance is imposed. Any destination off of the grid is treated as if it were 50 cells away from the edge.

3.3.2 Performance

In most cases the performance of the path navigator was essentially the same as the backtrack navigator. It also suffered from problems with sonar uncertainty and grid decay but for slightly different reasons. As previously mentioned it would quit prematurely when enclosed in an area which was accidentally closed off. While the backtrack navigator did this also it was not as common. Grid decay was a much larger problem. After entering and leaving an enclosed area, the walls would decay while Luey was off pursuing other avenues. If the wall decayed below the path threshold then Luey might return again. The backtrack navigator had a one cell cushion from the dead end marking and required decay to empty levels so was much less likely to return once an area was properly marked. Raising the path value would aid in the premature quit problem but worsen the decay problem. Of course turning off the decay effectively solved the decay problem and helped with the quitting problem.

A less serious problem which also occurred with the backtrack navigator is that Luey sometimes takes a less than direct route across empty space. This is due to two reasons. Because of the adjacent cell marking (as opposed to marking both adjacent and diagonal) in the construction of the direction grid, Luey tends to prefer directions which are multiples of 45 degrees. The second reason is that when choosing a new subdestination Luey has often not looked in the correct direction. Not having looked he couldn't see that the route is clear, and hence does not select that direction. While aesthetically displeasing, the deviations were not significant.

While the problems with indecision between two competing routes were reduced, they were not eliminated. Instead of following a route to its conclusion, Luey would turn and check out a route only marginally better. Until a route had been followed enough to be obviously better or worse, Luey would still try and follow two equal routes at once.

3.4 Incremental Navigator

Since encoding the path information in the direction grid allowed the removal of the error recovery code, perhaps it would also negate the need for subdestinations also. The *Incremental* navigator applies the new direction grid

directly to the local navigator bypassing the subdestinations entirely. The only question is how often to recalculate the direction grid. Ideally the local navigator would have the latest direction information every time it selected a move. This would mean redoing the direction grid every time the sonars were read as is done with the certainty and badness grids. This was deemed unnecessary because a single sonar reading does not usually provide significant global information. In fact, a single reading is more likely to provide erroneous data which would not be present after several readings. This is similar to the quitting problem with the backtrack and path navigators. The immediate need for the latest sonar data is in obstacle avoidance and this is already covered by the certainty grid. The path navigator redoes the grid at each checkpoint, but this interval is large enough to cause the direct path deviations mentioned in section 3.3.2. To have the least impact on Luey's thinking time, he recalculates the grid whenever he has already stopped. Since he stops all to frequently, the grid is recalculated often. When he does not stop it means that there are no obstacles or direction changes to stop for. A new direction grid is probably not needed under these circumstances.

3.4.1 Performance

The incremental navigator was at least the equal of the previous navigators. While slightly slower at its basic movement, it made up for it in its route selection. The oscillation problem was cleared up and even unanticipated side trips caused by obstacle decay were reduced. Instead of being indecisive about a route, Luey would now follow one until it was a dead end or worked out.

3.5 Navigator Summary

All the navigators suffered from the problems with the sensors. Because of the catastrophic effects of even small collisions, the local navigator was forced to be very cautious at the expense of performance. In the simulator, the navigators performed well with Luey avoiding obstacles, finding doors and reaching his destination without to many problems. In the real world however, small differences in badness potential often caused Luey to hesitate or avoid non-existent obstacles. One possible area of improvement would

be in the relative importances of the certainty and direction grids. Small differences in the certainty grid could be ignored but small differences in the direction grid were important.

In the global navigation contest, the incremental navigator came out on top. Not only was it simpler but it also used the best routes. The backtrack navigator's main recommendation is that it generates reproducible paths on the destination stack. Drawbacks included its complexity and indecision over competing routes. The path navigator fell between the two.

The global navigators were all affected by the decay of the certainty grid. Whenever the navigator was required to distinguish between two routes, the grid decay of a previously traveled area could cause new paths to open up. The backtrack and path navigators would immediately prefer them and even the incremental navigator was not completely immune to side trips. Lowering the path value eased this problem but caused other problems. Lower path values kept Luey from finding the best routes and led to inadvertent closings of small openings. Luey was much more likely to find the destination unreachable and quit. This was especially true when the decay was on and effectively negated any gain from a lower path value. The only workable solution was to turn off the decay procedure for any but the simplest routes. With the decay procedure off, performance was improved by raising the path value. This helped to keep accidental closures to a minimum. Below is a table of the path values used by the navigators.

Navigator	Min Path	Max Path	Comments
Local	THE	THO	
Backtrack	THE	< Background	Path navigator if above background
Path	Background	< THO	Above THO collisions likely
Increment	Background	< THO	Poor local navigator if 0.0 or 1.0

4 Driver

The driver contains the main procedure which ties the Cgrid, Oman and Display modules together. After an initialization procedure it runs a menu which allows the user to specify some options or have Luey do his thing. Information on starting the driver and setting up Luey can be found in the

README file or in appendix B. A short description of the main menu options follows:

0. Halt Program - Ends the program.

1. Set Parameters - Allows the user to set the navigation parameters. The most important are the sonar configuration, navigation method, and decay option. The sonar option tells Luey what configuration his sonars are in: front, square, off, or test. The front and square arrays are described in section 2.3.1. Test mode was used with the simulator to turn the sonars off but still simulate the time they require. The sonar simulator is recommended instead. The navigation method tells Luey which of his three methods to use: Local, Backtrack or Path. They are described in section 3. Setting the navigator will automatically change the parameters for the decay and the three thresholds: empty, occupied and path. The decay option is used to turn off the map decay as Luey moves. If the sonars are off, the decay should be turned off also. Changing the thresholds can have a significant effect on navigator performance as described in section 3. fine tuning navigator performance. The current parameter settings are shown in the top two lines of the curses display.

2. Find Background - Allows the user to set a new background or have Luey calculate it as described in section 2.1. All certainty values near the old background or between the empty and occupied threshold are set to the new background. To change all cells to the background use option 14 (reset). The current background value is shown on the top two lines of the curses display.

3. Read Sonars - Reads the sonars one time and updates the map accordingly.

4. Circle Scan - Does a circular scan of the robot's surroundings and updates the map accordingly. The user will be prompted for the angle between readings and the number of moves.

5. Rotate Luey - Rotates Luey a user specified number of degrees and updates his position. Positive degrees are counter-clockwise.

6. **Translate Luey** - Translates Luey a user specified number of centimeters and updates his position. He will travel backwards if the number of centimeters is negative.

7. **Set Cgrid** - Allows the user to set the certainty and confidence values of individual cells in Luey's map. The cell is highlighted with a red dot on the X display so the user can see where it is.

8. **Read Cgrid** - Reads in the certainty and confidence values from a user specified external file. Some sample files are provided with the names *.map .

9. **Read Sonar Map** - Reads in a sonar map from a user specified external file. This option is only available when in simulator mode. The format of the file is the same as that for loading the certainty grid so the same file may be used. Only certainties above the occupied threshold are significant. If the occupancy threshold is changed then the sonar map must be reread to be accurate.

10. **Query Grids** - Displays the values of the four potential grids, the sonar map and the dead end grid for a user specified cell. The cell is highlighted with a red dot on the X display so the user can see where it is.

11. **Get destination** - Allows the user to specify a destination for Luey. The destination is in grid coordinates although it may exceed the grid dimensions.

12. **Travel** - Runs the navigator routine. If a destination has not been specified then it will be prompted for. Once the travel routine starts, Luey is no longer under menu control until he reaches his destination or quits. Ctl C can be used to interrupt him and return control to the menu. This should be used sparingly as it can cause havoc if it occurs at the wrong time. The interrupt should cause the robot to immediately halt, but control only returns to the menu at the end of one travel cycle. This is to leave the navigator at a state which can be returned to. Section 3.2.5 describes what happens to th destination stack during a user abort. Menu option 13 should be used to return to the navigator following an interrupt.

13. **Continue** - Returns to the navigator routine and clears the abort condition caused by a user interrupt.

14. **Reset** - Restores the grids to their initial states. Luey's current position is the center of the grid and his current orientation the 0 degree heading. Any loaded sonar map is not reset nor are the navigation parameters changed.

15. **Update** - Updates Luey's position and reprints both displays.

16. **Save Cgrid** - Writes the current certainty and confidence grids to an external file in the format used by *Read Cgrid* and *Read Sonar Map.*

17. **Position Luey** - Allows the user to put Luey in a new location in his map. His angle and distance are both reset to 0.

18. **Test Speech** - Tests the speech routine. Enter a string and Luey or the terminal speech simulator will say it. Only strings of 14 characters or less can be input.

19. **Set Speech** - Allows the user to specify which speech option Luey is to use. The options are described in section 4.1. Having Luey speak is an interesting novelty but is not recommended because of the communication delay and a suspected bug in the speech software.

20. **Set Sonar Map** - Allows the user to change individual values in the sonar map. This option is only available when in simulator mode. The cell is highlighted with a red dot on the X display so the user can see where it is.

Default - Reprints the entire curses display and prompts for input.

4.1 Display

The display module consists of three parts: written output using the Curses screen handling package, pictorial output using an X window, and audio output using either the robot or the terminal speaker. Appendix A shows a sample of the curses and X displays.

The curses display is divided into 5 sections. The first two lines hold the current navigator settings. These can be changed by the user using options 1 and 2 in the menu. The next three lines hold the current robot values. These are updated automatically as the robot moves. The next two areas are the menu and the menu query area. The menu lists the options which the user may select while the query area is for more specific I/O requests. The remaining section is the log file area. Luey prints various information and debugging messages here as well as in the log file oman.log. When the 25 lines alloted to this area are full, the cursor moves to the top and writes over previous messages.

The pictorial display shows the certainty grid and badness grid maps in an X-window using gray scale to show occupancy probabilities and badness potential. Occupied/bad cells are shown in black and empty/good cells are shown in white. The certainty grid uses 32 levels of gray scale to cover its range of 0 - 1. Since the badness potentials have no upper limit, the same 32 levels are used to cover the range of 0 - 3. Any badness over 3 is shown as black. The badness display also uses some color to distinguish special cells. Any cell with a certainty over the occupied threshold is shown in purple and cells which have been marked as dead ends have an orange dot. The robot is shown as a green triangle pointing in the proper direction while the destinations are in pink(ultimate) and red(sub). In order to save time only the sections of the map near Luey are constantly displayed while he is in motion. Whenever Luey stops the entire map is redisplayed.

Limited additional information can also be output using the speech generator on the robot. This consists primarily of collision warnings and state of the navigator messages. Because of software problems with the robot speech routine it often interferes with more important robot commands and is not recommended. The terminal simulator should be used instead. In the simulator only specific things which have been prerecorded can be said, the rest are printed on the terminal and signaled with the bell. If the speech routine is turned off then all audio output will be printed in the menu query area and signaled by the bell.

4.2 Simulator

For test purposes a limited simulator is also available. The robot routines are replaced by dummies which calculate reasonable return values. The time values used for determining robot travel have been determined empirically. The sonar simulator uses a modified version of the sonar code to generate bearings and distances from a sonar map prepared by the user. The map must be prepared in advance and is limited to a single full grid. If the robot travels enough to require recentering, then the sonar simulator will be wrong. The format of the sonar file is described in appendix B. Choosing sonar array type 2 (test) will turn the sonar's off but still simulate the time they would require. Note that the sonar simulator does not suffer from the specular reflection problem that plagues the real sonars. This difference is what has allowed many of Luey's problems to be traced to the sonar sensors.

5 The Robot

Luey is one of two mobile robot platforms in the Brown AI lab built by Real World Interface.[9] (The other is Huey.) He has three wheels which turn at the same time so that his base angle and direction of travel are always the same (or 180 degrees apart since he can travel backward). Attached to the base is the sonar controller, an Arlan radio link, and a speech generator. He is controlled by a serial line going to an on-board NMI board containing a Motorola 6811 microcomputer and running Forth code. The serial line is connected to the Arlan radio link which goes to the Sun SPARC workstation *Tahoe* in the AI lab. The line can also be hardwired to the back of Tahoe but this limits Luey's movements to the length of the connecting cable. The NMI board controls the sonar and voice box while passing other commands on to the base.

The robot movement can be described by six parameters:

1. Location relative to some start location
2. Base orientation relative to some starting orientation
3. speed

4. acceleration

5. turning speed

6. turning acceleration

Luey keeps track of four of the six but ignores the two accelerations.

5.1 Sonars

Luey uses 8 Polaroid sonar transducers controlled by a Range Transducer Control Module board manufactured by Denning Mobile Robots. Control logic is provided by a Motorola 6811 microprocessor. The sonar's use a single threshold to register the closest object in an approximately 30 degree cone. An initial wait period sets their minimum distance at approximately 3 cm while a timeout on the return echo limits their range to 6 meters. The single threshold means that they are very vulnerable to errors from specular reflections, particularly those which do not return to the sonar. A more complete description of the hardware is in [5].

The sonars are arrayed on a square tower mounted on Luey's base as shown in fig. 4 in section 2. Sonars on the same corner are separated vertically so they do not physically interfere with each other. The tower is not centered on Luey's base but is instead located 4cm forward and 1/2 cm to the left. This necessitates a lot of extra calculation because each sonar must have it position determined independently, but allows the sonar to be relocated. The table below shows the location of each sonar relative to the robot center.

Sonar #	Location	Sonar angle deg	radians	Offset (base = 0 deg) x	y	r	Diff. from sonar angle deg	rad
1	Left rear	90	1.5708	-8.2	9.5	12.5	+40.80	+0.7121
7	Left forw	90	1.5708	15.8	9.5	18.4	-58.98	-1.0294
1,7	Left pair	90	1.5708	3.8	9.5	10.2	-21.80	-0.3805
0	Right rear	-90	-1.5708	-8.2	-8.5	11.8	-43.97	-0.7674
6	Right forw	-90	-1.5708	15.8	-8.5	17.9	+61.72	+1.0773
0,6	Right pair	-90	-1.5708	3.8	-8.5	9.3	+24.09	+0.4204
5	Front left	0	0.0000	13.3	12.0	17.9	+42.06	+0.7341
4	Front right	0	0.0000	13.3	-11.0	17.3	-39.59	-0.6910
5,4	Front pair	0	0.0000	13.3	0.5	13.3	+2.12	+0.0376
2	Back left	180	3.1416	-5.7	12.0	13.3	-64.59	-1.1273
3	Back right	180	3.1416	-5.7	-11.0	12.4	+62.61	+1.0927
2,3	Back pair	180	3.1416	-5.7	0.5	5.7	-5.01	-0.0875
3	Angle left	45	0.7853	11.6	11.2	16.1	-1.00	-0.0175
2	Angle right	-45	-0.7853	11.6	-10.2	15.4	+3.67	+0.0641

6 Discussion

Luey was able to perform as expected in open or uncluttered areas but suffered from major sonar problems in enclosed or dense areas. The problem of specular sonar reflections yielding false empty readings could not be completely overcome. While the certainty grid allowed the readings from all sonars to be combined to reduce random errors it was not able to eliminate systemic errors occurring in the sonars. These included specular reflections not returning to the sonar and erroneous readings of very close objects. In order to get the full benefits of the certainty grid, some additional types of sensors are needed. As a minimum, Luey needs a collision sensor or an accurate close range sensor covering his forward directions. These would mesh nicely with his sonars by filling in the holes left by their failings. In addition, he would be able to concentrate more on his route selection and less on collision avoidance. Increasing the number and separation of the sonars would have been useful as was evidenced by Luey's performance in uncluttered environments. His movement allowed readings from several directions which were combined into a fairly accurate map. In hallways and crowded areas; however, something was constantly in Luey's blind spots. These caused

collisions or degraded his performance as objects seemed to appear and disappear when he turned to avoid them. Because of the congestion, he was not able to move far enough away to resolve the map errors before running into trouble or encountering another discrepancy. Possible software solutions to these problems would have required Luey to move solely for the purpose of repositioning his sonars. This would have effectively killed any real time navigation. In addition, Luey already travels at a speed limited by his calculation and communication times; any extra movements (and the resulting communication delays) would further slow him.

Interestingly enough; Elfes [2, sec. 5.7] expected problems due to the intrinsic limitations of the sonar devices employed but did not encounter them. Perhaps his sonars were more sensitive. Less sensitive sonars would handle high-order specular reflections better but be more likely to return false negatives. This is supported by the results achieved here and by Tsai [12] at avoiding high-order reflections. Tsai presumably did not have the problem of false negatives causing problems due to his limited domain (hallways) and wall-follower navigator.

Some of the specific features of the project and how well they worked are described below:

Certainty Grid: The certainty grid worked as promised. It combined the various sonar readings into a recognizable map. The full power of the certainty grid was not used due to only a single sensor type, but it was easy to see how additional sensors could be added. Some additional work on determining the optimal cell size is indicated as testing did not provide conclusive evidence that Luey used the best size.

Background Determination: This worked fine but proved unnecessary. The wide angle sonars easily covered all nearby cells negating any need for an initial value. Any value below the path threshold seemed to work fine. The background value only became significant if the sonars were turned off and the preloaded map included only obstacles and not spaces. In this case a low background was needed in order to have a large potential change between occupied and unoccupied cells, and a small potential change between untraveled and traveled (known to be empty) cells.

Sonar Sensors: While I have complained about the sonars at almost every opportunity, they actually worked quite well. The problem was not so much the limitations of the sonars but the lack of an alternate sensor system to complement them. A previously undiscussed problem with the sonars was the real time spend getting a reading. After the communication and processing time was added to the sonar execution time, close to one second of real time had elapsed.

Paired Sonars: Pairing the sonars proved effective in increasing the sonar accuracy. Having the sonars in pairs allowed the mapper to narrow the possible locations of an object without sacrificing a lot of processor time. There was a loss of sonar readings from different directions but the map did not seem to be negatively affected. The large angle covered by each sonar and robot movement seemed to take care of this. If additional sonars had been available, the front angle sonars could probably have benefited from pairing, or the sonars could have been used as triples to further narrow the spatial uncertainty.

Regions of Constant Depth: The RCD approach was stretched a little beyond what it was capable of handling. I had hoped that they could overcome all of the sonar problems even though the RCDs were being generated without tracking. This proved to be to much. They were useful for determining confidences after multiple readings but could not cover up all the the sonar deficiencies. If the RCDs had been determined by a more accurate method which tracked and labeled objects, then they may have helped more.

Blind Spot Sonars (Front Array): There were mixed results from moving the rear sonar pair to the front corners. Because of their locations and being unpaired these sonars were very vulnerable to specular reflections not returning to the sonars. As mapping sonars, therefore they were almost useless. This was originally a problem until they were restricted to a small range and used only as a type of collision sensor. While not perfect at avoiding glancing collisions they were better than nothing. The loss of the data to the rear was usually not significant since the robot was leaving that area anyway.

Map Decay: The map decay worked all right but was of limited use. Near

the robot, the map decay was overwhelmed by new sonar information while far from the robot either the navigator didn't care what the map looked like or was negatively affected. The global portions of the navigator all depended on the persistence of objects, while the decay made them disappear over time. For long routes, the disappearing objects negated the route selection, causing unnecessary side trips and in some cases infinite oscillation between a correct path and a distant dead end. A completely different approach to the decay is needed. The current technique not only forgets where objects were over time, it also forgets that there were objects. A new decay must somehow preserve paths, obstacles and local relationships while still making their locations doubtful. In addition, the processor time used to calculate the decay was significant.

Navigator Decay The local decay done by the navigator was much more effective. Before implementation, Luey often approached too closely to objects or tried to squeeze through openings to small for him. By fuzzing the objects in his immediate vicinity prior to making navigation decisions, he became much more adept at selecting clear paths and maintaining a safe distance from obstacles.

Blank-Map Navigators This is discussed in section 3.5.

Elimination Arrays These worked but not as well as expected. In addition, they were only useful for improving the local navigator from a horrible global navigator to a very bad global navigator. On one hand, they allowed Luey to avoid simple dead ends but on the other they occasionly prevented him from turning to correct an accidental error. They are only used with the local navigator as it attempts to be a global navigator. When the local navigator is used as a route follower they are excluded.

Side Obstacle Extension The extension of side obstacles was needed because of glancing collisions which tended to occur in hallways. It was at least partially successful at avoiding these collisions but had the side effect of often causing Luey to make wild turns. Due to the granularity of the map, it was difficult to determine exactly how rapidly he was approaching a wall and the overly cautious Luey often overreacted. The

simple solutions would have been to have used the side sonars directly as a wall follower but this would have violated our rule of navigating only from the map.

Position Determination: The present method of position determination suffers from the errors described in section 3.1.6 as well as limiting the navigation options in order to minimize those errors. Because Luey determines his position solely from his movement, he must always know his speed, turning speed and direction.

Simulator The simulator worked better than anticipated. It allowed testing of features without the uncertainty imposed by the sonar inaccuracy, the time delays inherent in using the robot, or the problems of battery discharge. By the same token; however, it was difficult to determine actual robot performance from the simulator because the real world performance of Luey was so dependent on the sonars.

6.1 Future Work

Many interesting improvements suggested themselves during the project. Several of them are listed below:

Navigator controlled sonars One or two sonars under direct navigator control could be of immense value. They could allow tracking of an object for RCD determinations, or map matching and position determination. If nothing else they could be used to sense for impending collisions and avoid the object boundary uncertainties present when using the certainty grid.

Position verification from the map: To help minimize the error in Luey's position determination, he should be able to verify his position from a comparison of his map with what he sees. Some sort of map matching algorithm as in [8] would be needed.

Transient object tracking: If Luey was able to track and identify objects as in the previous two cases, he should also be able to distinguish transient objects. This would be very important if he was building a permanent map.

External map building: Once a robot has been through an unknown area, it should no longer be unknown. A major application of a blank map navigator is build a map for others. Luey's mapper could be extended so that the map is saved as it is constructed.

Fuzzy destinations: Because of the errors inherent in the position tracking by Luey the map is decayed as he moves. Nothing, however is done about the destination. There should be a way of fuzzing the destination to reflect the uncertainty of its position also. Another possibility is to allow a fuzzy destination in the first place. Instead of a specific spot, it would be nice to have a general area as the destination.

Destination determination: The current method of specifying a destination for Luey is very cumbersome and requires some domain knowledge on the part of the operator. In addition to the fuzzy destinations mentioned above, it would be useful to allow the operator to easily replace the current destination with a new one. A random or systemic search pattern could also be employed to have Luey map new areas.

Variable sized neighborhoods: When making decisions, Luey looks at only a fixed sized neighborhood. This could be extended so that he used different sized neighborhoods under different conditions.

Retracing previous routes: Since Luey keeps a record of where he has gone on his current route, it should be possible to have him retrace his path.

Path error recovery: While the backtrack navigator can be converted to the path navigator simply by changing the path threshold, the reverse is not true. The checkpoints generated are not reflective of the true path. The path navigator could be changed to monitor its subdestinations and remove dead ends from the stack yielding checkpoints for error recovery or reproducible routes.

Better interface: Replace the curses interface with an improved X one.

References

[1] Chang, Woong 0. *Management of a Certainty Grid.* Master's Thesis, Computer Science Dept., Brown University, Feb. 1988.

[2] Elfes, Alberto. *Occupancy Grids, A Probablistic Framework For Mobile Robot Perception and Navigation.* PhD Thesis, Electrical and Computer Engineering Dept./ Robotics Inst., Carnegie-Mellon University, 1989.

[3] Elfes, Alberto. "Using Occupancy Grids For Mobile Robot Perception and Navigation". *Computer,* Vol. 22, No. 8, June 1989.

[4] Koditsckek, Daniel E. *Robot Planning and Control Via Potential Functions.* Center for Systems Science, Yale University, Department of Electrical Engineering, June 1988.

[5] Lejter, Moises. *A Sonar Sensor for a Mobile Robot - (Draft).* Computer Science Dept., Brown University, May. 1989.

[6] Leonard, John J. & Durrant-Whyte Hugh F. *A Unified Approach to Mobile-Robot Navigation.* Department of Engineering Science, University of Oxford, Sept. 28 1989.

[7] Moravec, Hans P. "Certainty Grids for Mobile Robots". Robotics Institute, Carnegie Mellon University.

[8] Moravec, Hans P. & Elfes, Alberto. "High Resolution Maps from Wide Angle Sonar". *Proc. IEEE International Conference on Robotics and Automation,* pp. 116-121, March 1985.

[9] Real World Interface, *Mobile Robot Base B12 Guide to Operations.* Real World Interface P.O. Box 270, Dublin, NH 03444, 1988.

[10] Slack, Marc G. & Miller, David P. *Route Planning in a Four-Dimensional Environment.* Department of Computer Science, Virginia Tech, Blacksburg, VA 24061, 1986.

[11] Smith, Randall C. & Cheeseman, Peter. "On the Representation and Estimation of Spatial Uncertainty". *The International Journal of Robotics Research* Vol 5, Winter 1986.

[12] Tsai, Tu-Tsin. *An implementation of Certainty Grids For Mobile Robot Exempt From the Higher Order Echo Reflection.* Master's Thesis, Computer Science Dept., Brown University, Dec. 1990.

A Sample Output

- Curses Display.
- X Display with preloaded map of AI lab.
- X Display of simulator run on AI lab map.
- X Display of robot run in AI lab.

B Instructions

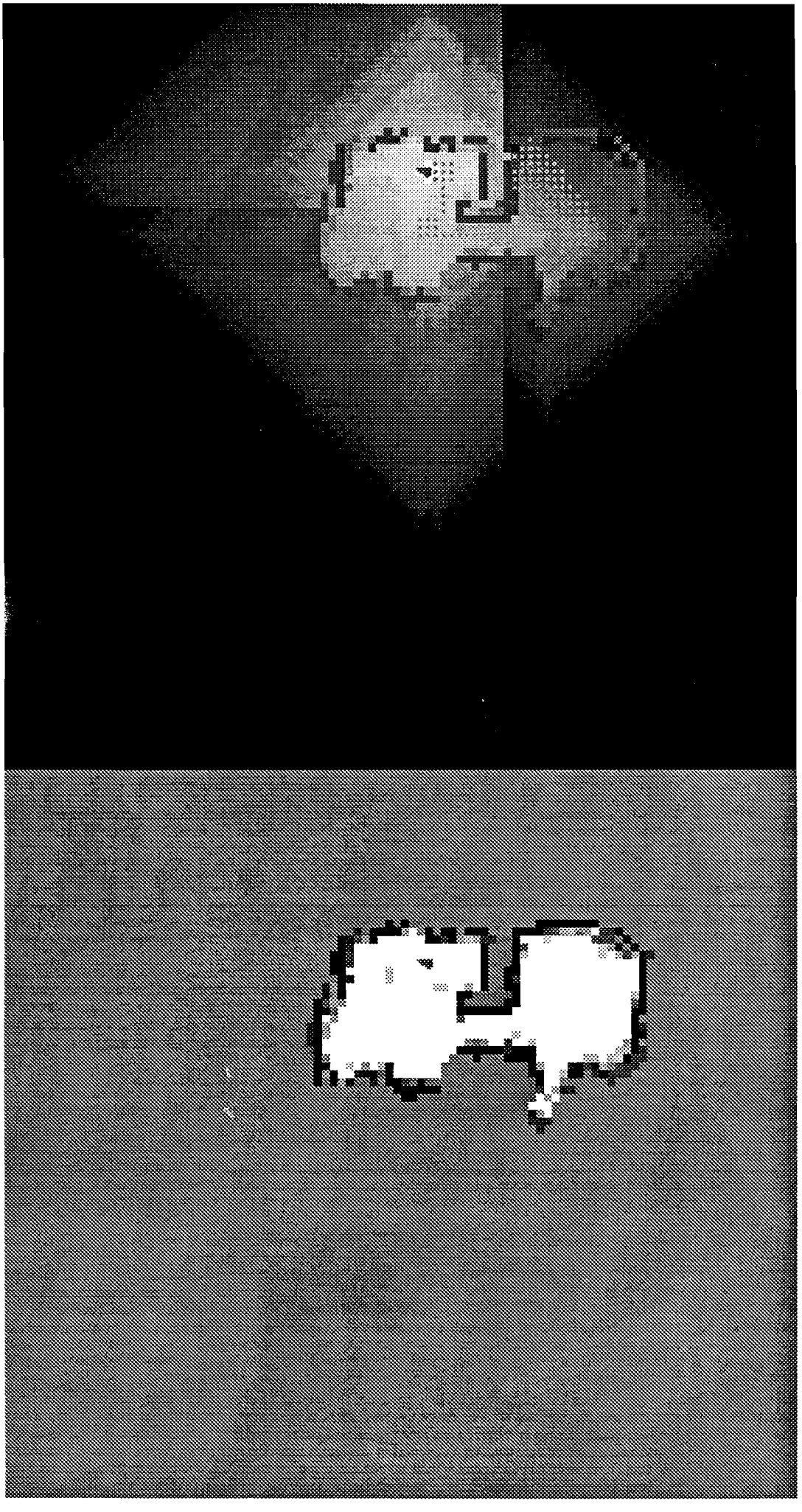
Attached is a copy of the file /u/ttg/omandir/README. It may no longer be up to date. See the README file in the directory where the program is stored for the latest information.

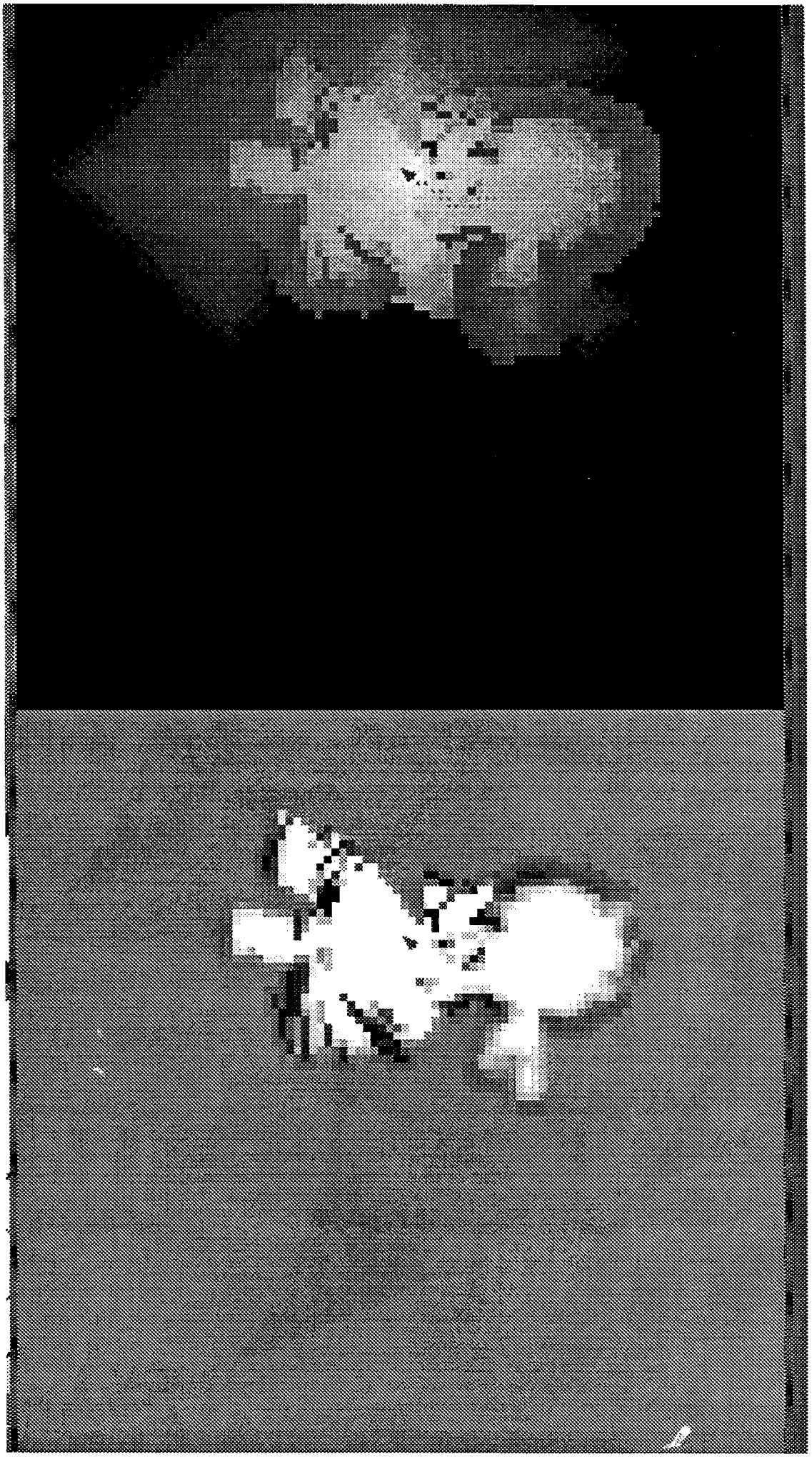
C List of Source Codes

- cgrid.h
- cgrid.C - Certainty Grid Module
- oman.h
- oman.C - Navigator Module
- display.h
- display.C - Display Module
- driver.C
- destack.h
- destack.C - Destination stack class
- RWITim.h

- RWITim.C - Robot Interface
- simulat.C - Simulated Robot Interface

```
Sonars: FRONT   Navigator: BACKTRACK  Decay: OFF
Empty:  0.25    Occupied:  0.75       Path value: 0.25  Background:  0.50

Position:   46, 24; Destination: 46, 24;  Subdest.: 46, 24  Checkpt: 46, 32
Bearing:   -91.3    Distance:   742.6     Voltage:  12.0
Slew:        0.0    Speed:        0       Desttype: SUBDEST

What do you want to do? 
 0: Halt Program    1: Set Parameters  2: Find Background 3: Read Sonars
 4: Circle Scan     5: Rotate Luey     6: Translate Luey  7: Set Cgrid
 8: Read Cgrid      9: Read Sonar Map 10: Query Grids    11: Set Destination
12: Travel         13: Continue       14: Reset          15: Update
16: Save Cgrid     17: Position Luey  18: Test Speech    19: Set Speech
20: Set Sonar Map

Luey has reached destination

Voice: destination


20 *TRAVEL: Robot has reached destination  46  24

22 Sonars read
23 SELECTMOVE: deg, base, estbase  -90.49  -91.26  -91.26
24 *FINDPOSITION: new cell 46  27  2  48
25 SELECTMOVE: deg, base, estbase  -90.12  -91.26  -91.26
26 *Speed changed to:   2.00
27 Sonars read
28 *FINDPOSITION: new cell 46  26  0  87
29 SELECTMOVE: deg, base, estbase  -90.00  -91.26  -91.26
30 SELECTMOVE: deg, base, estbase  -89.86  -91.26  -91.26
31 Sonars read
32 SELECTMOVE: deg, base, estbase  -89.64  -91.26  -91.26
33 Sonars read
34 SELECTMOVE: deg, base, estbase  -88.33  -91.26  -91.26
35 *FINDPOSITION: new cell 46  25  -5  63
36 SELECTMOVE: deg, base, estbase  -88.51  -91.26  -91.26
37 Sonars read
38 SELECTMOVE: deg, base, estbase  -88.44  -91.26  -91.26
39 Sonars read
40 *FINDPOSITION: new cell 46  24  -9  77
41 *STOPME: *****  46  24
42 checkpoint
43 *TRAVEL: Robot has reached sub-destination  46  24
44 destination
```





```
This directory contains stuff relating to Masters project

cgrid.C     has routines for handing a certainty grid
oman.C      has routines for navigator
driver.C    has the main routine and a menu for running cgrid and oman
display.C   has routines for X window display of certainty grid & curses
            display of operating data
simulat.C   has simulator routines to replace the robot ones so tests
            can be run without the robot. It replaces RWISimple.C,
            Sonar.C, Speech.C and RWIComplete.C but uses the same .h files
RWITim.C    has robot control routines, it is based on RWISimple.C in
            /pro/ai/robot/src but has additions and modifications
destack.C   has routines for manipulating a stack of destinations
*.map       contains a map which can be preloaded into the certainty grid or
            used with the sonar simulator, format is as follows:
            x y a.a b.b   x and y are integer grid coordinates (0-99)
                 ...      a.a is decimal certainty value (0.0 - 1.0)
                 ...      b.b is decimal confidence value (-6.0 - 6.0)
            999 999 0.0 0.0

docdir      is the directory with the documentation, to print it:
            dvips -Pfastdup oman.dvi

sounddir    is the directory with prerecorded speech for the speech simulator

make tester will compile with the simulator routines to creat testme
make        will compile with the RWITim.C robot routines to create runme

testme      is the simulator executable, it is run without the robot and
            simulates the robot responses,
runme       is the executable

***** TO RUN SIMULATOR ****************************************
start up X
(IF you wish to use the sonar simulator)
   create a fname.map file with the configuration you wish
   or use one of the test files.  ailab.map  maze.map  grid.map
   menu option  8 will allow you to view them
   menu option 14 will clear them
testme    to run the program
  menu option  1 to select navigation options
  menu option  9 to load the sonar map file
  menu option 19 to turn the sound on

***** TO RUN ROBOT*********************************************
------------Set up the environment-----------------------------
   copy .tiprc from /u/kjb if not present
   have /pro/ai/robot/bin in your path
   setenv RES_DIR '/u/jak/LIB/RES'
   setenv PARAM_DIR '/u/ttg/omandir/'
   start up X windows
   (If not using tahoe)  - must run on tahoe to use the serial ports
      xhost +            - to allow tahoe to write on your terminal
      rlogin tahoe       -
      setenv DISPLAY  termname:0.0    send tahoe X output to your terminal

------------Set up robot---------------------------------------
   check sonar configuration, for front configuration sonar 2 should be moved
      from back to the front below nos. 6 & 4 and sonar 3 below nos. 7 & 5.
   remove charger from robot
   turn on robot - Should say "OK" if speech is connected
   put switch on NMI box in NMI-BASE position
   check cable connections
   (IF using Arlan transmitters)
      connect Arlan cable on robot to NMI input
      connect I/O port 2 cable on main Arlan to rightmost serial port on Tahoe
      turn on Arlan transmitter box on robot
      turn on Arlan transmitter box by Tahoe
      turn on Arlan main box
      wait for connection indicator lights on both robot and base transmitters
      power cycle main Arlan box if connection not reached
   (ELSE IF hardwired)
     connect cable b (sonar) to NMI input on robot
     connect cable b (sonar) to rightmost serial port on back of Tahoe
   download  /pro/ai/robot/src/forth/one.forth
   -wait for download to complete
   tip hardwire
   -should return  connected
   hit return a few times until you receive OK
   type in  Multiplex
   -you should recieve  Ignored
   -if it hangs push the red button on the NMI box and type Multiplex again
   hit return a few times until you receive Ignored
   (IF rlogin to Tahoe)
     ~~.                        to exit tip
   (ELSE)
     ~.                         to exit tip
---------- Run the program ------------------------------------------------
   place robot in starting location
   runme                                 to run program
-----------Put everything away-----------------------------------------------
   turn off robot
   (If sonars in front position)
      return sonar 3 to left back
      return sonar 2 to right back
   (IF using Arlans)
     turn off Arlan on robot
     turn off Arlan transmitter
     turn off Arlan main box
   (ELSE IF hardwired)
      disconnect cable b from robot and tahoe
      reconnect Arlan cable on robot to NMI input
      reconnect I/O port 2 cable on main Arlan to right serial port on Tahoe
   put robot near charger
   plug charger into robot
   turn on charger
   push red button on robot (red charging light should come on)
```

```
/*****************************************************************************/
/*  cgrid.h  04/03/92  Timothy T. Good                                       */
/*    Contains constants and class declaration for the certainty grid module */
/*    The procedure definitions are in the file cgrid.C                      */
/*****************************************************************************/

#ifndef CGRID_H
#define CGRID_H

#include "RWITim.h"
#include <Sonar.h>            /* /pro/ai/robot/include/Sonar.h */

const int   SZE         = 100;   /* Size of occupancy grid                  */
const int   CELLCM      = 20;    /* Occupancy grid cell size in cm          */
const int   CELL    = 10 * CELLCM;/* Occupancy grid cell size in mm         */
const int   SONAR_RANGE = 5990;  /* Distance(mm) over which sonar can operate*/

enum sonarType { S_OFF, S_SQUARE, S_TEST, S_FRONT};

/*****************************************************************************/
/* Certgrid is used to create and maintain a certainty grid map.             */
/*****************************************************************************/

class Certgrid {
 public:
   Certgrid(Sonar *spt, RWITim *rpt, int xstart, int ystart);
   ~Certgrid();
   /************************************************************************/
   /* The next 7 functions are inline functions used to access some of   */
   /* the private data directly.                                         */
   /* The 4 setXxxxx functions are provided for test use. They should    */
   /* not be used in general.                                            */
   /************************************************************************/
   inline void  setBackgd(float bg)            { back_grnd = edge_bg = bg; }
   inline float getBackgd(void)                { return(back_grnd); }
   inline void  setSonarType(sonarType stype)  { sonar_type = stype; }
   inline sonarType getSonarType(void)         { return(sonar_type); }
   inline float getCg(int x, int y)            { return(cgrid[x][y]); }
   inline float getConfi(int x, int y)         { return(confi[x][y]); }
   inline void  setGrid(int x, int y, float cert, float conf)
                        { cgrid[x][y] = cert; confi[x][y] = conf; }
   void  cgridInit(int x,int y, float bsedist, short clear);
         /* Initialize certainty grid to background, confidences & RCDs to 0 */
   int   cgridRead(char *fn);
         /* Reads external file to initialize certainty and confidence       */
   void  circleScan(float baseAngle,int increment,int number,int robx,int roby,
                    int offx,int offy,int retflg);
         /* Takes multiple sonar readings while rotating, updates grids      */
   short decay(short decay_flag, float distance);
         /* Calculates weights for grid fuzzing,  fuzzes the grids           */
   float findBg(float baseangle,float newbg ,int robx,int roby,int offx,
                int offy);
         /* Determines local background from surroundings                    */
   short queryAtBgnd(int x, int y);
         /* Determine if a cell is near the background certainty             */
   int   saveMap(char *fnam);
         /* Outputs certainty grid and confidences to file                   */
   void  setCgrid(int x, int y, float cert, float conf);
         /* Sets certainty and confidence grid values using confidences      */
   void  shift(int dx, int dy);
         /* Shifts certainty & confidence grids by x & y cell displacement   */
   void  sonarScan(float baseAngle,int robx, int roby, int offx, int offy);
         /* Does a complete scan using 8 sonars, updates grids               */

 private:

   float cgrid[SZE][SZE]; /* Certainty grid of robot's surroundings          */
   float confi[SZE][SZE]; /* Confidence of certainty values                  */
   float back_grnd;       /* Background certainty value                      */
   float edge_bg;         /* Edge certainty value                            */
   int   loc_robx;        /* Local copy of robot x grid coordinate           */
   int   loc_roby;        /* Local copy of robot y grid coordinate           */
   RWITim *RwitPt;        /* Pointer to RWITim class w/robot routines        */
   int   s_mean[8];       /* Mean of sonar readings for current RCD          */
   int   s_num[8];        /* Number of sonar readings in current RCD         */
   int   s_var[8];        /* Variance of sonar readings in current RCD       */
   Sonar *SonarPt;        /* Pointer to sonar class with sonar routines      */
   sonarType sonar_type;  /* Sonar arrangement (S_OFF,S_SQUARE,S_TEST,S_FRONT*/

   void  newGrid(float weight);
         /* changes certainty & confidence to weighted ave. of neighbors     */
   void  rcdUpdate(int sonar,int range);
         /* Checks for membership in current RCD, updates factors            */
   void  sonarMarkObject(int xr, int yr, int xl, int yl,float cert,float conf);
         /* Marks an occupied area in the certainty grid                     */
   void  sonarScan1(int dist, int angle, float baseAngle, int offx, int offy,
                    int spread, float cert, int conf, int objflg);
         /* Updates certainty grid with 1 sonar reading                      */
   void  sonarInit(void);
         /* Initializes mean,number & variance of sonar RCDs                 */
   void  sonarPair(int s1, int s2, int r1, int r2, int ang,
                   float baseAngle, int offx, int offy);
         /* Combines range readings from a pair of sonars                    */
} ;
#endif  /* CGRID_H */
```

```
/*****************************************************************************/
/*      CGRID.C   04/03/92  Timothy T. Good                                  */
/*    This file contains the procedures necessary to build and maintain      */
/*    a certainty grid on the robot Luey in the Brown AI lab.  Except for    */
/*    some inline utilities these procedures are part of the Certgrid class  */
/*    declared in cgrid.h .                                                  */
/*****************************************************************************/

#define CGRID_C

#include <stdio.h>
#include <math.h>
#include "cgrid.h"
#include "RWITim.h"        /* Robot control routines */
#include "Sonar.h"         /* Robot sonar routines   */

/*****************************************************************************/
/* Constants:                                                                */
/*****************************************************************************/

#define  OK    1
#define  ERR   0
#define  TRUE  1
#define  FALSE 0
static const float CB      =  0.5;     /* Default background occupancy value */
static const float HIGHO   =  1.0;     /* High occupancy probability         */
static const float MEDO    =  0.75;    /* Medium occupancy probability       */
static const float LOWO    =  0.62;    /* Low occupancy probabilty           */
static const float HIGHE   =  0.0;     /* High empty probability             */
static const float MEDE    =  0.25;    /* Medium empty probality             */
static const float LOWE    =  0.38;    /* Low empty probablity               */
static const int   NEAR    = 500;      /* Dist. (mm) for objects to be near  */
static const float MAXCONF =   6.0;    /* Maximum confidence value           */
static const int   THRESH  = CELL;     /* Threshold for sonar agreement(mm)  */
static const float DECAYV  =  0.05;    /* Decay weight for 1 cell of movement*/
static const int   HALFCELL = CELL/2;  /* Half the cell size in mm           */

/*****************************************************************************/
/* Global data:                                                              */
/*    All local global data is declared in the private section of the class  */
/*    declaration found in cgrid.h . There is no external global data.       */
/*****************************************************************************/

/* float      back_grnd;       Background certainty value                    */
/* float      cgrid[SZE][SZE]; Certainty grid of robot's surroundings        */
/* float      confi[SZE][SZE]; Confidence of certainty values                */
/* float      edge_bg;         Edge certainty value                          */
/* int        loc_robx;        Local copy of robot x grid coordinate         */
/* int        loc_roby;        Local copy of robot y grid coordinate         */
/* int        s_mean[8];       Mean of sonar readings for current RCD        */
/* int        s_num[8];        Number of sonar readings in current RCD       */
/* int        s_var[8];        Variance of sonar readings in current RCD     */
/* RWITim    *RwitPt;          Pointer to RWITim class w/robot routs.        */
/* sonarType  sonar_type;      Sonar array (S_OFF,S_SQUARE,S_TEST,S_FRONT)   */
/* Sonar     *SonarPt;         Pointer to sonar class with sonar routines    */
```

```
/*****************************************************************************/
/* Procedures:                                                               */
/*    Declarations are in the Certgrid class declaration found in cgrid.h    */
/*    After the inline, constuctor,and destructor functions they are defined */
/*    in alphabetical order.                                                 */
/*****************************************************************************/

   inline float RADIAN(float degrees);/* Converts degrees to radians - local */
   inline int   ROUND(float fval);    /* Rounds floats to integers   - local */
          int   CEILING(float fval);  /* Rounds floats to larger int - local */

/* inline void  setBackgd(float bg)      Sets background              - public*/
/* inline float getBackgd(void)          Returns background           - public*/
/* inline void  setSonarType(sonarType stype) Sets sonar arrangement - public*/
/* inline sonarType getSonarType(void)   Returns sonar arrangement    - public*/
/* inline float getCg(int x, int y)      Returns certainty value      - public*/
/* inline float getConfi(int x, int y)   Returns confidence value     - public*/
/* inline void  setGrid(i,i,f,f)         Sets certainty & confidence - public*/

/* Certgrid(Sonar,RWITim,int,int)        Constructor for Certgrid     - public*/
/* ~Certgrid()                           Destructor for Certgrid      - public*/

/* void  cgridInit(i,i,f,s)              Initializes grids & RCDs     - public*/
/* int   cgridRead(char *fn)             Reads grids from a file      - public*/
/* void  circleScan(f,i,i,i,i,i,i,i)     Rotates and scans            - public*/
/* short decay(s,f)                      Fuzzes the certainty grid    - publie*/
/* float findBg(f,f,i,i,i,i)             Determines local background - public*/
/* void  newGrid(float weight)           Fuzzes the grids             - priv  */
/* short queryAtBgnd(int x, int y)       Is a cell near background    - public*/
/* void  rcdUpdate(int sonar,int range)  Checks & updates RCDs        - priv  */
/* int   saveMap(char *fnam)             Outputs grids to a file      - public*/
/* void  setCgrid(i,i,f,f)               Sets certainty & confidence - public*/
/* void  shift(int dx, int dy)           Shifts grids                 - public*/
/* void  sonarInit(void)                 Initializes RCD factors      - priv  */
/* void  sonarMarkObject(i,i,i,i,f,f)    Marks an occupied area       - priv  */
/* void  sonarPair(i,i,i,i,i,f,i,i)      Combines 2 sonar readings    - priv  */
/* void  sonarScan(f,i,i,i,i)            Reads sonars & updates grids- public*/
/* void  sonarScan1(i,i,f,i,i,i,f,i,i)   Updates grids with 1 sonar   - priv  */
```

```
, ************************************************************************/
/* Procedure: Certgrid                                                   */
/* Called by: <external>                                                 */
/* Calls:     <none>                                                     */
/*    This is the constructor for the certainty grid class Certgrid. It  */
/*    does some initialization and returns.                              */
/************************************************************************/
Certgrid::Certgrid(Sonar *spt, RWITim *rpt, int xstart, int ystart)
{
 SonarPt = spt;
 RwitPt = rpt;
 back_grnd = CB;
 edge_bg = CB;
 sonar_type = S_FRONT;
 cgridInit(xstart,ystart,0.0,TRUE);
}
/************************************************************************/
/* Procedure: ~Certgrid                                                  */
/* Called by: <external>                                                 */
/* Calls:     <none>                                                     */
/*    This is the destructor for the certainty grid class Certgrid. It does  */
/*    nothing.                                                           */
/************************************************************************/
Certgrid::~Certgrid()
{}
/************************************************************************/
/* Procedure: RADIAN                                                     */
/* Calls:     <none>                                                     */
/* Returns:   float - angle in radians                                   */
/************************************************************************/
inline float RADIAN(float degrees)
{
 return(degrees * 0.017453293);     /* degrees * pi / 180.0 */
}
/************************************************************************/
/* Procedure: ROUND                                                      */
/* Calls:     <none>                                                     */
/* Returns:   int - agument rounded to nearest integer                   */
/************************************************************************/
int ROUND(float fval)
{
 return   (  (fval >= 0)  ? (int)((fval) + 0.5)  : (int)((fval) - 0.5)  );
}
/************************************************************************/
/* Procedure: CEILING                                                    */
/* Calls:     <none>                                                     */
/* Returns:   int - argument rounded to integer with larger absolute value   */
/************************************************************************/
int CEILING(float fval)
{
 return (  (fval >= 0) ? (int)(ceil(fval)) : (int)(floor(fval))  );
}
```

```
/************************************************************************/
/* Procedure: setBackgd(float bg),         getBackgd(void)               */
/* Procedure: setSonarType(sonarType st),  getSonarType(void)            */
/* Procedure: setGrid(int x, int y, float cert, float conf)              */
/* Procedure:                              getCg(int x, int y)           */
/* Procedure:                              getConfi(int x, int y)        */
/*    These are inline functions which allow access to the private variables */
/*    back_grnd, edge_bg, sonar_type, cgrid[][], confi[][]. They are defined */
/*    in the class declaration in cgrid.h                                */
/*  **Warning: setxxxx is available only to allow testing from the driver,   */
/*    they are not intended for general use. No checking is done as in the   */
/*    the normal function setcgrid defined later.                        */
/************************************************************************/

/************************************************************************/
/* Procedure: cgridInit                                          Done    */
/* Called by: findBg, <external>                                         */
/* Calls:     sonarInit - initializes sonar factors for finding RCDs     */
/* Globals:   cgrid, confi  (Write only)                                 */
/*   Initializes the certainty grid to the background value and sets     */
/*   confidence factors to 0. (Except for the robots position which is   */
/*   empty with high confidence) The RCD factors and the decay accumulator   */
/*   are also reset.                                                     */
/************************************************************************/
void Certgrid::cgridInit(
   int   xstart, int ystart,  /* robots starting position               */
   float base_dis,            /* Starting distance traveled (cm)        */
   short clear)               /* Flag for reset cgrid to background     */
{
 int   i,j;              /* Loop indices                                */
 if (clear) {
    for (i=0; i<SZE; i++) {
       for (j=0; j<SZE; j++) {
          cgrid[i][j] = back_grnd;     /* Set grid values to background  */
          confi[i][j] = 0;             /* Set confidence to 0            */
       }
    }
 }
 decay(0,base_dis);                /* Initialize decay distance         */
 sonarInit();                      /* Initialize RCD factors for sonars */
 if ((xstart < 0)||(xstart >= SZE)||(ystart < 0)||(ystart >= SZE)) return;
 cgrid[xstart][ystart] = 0.0;      /* Cell containing robot is empty    */
 confi[xstart][ystart] = -MAXCONF; /* With maximum confidence           */
} /** end cgridInit() **/
```

```
/*****************************************************************************/
/* Procedure: cgridRead                                                 Done */
/* Called by: <external>                                                     */
/* Calls:     <none>                                                         */
/* Returns:   FALSE if file could not be opened, or TRUE                     */
/* Globals:   cgrid, confi (Write only)                                      */
/* Files:     external file  read only - initialization data                 */
/*    Reads an external file to initialize the certainty and confidence      */
/*    grids. They are unchanged if the file cannot be opened. If the filename*/
/*    starts with 0 then the certainty grid is set to the background.        */
/*****************************************************************************/
int Certgrid::cgridRead(char *fname)
{
 int   i,j;              /* Indices to array                                 */
 float cert, conf;       /* Certainty and confidence values read from file   */
 FILE *fpin;             /* Input file pointer,                              */
 if (fname[0] == '0') {
    for (i=0; i<SZE; i++) {
       for (j=0; j<SZE; j++) {
          cgrid[i][j] = back_grnd;     /* Set grid values to background      */
          confi[i][j] = 0;             /* Set confidence to 0                */
       }
    }
    return(TRUE);
 }
/***** Read in initialization values                                  *****/
 fpin = fopen(fname,"r");
 if (fpin == NULL)  return(FALSE);
 fscanf(fpin,"%3d %3d %f %f",&i,&j,&cert,&conf);
 while (i != 999) {
    if ((i<SZE) && (j<SZE) && (i>=0) && (j>=0)) {
       cgrid[i][j] = cert;
       confi[i][j] = conf;
    }
    fscanf(fpin,"%3d %3d %f %f",&i,&j,&cert,&conf);
 }
 fclose(fpin);
 return(TRUE);
} /** end cgridRead() **/
```

```
/*****************************************************************************/
/* Procedure: circleScan                                                Done */
/* Called by: (driver)test1, findBg, (oman)travel_aux                        */
/* Calls:     RwitPt->executeTurn - turns the robot                          */
/* Globals:   sonarScan - does a scan using all 8 sonars                     */
/*    Takes sonar readings at a number of fixed degree increments of rotation*/
/*    by the robot. If the return flag is set then the robot is returned to  */
/*    its original bearing.                                                  */
/*****************************************************************************/
void Certgrid::circleScan(
   float base_ang,          /* Robot's current direction                     */
   int   increment,         /* No. of degrees between sonar readings         */
   int   number,            /* No. of increments to move                     */
   int   robx, int roby,    /* Robot's current cell                          */
   int   offx, int offy,    /* Distances from robot center to cell center    */
   int   retflg)            /* Flag for return (1) or keep new angle (0)     */
{
 int i;                     /* Loop index                                    */
 int degrees = 0;           /* Counts total degrees moved                    */
 if (sonar_type == S_OFF) return;
 if (number <= 0) return;                    /* If bad values return       */
 if ((increment>90)||(increment < -90)) return; /*  without scanning         */
 if ((increment * number) > 360) return;
 sonarScan(base_ang,robx,roby,offx,offy);
 for (i=1; i<=number; i++) {
    RwitPt->executeTurn((float)(-increment),15.0,1);
    degrees = degrees + increment;
    sonarScan(base_ang+(float)degrees,robx,roby,offx,offy);
 }
 if (retflg == TRUE) RwitPt->executeTurn((float)degrees,15.0,1);
} /** end circleScan() **/
```

```
/*****************************************************************************/
/* Procedure: decay                                                     Done */
/* Called by: <external>                                                     */
/* Calls:     newGrid - Does the actual decay                                */
/* Returns:   OK or ERR if movement since last decay exceeded 3 cells        */
/*   Calculates the weighting value to be used when decaying the certainty   */
/*   grid. The weight ranges from 1.0 when the robot has not moved to 0.9    */
/*   when it has moved an entire CELL distance. If the robot has moved over  */
/*   3 cells since the last decay, no decay is done and an error(0) returned.*/
/*****************************************************************************/
short Certgrid::decay(
   short decay_flag,     /* Flag for decay turned off (0) or on (1)          */
   float distance)       /* Current distance (cm) traveled by Luey           */
{
 float weight;           /* Weighting value for center cell                  */
 float dist,dist2;       /* Distance(cm) traveled since last decay           */
 static float lastdist =0.0; /* Distance (cm) robot has traveled             */
 dist = abs(distance - lastdist);
 lastdist = distance;
 if (decay_flag == FALSE) return(OK);     /* If decay off, dont calculate   */
 dist2 = ((int)dist > (3*CELLCM)) ? (float)(3*CELLCM) : dist ;
 weight = 1.0 - (DECAYV * dist2 / (float)CELLCM);
 newGrid(weight);
 if (dist > (float)(3*CELLCM)) return(ERR);
 return(OK);
} /** end decay() **/
```

```
/*************************************************************************/
/* Procedure: findBg                                              Done */
/* Called by: (driver)test1                                            */
/* Calls:     cgridInit  - Sets certainty grid to background value     */
/*            circleScan - Take multiple sonar scans while rotating    */
/*            newGrid  - Decays the grid                               */
/* Returns:   float - Local background occupancy value                 */
/* Globals:   cgrid (Write only) back_grnd (Read only)                 */
/*    Sets a user defined background or calculates a new one if the user one */
/*    is out of the range.  (0.0 - 1.0)                                */
/*    Calculates the local occupancy background level. Takes a complete scan */
/*    of the area, runs through 7 decay cycles, and takes the average of the */
/*    certainty values for the area around the robot. The area is the square */
/*    with sides 3/4 the length of the SONAR_RANGE.  If the sonars are turned*/
/*    off then it will not calculate a new background.                 */
/*************************************************************************/
float Certgrid::findBg(
   float base_ang,       /* Robot's current direction                  */
   float newbg,          /* New background to use                      */
   int   robx, int roby, /* Robot's current cell                       */
   int   offx, int offy) /* X,Y distances(mm) from cell to robot center */
{
 const int REACH = (int)((3 * SONAR_RANGE)/(CELL * 8));
 float bgnd = 0.0;                    /* New background value returned */
 int   i,j;                           /* Loop indices                  */
 int   counter = 0;                   /* Counts cells summed           */
 if ((newbg < 0.0) || (newbg > 1.0)) {
    if ((sonar_type == S_OFF) || (sonar_type == S_TEST)) return(back_grnd);
    circleScan(base_ang,15,23,robx,roby,offx,offy,1);
    for (i=1; i<=7; i++) newGrid(0.9);
    for (i=robx-REACH;i<=robx+REACH;i++) {
       if ((i >= 0) && (i < SZE)) {
          for (j=roby-REACH;j<=roby+REACH;j++) {
             if ((j >= 0) && (j < SZE))  {
                bgnd = bgnd + cgrid[i][j];
                counter++;
             }
          }
       }
    }
    bgnd = bgnd / (float)counter ;
 } else bgnd = newbg;
 for (i=0; i<SZE; i++) {
    for (j=0; j<SZE; j++) {
       if (((cgrid[i][j] < MEDO) && (cgrid[i][j] > MEDE)) ||
           (cgrid[i][j] == back_grnd)) {
         cgrid[i][j] = bgnd;
         confi[i][j] = 0.0;
       }
    }
 }
 back_grnd = bgnd;
 edge_bg = bgnd;
 return(bgnd);
} /** end findBg() **/
```

```
/*************************************************************************/
/* Procedure: newGrid                                             Done */
/* Called by: decay, findBg                                            */
/* Calls:     <none>                                                   */
/* Globals:   cgrid - the certainty grid                               */
/*   Decays the certainty grid to the background value. The decay fuzzes     */
/*   the certainty grid to represent inaccuracies in the robot drive systems.*/
/*   Changes the certainty grid values by taking a weighted average of the   */
/*   current cell with a weighted average of its 8 neighbors.  The neighbors */
/*   are averaged using modified confidence values as their weights so that  */
/*   cells of high confidence will have a greater effect. The confidence     */
/*   values are modified by taking the absolute value and adding 1. This     */
/*   insures that all neighbors will have some effect.                       */
/*   This is a time consuming procedure and should not be called during      */
/*   crucial robot movements.                                                */
/*************************************************************************/
void Certgrid::newGrid(
   float weight)         /* weight value (based on distance moved)        */
{
 float curr[SZE];        /* current row of weighted cgrid values pre update  */
 float ccon[SZE];        /* current row of modified confidences pre update   */
 float conchg;           /* change in confidence value for current cell      */
 float csum;             /* sum of modified confidence values for neighbors  */
 float next[SZE];        /* next row of weighted cgrid values pre update     */
 float ncon[SZE];        /* next row or modified confidences pre update      */
 float prev[SZE];        /* previous row of weighted cgrid values pre update */
 float pcon[SZE];        /* previous row of modified confidences pre-update  */
 float nbors;            /* sum of weighted cgrid values for 8 neighbor cells*/
 float weight2;          /* weighting value for neighbors (1-weight)         */
 float wgtnbor;          /* weighted average of neighbor cells               */
 int x,y;                /* array indices, x and y coords in cgrid           */
 int ym,xm,xp;           /* y-1, x-1, x+1, to save multiple recalculations   */
 if (weight == 0.0) return;
 weight2 = 1.0 - weight;
 y = SZE-1;
 ym = SZE -2;
 for (x=0; x<=SZE-1; x++) {      /* initialize previous, current and next row*/
    prev[x] = edge_bg;           /* initialize upper edge to background      */
    pcon[x] = 1;                 /* modified edge confidences are 1          */
    ccon[x] = (float)fabs((double)confi[x][y]) + 1.0; /* modify conf. values */
    curr[x] = cgrid[x][y] * ccon[x];        /* calc. weighted cert. values   */
    ncon[x] = (float)fabs((double)confi[x][ym]) + 1.0;   /* calc. next row   */
    next[x] = cgrid[x][ym] * ncon[x];       /* calc. weighted cert. values   */

 }
 for (y=SZE-1; y>=0; y--) {
/***** Do the first one in the row  (x=0) ********************************/
    nbors = (3.0 * edge_bg) + prev[0] + prev[1] + curr[1] + next[0] + next[1];
    csum = pcon[0] + pcon[1] + ccon[1] + ncon[0] + ncon[1] + 3.0;
    wgtnbor = nbors / csum;
    conchg = cgrid[0][y] - wgtnbor;
    conchg = (float)fabs((double)conchg);
    if (confi[0][y] > conchg ) confi[0][y] = confi[0][y] - conchg;
    else if (confi[0][y] < -conchg) confi[0][y] = confi[0][y] + conchg;
    else confi[0][y] = 0.0;
    cgrid[0][y] = weight * cgrid[0][y] + (weight2 * wgtnbor);
/***** Do rest of row except for last element ****************************/
    for (x=1; x<SZE-1; x++) {
       xm = x - 1;
       xp = x + 1;
       nbors =  prev[xm] + prev[x] + prev[xp] + curr[xm] + curr[xp] +
                next[xm] + next[x] + next[xp];
       csum = pcon[xm] + pcon[x] + pcon[xp] + ccon[xm] + ccon[xp] +
```

```
            ncon[xm] + ncon[x] + ncon[xp];
      wgtnbor = nbors / csum;
      conchg = cgrid[x][y] - wgtnbor;
      conchg = (float)fabs((double)conchg);
      if (confi[x][y] > conchg ) confi[x][y] = confi[x][y] - conchg;
      else if (confi[x][y] < -conchg) confi[x][y] = confi[x][y] + conchg;
      else confi[x][y] = 0.0;
      cgrid[x][y] = (weight * cgrid[x][y]) + (weight2 * wgtnbor);
   }
/***** Do the last one in the row (x=SZE-1) ********************************/
   x = SZE-1;
   xm = SZE-2;
   nbors = (3.0 * edge_bg) + prev[xm] + prev[x] + curr[xm] +
                             next[xm] + next[x];
   csum = pcon[xm] + pcon[x] + ccon[xm] + ncon[xm] + ncon[x] + 3.0;
   wgtnbor = nbors / csum;
   conchg = cgrid[x][y] - wgtnbor;
   conchg = (float)fabs((double)conchg);
   if (confi[x][y] > conchg ) confi[x][y] = confi[x][y] - conchg;
   else if (confi[x][y] < -conchg)
       confi[x][y] = confi[x][y] + conchg;
   else confi[x][y] = 0.0;
   cgrid[x][y] = weight * cgrid[x][y] + (weight2 * wgtnbor);
/***** Set up rows for next iteration **************************************/
   ym = y-2;
   for (x=0; x<SZE; x++) {
      prev[x] = curr[x];                    /* previous row = current row    */
      pcon[x] = ccon[x];
      ccon[x] = ncon[x];                    /* current row = next row        */
      curr[x] = next[x];
      if (y > 1) {
         ncon[x] = (float)fabs((double)confi[x][ym]) + 1.0;/* calc. next row*/
         next[x] = cgrid[x][ym] * ncon[x];    /* calc. weighted cert. values*/
      } else {
        next[x] = edge_bg;
        ncon[x] = 1;
      }
   }
  }
} /** end newGrid() **/
```

```
/*****************************************************************************/
/* Procedure: queryAtBgnd                                                Done */
/* Called by: <external>                                                     */
/* Calls:     <none>                                                         */
/* Globals:   cgrid, confi, back_grnd (Read only)                            */
/* Returns:   short   TRUE if near background, FALSE if not                  */
/*    Checks the indicated cell. If the certainty is close to the background */
/*    level and the confidence level is near zero, then the cell is assumed  */
/*    to be at background.                                                   */
/*****************************************************************************/
short Certgrid::queryAtBgnd(
   int x, int y)                    /* Coordinates of cell */
{
 if ((cgrid[x][y] > (back_grnd - 0.1))&& (cgrid[x][y] < (back_grnd + 0.1))) {
    if ((confi[x][y] >= -1.0) && (confi[x][y] <= 1.0))
       return(TRUE);
 }
 return(FALSE);
} /** end queryAtBgnd() **/

/*****************************************************************************/
/* Procedure: rcdUpdate                                                  Done */
/* Called by: sonarScan                                                      */
/* Calls:     <none>                                                         */
/* Globals:   s_num,s_mean,s_var (READ, WRITE)                               */
/*    Determines if the latest sonar range reading belongs in the current    */
/*    RCD. Updates the RCD accordingly or begins a new one.  Since there is  */
/*    an arbitrary confidence limit of MAXCONF, sonar factors are not        */
/*    recalculated once an RCD has MAXCONF readings in it.                   */
/*****************************************************************************/
void Certgrid::rcdUpdate(
   int sonar,                /* Sonar taking the reading                     */
   int range)                /* Distance (mm) to object                      */
{
 float meanf;                /* Mean of range readings for current RCD       */
 int   nump;                 /* s_num[sonar] + 1                             */
 meanf = (float)s_mean[sonar];
 if (((float)range < (meanf * 1.16)) && ((float)range > (meanf * 0.84)) &&
  (s_var[sonar] < 1000)) {
    if (s_num[sonar] < MAXCONF) {
       nump = s_num[sonar] + 1;
       s_var[sonar] = ((s_var[sonar] * s_num[sonar]) +
                    ((s_mean[sonar] - range) * (s_mean[sonar] -  range)))/ nump;
       s_mean[sonar] = ((s_num[sonar] * s_mean[sonar]) + range) / nump;
       s_num[sonar] = nump;
    }
 } else {
    s_num[sonar] = 1;
    s_mean[sonar] = range;
    s_var[sonar] = 0;
 }
} /** end rcdUpdate() **/
```

```
/*****************************************************************************/
/* Procedure: saveMap                                                  Done */
/* Called by: (driver)test1                                                 */
/* Calls:     <none>                                                        */
/* Returns:   FALSE if file could not be opened, or TRUE                    */
/* Globals:   cgrid, confi  (READ only)                                     */
/* I/O:       writes to a file                                              */
/*    Saves the current certainty and confidence grid in the argument file. */
/*    If the file could not be opened, does nothing.                        */
/*****************************************************************************/
int Certgrid::saveMap(char *fname)
{
 FILE *fpo;                  /* Output file pointer                       */
 int   i,j;                  /* Loop indices                              */
 fpo = fopen(fname,"w");
 if (fpo == NULL) return(FALSE);
 for (i=0; i<SZE; i++) {
    for (j=0; j<SZE; j++ ) {
       fprintf(fpo,"\n%3d %3d %5.2f %5.2f",i,j,cgrid[i][j],confi[i][j]);
    }
 }
 fprintf(fpo,"\n999 999 0.0   0.0  ");
 fclose(fpo);
 return(TRUE);
} /** end saveMap() **/
```

```
/*****************************************************************************/
/* Procedure: setCgrid                                                 Done */
/* Called by: (oman)findPosition, SonarScan1                                */
/* Calls:     <none>                                                        */
/* Globals:    cgrid, confi (Read Write)                                    */
/*    Checks the array indices to see if they are in the grid. The certainty */
/*    grid cell is set to the weighted average of the current cell value    */
/*    and the new value. The corresponding confidence values are used for   */
/*    the weights. The new confidence value is found by increasing the old  */
/*    value by one for an occupied cell or decreasing it by 1 if empty.     */
/*    If the occupied probability drops low with confidence of -1 or less   */
/*    then the probability is set to 0 to avoid a very picky navigator.     */
/*****************************************************************************/
void Certgrid::setCgrid(
   int   x, int y,           /* Indices to certainty grid cell              */
   float cert,               /* New value for certainty grid                */
   float conf)               /* Confidence value                            */
{
 if ((x>=SZE) || (y>=SZE) || (x<0) || (y<0)) return;        /* out of bounds */
 else if (conf < 0.0) {          /* Negative confidence suppresses averaging */
    cgrid[x][y] = cert;
    if (cert < 0.5) confi[x][y] = -MAXCONF;
    else confi[x][y] = MAXCONF;
 } else {                                        /* Calculate new certainty  */
    cgrid[x][y] = ((cgrid[x][y] * (float)fabs((double)confi[x][y]))
                  + (cert * conf)) / ((float)fabs((double)confi[x][y]) + conf);
    if (cert >= MEDO) {                          /* Calculate new confidence */
       confi[x][y] = confi[x][y] + 1.0;
       if (confi[x][y] > MAXCONF) confi[x][y] = MAXCONF;
    } else if (cert <= MEDE) {
       confi[x][y] = confi[x][y] - 1.0;
       if (confi[x][y] < -MAXCONF) confi[x][y] = -MAXCONF;
       if ((confi[x][y] <= -1) && (cgrid[x][y] < MEDE)) cgrid[x][y] = HIGHE;
     }
 }
} /** end setCgrid() **/
```

```
/*****************************************************************************/
/* Procedure: sonarInit                                                Done */
/* Called by: cgridInit                                                     */
/* Calls:     <none>                                                        */
/* Globals:   s_mean, s_num, s_var (Write Only)                             */
/*    Initializes the RCDs for all 8 sonars.                                */
/*****************************************************************************/
void Certgrid::sonarInit(void)
{
 int i;
 for (i=0; i<=7; i++) {
    s_mean[i] = 0;
    s_num[i] = 0;
    s_var[i] = 0;
 }
} /** end sonarInit() **/
```

```
/*****************************************************************************/
/* Procedure: shift                                                    Done */
/* Called by: (oman)center_robot                                            */
/* Calls:     <none>                                                        */
/* Globals:   cgrid, robx,roby (Read Write)                                 */
/*   Shifts the certainty grid to compensate for the robot movement passed  */
/*   as arguments. The robot is restored to the center of the grid.         */
/*   Certainty values which are shifted off the grid are lost while new     */
/*   cells are given the background value.                                  */
/*****************************************************************************/
void Certgrid::shift(
   int dx, int dy)         /* X & Y movement of robot in grid cells         */
{
 int x,y;
 if (dx >= 0) {
    if (dy >= 0) {                    /* Robot has moved forward and up     */
       for (x=0; x<SZE; x++) {
          for (y=0; y<SZE; y++) {
             if ((x+dx >= SZE) || (y+dy >= SZE)) {
                cgrid[x][y] = back_grnd;
                confi[x][y] = 0.0;
             } else {
                cgrid[x][y] = cgrid[x+dx][y+dy];
                confi[x][y] = confi[x+dx][y+dy];
       } } }
    } else {
       for (x=0; x<SZE; x++) {
          for (y=SZE-1; y>=0; y--) {   /* Robot has moved forward and down   */
             if ((x+dx >= SZE) || (y+dy < 0)) {
                cgrid[x][y] = back_grnd;
                confi[x][y] = 0.0;
             } else {
                cgrid[x][y] = cgrid[x+dx][y+dy];
                confi[x][y] = confi[x+dx][y+dy];
    } } } }
 } else {
    if (dy >= 0) {                    /* Robot has moved backward and up    */
       for (x=SZE-1; x>=0; x--) {
          for (y=0; y<SZE; y++) {
             if ((x+dx < 0) || (y+dy >= SZE)) {
                cgrid[x][y] = back_grnd;
                confi[x][y] = 0.0;
             } else {
                cgrid[x][y] = cgrid[x+dx][y+dy];
                confi[x][y] = confi[x+dx][y+dy];
       } } }
    } else {
       for (x=SZE-1; x>=0; x--) {     /* Robot has moved backward and down */
          for (y=SZE-1; y>=0; y--) {
             if ((x+dx < 0) || (y+dy < 0)) {
                cgrid[x][y] = back_grnd;
                confi[x][y] = 0.0;
             } else {
                cgrid[x][y] = cgrid[x+dx][y+dy];
                confi[x][y] = confi[x+dx][y+dy];
             }
          }
       }
    }
 }
} /** end shift() **/
```

```
/*****************************************************************************/
/* Procedure: sonarMarkObject                                          Done */
/* Called by: sonarScan1                                                    */
/* Calls:     <none>                                                        */
/*   Increments from one end of a perceived object to the other setting the */
/*   certainty grid for each cell along the way.                            */
/*****************************************************************************/
void Certgrid::sonarMarkObject(
   int xr, int yr,          /* Coordinates of left end of object  */
   int xl, int yl,          /* Coordinates of right end of object */
   float cert,              /* Certainty value  */
   float conf)              /* Confidence value */
{
 int   i,j;           /* Loop indices */
 int   dx,dy;         /* difference between two points     */
 int   minv, maxv;    /* Lesser, greater of x or y values  */
 float slope;         /* slope of line between object ends */
 float jf;            /* float index of non incremented pt */
 if (xr == xl) {
    minv = (yr >= yl) ? yl : yr;
    maxv = (yr >= yl) ? yr : yl;
    for (j = minv; j <= maxv; j++)  setCgrid(xr,j,cert,conf);
 } else if (yr == yl) {
    minv = (xr >= xl) ? xl : xr;
    maxv = (xr >= xl) ? xr : xl;
    for (i = minv; i <= maxv; i++)  setCgrid(i,yr,cert,conf);
 } else {
    dx = (xr - xl);
    dy = (yr - yl);
    if (abs(dx) >= abs(dy)) {  /* increment along x */
       if (xr > xl) {          /* order points by x value   xr < xl  */
          minv = xl;           /* exchange points */
          xl = xr;
          xr = minv;
          minv = yl;
          yl = yr;
          yr = minv;
       }
       slope = (float)dy / (float)dx;
       jf = (float)yr - slope;
       for (i = xr; i <= xl; i++) {
          jf = jf + slope;
          j = ROUND(jf);
          setCgrid(i,j,cert,conf);
       }
    } else {                   /* increment along y */
       if (yr > yl) {             /* order points by y value   yr < yl  */
          minv = xl;
          xl = xr;
          xr = minv;
          minv = yl;
          yl = yr;
          yr = minv;
       }
       slope = (float)dx / (float)dy;
       jf = (float)xr - slope;
       for (j = yr; j <= yl; j++) {
          jf = jf + slope;
          i = ROUND(jf);
          setCgrid(i,j,cert,conf);
 } } }
} /** end sonarMarkObject() **/
```

```
/*****************************************************************************/
/* Procedure: sonarPair                                                 Done */
/* Called by: sonarScan                                                      */
/* Calls:     sonarScan1 - update certainty grid with a single sonar reading */
/*    Combines the two range readings from a pair of sonars into a single    */
/*    reading used to update the grid. Determines the sonar, angle spread    */
/*    and certainty.                                                         */
/*    Below is a table with the hardcoded values givning each sonars position */
/*    they are x & y distance (cm) from robot center, radial distance from   */
/*    the robot center and the angular difference between the radial line    */
/*    to the sonar and the sonars direction                                  */
/*    # location      dir          x     y     r     deg     rad             */
/*    1 Left rear      90   1.5708   8.2   9.5  12.5  +40.80  +0.7121        */
/*    7 Left forw      90   1.5708  15.8   9.5  18.4  -58.98  -1.0294        */
/*  1,7 Left pair      90   1.5708   3.8f  9.5  10.2  -21.80  -0.3805        */
/*    0 Right rear   -90  -1.5708   8.2   8.5  11.8  -43.97  -0.7674        */
/*    6 Right forw   -90  -1.5708  15.8   8.5  17.9  +61.72  +1.0773        */
/*  0,6 Right pair   -90  -1.5708   3.8f  8.5   9.3  +24.09  +0.4204        */
/*    5 Front left     0   0.0     13.3  12.0  17.9  +42.06  +0.7341        */
/*    4 Front right    0   0.0     13.3  11.0  17.3  -39.59  -0.6910        */
/*  5,4 Front pair     0   0.0     13.3   0.51 13.3   +2.12  +0.0376        */
/*    2 Back left    180   3.1416   5.7  12.0  13.3  -64.59  -1.1273        */
/*    3 Back right   180   3.1416   5.7  11.0  12.4  +62.61  +1.0927        */
/*  2,3 Back pair    180   3.1416   5.7   0.51  5.7   -5.01  -0.0875        */
/*    3 Angle left    45   0.7853  11.6  11.2  16.1   -1.00  -0.0175        */
/*    2 Angle right  -45  -0.7853  11.6  10.2  15.4   +3.67  +0.0641        */
/*****************************************************************************/
void Certgrid::sonarPair(
   int   s1, int s2,          /* Numbers of paired sonars                    */
   int   r1, int r2,          /* range readings(mm) from sonar pair          */
   int   ang,                 /* sonar angle(deg) with respect to robot      */
   float base_ang,            /* robot's direction (deg)                     */
   int   offx, int offy)      /* dist.(mm) from robot center to cell center  */
{
 float angle;               /* Angle from robot cell center to sonar */
 float bangle;              /* base angle (radians)                  */
 short done = FALSE;        /* Loop exit condition                   */
 int dx,dy;                 /* distance(mm) of sonar from cell cent. */
 int mindis;                /* lower of the two range readings       */
 int meandis;               /* average of the two range readings     */
 int curson;                /* loop index,  sonar number             */
 bangle = RADIAN(base_ang);

/*****************************************************************************/
/* If the two range readings are within the threshold distance of each other */
/* we assume they see the same object in their overlap area. We mark as if a */
/* single sonar centered between them recorded the average                   */
/*****************************************************************************/
 if (abs(r1-r2) < THRESH) {
    meandis = (int)((r1+r2)/2);
    if ((s1 == 7) || (s1 == 1)) {          /* Left side */
       angle = bangle + 1.5708 - 0.3805;
       dx = offx + (int)(102.0 * cos(angle));
       dy = offy + (int)(102.0 * sin(angle));
    } else if ((s1 == 0) || (s1 == 6)) {    /* Right side */
       angle = bangle - 1.5708 + 0.4204;
       dx = offx + (int)(93.0 * cos(angle));
       dy = offy + (int)(93.0 * sin(angle));
    } else if ((s1 == 4) || (s1 == 5)) {    /* Front */
       angle = bangle + 0.0376;
       dx = offx + (int)(133.0 * cos(angle));
       dy = offy + (int)(133.0 * sin(angle));
    } else if ((s1 == 2) || (s1 == 3)) {
       angle = bangle + 3.1416 - 0.0875;
       dx = offx + (int)(57.0 * cos(angle));
       dy = offy + (int)(57.0 * sin(angle));
    }
    if (s_num[s1] >= s_num[s2])
       sonarScan1(meandis,ang,base_ang,dx,dy,8,HIGHO,s_num[s2],1);
 } else {
  /*****************************************************************************/
  /* if the sonars disagree then each object is marked independently from   */
  /* the sonar which recorded it                                               */
  /*****************************************************************************/
   mindis = (r1 <= r2) ? r1 : r2;
   curson = s1;
   while (!done) {
      switch (curson) {

      case  1:   /* Left rear */
         angle = bangle + 1.5708 + 0.7121;
         dx = offx + (int)(125.0 * cos(angle));
         dy = offy + (int)(125.0 * sin(angle));
      break;
      case 7:   /* Left forward */
         angle = bangle + 1.5708 - 1.0294;
         dx = offx + (int)(184.0 * cos(angle));
         dy = offy + (int)(184.0 * sin(angle));
      break;
      case 0:   /* Right rear */
         angle = bangle - 1.5708 - 0.7674;
         dx = offx + (int)(118.0 * cos(angle));
         dy = offy + (int)(118.0 * sin(angle));
      break;
      case 6:  /* Right forward */
         angle = bangle - 1.5708 + 1.0773;
         dx = offx + (int)(179.0 * cos(angle));
         dy = offy + (int)(179.0 * sin(angle));
      break;
      case 5:  /* Front left  */
         angle = bangle + 0.7341;
         dx = offx + (int)(179.0 * cos(angle));
         dy = offy + (int)(179.0 * sin(angle));
      break;
      case 4:  /* Front right */
         angle = bangle - 0.6910;
         dx = offx + (int)(173.0 * cos(angle));
         dy = offy + (int)(173.0 * sin(angle));
      break;
      case  2:  /* Back left */
         angle = bangle + 3.1416 - 1.1273;
         dx = offx + (int)(133.0 * cos(angle));
         dy = offy + (int)(133.0 * sin(angle));
      break;
      case 3:   /* Back right */
         angle = bangle + 3.1416 + 1.0927;
         dx = offx + (int)(124.0 * cos(angle));
         dy = offy + (int)(124.0 * sin(angle));
      break;
         default:
      break;
      }
```

```
      /*************************************************************************/
      /* Distant objects are marked 4 degrees off line with medium             */
      /* probability. Close objects have high probability                      */
      /*************************************************************************/
      if (mindis > NEAR) {
         if (curson == s1)
            sonarScan1(r1,ang-4,base_ang,dx,dy,4,MEDO,s_num[s1],1);
         else sonarScan1(r2,ang+4,base_ang,offx,offy,4,MEDO,s_num[s2],1);
      } else
         if (curson == s1)
            sonarScan1(r1,ang,base_ang,dx,dy,8,HIGHO,s_num[s1],1);
         else sonarScan1(r2,ang,base_ang,dx,dy,8,HIGHO,s_num[s2],1);
      if (curson == s2) done = TRUE;
      else curson = s2;
   }
 }
} /** end sonarPair() **/
```

```
/*****************************************************************************/
/* Procedure: sonarScan                                                 Done */
/* Called by: (oman)turn, circleScan, (oman)travel_aux, (driver)test1        */
/* Calls:     rcdUpdate - Determines if latest reading is in current RCD     */
/*            SonarPt->read - read all 8 sonars                              */
/*            SonarScan1 - update cgrid with information from a single sonar */
/*            sonarPair  - update cgrid with information from a sonar pair   */
/*  Takes sonar readings from all 8 sonars.  The sonars are arranged in      */
/*  either a C or square configuration with a pair of sonars in the front    */
/*  and on each side.  The square array has a pair of sonars facing the rear */
/*  while the C array (default)   1  7 3               1   7                 */
/*  moves those sonars to the             5           2     5               */
/*  two front corners.          (Front)   Forward >   (square)   Forward >   */
/*  The sonar firing order is           4             3     4                 */
/*  0,5,2,7,4,1,6,3.              0  6 2               0   6                 */
/*  The values from each pair of same facing sonars are combined in the      */
/*  sonarPair routine before updating the grid. The two front angled sonars  */
/*  (2 and 3) are for close object detection to prevent collisions caused by */
/*  turning into an object. Because these sonars are very vulnerable to      */
/*  specular reflections when wall following, their values are only used     */
/*  when they indicate an object within 1000 mm.                             */
/*     The sonar specific values are in the table in sonarPair.              */
/*****************************************************************************/
void Certgrid::sonarScan(
   float base_ang,       /* Current robot orientation                       */
   int robx, int roby,   /* Current robot cell                              */
   int offx, int offy)   /* X,Y distances from robot center to cell center  */
{
 float angle;            /* angle(rad) to sonar   */
 float bangle;           /* base angle in radians */
 int   dx,dy;            /* dist. cell center to sonar */
 int   i;                /* Loop index       */
 int   sdat[8];          /* sonar range data */
 if ((sonar_type == S_OFF) || (sonar_type == S_TEST)) return;
 loc_robx = robx; loc_roby = roby;
 SonarPt->read(sdat);
 for (i=0; i<=7; i++) {
    if (sdat[i] < 0) sdat[i] = SONAR_RANGE;
    rcdUpdate(i,sdat[i]);
 }
 sonarPair(4,5,sdat[4],sdat[5],0,base_ang,offx,offy);  /* Front sonars */
 sonarPair(7,1,sdat[7],sdat[1],90,base_ang,offx,offy); /* Left  sonars */
 sonarPair(0,6,sdat[0],sdat[6],-90,base_ang,offx,offy);/* Right sonars */
 if (sonar_type == S_FRONT) {
     bangle = RADIAN(base_ang);
     if (sdat[3] < 1000) {
       angle = bangle + 0.7853 -0.0175;
       dx = offx + (int)(161.0 * cos(angle));
       dy = offy + (int)(161.0 * sin(angle));
       sonarScan1(sdat[3],45,base_ang,dx,dy,0,HIGHO,s_num[3],1);  //angle left
     }
     if (sdat[2] < 1000) {
       angle = bangle -0.7853 + 0.0641;
       dx = offx + (int)(154.0 * cos(angle));
       dy = offy + (int)(154.0 * sin(angle));
       sonarScan1(sdat[2],-45,base_ang,dx,dy,0,HIGHO,s_num[2],1); //angle right
    }
 }
 if (sonar_type == S_SQUARE)
    sonarPair(2,3,sdat[2],sdat[3],180,base_ang,offx,offy);/* Back sonars*/
} /** end sonarScan() **/
```

```
/*************************************************************************/
/* Procedure: sonarScan1                                          Done */
/* Called by: sonarScan, sonarPair                                     */
/* Calls:     <none>                                                   */
/* Globals:   loc_robx,loc_roby (read)   cgrid (write)                 */
/*   Uses a single sonar's distance and angle to an object to update the     */
/*   certainty grid. The cells with the object are marked occupied and all   */
/*   in the cone between the sonar and the object are marked open.  The size */
/*   of the cone and occupancy probabilities are passed as arguments.        */
/*      The marking is done using the following algorithm.  First find the   */
/*   two far endpoints of the cone swept by the sonar.  The line between     */
/*   these points is marked as occupied.  The slopes of the edges of the cone*/
/*   are found. Starting at the cell with the sonar, increment outward in x  */
/*   toward the closer (in x) of the two endpoints.  For each X find the     */
/*   corresponding Ys (using the slopes) of each edge of the cone. Increment */
/*   between them and mark each cell (X,Y) as empty. If the axis of the cone */
/*   is close to the x axis then the above is used. If close to the y axis   */
/*   then the procedure increments outward in Y then in X between the cone   */
/*   edges. This can leave some empty cells unmarked under the following     */
/*   conditions: object is several cells away, sonar angle is close to 45    */
/*   135, 225, or 315 degrees.                                               */
/*************************************************************************/
void Certgrid::sonarScan1(
   int   dist,           /* distance in mm to object                    */
   int   angle,          /* sonar angle wrt robot head                  */
   float base_ang,       /* Current robot direction                     */
   int   offx, int offy, /* offset(mm) of sonar from robot cell center  */
   int   spread,         /* Half angle of sonar spread                  */
   float cert,           /* Certainty value to use when updating grid   */
   int   conf,           /* Confidence, (number of readings in current RCD) */
   int   objflg)         /* 0 - space only, 1 - both, 2 - object only   */
{
 int   angle2;           /* direction(deg)  of sonar                    */
 float dir1,dir2;        /* direction to object ends                    */
 int   xg,yg;            /* grid coordinates of object center           */
 int   xgl,ygl,xgr,ygr;  /* grid coordinates of object ends             */
 int   xp1,yp1,xp2,yp2;  /* grid coordinates of unoccupied squares      */
 int   xc,yc;            /* X and Y distance to object in grid cells    */
 float x,y;              /* x and y distance(mm) to object center       */
 float x1,y1,x2,y2;      /* x and y distance(mm) to object ends         */
 float px1,py1,px2,py2;  /* coordinates of unoccupied squares           */
 float slope1,slope2;    /* slope of lines to object ends               */
 int   i,j;              /* Loop indices                                */
 float certe;            /* Certainty value for marking empty cells     */
 float conff;            /* confidence as a floating pt number          */
 int   sonx,sony;        /* Cell with sonar                             */
 int   offcx,offcy;      /* Offset(mm) of sonar from cell center        */

 conff = (float)conf;
/***** If sonar sees nothing set distance to the sonar's range      *****/
 if (dist >= SONAR_RANGE) dist = SONAR_RANGE;
/*************************************************************************/
/* Find which cell the sonar is in and how far off center it is in that cell */
/*************************************************************************/
 sonx = loc_robx; sony = loc_roby;
 offcx = offx;    offcy = offy;
 if (offcx > 0) offcx++;  /* increase offset by 1 to cover rounding error */
 if (offcx < 0) offcx--;
 if (offcy > 0) offcy++;
 if (offcy < 0) offcy--;
 if (offcx >= HALFCELL)  {
    sonx++;
    offcx = offcx - HALFCELL;
 } else if (offcx <= -HALFCELL)  {
    sonx--;
    offcx = offcx + HALFCELL;
 }
 while (offcx >= CELL) {
    sonx++;
    offcx = offcx - CELL;
 }
 while (offcx <= -CELL) {
    sonx--;
    offcx = offcx + CELL;
 }
 if (offcy >= HALFCELL)  {
    sony++;
    offcy = offcy - HALFCELL;
 } else if (offcy <= -HALFCELL)  {
    sony--;
    offcy = offcy + HALFCELL;
 }
 while (offcy >= CELL) {
    sony++;
    offcy = offcy - CELL;
 }
 while (offcy <= -CELL) {
    sony--;
    offcy = offcy + CELL;
 }
/*************************************************************************/
/* Find the coordinates of the left and right ends of the object. First find */
/* the angles to the two ends from the sonar. Find the X and Y components of */
/* the distance from sonar to end cell and add the offsets to get distance   */
/* from the center of the sonar cell. Finally convert to grid cells and add  */
/* to sonar cell coordinatates.                                              */
/*************************************************************************/
 dir1 = base_ang + (float)(angle + spread);   /* Get angles to object ends   */
 dir2 = base_ang + (float)(angle - spread);
 dir1 = RADIAN(dir1);                         /* Convert to radians          */
 dir2 = RADIAN(dir2);
 x1 = ((float)dist * cos(dir1)) + (float)offcx;
 y1 = ((float)dist * sin(dir1)) + (float)offcy;
 if (( x1 >= 100.0) || (x1 <= -100.0))  xc = CEILING(x1/CELL);
 else xc = 0;
 xgl = xc + sonx;
 if (( y1 >= 100.0) || (y1 <= -100.0))  yc = CEILING(y1/CELL);
 else yc = 0;
 ygl = yc + sony;
 x2 = ((float)dist * cos(dir2)) + (float)offcx;
 y2 = ((float)dist * sin(dir2)) + (float)offcy;
 if (( x2 >= 100.0) || (x2 <= -100.0)) xc = CEILING(x2/CELL);
 else xc = 0;
 xgr = xc + sonx;
 if (( y2 >= 100.0) || (y2 <= -100.0)) yc = CEILING(y2/CELL);
 else yc = 0;
 ygr = yc + sony;
```

```
/*****************************************************************************/
/* Mark the object in the certainty grid                                     */
/* If the object is in the robot cell. Mark both it and the cell farther out */
/* The next cell out is found by recalculating the object position with a    */
/* distance 1 half cell larger.  Otherwise just mark the object.             */
/*****************************************************************************/
 if ((dist != SONAR_RANGE)&&(objflg >= 1)) {
    if ((xgr == loc_robx) && (ygr == loc_roby)) {
       setCgrid(loc_robx,loc_roby,cert,conff);
       x1 = ((float)(dist + HALFCELL) * cos(dir1)) + (float)offcx;
       y1 = ((float)(dist + HALFCELL) * sin(dir1)) + (float)offcy;
       xc = CEILING(x1/CELL);
       yc = CEILING(y1/CELL);
       i = xc + sonx;
       j = yc + sony;
       setCgrid(i,j,cert,conff);
    }
    else sonarMarkObject(xgr,ygr,xgl,ygl,cert,conff);
 }
/*****************************************************************************/
/*  Mark empty space                                                         */
/*  First determine the closer of the two cone ends. Based on the sonar angle*/
/*  increment */
 if (objflg > 1) return;
 if (objflg == 0) certe = cert;
 else certe = HIGHE;
 if (abs(xgl - sonx) > abs(xgr - sonx)) xg = xgr;
 else (xg = xgl);
 if (abs(ygl - sony) > abs(ygr - sony)) yg = ygr;
 else (yg = ygl);
 x  = (x1 + x2)/2.0;
 y  = (y1 + y2)/2.0;
 angle2 = angle + (int)base_ang;
 while (angle2 > 180) angle2 = angle2 - 360;
 while (angle2 < -180) angle2 = angle2 + 360;
/***** Mark all cells between sonar and object as open                  *****/
 if ((angle2 >= -45) && (angle2 < 45)) {   /* positive x direction */
    slope1 = y1 / x1;
    slope2 = y2 / x2;
    py1 = (float)sony - slope1;
    py2 = (float)sony - slope2;
    for (xp1= sonx; xp1<xg;xp1++) {   /* For each X on line of sight */
       py1 = py1 + slope1;            /* calculate Ys of cone edge   */
       yp1 = ROUND(py1);
       py2 = py2 + slope2;            /* For each Y between cone edges,   */
       yp2 = ROUND(py2);              /* mark cell empty                  */
       if (yp1 <= yp2) for (j=yp1; j<=yp2; j++) setCgrid(xp1,j,certe,conff);
       else for (j=yp2; j<=yp1; j++) setCgrid(xp1,j,certe,conff);
    }
 ) else if (((angle2 >= 135) && (angle2 <  225)) ||
            ((angle2 < -135) && (angle2 >= -225))) { /* negative x direction */
    slope1 = y1 / x1;
    slope2 = y2 / x2;
    py1 = (float)sony + slope1;
    py2 = (float)sony + slope2;
    for (xp1=sonx; xp1>xg; xp1--) {
       py1 = py1 - slope1;
       yp1 = ROUND(py1);
       py2 = py2 -  slope2;
       yp2 = ROUND(py2);
       if (yp1 <= yp2) for (j=yp1; j<=yp2; j++) setCgrid(xp1,j,certe,conff);
       else for (j=yp2; j<=yp1; j++) setCgrid(xp1,j,certe,conff);
    }
 ) else if ((angle2 >= 45) && (angle2 < 135)) {    /* positive y direction */
    slope1 = x1 / y1;
    slope2 = x2 / y2;
    px1 = (float)sonx - slope1;
    px2 = (float)sonx - slope2;
    for (yp1=sony; yp1<yg; yp1++) {   /* For each Y on line of sight */
       px1 = px1 + slope1;            /* calculate Xs of cone edges  */
       xp1 = ROUND(px1);
       px2 = px2 + slope2;            /* For each X between cone edges,   */
       xp2 = ROUND(px2);              /* mark cell empty                  */
       if (xp1 <= xp2) for (i=xp1; i<=xp2; i++) setCgrid(i,yp1,certe,conff);
       else for (i=xp2; i<=xp1; i++) setCgrid(i,yp1,certe,conff);
    }
 ) else if ((angle2 < -45) && (angle2 >= -135)) {
    slope1 = x1 / y1;
    slope2 = x2 / y2;
    px1 = (float)sonx +slope1;
    px2 = (float)sonx + slope2;
    for (yp1=sony; yp1>yg; yp1--) {
       px1 = px1 - slope1;
       xp1 = ROUND(px1);
       px2 = px2 - slope2;
       xp2 = ROUND(px2);
       if (xp1 <= xp2) for (i=xp1; i<=xp2; i++) setCgrid(i,yp1,certe,conff);
       else for (i=xp2; i<=xp1; i++) setCgrid(i,yp1,certe,conff);
    }
 )
) /** end sonarScan1() **/

/** end of cgrid.C **/
```

```
/*****************************************************************************/
/*    oman.h   03/19/92  Timothy T. Good                                     */
/* External header file for programs using the blank map navigator in oman.C */
/*****************************************************************************/

#ifndef OMAN_H
#define OMAN_H

/*****************************************************************************/
/* Constansts:                                                               */
/*****************************************************************************/

static const int   SPEED_INCREMENT =  2; /* Multiplier for speed changes     */
static const int   CENX       =   27;    /* Robot's starting X position      */
static const int   CENY       =   32;    /* Robot's starting Y position      */
static const float THO        = 0.75;    /* Threshold for occupied           */
static const float THE        = 0.25;    /* Threshold for empty              */

enum NavMeth { N_LOCAL, N_BACKTRACK, N_PATH, N_INCREMENT } ;

#ifndef OMAN_C              /* The following is already declared in oman.C */

/*****************************************************************************/
/* Procedures:                                                               */
/*****************************************************************************/
 extern float   checkAhead(float bearing);
                        /* Checks cgrid in front of robot for obstacles */
 extern float   checkSides(float bearing, float speed);
                        /* Checks cgrid to sides of robot for obstacles */
 extern float   distance(int x1, int y1, int x2, int y2);
                        /* finds distance between two points            */
 extern float   getAngle(void);/* Gets and normalizes robot angle       */
 extern float   getBaseAngle(void);      /* Retrieve robot angle         */
 extern float   getBaseDist(void);       /* Retrieve robot distance      */
 extern float   getVolts(void);          /* Retrieve robot voltage       */
 extern int     getDecayFlag(void);      /* Retrieve decay flag          */
 extern void    getDest(int x, int y);   /* Sets new destination         */
 extern float   getConstEmpty(void);     /* Retrieve empty threshold     */
 extern float   getConstOccup(void);     /* Retrieve occupied threshold  */
 extern float   getConstPath(void);      /* Retrieve path threshold      */
 extern NavMeth getNavigator(void);      /* Retrieve navigation method   */
 extern void    setBaseAngle(float);     /* Store robot angle            */
 extern int     setDecayFlag(int);       /* Set decay flag               */
 extern float   setConstEmpty(float);    /* Set empty threshold          */
 extern float   setConstOccup(float);    /* Set occupied threshold       */
 extern float   setConstPath(float);     /* Set path threshold           */
 extern NavMeth setNavigator(NavMeth);   /* Set navigation method        */
 extern void    setVolts(float);         /* Store robot voltage          */

 extern void    mkbgrd(void);  /* Creates entire badness grid            */
 extern void    newCell(void); /* Performs new cell actions, recenter, decay */
                        /* Finds off center distance in given direction */
 extern float  offCenter(int x, int y, float direction);
                        /* Finds off center distance in given direction */
 extern void    omanInit(int x,int y,short destFlag);
                        /* Initializes the navigation package           */
 extern float   selectMove(int robx, int roby, int offx, int offy);
                        /* Selects direction of travel                  */
 extern int     travel(void);  /* Main navigation routine               */
 extern short   update(float newBearing, float newDistance);
                        /* Updates distance, angle and grid postion     */
#endif  /* OMAN_C */
```

```
/*****************************************************************************/
/*  oman.C      04/07/92  Timothy T. Good                                    */
/*    This file contains the procedures for a robot navigator using the      */
/*    cgrid certainty grid package.                                          */
/*****************************************************************************/
#define OMAN_C
#include "cgrid.h"              /* certainty grid module      */
#include "oman.h"               /* navigator module           */
#include "display.h"            /* display module             */
#include "destack.h"            /* destination stacks         */
#include "RWITim.h"             /* robot commands             */
#include <math.h>
#include <curses.h>

/*****************************************************************************/
/* Constants:  Local constants are listed below, additional are in oman.h    */
/*****************************************************************************/

/* #define TRUE 1, FALSE 0, OK 1, ERR 0 - already done in curses.h */
static const int   ROBOT_DIAMETER = 300; /* Robot diameter in mm             */
static const int   MAXFAC          =  6; /* Maximum speed factor             */
static const float TRAVEL_VALUE  =  0.5; /* Potential increase, traveled cell*/
static const float CLOSE = (float)((2*CELL)-10); /* Fake dist.(mm) to dest.  */
static const float ADJUSTS = ((float)(ROBOT_DIAMETER + CELL))/2.0;
static const float CHECKDIS = (float)(5*CELL); /* Checkahead distance (mm)   */
static const float QUERY_TIME = 0.33;  /* Simulated real time for robot query*/
static const float SET_TIME   = 0.55;  /* Simulated real time for robot set  */
static const int   XNBOR[8] = {1,-1, 0,0,1,-1, 1,-1); /* used to access      */
static const int   YNBOR[8] = {0, 0,-1,1,1,-1,-1, 1); /* neighbor cells      */
static const int   NEAR_EDGE1 =   9;     /* Index of cell near edge of grid  */
static const int   NEAR_EDGE2 = SZE - 10;/* Index of cell near edge of grid  */
static const float PATH_VALUE = 0.25;    /* Certainty limit for paths        */
static const float EXTEND = (float)(CELL - (ROBOT_DIAMETER/2));
     /* Off center distance for robot to extend to midpoint of neighbor cell */

/*****************************************************************************/
/* Type definitions:                                                         */
/*****************************************************************************/

typedef struct Cell {           /* for linked lists of cells */
   int x;                       /* x index value             */
   int y;                       /* y index value             */
   struct Cell *npt;            /* pointer to next list element */
};
```

```
/*****************************************************************************/
/* Global data:     Listed alphabetically by groups- extern, global, static  */
/*****************************************************************************/

extern Certgrid  *CgPt;       /*(driver) Certainty grid package (cgrid.C)    */
extern DisplayC  *DcPt;       /*(driver) Curses display         (display.C)  */
extern DisplayX  *DxPt;       /*(driver) X display              (display.C)  */
extern RWITim    *RwitPt;     /*(driver) Robot control routines (RWITim.C)   */
extern Voice     *VoicePt;    /*(driver) Robot speech routines  (display.C)  */
extern Destack   *DsPt;       /*(driver) Destination stack      (destack.C)  */

short abortFlg = FALSE;  /* Flag for user abort                              */
float base_angl=0.0;     /* Robot's orientation in degrees                   */
float base_dis=0.0;      /* Robot's distance traveled (cm) since start       */
float bgrid[SZE][SZE];   /* Badness grid, Combination of cgrid, dgrid & tgrid*/
float dgrid[SZE][SZE];   /* Directional grid, imposes bias toward destination*/
short egrid[SZE][SZE];   /* Dead end grid, holds global travel info          */
int   offx,offy;         /* Offsets of robot center from cell center in mm   */
float off_angle;         /* Offset between base_angl and robot's queryAngle  */
float off_dist;          /* Offset between base_dis and robot's queryDistance*/
int   robx,roby;         /* Robot's position in grid cells                   */
float slew=0.0;          /* Robot's turn rate, positive is counterclockwise  */
int   spdfac=0;          /* Speed factor, multiplier for speed increment     */
float tgrid[SZE][SZE];   /* Travel grid, indicates cells already traversed   */
float timer;             /* Approximates time for simulator                  */
float volts;             /* Robot voltage                                    */
int   xdest,ydest;       /* Robot's ultimate destination in grid coords      */
int   xsubd,ysubd;       /* Robot's subdestination                           */

static int   begx,begy;       /* Starting X & Y position in grid             */
static short elimCur[5][5];   /* Cells around current cell to eliminate      */
static short elimNex[5][5];   /* Cells around next cell to eliminate         */
static int   elmx,elmy;       /* Center cell of elimination array            */
static short mark[SZE][SZE];  /* Holds marks for breadth first grid traverse */
static float movx,movy;       /* Cumulative X & Y dist. (mm) robot has moved */
static short nwcell = FALSE;  /* Flag for new cell entered                   */
static int   n_decay = 0;     /* Decay flag 0 - off, 1 - on                  */
static NavMeth n_nav = N_BACKTRACK;  /* Navigation method                    */
static float n_path = PATH_VALUE; /* Upper threshold for paths               */
static float n_the = THE;     /* Upper threshold for empty cells             */
static float n_tho = THO;     /* Lower threshold for occupied cells          */
```

```
/*****************************************************************************/
/* Procedures:  Separated into groups of inline, access and other functions. */
/*              Within groups listed alphabetically.                         */
/*****************************************************************************/

inline float degToRadian(float degrees);
inline float radToDegree(float radians);
inline int   ROUND(float fval);

       float getConstEmpty(void);          /* Retrieve empty threshold      */
       float getConstOccup(void);          /* Retrieve occupied threshold   */
       float getConstPath(void);           /* Retrieve path threshold       */
       int   getDecayFlag(void);           /* Retrieve decay flag           */
       NavMeth getNavigator(void);         /* Retrieve navigator method     */
       float setConstEmpty(float);         /* Set empty threshold           */
       float setConstOccup(float);         /* Set occupied threshold        */
       float setConstPath(float);          /* Set path threshold            */
       int   setDecayFlag(int);            /* Set decay flag                */
       NavMeth setNavigator(NavMeth);      /* Set navigator method          */

       float getBaseAngle(void);           /* Retrieve robot angle          */
       float getBaseDist(void);            /* Retrieve robot distance       */
       float getVolts(void);               /* Retrieve robot voltage        */
       void  setBaseAngle(float);          /* Store robot angle             */
       void  setVolts(float);              /* Store robot voltage           */
static short atSDest(int x, int y);
                       /* Checks to see if a cell is the sub destination   */
static short atUDest(int x, int y);
                       /* Checks to see if a cell is the ultimate dest.    */
static short backGnd(int x, int y);
                       /* Checks if a cell is unmapped (at background)     */
static void  centerRobot(void);
                       /* Shifts the grid so the robot is in the center    */
static void  changeSpeed(int oldspeed, int newspeed);
                       /* Reserved for future use , sets the robot speed   */
       float checkAhead(float bearing);
                       /* Checks cgrid in front of robot for obstacles     */
static float checkAux(int i, int j, float adjust, int flg);
                       /* Assists checkAhead                               */
       float checkSides(float bearing, float speed);
                       /* Checks cgrid to sides of robot for obstacles     */
static void  collision(void);
                       /* Checks current and front cells for collision     */
static short destCheck(void);
                       /* Checks that the subdestination is unoccupied     */
       float distance(int x1, int y1, int x2, int y2);
                       /* Calculates distance between two grid cells       */
static void  elimInit();
                       /*  Initialize the elimination arrays               */
static void  elimStore(int xold,int yold);
                       /* Save cells eliminated for next selectMove        */
static float findAngle(int xsrce, int ysrce, int xdead, int ydead);
                       /* Finds the angle of the line from srce to dead    */
static short findPosition(int distance, int bearing);
                       /* Determines robot's grid position                 */
static void  fzbgrd(void);
                       /* Makes fuzzy local badness grid                   */
       float getAngle(void);
                       /* Gets and normalizes robot angle                  */
static float getDistance(void);
                       /* Gets and normalizes robot distance               */
       void  getDest(int x, int y);
                       /* Sets new destination, initializes start list     */
static void  look(void);
                       /* Makes sure some local map is present             */
static void  markDeadEndAll(int cx, int cy, int fx, int fy);
                       /* Sets dead end flags for all cells along route    */
static void  markDeadEndBox(int,int,int,int,int,int,int,int);
                       /* Sets dead end flags over a boxed area            */
static void  markLine(int x1, int y1, int x2, int y2);
                       /* Puts a line into the mark array                  */
       void  mkbgrd(void);
                       /* Creates entire badness grid                      */
static void  mkdgrd(int x,int y,int destx,int desty,float path,short redo);
                       /* Makes the directional grid                       */
static Cell *mkDgridAux(int, int, Cell *, Cell *);
                       /* Treats grid edge as a destination                */
static void  mktgrd(void);
                       /* Initializes the travel grid to 0.0               */
static void  moveMe(int sonar_array);
                       /* Determines & implements robot movement           */
static short nearTo(int x1, int y1, int x2, int y2);
                       /* Checks if two points are within 4 cells          */
static short neighbor(int x1, int y1, int x2, int y2);
                       /* Checks if two cells are adjacent                 */
       void  newCell(void);
                       /* Performs new cell actions, recentering, decaying */
       float offCenter(int x, int y, float direction);
                       /* Finds off center distance in given direction     */
       void  omanInit(int x, int y, short destFlag);
                       /* Initializes the navigation package               */
static short pathToDest(int sourcex, int  sourcey, int destx, int desty);
                       /* Determines if path from source to dest may exist */
static short quitOk(void);
                       /* Determines if robot should quit                  */
       float selectMove(int robx, int roby, int offx, int offy);
                       /* Selects direction of travel                      */
static void  shiftTravel(int dx, int dy);
                       /* Shifts the travel grid to recenter robot         */
static void  stopMe(void);
                       /* Stops the robot and sets spdfac & slew to 0      */
static short subDest(float pathvalue);
                       /* Determines the next subdestination               */
static void  subStart(void);
                       /* Determines new start node                        */
       int   travel(void);
                       /* Main navigation routine                          */
static void  travelAuxBack(void);
                       /* Assists travel, assigns subdestinations          */
static void  travelAuxIncrement(void);
                       /* Assists travel, initializes incremental navigator*/
static void  travelAuxLocal(void);
                       /* Assists travel, initializes local navigator      */
static void  travelAuxPath(void);
                       /* Assists travel, assigns subdestinations          */
static void  turn(float angle, short waitFlag);
                       /* Turns robot a fixed number of degrees            */
       short update(float newBearing, float newDistance);
                       /* Updates robot's distance, angle and grid postion */
```

```
/*****************************************************************************/
/* Procedure: degToRadian     Converts degrees to radians              Done */
/* Calls:     <none>                                                        */
/* Returns:   float - angle in radians                                      */
/*****************************************************************************/
inline float degToRadian(float degrees)
{
 return(degrees * 0.017453293);     /* degrees * pi    / 180.0 */
}
/*****************************************************************************/
/* Procedure: radToDegree     Coverts radians to degrees               Done */
/* Calls:     <none>                                                        */
/* Returns:   float - angle in degrees                                      */
/*****************************************************************************/
inline float radToDegree(float radians)
{
 return(radians * 57.29578);        /* radians * 180.0 / pi */
}
/*****************************************************************************/
/* Procedure: ROUND           Rounds floats to integer                 Done */
/* Calls:     <none>                                                        */
/* Returns:   int - argument rounded to nearest integer                     */
/*****************************************************************************/
inline int ROUND(float fval)
{
 return   (  (fval >= 0)  ? (int)((fval) + 0.5)  : (int)((fval) - 0.5)  );
}
/*****************************************************************************/
/* Procedure: getConstEmpty      setConstEmpty                         Done */
/*            getConstOccup      setConstOccup                              */
/*            getConstPath       setConstPath                               */
/*            getDecayFlag       setDecayFlag                               */
/*            getNavigator       setNavigator                               */
/* Globals:  n_the, n_tho, n_path, n_decay, n_nav                           */
/*   These are functions to allow access to the user settable navigator     */
/*   constants.  Changing these values can have a significant effect on     */
/*   navigator performance.  They will default to THE, THO, PATH_VALUE, off */
/*   and N_INCREMENT.                                                       */
/*****************************************************************************/

float   getConstEmpty(void)    ( return(n_the); )
float   getConstOccup(void)    ( return(n_tho); )
float   getConstPath(void)     ( return(n_path); )
int     getDecayFlag(void)     ( return(n_decay); )
NavMeth getNavigator(void)     ( return(n_nav); )

float setConstEmpty(float newthe)
   ( if ((newthe >= 0.0) && (newthe <= 0.5)) n_the = newthe;
     return(n_the);
   )
float setConstOccup(float newtho)
   ( if ((newtho >= 0.5) && (newtho <= 1.0)) n_tho = newtho;
     return(n_tho);
   )
float setConstPath(float newpath)
   ( if ((newpath >= 0.0) && (newpath <= 1.0)) n_path = newpath;
     return(n_path);
   )
int setDecayFlag(int newdecay)
   ( if ((newdecay == 1) || (newdecay == 0)) n_decay = newdecay;
     return(n_decay);
   )

NavMeth setNavigator(NavMeth newnav)
   ( n_nav = newnav;
     if (newnav == N_LOCAL)      ( n_path = n_the; n_decay = 1; )
     if (newnav == N_BACKTRACK)  ( n_path = n_the; n_decay = 0; )
     if (newnav == N_PATH)         ( n_path = CgPt->getBackgd(); n_decay = 0; )
     if (newnav == N_INCREMENT)    ( n_path = CgPt->getBackgd(); n_decay = 0; )
     return(n_nav);
   )
/*****************************************************************************/
/* Procedure: getBaseAngle                                              Done */
/*            getBaseDist                                                    */
/*            getVolts              setVolts                                 */
/*    These are functions to allow access to current robot values.          */
/*****************************************************************************/

float getBaseAngle(void)          ( return(base_angl); )
float getBaseDist(void)           ( return(base_dis);  )
float getVolts(void)              ( return(volts);     )
void  setBaseAngle(float newang)  ( base_angl = newang; )
void  setVolts(float newvolts)    ( volts = newvolts;  )

/*****************************************************************************/
/* Procedure: atSDest        Checks if a cell is the sub-destination    Done */
/* Called by: checkAhead, checkAux, moveMe, selectMove, travel               */
/* Calls:     <none>                                                         */
/* Globals:   xsubd,ysubd (Read only)                                        */
/* Returns:   short - TRUE if cell is subdestination, FALSE (0) if it is not */
/*    Checks to see if the indicated grid cell is the robot's current        */
/*    subdestination.                                                        */
/*****************************************************************************/
short atSDest(
   int x,int y)        /* Coordinates of cell being checked for destination  */
{
 if ((x == xsubd) && (y == ysubd)) return(TRUE);
 return(FALSE);
} /** end atSDest() **/

/*****************************************************************************/
/* Procedure: atUDest     Checks if a cell is the ultimate destination  Done */
/* Called by: travel, travelAuxBack                                          */
/* Calls:     <none>                                                         */
/* Globals:   xdest,ydest (Read only)                                        */
/* Returns:   short - TRUE (1) if cell is destination, FALSE (0) if it is not*/
/*    Checks to see if the indicated grid cell is the robot's ultimate       */
/*    destination.                                                           */
/*****************************************************************************/
short atUDest(
   int x,int y)        /* Coordinates of cell being checked for destination  */
{
 if ((x == xdest) && (y == ydest)) return(TRUE);
 return(FALSE);
} /** end atUDest() **/
```

```
/*****************************************************************************/
/* Procedure: centerRobot        Put robot back in center of the grids  Done */
/* Called by: newCell                                                        */
/* Calls:     (cgrid)shift     - moves the certainty grid                    */
/*            (destack)shiftDS - changes destinations in destination stacks  */
/*            shiftTravel      - moves the travel grid                       */
/*            stopMe           - Halts the robot                             */
/*            mkdgrd           - Make new direction grid                     */
/* Globals:   robx,roby (Read, Write )                                       */
/*            begx,begy,xdest,ydest,xsubd,ysubd (Read, Write)                */
/*    Determines shift needed to replace robot in the designated center of   */
/*    the grid. Uses CgPt->shift to move the certainty grid and shiftTravel  */
/*    to adjust the travel and dead end grids.  Resets the starting position */
/*    and destinations to reflect the new coordinate system &  recalculates  */
/*    the direction grid. Uses DsPt->shiftDS to change the destination stack.*/
/*****************************************************************************/
void centerRobot(void)
{
 int dx,dy;            /* X & Y distances (cells) robot is offcenter        */
 stopMe();
 dx = robx - CENX;
 dy = roby - CENY;
 CgPt->shift(dx,dy);
 robx = robx - dx;
 roby = roby - dy;
 shiftTravel(dx,dy);
 begx = begx - dx;   begy = begy - dy;   /* Reset position reference cell    */
 xsubd = xsubd - dx; ysubd = ysubd - dy; /* Change subdest. to new coords    */
 xdest = xdest - dx; ydest = ydest - dy; /* Change dest. to new coordinates  */
 DsPt->shiftDS(dx,dy);
 mkdgrd(robx,roby,xsubd,ysubd,n_path,TRUE); /* Recalculate directional grid  */
} /** end centerRobot() **/

/*****************************************************************************/
/* Procedure: changeSpeed     Changes robot speed                    Done */
/* Called by: moveMe                                                         */
/* Calls:     (RWITim)setSpeed - sets the robot's speed                      */
/* Globals:   timer (Read, Write)                                            */
/*    This routine is in case I wish to tweak the acceleration parameters    */
/*    when changing speed. It currently does nothing except set the new      */
/*    speed using RwitPt->setSpeed();                                        */
/*****************************************************************************/
void changeSpeed(
   float,              /* oldspd: old robot speed in cm/sec (not used)      */
   float newspd)       /* Speed to change to                                */
{
 RwitPt->setSpeed(newspd);
 timer = timer + SET_TIME;  /* Simulated time for setspeed command          */
 DcPt->prtLne("*Speed changed to:"); DcPt->prtFlt(newspd);
 }  /** end changeSpeed() **/
```

```
/*****************************************************************************/
/* Procedure: checkAhead          Checks for obstacles along path      Done */
/* Called by: moveMe, checkSides                                             */
/* Calls:     atSDest    - Checks if a cell is the subdestination            */
/*            checkAux   - Does the distance calculation                     */
/* Globals:   offx, offy, robx,roby (Read only)                              */
/* Returns:   float - distance(mm) to the nearest object along bearing given */
/*    Checks the certainty grid for 5 cells along the bearing given.         */
/*    Returns the distance to the closest object found on this path or       */
/*    CHECKDIS if nothing is found.  If the first cell along the path has a  */
/*    higher badness potential than the current cell (indicating a           */
/*    possible direction change) then the distance returned is negative.     */
/*    The destination is treated as an obstacle so that the robot will slow  */
/*    as it is approached.                                                   */
/*****************************************************************************/
float checkAhead(
   float bearing)      /* Bearing (degrees) along which to check            */
{
 float angle;          /* Bearing in radians along which to check           */
 float dist = CHECKDIS;/* Distance (mm) to nearest object along bearing     */
 float x,y;            /* X,Y components of unit length along bearing       */
 float slope;          /* Change along Y direction for unit change in X     */
                       /* Also change along X for unit change in Y          */
 float yval;           /* Accumulator for changes along Y direction         */
 float xval;           /* Accumulator for changes along X direction         */
 int   flg = 1;        /* Flag for next cell has higher potential (-1)      */
 int   i,j;            /* Loop indices                                      */
 angle = degToRadian(bearing);
 x = (float)cos((double)angle);
 y = (float)sin((double)angle);
 if (abs(x) >= abs(y)) {          /* Increment along X axis                  */
    slope = y/x;
    yval = (float)roby + (offy/(float)CELL);
    if (x > 0.0) {                /* Increment in positive X direction       */
       if (bgrid[robx+1][(ROUND(yval+slope))] > bgrid[robx][roby]) flg = -1;
       for (i=robx; i <= robx + 5; i++) {
          j = ROUND(yval);
          if ((CgPt->getCg(i,j) >= n_tho) || (atSDest(i,j) == TRUE))
             return(checkAux(i,j,bearing,flg));
          if ((CgPt->getCg(i+1,j) >= n_tho) || (atSDest(i+1,j) == TRUE))
             return(checkAux(i+1,j,bearing,flg));
          yval = yval + slope;
       }
    } else {                      /* Increment in negative X direction       */
       if (bgrid[robx-1][(ROUND(yval-slope))] > bgrid[robx][roby]) flg = -1;
       for (i=robx; i >= robx - 5; i--) {
          j = ROUND(yval);
          if ((CgPt->getCg(i,j) >= n_tho) || (atSDest(i,j) == TRUE))
             return(checkAux(i,j,bearing,flg));
          if ((CgPt->getCg(i-1,j) >= n_tho) || (atSDest(i-1,j) == TRUE))
             return(checkAux(i-1,j,bearing,flg));
          yval = yval - slope;
    }  }
 } else {                         /* Increment along Y axis                  */
    slope = x/y;
    xval = (float)robx + (offx/(float)CELL);
    if (y > 0) {                  /* Increment in positive Y direction       */
       if (bgrid[(ROUND(xval+slope))][roby+1] > bgrid[robx][roby]) flg = -1;
       for (j=roby; j <= roby + 5; j++) {
          i = ROUND(xval);
          if ((CgPt->getCg(i,j) >= n_tho) || (atSDest(i,j) == TRUE))
             return(checkAux(i,j,bearing,flg));
          if ((CgPt->getCg(i,j+1) >= n_tho) || (atSDest(i,j+1) == TRUE))
```

```
            return(checkAux(i,j+1,bearing,flg));
         xval = xval + slope;
      )
   ) else (                    /* Increment in negative Y direction      */
      if (bgrid[(ROUND(xval-slope))][roby-1] > bgrid[robx][roby]) flg = -1;
      for (j=roby; j >= roby - 5; j--) {
         i = ROUND(xval);
         if ((CgPt->getCg(i,j) >= n_tho) || (atSDest(i,j) == TRUE))
            return(checkAux(i,j,bearing,flg));
         if ((CgPt->getCg(i,j-1) >= n_tho) || (atSDest(i,j-1) == TRUE))
            return(checkAux(i,j-1,bearing,flg));
         xval = xval - slope;
      )
   )
)
dist = (float)flg * dist;
return(dist);
}  /** end checkAhead() **/

/*****************************************************************************/
/* Procedure: checkAux          Assists checkAhead                    Done */
/* Called by: checkAhead                                                     */
/* Calls:     atSDest    - Check if cell is the subdestination               */
/*            distance   - Determines distance betweeen two points          */
/*            offCenter  - Find component of x,y vector in new direction    */
/* Returns:   float - adjusted distance(mm) between robot and indicated cell */
/* Globals:   offx,offy,robx,roby (Read only)                                */
/*   Calculates the distance between the certainty grid cell and the robot's */
/*   cell. Makes adjustments to this distance to account for the robot not   */
/*   being centered in its cell. If the distance is negative (a possiblity   */
/*   due to the adjustments) it it set to 1.0. If the grid cell is also the  */
/*   destination, then the distance is set to the larger of the actual       */
/*   distance and a dummy value. The dummy value is used to cause the robot  */
/*   to slow but not stop as it would for a close object.                    */
/*****************************************************************************/
float checkAux(
   int i, int j,       /* Grid coordinates of obstacle or destination       */
   float bearing,      /* Direction being checked                           */
   int flg)            /* Flag for first cell along bearing has higher poten.*/
{
 float dist;           /* Distance (mm) from robot to obstacle or destination*/
 float adjust;         /* Boundary Adjustments (mm) to dist read from cgrid  */
 adjust = ADJUSTS + offCenter(offx,offy,bearing);
 dist = distance(robx,roby,i,j);
 dist = (dist * (float)CELL) - adjust;
 if (atSDest(i,j) == TRUE) dist = ((dist >= CLOSE) ? dist : CLOSE);
 if (dist < 0.0) dist = 1.0;
 else dist = (float)flg * dist;
 return(dist);
} /** end checkAux() **/

/*****************************************************************************/
/* Procedure: checkSides     Check for obstacles to sides                Done */
/* Called by: moveMe                                                         */
/* Calls:     checkAhead - checks a certain direction for obstacles          */
/*            VoicePt->Say - robot speech routine                            */
/* Returns:   float - slew rate to turn away from perceived obstacle         */
/* Globals:   base_dis (Read only)                                           */
/*    Checks the distance to the nearest object to either side of the robot. */
/*    Compares the distance with the previous distance and calculates a turn */
/*    rate away if the distance is decreasing and small.                     */
/*    If both sides have objects then the turn rates are combined. When the  */
/*    distance has decreased by over a cell then it is assumed to be due to  */
/*    the granularity of the map. A smaller value (1/3) is used instead.     */
/*    Right turns are negative and left turns are positive.                  */
/*****************************************************************************/
float checkSides(
   float bearing,        /* Direction (degrees) of robot travel             */
   float speed)          /* Robot's current speed (cm/sec)                  */
{
 float distl,distr;      /* Distance (mm) to closest object to left, right  */
 float side;             /* Decrease (mm) in side distance to object        */
 float fwrd;             /* Distance (mm) traveled by robot since last check */
 float theta;            /* Angle of travel line to wall                    */
 float chgslew = 0.0;    /* Slew rate to turn from obstacle                 */
 static float olddl = 0.0; /* Previous distance (mm) to left object         */
 static float olddr = 0.0; /* Previous distance (mm) to right object        */
 static float oldis = 0.0; /* Previous distance (cm) traveled by robot      */
 distl = checkAhead(bearing+90.0);
 distr = checkAhead(bearing-90.0);
 distl = (float)fabs((double)distl);
 distr = (float)fabs((double)distr);
 if (speed != 0.0) {                        /* Only check when robot moving */
    fwrd = (base_dis - oldis) * 10.0;
    if ((distl < olddl) && (distl < (2.0 * (float)CELL))) {
       if (olddl >= CHECKDIS) chgslew = 50.0;
       else {
          side = olddl - distl;
          if (side >= (float)CELL) side = side - (float)CELL + 66.0;
          theta = atan((double)(side/fwrd));
          theta = radToDegree(theta);
          chgslew = -theta * speed / CELLCM;   /* turn in negative directiion*/
          VoicePt->say("wallleft");
          DcPt->prtLne("CHECKSIDES: Wall left"); DcPt->prtFlt(distl,olddl);
          DcPt->prtFlt(fwrd,side);  DcPt->prtFlt(theta,chgslew);
       }
    )
    if ((distr < olddr) && (distr < (2.0 * (float)CELL))) {
       if (olddr >= CHECKDIS) chgslew = 50.0;
       else {
          side = olddr - distr;
          if (side >= (float)CELL) side = side - (float)CELL + 66.0;
          theta = atan((double)(side/fwrd));
          theta = radToDegree(theta);
          chgslew = chgslew + (theta * speed / CELLCM);     /* turn in + dir.*/
          VoicePt->say("wallright");
          DcPt->prtLne("CHECKSIDES: Wall right"); DcPt->prtFlt(distr,olddr);
          DcPt->prtFlt(fwrd,side); DcPt->prtFlt(theta,chgslew);
       }
    )
 )
 olddl = distl;  olddr = distr;
 oldis = base_dis;
 if (olddl <= 1.0) olddl = 25.0;    /* If distance is <= 1 set to 25 so can  */
```

```
 if (olddr <= 1.0) olddr = 25.0;    /* decrease for next check              */
 return(chgslew);
} /** end checkSides() **/
```

```
/*****************************************************************************/
/* Procedure: collision       Check if robot is in occupied cell        Done */
/* Called by: travel                                                         */
/* Calls:     (cgrid)  setCgrid    - change a certainty grid value           */
/*            (display)say         - robot speech routine                    */
/*            (RWITim) executeMove - back Luey up                            */
/*            (RWITim) kill        - halt movement                           */
/*            newCell              - housekeeping upon entering new cell     */
/*            update               - determine robot location                */
/* Globals:   base_angl, base_dis, robx, roby (Read only)                    */
/*    This is a dummy routine used as a cheap collision sensor. Instead of   */
/*    reading an external sensor, it merely looks at the map to see if the   */
/*    current cell or the 3 cells in front of the robot are occupied. If the */
/*    robot projects more than 1/2 way into an occupied cell then a collision*/
/*    is assumed. Because the robot is larger than a cell it only has to be  */
/*    CELL - ROBOT_DIAMETER/2  = 50mm off center in a cell to project across */
/*    the middle of the next.  If a collision occurs and the robot was moving*/
/*    then it is stopped and backed up a little.                             */
/*****************************************************************************/
void collision(void)
{
 float angle;               /* Robot angle / 45                */
 int   dir;                 /* Octant robot is facing          */
 short hit = FALSE;         /* Flag for robot in occupied cell */
 int   i;                   /* Loop indes                      */
 short moving = TRUE;       /* Flag for robot moving           */
 int   occx, occy;          /* Cell where collision occurred   */
 float cert, conf;          /* Certainty and confidence of cell*/
 const float BACKUP = (EXTEND +10.0)/10.0;/* Amount to backup after collision*/
 if (spdfac == 0) moving = FALSE;
 angle = base_angl;
 if (angle < 0.0) angle = angle + 360.0;  /* determine which of 8 directions */
 angle = angle / 45.0;                    /* robot is going                  */
 dir = ROUND(angle);
 for (i = dir-1; i<= dir+1; i++){         /* for three cells in that dir.    */
    if (!hit)  switch(i) {
    case 0: case 8:
       if ((CgPt->getCg(robx+1,roby) >= n_tho) && (offx > EXTEND)) {
          hit = TRUE;
          tgrid[robx+1][roby] = tgrid[robx+1][roby] + TRAVEL_VALUE;
          occx = robx+1; occy = roby;
       }
    break;
    case 1: case 9:
       if ((CgPt->getCg(robx+1,roby+1) >= n_tho) && (offx > EXTEND) &&
                                                    (offy > EXTEND)) {
          hit = TRUE;
          tgrid[robx+1][roby+1] = tgrid[robx+1][roby+1] + TRAVEL_VALUE;
          occx = robx+1; occy = roby+1;
       }
    break;
    case 2:
       if ((CgPt->getCg(robx,roby+1) >= n_tho) && (offy > EXTEND)) {
          hit = TRUE;
          tgrid[robx][roby+1] = tgrid[robx][roby+1] + TRAVEL_VALUE;
          occx = robx; occy = roby+1;
       }
    break;
    case 3:
       if ((CgPt->getCg(robx-1,roby+1) >= n_tho) && (offx < EXTEND) &&
                                                    (offy > EXTEND)) {
          hit = TRUE;
          tgrid[robx-1][roby+1] = tgrid[robx-1][roby+1] + TRAVEL_VALUE;
```

```
         occx = robx-1; occy = roby+1;
      )
   break;
   case 4:
      if ((CgPt->getCg(robx-1,roby) >= n_tho) && (offx < EXTEND)) {
         hit = TRUE;
         tgrid[robx-1][roby] = tgrid[robx-1][roby] + TRAVEL_VALUE;
         occx = robx-1; occy = roby;
      )
   break;
   case 5:
      if ((CgPt->getCg(robx-1,roby-1) >= n_tho) && (offx < EXTEND) &&
                                                   (offy < EXTEND)) {
         hit = TRUE;
         tgrid[robx-1][roby-1] = tgrid[robx-1][roby-1] + TRAVEL_VALUE;
         occx = robx-1; occy = roby-1;
      )
   break;
   case 6:
      if ((CgPt->getCg(robx,roby-1) >= n_tho) && (offy < EXTEND)) {
         hit = TRUE;
         tgrid[robx][roby-1] = tgrid[robx][roby-1] + TRAVEL_VALUE;
         occx = robx; occy = roby-1;
      )
   break;
   case 7: case -1:
      if ((CgPt->getCg(robx+1,roby-1) >= n_tho)&&(offx > EXTEND) &&
                                                 (offy < EXTEND)) {
         hit = TRUE;
         tgrid[robx+1][roby-1] = tgrid[robx+1][roby-1] + TRAVEL_VALUE;
         occx = robx+1; occy = roby-1;
      )
   break;
   default:
      DcPt->prtMsg("*COLLISION: switch out of bounds",i,(int)angle);
   )
)
if (CgPt->getCg(robx,roby) >= n_tho) {
   hit = TRUE;
   egrid[robx][roby] = 1;
   occx = robx; occy = roby;
)
if (hit == TRUE) {
   RwitPt->kill();
   VoicePt->say("boom");
   DcPt->prtMsg("*COLLISION at cell",occx,occy);
   DcPt->prtFlt(CgPt->getCg(occx,occy),CgPt->getConfi(occx,occy));
   DcPt->prtMsg("    Robot at ",robx,roby); DcPt->prtFlt(offx,offy);
   nwcell = update(0.0,-1.0) || nwcell;
   if (nwcell) newCell();
   spdfac = 0;
   slew = 0.0;
   if (moving) {
      cert = CgPt->getCg(occx,occy);       /* save the values for the cell */
      conf = CgPt->getConfi(occx,occy);    /* collided with                */
      RwitPt->executeMove(-5,BACKUP,1);
      nwcell = update(base_angl,base_dis-BACKUP);
      DcPt->prtMsg("   Backing up to ",robx,roby); DcPt->prtFlt(offx,offy);
      for (i=0; ((i <= 2) && (CgPt->getCg(robx,roby) >= n_tho)); i++) {
         RwitPt->executeMove(-5,BACKUP,1);
         nwcell = update(base_angl,base_dis-BACKUP);
         DcPt->prtMsg("   Backed more to ",robx,roby);DcPt->prtFlt(offx,offy);
      )
   CgPt->setGrid(occx,occy,cert,6.0);      /* Reset grid values clobbered */
      )
      if (neighbor(robx,roby,xsubd,ysubd)) {
         DcPt->prtMsg("Moving sub-destination to",robx,roby);
         egrid[xsubd][ysubd] = 1;
         xsubd = robx;
         ysubd = roby;
         DsPt->changeDS(robx,roby);
      )
   )
) /** end collision() **/
```

```
/*****************************************************************************/
/* Procedure: destCheck    Checks that subdestination is unoccupied     Done */
/* Called by: travel                                                         */
/* Calls:     (cgrid)  getCg     - get certainty values                      */
/*            (destack)changeDS  - change current subdestination             */
/*            (display)prtMsg    - output message to terminal                */
/* Returns:   short - quit flag, TRUE if subdest cannot be reached.          */
/*    If the subdestination is occupied then the N1 neighbor with the lowest */
/*    certainty value is used to replace it and the destination grid and     */
/*    stack modified accordingly. If no unoccupied neighbor exists then      */
/*    the current subdestination cannot be reached and TRUE is returned.     */
/*****************************************************************************/
short destCheck(void)
{
 float dissqr;          /* change in directional potential for adjacent cell */
 int   minx,miny;       /* neighbor cell with lowest certainty               */
 float mincert = n_tho; /* certainty of that cell                            */
 int i;
 if (CgPt->getCg(xsubd,ysubd) >= 0.66) {
    minx = xsubd; miny = ysubd;
    for (i=0; i<=7; i++) {
       if (CgPt->getCg(xsubd+XNBOR[i],ysubd+YNBOR[i]) < mincert) {
          minx = xsubd + XNBOR[i];
          miny = ysubd + YNBOR[i];
          mincert = CgPt->getCg(minx,miny);
       }
    }
    if ((minx != xsubd) || (miny != ysubd)) {
       dissqr = dgrid[xsubd+1][ysubd+0] / 2.0;       /* adjust dgrid */
       dgrid[minx][miny] = 0.0;
       for (i=0; i<=7; i++) dgrid[minx+XNBOR[i]][miny+YNBOR[i]] = dissqr;
       egrid[xsubd][ysubd] = 1;       /* mark previous cell as dead end */
       xsubd = minx; ysubd = miny;                 /* replace subdestination */
       DsPt->changeDS(minx,miny);
       DcPt->prtMsg("*DESTCHECK: subdest occupied, shifting to",minx,miny);
    } else {
       DcPt->prtMsg("*DESTCHECK: subdest occupied, no path",xsubd,ysubd);
       DsPt->changeDS(robx,roby);
       xsubd = robx; ysubd = roby;
       return(TRUE);
    }
 }
 return(FALSE);
}  /** end destCheck() **/
/*****************************************************************************/
/* Procedure: distance     Finds distance between two points           Done */
/* Called by: checkAux, mkdgrd, (simulat)sondist                             */
/* Calls:     <none>                                                         */
/* Returns:   float: distance between the two points                         */
/*    Calculates the distance between two points in the XY cartesian plane   */
/*    using the pythagorean theorem.                                         */
/*****************************************************************************/
float distance(
   int x1, int y1, int x2, int y2) /* Coordinates of the two points          */
{
 int   xs,ys;            /* Square of X & Y distances                        */
 float dist;             /* Distance between the points                      */
 xs = ((x1 - x2) * (x1 - x2));
 ys = ((y1 - y2) * (y1 - y2));
 dist = (float)sqrt((double)(xs + ys));
 return(dist);
} /** end distance() **/
```

```
/*****************************************************************************/
/*           the elimination arrays, elimCur and elimNex                     */
/*    Each element of the elimCur and elimNex array tells whether a specific */
/*    cell has been eliminated from consideration as the next local          */
/*    destination. A cell is eliminated if it was within the current or      */
/*    previous search neighborhood (within 2 cells) and was not the one      */
/*    selected as having the lowest potential.  This is useful in keeping    */
/*    Luey moving when his route requires an increase in potential. ElimCur  */
/*    contains cells which have been previously eliminated from other        */
/*    locations while elimNex contains cells which were eliminated from the  */
/*    current cell. When Luey changes cells, elimNex is copied to ElimCur    */
/*    (with appropriate index changes) and elimNex reinitialized.            */
/*    The elements are accessed directly using the indices  [dx+2][dy+2]     */
/*    where dx and dy are cell offsets from the current cell.                */
/*****************************************************************************/
/* Procedure: elimInit     Initializes the elimination arrays          Done */
/* Called by: travel                                                         */
/* Calls:     <none>                                                         */
/* Globals:   elimCur, elimNex (Write only)                                  */
/*    Initializes the elimination arrays. The current array is set to FALSE  */
/*    (no cells eliminated) and the next array is set to TRUE (all elim.)    */
/*****************************************************************************/
void elimInit( void )
{
 int x,y;                /* Array indices                                    */
 for (x=0; x<5; x++) {
    for (y=0; y<5; y++) {
       elimCur[x][y] = FALSE;
       elimNex[x][y] = TRUE;
    }
 }
} /** end elimInit() **/
/*****************************************************************************/
/* Procedure: elimStore     Transfers elimination arry to a new point  Done */
/* Called by: selectMove                                                     */
/* Calls:     <none>                                                         */
/* Globals:   elmx,elmy,elimNex (Read Write)  elimCur (Write only)           */
/*    Copies the elimNex array into the elimCur array and shifts it to allow */
/*    for the change in current cell. Cells which where not in the elimNex   */
/*    array are set to FALSE. Initiializes a new elemNex at TRUE.            */
/*****************************************************************************/
void elimStore(
 int newx, int newy)     /* New center cell for elimination array            */
{
 int dx,dy;              /* Difference in x,y indices from old to new cell   */
 int x,y;                /* Array indices                                    */
 dx = newx - elmx;
 dy = newy - elmy;
 elmx = newx;  elmy = newy;
 for (x=0; x<5; x++) {
    for (y=0; y<5; y++) {
       if ((x+dx >= 5) || (y+dy >= 5) || (x+dx < 0) || (y+dy < 0))
          elimCur[x][y] = FALSE;
       else elimCur[x][y] = elimNex[x+dx][y+dy];
    }
 }
 for (x=0; x<5; x++) {       /* Initialize elimNex at TRUE (all eliminated)  */
    for (y=0; y<5; y++) {
       elimNex[x][y] = TRUE;
    }
 }
} /** end elimStore() **/
```

```
/*****************************************************************************/
/* Procedure: findPosition     Finds robot position in grid              Done */
/* Called by: update                                                         */
/* Calls:     (cgrid)setCgrid - Mark cell just traversed as empty            */
/* Globals:   cgrid (Write only) base_angl(Read only)                        */
/*            robx,roby,movx,movy,offx,offy,tgrid,bgrid (Read Write)         */
/* Returns:   short - TRUE(1) if new cell is entered, FALSE (0) if not       */
/*    Determines the robot's current position in the grid by calculating its */
/*    X & Y displacement from its last known position.  Uses the distance    */
/*    traveled and the previous and current direction to make the calculation*/
/*    If the new position is in a different grid cell than the previous      */
/*    position, then the travel grid is updated and the previous cell marked */
/*    as empty.  For constant angles, negative distances are allowed.        */
/*    Not verified for negative distance when angle changes.                 */
/*****************************************************************************/
int findPosition(
   float distance,       /* Distance (cm) traveled by robot since last update*/
   float bearing)        /* Robot's angle in degrees                         */
{
 float  ang;             /* Average direction of robot travel in degrees     */
 float  angle;           /* Average direction of robot travel in radians     */
 float  delta;           /* Direction change since last findPosition         */
 double deltad;          /* Half of direction change since last findPosition */
 float  dist;            /* Distance (mm) of robot travel                    */
 short  nwCell = FALSE;  /* Flag for new grid cell entered                   */
 float  radius;          /* Radius (mm) of circular path traveled by robot   */
 float  chord;           /* Length (mm) of chord of robots path              */
 int    cellx,celly;     /* Robot's new position in grid                     */
 dist = distance * 10.0; /* Convert to millimeters                           */
 if (abs(bearing - base_angl) < 0.2) {   /* Robot went in a straight line    */
    angle = degToRadian(bearing);
    movx = movx + (dist * (float)cos((double)angle));   /* X & Y components  */
    movy = movy + (dist * (float)sin((double)angle));   /* of movement       */
 } else {                               /* Robot went in circular arc       */
    delta = (float)(base_angl - bearing);
    delta = degToRadian(delta);
    radius = dist/delta; /*????? abs? */
    deltad = (double)(delta/2.0);
    chord = 2.0 * radius * (float)sin(deltad); /* Straight line dist traveled*/
    ang = (base_angl + bearing) / 2.0;   /* Average direction of travel     */
    angle = degToRadian(ang);
    movx = movx + (chord * (float)cos((double)angle)); /* X & Y components  */
    movy = movy + (chord * (float)sin((double)angle)); /* of movement       */
 }
 cellx = begx + (ROUND(movx/((float)CELL)));   /* Determine current cell     */
 celly = begy + (ROUND(movy/((float)CELL)));
 offx = (int)(movx - (CELL * (cellx - begx))); /* Determine position in cell */
 offy = (int)(movy - (CELL * (celly - begy)));
 if ((cellx != robx) || (celly != roby)) {     /* If Luey is in a new cell   */
    tgrid[robx][roby] = tgrid[robx][roby] + TRAVEL_VALUE;
    egrid[robx][roby] = 1;
    CgPt->setCgrid(robx,roby,0.0,-1.0);
    bgrid[robx][roby] = bgrid[robx][roby] + TRAVEL_VALUE;      /*?????*/
    robx = cellx;  roby = celly;
    nwCell = TRUE;
    DcPt->prtMsg("*FINDPOSITION: new cell",robx,roby); DcPt->prtInt(offx,offy);
 }
 return(nwCell);
} /** end findPosition() **/
```

```
/*****************************************************************************/
/* Procedure: fzbgrd     Fuzzes local badness grid                       Done */
/* Called by: getDest, stopMe, travelAuxBack                                 */
/* Calls:     <none>                                                         */
/* Globals:   dgrid,tgrid,cgrid,robx,roby (Read only)   bgrid (Write only)   */
/*    Combines the direction, travel and certainty grids by summing the      */
/*    corresponding elements. The certainty values are fuzzed slightly by    */
/*    averaging neighbor cells before summing in the badness grid. This      */
/*    makes obstacles appear slightly larger which aids in avoiding them.    */
/*    Occupied cells are not fuzzed to avoid accidental erasures.            */
/*    Since the robot only looks at cells within 2  of its current position  */
/*    only the small section (7x7) surrounding the robot is calculated.      */
/*****************************************************************************/
void fzbgrd(void)
{
int   i,j;                  /* Loop indicies */
int   ic,jc;                /* Loop indices ajusted to access grids          */
int   ip,im,jp,jm;          /* ic+1, ic-1, jc+1, jc-1 for efficiency         */
float tmp;                  /* Sum of certainty values for surrounding cells */
float tmpgrd[9][9];         /* Local cgrid  after 1 fuzz  cycle              */
 for (i=0; i<=8; i++) {
    ic = i + robx - 4;
    ip = ic + 1;
    im = ic - 1;
    for (j=0; j<=8; j++) {
       jc = j +  roby - 4;
       if (CgPt->getCg(ic,jc) >= n_tho) tmpgrd[i][j] = CgPt->getCg(ic,jc);
       else {
          jm = jc - 1;
          jp = jc + 1;
          tmp = CgPt->getCg(im,jp) + CgPt->getCg(ic,jp) + CgPt->getCg(ip,jp) +
                CgPt->getCg(im,jc) +                      CgPt->getCg(ip,jc) +
                CgPt->getCg(im,jm) + CgPt->getCg(ic,jm) + CgPt->getCg(ip,jm);
          tmpgrd[i][j] = (0.7 * CgPt->getCg(ic,jc)) + (0.3 * tmp / 8.0);
       }
    }
 }
 for (i=1; i<=7; i++) {
    ic = i + robx - 4;
    ip = i + 1;
    im = i - 1;
    for (j=1; j<=7; j++) {
       jc = j + roby - 4;
       if (CgPt->getCg(ic,jc) >= n_tho) bgrid[ic][jc] = CgPt->getCg(ic,jc) +
                   dgrid[ic][jc];
       else {
          jp = j + 1;
          jm = j - 1;
          tmp = tmpgrd[im][jp] + tmpgrd[i][jp] + tmpgrd[ip][jp] +
                tmpgrd[im][j]  +                 tmpgrd[ip][j]  +
                tmpgrd[im][jm] + tmpgrd[i][jm] + tmpgrd[ip][jm];
          tmp = (0.7 * tmpgrd[i][j]) + (0.3 * tmp / 8.0);
          if (tmp < 0.12) tmp = 0.0;          /* remove small perturbations */
          bgrid[ic][jc] = tmp + dgrid[ic][jc];
 ) ) )
 /**** add travel grid values if local navigator ****/
 if (n_nav == N_LOCAL) {
    for (i = robx-3; i <= robx + 3; i++) {
       for (j = roby - 3; j<= roby + 3; j++) {
          bgrid[i][j] = bgrid[i][j] + tgrid[i][j];
 ) ) )
} /** end fzbgrd() **/
```

```
/*****************************************************************************/
/* Procedure: getAngle     Gets and normalizes robot angle              Done */
/* Called by: update                                                         */
/* Calls:     (RWITim)queryAngle - get robot's orientation                   */
/* Globals:   off_angle (Read only)                                          */
/* Returns:   float - the current angle(degrees) of the robot                */
/*    Reads the current robot direction and converts it to a counterclockwise*/
/*    angle between -180 and 180 degrees ready for use by the program. Since */
/*    robot is clockwise positive the angle is multiplied by -1.             */
/*****************************************************************************/
float getAngle(void)
{
 float angle;                              /* Robot's current orientation  */
 angle = RwitPt->queryAngle() + off_angle;
 angle = -1.0 * angle;
 if (angle > 180.0) angle = angle - 360.0;
 else if (angle < -180.0) angle = angle + 360.0;
 return(angle);
} /** end getAngle() **/

/*****************************************************************************/
/* Procedure: getDest         Sets up a new ultimate destination        Done */
/* Called by: (driver)test1                                                  */
/* Calls:     (destack)clearCS - Clears the destination stack                */
/*            (destack)pushDS  - Pushes a destination onto the stack         */
/*            fzbgrd           - fuzzes the local badness grid               */
/*            mkbgrd           - makes the badness grid                      */
/*            mkdgrd           - makes the destination grid                  */
/*            mktgrid          - initializes the travel grid                 */
/* Globals:   xdest, ydest, xsubd, ysubd (Write only)                        */
/*            robx, roby, cgrid (Read only)                                  */
/*    Sets up navigator for a traverse to a new destination.  Clears the     */
/*    destination stacks then reinitializes it with a D_START node           */
/*    containing the current robot position. Resets the travel grid, and     */
/*    recalculates the destination and badness grids.                        */
/*****************************************************************************/
void getDest
    (int xd, int yd)          /* destination coordinates    */
{
 int i,j;                     /* loop indices               */
 xdest = xd;  ydest = yd;
 xsubd = robx;  ysubd = roby;
 DsPt->clearDS();
 DsPt->pushDS(robx,roby,robx,roby,D_START);
 for (i=0; i<SZE; i++) {                    /* initialize dead end data */
    for (j=0; j<SZE; j++) {
       egrid[i][j] = 0;
    }
 }
 mkdgrd(robx,roby,xd,yd,n_path,TRUE);       /* make new direction grid */
 mktgrd();                                  /* initialize travel grid  */
 mkbgrd();                                  /* make new badness grid   */
 fzbgrd();                                  /* fuzz local badness grid */
}  /** end getDest() **/
```

```
/*****************************************************************************/
/* Procedure: getDistance     Gets and normalizes robot distance        Done */
/* Called by: update                                                         */
/* Calls:     (RWITim)queryDistance - get robot distance                     */
/* Globals:   base_dis (Read only) off_dist (Read, Write)                    */
/* Returns:   int - distance (cm) traveled by robot since program started    */
/*    Queries the robot for the total distance traveled and adjusts it to    */
/*    distance traveled since the program was started. If the new distance   */
/*    is no where near the previous distance (which occurs following a       */
/*    communications error) an error message is printed and offset distance  */
/*    reset.                                                                 */
/*****************************************************************************/
float getDistance(void)
{
 float dist;                               /* Distance (cm) traveled by robot */
 dist = RwitPt->queryDistance() + off_dist;
 if (abs(dist-base_dis) > 1000) {
    DcPt->prtLne("GETDISTANCE: ** ERROR ** Position lost, Resetting off_dist");
    off_dist = base_dis - dist + off_dist;
 }
 return(dist);
}  /** end getDistance() **/

/*****************************************************************************/
/* Procedure: findAngle      Finds angle of line between two points     Done */
/* Called by: markDeadEndBox, markDeadEndAll                                 */
/* Calls:     <none>                                                         */
/*    Using arctan finds the angle of the line between two points.           */
/*****************************************************************************/
float findAngle(
    int xsrce, int ysrce, int xdead, int ydead) /* Source & Dead end points  */
{
 float angle;            /* Angle of line from checkpoint to deadend         */
 float tanang;           /* tangent of angle to deadend                      */
 float diffx, diffy;     /* Cell difference between endpoints of direction   */
/*************** Find the angle of line from source to deadend ***********/
 diffx = xdead - xsrce;  diffy = ydead - ysrce;
 if ((diffx > 0) && (diffy >= 0)) {               /* First quadrant */
    tanang = ((float)diffy) / ((float)diffx);
    angle = (float)atan((double)tanang);
 } else if ((diffx < 0) && (diffy >= 0)) {       /* Second quadrant */
    tanang = ((float)-diffy) / ((float)diffx);
    angle = 3.1416 - (float)atan((double)tanang);
 } else if ((diffx < 0) && (diffy < 0)) {        /* Third quadrant */
    tanang = ((float)diffy) / ((float)diffx);
    angle = -3.1416 + (float)atan((double)tanang);
 } else if ((diffx > 0) && (diffy < 0)) {       /* Forth quadrant  */
    tanang = ((float)diffy) / ((float)diffx);
    angle = (float)atan((double)tanang);
 } else                                          /* diffx = 0     */
    angle = (diffy >= 0) ? 1.570796: -1.570796 ;  /* +/- Pi/2 */
 return(angle);
} /** end findAngle() **/
```

```
/*****************************************************************************/
/* Procedure: look          Make a little map in none present          Done */
/* Called by: travelAuxBack,travelAuxIncrement,travelAuxPath,travelAuxLocal  */
/* Calls:     (cgrid)sonarScan   - read the sonars                           */
/*            (cgrid)circleScan  - rotate and read sonars                    */
/*            getAngle           - find robot's orientation                  */
/*            (display)drawCgrid - Redraw and display the map                */
/* Globals:   robx, roby (Read only): base_angl (Read, Write)                */
/*    Checks for a map of the immediate area. Does a circle scan or a single */
/*    sonar read depending on if a map is already present.  Should only be   */
/*    called when robot is not moving.                                       */
/*****************************************************************************/
void look(void)
{
 if ((CgPt->getCg(robx-1,roby) < n_the)||(CgPt->getCg(robx+1,roby) < n_the) ||
     (CgPt->getCg(robx,roby-1) < n_the)||(CgPt->getCg(robx,roby+1) < n_the)) {
     CgPt->sonarScan(base_angl,robx,roby,offx,offy);
 } else {
    CgPt->circleScan(base_angl,15,23,robx,roby,offx,offy,0);
    base_angl = getAngle();
 }
 DxPt->drawCgrid(1);
}
```

```
/*****************************************************************************/
/* Procedure: markDeadEndAll   Set dead end flags from dead end to source    */
/* Called by: travelAuxBack                                                  */
/* Calls:     findAngle - Find angle of a line                               */
/*            markLine  - Put a line into the mark array                     */
/*    Does a breadth first traverse through all non occupied cells starting  */
/*    from the far point and stopping when the checkpoint is reached. Sets   */
/*    the dead end flag for the neighbors of every cell traversed.           */
/*****************************************************************************/
void markDeadEndAll(
    int xsrce, int ysrce,      /* previous checkpoint              */
    int xdead, int ydead)      /* farthest point reached in deadend */

{
 float angle;            /* Angle of line from checkpoint to deadend     */
 float dir1;             /* directions of normals to the line            */
 int   x1d,y1d,x2d,y2d;  /* coordinates of limit line through dead end   */
 int   xlim,ylim;        /* Center of limit line                         */
 int   xc,yc;            /* distance(cells) from line to box coordinates */
 float xx,yy;            /* Offset of box corner from source or dest.    */
 Cell *botpt;            /* pointer to bottom of list                    */
 Cell *curpt;            /* pointer to current node of list              */
 Cell *newpt;            /* pointer to node being added to list          */
 Cell *toppt;            /* pointer to top of list                       */
 short        done = FALSE;
 int          nborx,nbory; /* indices of cells next to current cell          */
 int          x,y,i;     /* loop indices, indices of current cell          */
 for (x=0; x<SZE; x++) {                      /* unmark all cells         */
    for (y=0; y<SZE; y++) mark[x][y] = FALSE;
 }
/******** Put in a limiting line at right angle to line through points *******/
 angle = findAngle(xsrce,ysrce,xdead,ydead);
 dir1 = (angle + 1.570796);              /* 90 deg * pi / 180 */
 xx = 20.0 * (float)cos((double)dir1);
 yy = 20.0 * (float)sin((double)dir1);
 xc = ROUND(xx);
 yc = ROUND(yy);
 xlim = (xsrce < xdead) ? (xdead + 1) : (xdead - 1);
 ylim = (ysrce < ydead) ? (ydead + 1) : (ydead - 1);
 x1d = xlim + xc;     y1d = ylim + yc;
 x2d = xlim - xc;     y2d = ylim - yc;
 markLine(x1d,y1d,x2d,y2d);
/****** Now set the dead end flags for all points from limit line to sourc **/
 toppt = new(Cell);                          /* Store position in list     */
 botpt = toppt;
 toppt->x = xdead;
 toppt->y = ydead;
 toppt->npt = NULL;
 mark[xdead][ydead] = TRUE;
 egrid[xdead][ydead] = 1;
 while (toppt != NULL) {          /* While list not empty */
    curpt = toppt;                /*    pop top node      */
    toppt = toppt->npt;
    x = curpt->x;
    y = curpt->y;
    if ((x == xsrce) && (y == ysrce))  done = TRUE;
    for (i = 0; i<= 3; i++) {            /* for the 4 adjacent neighbors */
       nborx = x + XNBOR[i];
       nbory = y + YNBOR[i];
       if ((nbory >= 0) && (nbory < SZE) && (nborx >= 0) && (nborx < SZE)
                                          && (!mark[nborx][nbory])) {
          if ((nborx == xsrce) && (nbory == ysrce)) done = TRUE;
          mark[nborx][nbory] = TRUE;
```

```
         egrid[nborx][nbory] = 1;
         if (( CgPt->getCg(nborx,nbory) <= n_the) && (!done)) {
            if (curpt != NULL) {           /* Reuse current node    */
               newpt = curpt;
               curpt = NULL;
               if (toppt == NULL) toppt = newpt;
            } else newpt = new(Cell);      /* Allocate new node     */
            newpt->x = nborx;              /* Add to bottom of list */
            newpt->y = nbory;
            newpt->npt = NULL;
            if (botpt != newpt) botpt->npt = newpt;
            botpt = newpt;
         )
      )
   )
   if (curpt != NULL) delete(curpt);  /* Delete current node if not reused */
 )
} /** end markDeadEndAll() **/
```

```
/*****************************************************************************/
/* Procedure: markDeadEndBox     Set dead end flags for boxed area      Done */
/* Called by: travelAuxBack                                                  */
/* Calls:     markLine      - sets mark array for points on a line segment   */
/* Globals:   mark, egrid (Write only)                                       */
/*    Sets the dead end flag for cells enclosed in the box 6 wide along the  */
/*    line from the source to the dead end point. Finds angle of line thru   */
/*    the points & uses normal  to find the 4 corner points.  Using the mark */
/*    array, marks the lines between the corner points and the points        */
/*    themselves. The line on the dead end is marked one cell farther from   */
/*    the source so that the dead end point will lie inside the box and can  */
/*    be used as the starting point for a breadth first traverse of all      */
/*    empty cells enclosed in the box. Sets the dead end flag for all        */
/*    neighbors of each traversed cell. Does not do a complete job on small  */
/*    boxes or those with partitioned (by obstacles) interiors. Boxes with   */
/*    no interior cells have the dead end flag set for only the two line     */
/*    endpoints. (Should happen only when source and dead points are same)   */
/*****************************************************************************/
void markDeadEndBox(
    int xsrce, int ysrce, int xdead, int ydead) /* Source & Dead end points  */
{
 float angle;           /* Angle of line from checkpoint to deadend     */
 float dir1;            /* directions of normals to the line            */
 int   x1s,y1s,x2s,y2s; /* coordinates of box near source               */
 int   x1d,y1d,x2d,y2d; /* coordinates of box near dead end             */
 int   xlim,ylim;       /* Center of limit line near dead end           */
 int   xc,yc;           /* distance(cells) from line to box coordinates */
 float xx,yy;           /* Offset of box corner from source or dest.    */
 int   i;               /* Loop index                                   */
 int   nborx,nbory;     /* Indices of square neighbor cells             */
 int   x,y;             /* Indices of cell under consideration          */
 Cell *toppt;           /* Top node of list                             */
 Cell *botpt;           /* Bottom node of list                          */
 Cell *curpt;           /* Current node of list                         */
 Cell *newpt;           /* Node being added to list                     */
/**************** Find the cell coordinates of the 4 corners **************/
 angle = findAngle(xsrce,ysrce,xdead,ydead);
 dir1 = (angle + 1.570796);             /* 90 deg * pi / 180 */
 xx = 3.0 * (float)cos((double)dir1);
 yy = 3.0 * (float)sin((double)dir1);
 xc = ROUND(xx);
 yc = ROUND(yy);
 xlim = (xsrce < xdead) ? (xdead + 1) : (xdead - 1);
 ylim = (ysrce < ydead) ? (ydead + 1) : (ydead - 1);
 x1d = xlim + xc;     y1d = ylim + yc;
 x2d = xlim - xc;     y2d = ylim - yc;
 x1s = xsrce + xc;     y1s = ysrce + yc;
 x2s = xsrce - xc;     y2s = ysrce - yc;
/****************** Set the dead end flags for the box formed ***************/
 egrid[xsrce][ysrce] = 1;
 egrid[xdead][ydead] = 1;
 for (x=0; x<SZE; x++) {                        /* unmark all cells         */
    for (y=0; y<SZE; y++) mark[x][y] = FALSE;
 )
 markLine(x1s,y1s,x2s,y2s);                     /* Mark the box sides       */
 markLine(x1d,y1d,x2d,y2d);
 markLine(x1s,y1s,x1d,y1d);
 markLine(x2s,y2s,x2d,y2d);
 mark[x1s][y1s] = TRUE;                         /* Mark the four corners    */
 mark[x2s][y2s] = TRUE;
 mark[x1d][y1d] = TRUE;
 mark[x2d][y2d] = TRUE;
 if (mark[xdead][ydead]) return;               /* Box has no interior      */
```

```
/******* Start the traverse of interior cells, and set dead end flags  *******/
 toppt = new(Cell);                              /* Store start point in list */
 botpt = toppt;
 toppt->x = xdead;
 toppt->y = ydead;
 toppt->npt = NULL;
 mark[xdead][ydead] = TRUE;
 while (toppt != NULL) {          /* While list not empty */
    curpt = toppt;                /*    pop top node      */
    toppt = toppt->npt;
    x = curpt->x;
    y = curpt->y;
    for (i = 0; i<=3; i++) {
        nborx = x + XNBOR[i];
        nbory = y + YNBOR[i];
       if ((nbory >= 0) && (nbory < SZE) &&  (nborx >= 0) && (nborx < SZE)) {
          egrid[nborx][nbory] = 1;
          if (mark[nborx][nbory] == FALSE) {
             mark[nborx][nbory] = TRUE;
             if (CgPt->getCg(nborx,nbory) <= n_the) {
                if (curpt != NULL) {           /* Reuse current node    */
                   newpt = curpt;
                   curpt = NULL;
                   if (toppt == NULL) toppt = newpt;
                } else newpt = new(Cell);      /* Allocate new node     */
                newpt->x = nborx;              /* Add to bottom of list */
                newpt->y = nbory;
                newpt->npt = NULL;
                if (botpt != newpt) botpt->npt = newpt;
                botpt = newpt;
             }
          }
       }
    }
    if (curpt != NULL) delete(curpt);  /* Delete current node if not reused */
 }
}  /** end markDeadEndBox() **/
```

```
/*****************************************************************************/
/* Procedure: markLine         mark a line in the mark array            Done */
/* Called by: markDeadEndBox, markDeadEndAll                                 */
/* Calls:     ROUND                                                          */
/* Globals:   mark (Write only)                                              */
/*    Finds all cells along the line determined by the two endpoints and sets*/
/*    the mark flag TRUE for them. The two endpoints are not set.            */
/*    Increments from one point to the other along either the X or Y axis.   */
/*    Determines the corresponding other coordinate from the slope, and      */
/*    marks the indicated cell.  -- Used to set a boundary for a breadth     */
/*    traverse of some portion of the grid. Be sure to clear the mark array. */
/*****************************************************************************/
void markLine(int x1, int y1, int x2, int y2) {
 float slope;        /* slope (or inverse slope) of line between points */
 int dx,dy;          /* difference between the two points               */
 float fx,fy;        /* accumulator for index determined from slope.    */
 int ix,iy;          /* indices of cell being marked                    */
 dx = x2 - x1;
 dy = y2 - y1;
 if ((abs(dx) >= abs(dy)) && (dx != 0)) {     /* increment X if dx larger  */
    slope = (float)(dy)/(float)(dx);
    fy = (float)y1;
    if (x1 < x2) {                            /* positive increment x1 to x2 */
       for (ix = x1 + 1; ix < x2; ix++) {
          fy = fy + slope;
          iy = ROUND(fy);
          if ((ix<SZE)&&(ix >= 0)&&(iy < SZE)&&(iy >= 0)) mark[ix][iy] = TRUE;
       }
    } else {                                  /* negative increment x1 to x2 */
       for (ix = x1 - 1; ix > x2; ix--) {
          fy = fy - slope;
          iy = ROUND(fy);
          if ((ix<SZE)&&(ix >= 0)&&(iy < SZE)&&(iy >= 0)) mark[ix][iy] = TRUE;
       }
    }
 } else if (abs(dy) > abs(dx)) {        /* increment Y, y change is greatest */
    slope = (float)(dx)/(float)(dy);
    fx = (float)x1;
    if (y1 < y2) {                            /* positive increment y1 to y2 */
       for (iy = y1 + 1; iy < y2; iy++) {
          fx = fx + slope;
          ix = ROUND(fx);
          if ((ix<SZE)&&(ix >= 0)&&(iy < SZE)&&(iy >= 0)) mark[ix][iy] = TRUE;
       }
    } else {                                  /* negative increment y1 to y2 */
       for (iy = y1 - 1; iy > y2; iy--) {
          fx = fx - slope;
          ix = ROUND(fx);
          if ((ix<SZE)&&(ix >= 0)&&(iy < SZE)&&(iy >= 0)) mark[ix][iy] = TRUE;
       }
    }
 }
}
```

```
/*****************************************************************************/
/* Procedure: mkbgrd        makes the badness grid                     Done */
/* Called by: getDest, omanInit, stopMe,(driver)test1, travelAuxBack        */
/* Calls:     <none>                                                        */
/* Globals:   cgrid, dgrid, tgrid (Read only) bgrid (Write only)            */
/*    Creates the entire badness grid by summing the corresponding elements */
/*    of the direction (dgrid), travel (tgrid), and certainty (cgrid) grids.*/
/*    In order to save time, should only be called when the robot is stopped.*/
/*    fzbgrd should be used to generate the badness values used for         */
/*    navigation since it includes fuzzing and only calculates values near  */
/*    the robot.                                                            */
/*****************************************************************************/
void mkbgrd(void) {
 int i,j;                                  /* loop indicies */
 for (i=0; i<SZE; i++) {
    for (j=0; j<SZE; j++) {
       bgrid[i][j] = CgPt->getCg(i,j) + dgrid[i][j];
       if (n_nav == N_LOCAL)               /* add travel value if local nav */
          bgrid[i][j] = bgrid[i][j] + tgrid[i][j];
    }
 } } /** end mkbgrd() **/
```

```
/*****************************************************************************/
/* Procedure: mkdgrd           makes the direction grid                      */
/* Called by: centerRobot, getDest, omanInit, travelAuxBack                  */
/* Calls:     distance - Determine the distance between two points           */
/* Globals:   dgrid,mark (Write only); cgrid (Read only)                     */
/*    The directional grid assigns a potential value to each location based  */
/*    on how far it is from the robots destination. The cells which are far  */
/*    from the destination have high potentials while cells near the         */
/*    destination have low potentials.  How far a cell is from the dest.     */
/*    is determined not just by the actual distance but how far Luey would   */
/*    have to travel to avoid known obstacles.  A two stage                  */
/*    breadth first traverse is used is mark each grid element with a        */
/*    a potential slightly higher than then next closer cell. Any path       */
/*    (certainty values below the path threshold) starting from the          */
/*    destination is traversed first, then the remaining cells starting from */
/*    those already done or the destination if no path was present. This can */
/*    result in discontinuities in the grid values for the remaining cells   */
/*    but the grid should be recalculated before the robot traverses those   */
/*    areas. Also note that the potential field is not exactly proportional  */
/*    to the distance from the destination.  When doing the traverse, only   */
/*    adjacent cells are considered neighbors. This gives cells on the       */
/*    diagonal a higher potential then if straight distances were used.      */
/*       When the destination is unmapped the entire edge of the grid        */
/*    closest to the destination is used as  the starting point for the      */
/*    traverse.                                                              */
/*****************************************************************************/
void mkdgrd(
   int xrob, int yrob,     /* robot's location               */
   int destx, int desty,   /* destination in grid coordinates */
   float pathvalue,        /* maximum value for connected path*/
   short redo)             /* Flag to force recalculation     */
{
 int i;                /* loop indicies */
 float norm;           /* Normalizing value so potential in good range */
 Cell *botpt;          /* pointer to bottom of path list      */
 Cell *botun;          /* pointer to bottom of unknown list   */
 Cell *curpt;          /* pointer to current node of list     */
 Cell *newpt;          /* pointer to node being added to list */
 Cell *toppt;          /* pointer to top of path list         */
 Cell *topun;          /* pointer to top of unkown list       */
 int nborx,nbory;      /* indices of neighbor cells           */
 int x,y;              /* loop indices,  current cell         */
 float sqreval;        /* additional potential, adjacent cell */
 static int olddx = -1,olddy = -1; /* destination last time  */
 static float oldpath = -1.0;      /* path value last time   */
 if ((olddx == destx)&&(olddy == desty)&&(pathvalue == oldpath)&&(!redo)) {
    return;
 }
 olddx = destx;
 olddy = desty;
 oldpath = pathvalue;
 DcPt->prtMsg("MKDGRD:",destx,desty); DcPt->prtInt(xrob,yrob);
 DcPt->prtFlt(pathvalue);
/******* If destination is off of grid, give it a location not to far away ***/
    if (destx >= SZE) olddx = 150;
    else if (destx < 0) olddx = -50;
    if (desty >= SZE) olddy = 150;
    else if (desty < 0) olddy = -50;
/****** Determine a suitable normalizing value and 1 cell potential change ***/
    norm = distance(olddx,olddy,xrob,yrob);
    norm = ((20.0 >= norm) ? 20.0 : norm);
    sqreval = 1.0/norm;
 for (x=0; x<SZE; x++) {                          /* unmark all cells */
```

```
    for (y=0; y<SZE; y++) mark[x][y] = FALSE;
 )
 toppt = new(Cell);              /* Dummy header node starts path cell list */
 topun = new(Cell);              /* Dummy header node starts unknown list */
/*************** Put the starting cell(s) in the list ********************/
 if ((destx < 0) || (destx >= SZE) || (desty < 0) || (desty >= SZE)) {
    botpt = mkDgridAux(olddx,olddy,toppt,topun);
    botun = botpt->npt;
    botpt->npt = NULL;
 } else {
    topun->npt = NULL;           /* Unknown list is empty               */
    botun = topun;
    botpt = new(Cell);           /* Store destination in path cell list */
    toppt->npt = botpt;
    botpt->x = destx;
    botpt->y = desty;
    botpt->npt = NULL;
    dgrid[destx][desty] = 0.0;
    mark[destx][desty] = TRUE;
 )
/*****************************************************************************/
/* Mark and determine potentials for all cells on the path from the          */
/* destination. Occupied or unknown cells adjacen to the path cells are      */
/* marked and their potential found then they are put in the unknown list.   */
/*****************************************************************************/
 while (toppt->npt != NULL) {     /* While list not empty */
    curpt = toppt->npt;           /* pop top node */
    toppt->npt = curpt->npt;
    x = curpt->x;
    y = curpt->y;
    for (i = 0; i <= 3; i++) {
       nborx = x + XNBOR[i];
       nbory = y + YNBOR[i];
       if ((nbory >= 0) && (nbory < SZE) && (nborx >= 0) && (nborx < SZE)) {
          if (mark[nborx][nbory] == FALSE) {
             mark[nborx][nbory] = TRUE;
             dgrid[nborx][nbory] = dgrid[x][y] + sqreval;
             if (curpt != NULL) { /* Reuse current node */
                newpt = curpt;
                curpt = NULL;
             } else newpt = new(Cell); /* Allocate new node */
             newpt->x = nborx;
             newpt->y = nbory;
             newpt->npt = NULL;
             if (CgPt->getCg(nborx,nbory) <= pathvalue) { /* to path list */
                if (toppt->npt == NULL) botpt = toppt;
                botpt->npt = newpt;
                botpt = newpt;
             } else {                              /* add to unkown list */
                botun->npt = newpt;
                botun = newpt;
             }
          }
       }
    }
    if (curpt != NULL) delete(curpt); /* Delete current node if not reused */
 }
/*****************************************************************************/
/* Mark and determine potentials for all remaining cells.  (unknown list)    */
/*****************************************************************************/
 while (topun->npt != NULL) {          /* While list not empty */
    curpt = topun->npt;                /*    pop top node      */
    topun->npt = curpt->npt;
    x = curpt->x;
    y = curpt->y;
    for (i = 0; i <= 3; i++) {
       nborx = x + XNBOR[i];
       nbory = y + YNBOR[i];
       if ((nbory >= 0) && (nbory < SZE) && (nborx >= 0) && (nborx < SZE)) {
          if (mark[nborx][nbory] == FALSE) {
             mark[nborx][nbory] = TRUE;
             dgrid[nborx][nbory] = dgrid[x][y] + sqreval;
             if (curpt != NULL) {           /* Reuse current node    */
                newpt = curpt;
                curpt = NULL;
             } else newpt = new(Cell);      /* Allocate new node     */
             newpt->x = nborx;
             newpt->y = nbory;
             newpt->npt = NULL;
             if (topun->npt == NULL) botun = topun;
             botun->npt = newpt;
             botun = newpt;
          }
       }
    }
    if (curpt != NULL) delete(curpt); /* Delete current node if not reused*/
 )
 delete(topun); delete(toppt);         /* Free header nodes */
} /** end mkdgrd() **/
```

```
/*****************************************************************************/
/* Procedure: mkDgridAux       treats grid edge as destination               */
/* Called by: mkdgrd                                                         */
/* Calls:     (cgrid)getCg     - get certainty value                         */
/* Globals:  mark, dgrid (Write only)                                        */
/*    Initializes the stack for mkdgrd when the destination is not on the    */
/*    grid. Treats all cells along the grid edges closest to the destination */
/*    as starting points for the traverse of the grid. Each edge cell is     */
/*    marked as done, its direction potential set to 0.0 and it is put on    */
/*    the bottom of one of the two stacks used to make the direction grid.   */
/*    If the certainty is below the path value then it is put on the path    */
/*    stack, if above the path value the unknown stack.                      */
/*****************************************************************************/
Cell *mkDgridAux(
   int x,int y,                    /* destination cell              */
   Cell *toppath, Cell *topunkn)   /* top of path and unknown lists */
{
 int   i,j;     /* loop indices            */
 Cell *newpt;   /* new node to add to list */
 Cell *botpt;   /* bottom of path list     */
 Cell *botun;   /* bottom of unknown list  */
 botpt = toppath;
 botun = topunkn;
 if (x >= SZE) {
    for (j=0; j<SZE; j++) {
       newpt = new(Cell);
       newpt->x = SZE-1;
       newpt->y = j;
       mark[SZE-1][j] = 1;
       dgrid[SZE-1][j] = 0.0;
       if (CgPt->getCg(SZE-1,j) <= n_path) {  /* put node on correct stack */
          botpt->npt = newpt;
          botpt = newpt;
       } else {
          botun->npt = newpt;
          botun = newpt;
       }
    }
 } else if (x < 0) {
    for (j=0; j<SZE; j++) {
       newpt = new(Cell);
       newpt->x = 0;
       newpt->y = j;
       mark[0][j] = 1;
       dgrid[0][j] = 0.0;
       if (CgPt->getCg(0,j) <= n_path) {  /* put node on correct stack */
          botpt->npt = newpt;
          botpt = newpt;
       } else {
          botun->npt = newpt;
          botun = newpt;
       }
    }
 }
 if (y >= SZE) {
    for (i=0; i<SZE; i++) {
       newpt = new(Cell);
       newpt->x = i;
       newpt->y = SZE-1;
       mark[i][SZE-1] = 1;
       dgrid[i][SZE-1] = 0.0;
       if (CgPt->getCg(i,SZE-1) <= n_path) {  /* put node on correct stack */
          botpt->npt = newpt;
          botpt = newpt;
       } else {
          botun->npt = newpt;
          botun = newpt;
       }
    }
 } else if (y < 0) {
    for (i=0; i<SZE; i++) {
       newpt = new(Cell);
       newpt->x = i;
       newpt->y = 0;
       mark[i][0] = 1;
       dgrid[i][0] = 0.0;
       if (CgPt->getCg(i,0) <= n_path) {  /* put node on correct stack */
          botpt->npt = newpt;
          botpt = newpt;
       } else {
          botun->npt = newpt;
          botun = newpt;
       }
    }
 }
 botun->npt = NULL;
 botpt->npt = botun;                    /* want to return both bottoms */
 return(botpt);
} /** end mkDgridAux() **/

/*****************************************************************************/
/* Procedure: mktgrd                Clear the local travel grid      Done */
/* Called by: getDest, omanInit                                              */
/* Calls:     <none>                                                         */
/* Globals:   tgrid (Write only)                                             */
/*    The travel grid contains a record of which cells have been traversed.  */
/*    It is initialized to all 0 (no cell has been traversed)                */
/*****************************************************************************/
void mktgrd(void)
{
 int i,j;                   /* loop indicies */
 for (i=0; i<SZE; i++) {
    for (j=0; j<SZE; j++) {
       tgrid[i][j] = 0.0;
    }
 }
}  /** end mktgrd() **/
```

```
/*****************************************************************************/
/* Procedure: moveMe          Adjusts robot speed and direction         Done */
/* Called by: travel                                                         */
/* Calls:     (RWITim)executeTurn - Turn the robot                           */
/*            (RWITim)setSlew     - Set robot's turn rate                    */
/*            atSDest      - Check if a cell is the subdestination           */
/*            changeSpeed - Change robot's speed                             */
/*            checkAhead  - Check for obstacles in robot's path              */
/*            checkSides  - Check for obstacles to sides of robot            */
/*            selectMove  - Determine bearing toward destination             */
/*            stopMe      - Stops robot movement, rotation & finds position  */
/*            turn        - Turn the robot and read sonars                   */
/*            update      - Update robot's current position                  */
/* Globals: nwcell, slew, spdfac, timer (Read,Write)                         */
/*          base_angl, offx, offy, robx, roby (Read only)                    */
/*    Determines and implements changes to the robot's direction and speed.  */
/*    selectMove is called to determine which direction the robot favors. If */
/*    the robot is starting from rest or the direction is over 30 degrees    */
/*    from its current orientation, then the robot will turn in place before */
/*    starting at a slow speed. If the robot is already moving, then its     */
/*    speed is changed based on the current speed and how far ahead is clear.*/
/*    If a direction change of less than 30 degrees is needed or a glancing  */
/*    collision is anticipated then a turn is started.                       */
/*****************************************************************************/
void moveMe(void)
{
 short change = FALSE; /* Flag for probable dir. change, cell ahead has > pot*/
 float deg;            /* Robot's estimated new direction                    */
 float dist;           /* Distance(mm) to closest object in direct. of travel*/
 float diff;           /* Angle (deg) between current and intended direction */
 int   esox, esoy;     /* Estimate of robot position in cell at change time  */
 int   estx=0, esty=0; /* Estimate of robot cell at speed, slew change time  */
 float oldspd;         /* Speed (cm/sec) before change                       */
 float chslew;         /* Direction adjustment to avoid glancing off of wall */
 int   newfac;         /* New speed factor                                   */
 float newslew;        /* New slew rate                                      */
 float newspd;         /* New speed                                          */
 static int cntturn=0; /* Counts consecutive turns without forward progress  */
/*****************************************************************************/
/* Estimate the robot's position at next change of direction or speed.       */
/* Using current speed and slew, where will robot be in 2 seconds.           */
/* Estimated direction is base_angl + 2*slew                                 */
/*****************************************************************************/
 oldspd = spdfac * SPEED_INCREMENT;
 deg  = base_angl + slew;
 deg  = degToRadian(deg);
 esox = offx + (int)(20.0 * oldspd * cos((double)deg));
 esoy = offy + (int)(20.0 * oldspd * sin((double)deg));
 if (esox >= 100)       { estx =  1; esox = esox - 200;  }
 else if (esox <= -100) { estx = -1; esox = esox + 200;  }
 if (esoy >= 100)       { esty =  1; esoy = esoy - 200;  }
 else if (esoy <= -100) { esty = -1; esoy = esoy + 200;  }
/*****************************************************************************/
/* Select new direction and calculate slew rate for it.                      */
/*****************************************************************************/
 diff = selectMove(estx+robx,esty+roby,esox,esoy) - base_angl - slew - slew;
 if (diff >  180.0) diff = diff - 360.0;        /* Put between -180 & 180    */
 else if (diff < -180.0) diff = diff + 360.0;
 if  (abs(diff) < 1.0) diff = 0.0;
 if ((abs(diff) <= 30.0) && (spdfac != 0)) {/* If moving and small angle     */
    newslew = (diff * oldspd)/(float)CELLCM;    /* New turn rate             */
    deg = base_angl + slew + slew + newslew;    /* Estimated new direction   */
/*****************************************************************************/
/* Determine new speed. Check estimated path ahead, speed up if clear, slow  */
/* down or stay the same if obstacle ahead. If a direction change is antici- */
/* pated, slow to minimum speed.                                             */
/*****************************************************************************/
    dist = checkAhead(deg);                     /* Check path ahead          */
    if (dist < 0.0) change = TRUE;
    dist = (float)fabs((double)dist);
    newfac = spdfac;                            /* adjust speed              */
    if (dist >= CHECKDIS) newfac++;             /* if path clear, speed up   */
    else {
       newfac = (int)(dist)/CELL;                       /* object ahead, slow */
       newfac = ((newfac <= spdfac) ? newfac : spdfac);/* don't speed up also*/
    }
    if (change == TRUE) newfac = ((newfac <= 1) ? newfac : 1); /* slow       */
    else if (newfac > MAXFAC) newfac = MAXFAC;  /* Limit maximum speed       */
    newspd = newfac * SPEED_INCREMENT;
/*****************************************************************************/
/* Adjust slew rate. Slew rate is set so robot will turn to direction indi-  */
/* cated by selectMove in time robot takes to go 1 cell. If the side check   */
/* shows a wall is near & getting closer, (probable collision) the slew rate */
/* is set to turn robot parallel to wall in time to travel 1 cell. Maximum   */
/* slew rate is +-10.                                                        */
/*****************************************************************************/
    chslew = checkSides(deg,newspd);            /* Check for side obstacles */
    if (chslew == 50.0) {                       /* If single reading of wall*/
       nwcell = update(0.0,-1.0) || nwcell;     /*    check again           */
       chslew = checkSides(deg,newspd);
    }
    if (chslew > 0) newslew = ((newslew >= chslew) ? newslew : chslew);
    else if (chslew < 0) newslew = ((newslew <= chslew) ? newslew : chslew);
    if (newslew > 10.0) newslew = 10.0;
    else if (newslew < -10.0) newslew = -10.0;
    newslew = (float)ROUND(newslew);            /* Round slew rate to int.  */
    timer = timer + 1.12;                       /* Simulated time for moveMe */
    if ((newslew != slew) || (newspd != oldspd)) {
       nwcell = update(0.0,-1.0) || nwcell;     /* Get robot position       */
    }
/*****************************************************************************/
/* If the slew rate is changed the robot will not speed up. It may slow down.*/
/*****************************************************************************/
    if (atSDest(robx,roby) == TRUE) stopMe();
    else {
       if (newslew != slew) {               /* set slew before speed         */
          slew = newslew;
          RwitPt->setSlew(-slew);           /* robot directions are reversed */
          timer = timer + SET_TIME;         /* Simulated time for set command*/
    DcPt->prtLne("*MOVEME: set slew to "); DcPt->prtFlt(slew);
          if (newspd < oldspd) {            /* Decrease speed                */
             changeSpeed(oldspd,newspd);
             spdfac = newfac;
          }
       }
       else if (newspd != oldspd) {         /* Increase speed                */
          changeSpeed(oldspd,newspd);
          spdfac = newfac;
       }
    }
/*****************************************************************************/
/* If a large turn is needed or the robot is at rest.  Robot is stopped and  */
/* the turn recalculated then executed.                                      */
/*****************************************************************************/
 } else {                                   /* From rest or large angle      */
    if ((abs(diff)>1.0) && (cntturn < 5)){  /*    If large angle             */
      if ((spdfac != 0) || (slew != 0.0)) { /*         halt & find location  */
```

```
        stopMe();
        diff = selectMove(robx,roby,offx,offy) - base_angl;
        if (diff >  180.0) diff = diff - 360.0;
        else if (diff < -180.0) diff = diff + 360.0;
      )
      if (cntturn == 1) {                    /* no sonars if turning back     */
         RwitPt->executeTurn(-diff,10.0,TRUE);
        DcPt->prtLne("*MOVEME: TURN change is  "); DcPt->prtFlt(diff);
      )
      else turn(diff,TRUE);         /*       Rotate robot with sonar scan     */
      cntturn++;
    }
    else {                          /* When starting from rest use slow speed  */
       if (cntturn == 5)
          DcPt->prtLne("*MOVEME: excessive turns, moving forward");
       spdfac = 1;
       chslew = checkSides(base_angl,0.0);
                                    /* Get base line data for side checks*/
       changeSpeed(0.0,(float)(spdfac * SPEED_INCREMENT));
       cntturn = 0;
    }
 )
) /** end moveMe() **/
```

```
/*****************************************************************************/
/* Procedure: nearTo         Checks if two cells are within 4          Done */
/* Called by: travelAuxBack                                                  */
/* Returns:   short - TRUE if points are within 4 cells FALSE otherwise      */
/*****************************************************************************/
short nearTo(
    int x1,int y1,int x2,int y2)    /* Two points */
{
 if ((abs(x1-x2) <= 4) && (abs(y1-y2) <= 4)) return(TRUE);
 return(FALSE);
) /** end nearTo() **/

/*****************************************************************************/
/* Procedure: neighbor           Checks if two cells are neighbors     Done */
/* Called by: collision                                                      */
/* Calls:     <none>                                                         */
/* Returns:    short - TRUE if cells adjacent, FALSE otherwise               */
/*    Checks if two cells are adjacent.                                      */
/*****************************************************************************/
short neighbor(
   int x1, int y1, int x2, int y2)    /* Coords. of two points              */
{
 int i;
 for (i=0; i<= 7; i++ ) {
    if (((x1 + XNBOR[i]) == x2) && ((y1 + YNBOR[i]) == y2))
       return(TRUE);
 )
 return(FALSE);
)  /** end neighbor() **/

/*****************************************************************************/
/* Procedure: newCell        Decays the certainty grid, checks edges   Done  */
/* Called by: collision, travel, moveMe, stopMe                              */
/* Calls:     centerRobot  - shifts grid indices to recenter robot           */
/*            CgPt->decay  - decays the certainty grid                       */
/* Globals:   base_dis, robx, roby, (Read); timer (Read,Write); nwcell(Write)*/
/*    Does what is needed when a new grid cell is entered. This includes     */
/*    decaying the certainty grid and checking to see if the robot must be   */
/*    recentered. Because of the time consuming nature of these procedures   */
/*    newCell should be called sparingly. It is not called immediately after */
/*    a new cell is entered, but is instead called the next time the robot   */
/*    halts.                                                                 */
/*****************************************************************************/
void newCell(void)
{
 if ((robx < NEAR_EDGE1) || (robx > NEAR_EDGE2) ||
     (roby < NEAR_EDGE1) || (roby > NEAR_EDGE2)) centerRobot();
 if (CgPt->decay(n_decay,base_dis) == ERR)
    DcPt->prtLne("*NEWCELL: error in decay, excessive distance");
 if (n_decay) timer = timer + 0.45; /* Simulated time for decay (OK 10/31)   */
 nwcell = FALSE;              /* Turn off new cell flag                       */
)  /** end newCell() **/
```

```
/*****************************************************************************/
/* Procedure: offCenter                                                 Done */
/* Called by: checkAux                                                       */
/* Calls:     <none>                                                         */
/* RETURNS:   float - length of compenent                                    */
/*    Finds the length of the component of the vector (x,y) in the direction */
/*    dir. Units are the same as those for X & Y.                            */
/*****************************************************************************/
float offCenter(
   int   x, int y,            /* x,y coordinates of vector                 */
   float dir)                 /* Direction of component                    */
{
 float angle1;                /* Angle (radians) of vector (x,y)           */
 float angle2;                /* Angle (radians) between (x,y) and dir     */
 float dist;                  /* Length of (x,y)                           */
 float offc;                  /* Length of component in direction dir      */
 float fx,fy;                 /* Float versions of x & y                   */
 fx = (float)x;
 fy = (float)y;
 if (x == 0) angle1 =  1.570796; /* Pi/2 */
 else angle1 = atan(fy/fx);
 angle2 = degToRadian(dir) - angle1;
 dist = sqrt( (fx*fx) + (fy*fy));
 if (x >= 0) offc = dist * (float)cos((double)angle2);
 else offc = -dist * (float)cos((double)angle2);
 return(offc);
} /** end offCenter() **/
/*****************************************************************************/
/* Procedure: omanInit       Initializes the navigator                  Done */
/* Called by: (driver)main, (driver)test1                                    */
/* Calls:     (destack)clearDS      - Clears the checkpoint stack            */
/*            (RWITIm)queryAngle    - Get robot's value for angle            */
/*            (RWITim)queryDistance - Get robot's value for distance         */
/*            mkbgrd           - Make the badness grid                       */
/*            mkdgrd           - Make directional grid                       */
/*            mktgrd           - Initialize travel grid                      */
/* Globals:   base_angl, base_dis, bgrid, dgrid, tgrid, movx, movy, off_angle*/
/*            off_dist, offx, offy, xdest, ydest, xsubd, ysubd (Write only)  */
/*            robx, roby (Read only)                                         */
/*    Initializes the robot's position, the travel, destination and badness  */
/*    grids. Also zeros the movement accumulators and the robot's off center */
/*    of cell distance.  If the destination flag is set then the destination */
/*    stack is cleared and the destination set to the robots current position*/
/*****************************************************************************/
void omanInit(int xcen, int ycen, short destFlag)
{
 if (destFlag) {
    DsPt->clearDS();
    xdest = xcen; ydest = ycen; /* Set destination as robot's position       */
    xsubd = xcen; ysubd = ycen; /* Set subdestination as same                */
 }
 begx = xcen; begy = ycen; /* Set robot's start location for position finding*/
 movx = movy = 0.0;        /* Initialize movement accumulators              */
 offx = offy = 0;          /* Initialize offset from cell center            */
 mktgrd();                 /* Initialize travel grid                        */
 mkdgrd(robx,roby,xdest,ydest,n_path,TRUE); /* Initialize direction grid    */
 mkbgrd();                 /* Initialize badness grid                       */
 base_angl =  0.0;
 base_dis  =  0.0;
 off_angle = -1.0 * RwitPt->queryAngle();          /* determine offsets for */
 off_dist  = -1.0 * RwitPt->queryDistance();       /*  robot queries        */
} /* end omanInit() */
```

```
/*****************************************************************************/
/* Procedure: pathToDest     Checks for a possible path to destination       */
/* Called by: quitOk, travelAuxBack                                          */
/* Calls:     <none>                                                         */
/*    Does a breadth first traverse through all non occupied cells (or path  */
/*    cells when using the incremental navigator) starting                   */
/*    from the robot position and stopping when the destination is reached.  */
/*    Returns TRUE if it was or FALSE if it was not. If the destination is   */
/*    off of the grid then TRUE is returned if a path exists to the edge of  */
/*    the grid closest to the destination.                                   */
/*****************************************************************************/
short pathToDest(
    int sx, int sy,      /* Robot position */
    int dx, int dy)      /* Destination    */
{
 Cell *botpt;            /* pointer to bottom of list           */
 Cell *curpt;            /* pointer to current node of list     */
 Cell *newpt;            /* pointer to node being added to list */
 Cell *toppt;            /* pointer to top of list              */
 float path;             /* path value, depends on navigator    */
 short done = FALSE;     /* flag for destination reached        */
 short offgrid = FALSE;  /* flag for destination off of grid    */
 int   nborx,nbory;      /* indices of neighbor cells           */
 int   x,y,i;            /* loop indices,  current cell         */
 if ((dx < 0) || (dx >= SZE) || (dy < 0) || (dy >= SZE)) offgrid = TRUE;
 for (x=0; x<SZE; x++) {                          /* unmark all cells  */
    for (y=0; y<SZE; y++) mark[x][y] = FALSE;
 }
 if (n_nav == N_INCREMENT) path = n_path;
 else path = n_tho;
 toppt = new(Cell);                          /* Store position in list  */
 botpt = toppt;
 toppt->x = sx;
 toppt->y = sy;
 toppt->npt = NULL;
 mark[sx][sy] = TRUE;
 while ((toppt != NULL) && (!done)) {      /* While list not empty */
    curpt = toppt;                         /*    pop top node      */
    toppt = toppt->npt;
    x = curpt->x;
    y = curpt->y;
    if (offgrid) {
       if ((dx >= SZE) && (x == (SZE-1))) done = TRUE;
       if ((dx < 0)    && (x == 0))       done = TRUE;
       if ((dy >= SZE) && (y == (SZE-1))) done = TRUE;
       if ((dy < 0)    && (y == 0))       done = TRUE;
    } else if ((x == dx)   && (y == dy))      done = TRUE;
    for (i = 0; i<= 3; i++) {            /* for the 4 adjacent neighbors */
       nborx = x + XNBOR[i];
       nbory = y + YNBOR[i];
       if ((nbory >= 0) && (nbory < SZE) && (nborx >= 0) && (nborx < SZE)) {
       if ((CgPt->getCg(nborx,nbory) < path) &&(mark[nborx][nbory] == FALSE)) {
          mark[nborx][nbory] = TRUE;
          if (curpt != NULL) {           /* Reuse current node    */
             newpt = curpt;
             curpt = NULL;
             if (toppt == NULL) toppt = newpt;
          } else newpt = new(Cell);      /* Allocate new node     */
          newpt->x = nborx;              /* Add to bottom of list */
          newpt->y = nbory;
          newpt->npt = NULL;
          if (botpt != newpt) botpt->npt = newpt;
          botpt = newpt;
```

```
      ))
    )
    if (curpt != NULL) delete(curpt);  /* Delete current node if not reused */
 )
 while (toppt != NULL) (          /* Free the remaining list */
    curpt = toppt;
    toppt = toppt->npt;
    delete(curpt);
 )
 if (done) return(TRUE);
 else (
    DcPt->prtMsg("PATHTODEST: No path to ",dx,dy);
    return(FALSE);
 )
) /** end pathToDest() **/

/*****************************************************************************/
/* Procedure: quitOk                                                   Done */
/* Called by: travel                                                         */
/* Calls:     (display)prtMsg - Output message to terminal                   */
/*            pathToDest      - Check for existance of possible path         */
/*            stopMe          - halt the robot                               */
/* Returns:   short - TRUE to quit local navigator, 2 to quit entire         */
/*                    navigator, FALSE - continue                            */
/*    Determines if the robot navigator is too confused to continue. If the  */
/*    current cell has been traversed over 3 times on the local travel grid, */
/*    then no progress is being made and the navigator gives up on the       */
/*    current subdestination. The current location is made the subdestination*/
/*    and the destination stack changed accordingly.                         */
/*         If the destination stack has been reduced to just a quit node     */
/*   then no path to the destination exists (at least we can't find it). We  */
/*   give up entirely.                                                       */
/*****************************************************************************/
short quitOk()
{
 if (tgrid[robx][roby] > (2.0 * TRAVEL_VALUE)) (
    stopMe();
    VoicePt->say("confused");
    DcPt->prtMsg("*QUITOK: Navigator confused. Stopping to think",robx,roby);
    ysubd = roby; xsubd = robx;
    DsPt->changeDS(robx,roby);
    if (n_nav == N_LOCAL) return(2);
    if (pathToDest(robx,roby,xdest,ydest)) return(TRUE);
    else return(2);
 ) else if (DsPt->getDt() == D_QUIT) (
    return(2);
 ) else if ((CgPt->getCg(xdest,ydest) >= n_tho) &&
            neighbor(robx,roby,xdest,ydest)) (
    DcPt->prtMsg("*QUITOK: Destination is occupied",xdest,ydest);
    return(2);
 ) else return(FALSE);
)
```

```
/*****************************************************************************/
/* Procedure: selectMove                                               Done */
/* Called by: moveMe                                                         */
/* Calls:     (cgrid)getSonarType - check if sonars are on                   */
/*            atSDest - Check if a cell is the subdestination                */
/*            stopMe - Halts the robot if destination is occupied            */
/* Globals:   bgrid, cgrid, elimCur, robx, roby, slew (Read only)            */
/*            elimNex (Write only)                                           */
/* Returns:   float - Absolute angle of open direction toward destination    */
/*    Checks the N1 neighbors of the current cell (where the robot expects   */
/*    to  be). Selects the one with the lowest potential in the badness grid.*/
/*    Using this cell selects its N1 neighbor which is in N2 of the original */
/*    cell with the lowest badness. Determines the absolute direction to     */
/*    the center of that cell and returns the angle. The robot's position in */
/*    its cell is taken into account and adjustments made to avoid occupied  */
/*    cells along the direction selected. The subdestination will preempt    */
/*    any other selected cell.  Occupied cells are never considered.  When   */
/*    the N_LOCAL navigator is used, cells which have previously been        */
/*    bypasses are also not considered.                                      */
/*    If no N2 cell is available then the N1 cell will be                    */
/*    used, if no N1 is available  then a direction 180 degrees from the     */
/*    current one is returned.  This will force the robot to stop and look   */
/*    around.                                                                */
/*     ** This routine is the heart of the potential based local navigator.  */
/*****************************************************************************/
float selectMove(
 int estx, int esty,   /* Estimate of robot's cell at next speed/slew change */
 int esox, int esoy)   /* Estimate of robot's position in cell               */
(
 float deg;            /* Angle (degrees) to neighbor with lowest potential  */
 int   i1,j1;          /* X,Y cell offset to N1 neighbor with least potential*/
 int   i,j;            /* Loop indices, X,Y cell offset to lowest N2 neighbor*/
 int   m1i,m1j;        /* Coordinates of neighbor cell with lowest potential */
 int   mini,minj;      /* Coords of n2 neighbor with least potential         */
 float minpot;         /* Potential of neighboring cell with lowest potential*/
 float tanang;         /* Tangent of angle to neighboring cell with less pot.*/
/*****************************************************************************/
/* If already at or heading for destination, make no changes                 */
/*****************************************************************************/
 if ((atSDest(robx,roby) == TRUE) || (atSDest(estx,esty))) { /* At destinatn */
    deg = (float)base_angl + (2.0 * slew);
    return(deg);
 )
 if   ((n_nav == N_LOCAL) &&  ((estx != elmx) || (esty != elmy)))
    elimStore(estx,esty);
/*****************************************************************************/
/* Determine empty N1 neighbor with lowest potential badness.                */
/*****************************************************************************/
 minpot = 10.0;
 mini = estx; minj = esty;
 for (i = estx-1; i <= estx+1; i++) (
    for (j = esty-1; j <= esty+1; j++) (
       if (atSDest(i,j) == TRUE) (           /* If cell is subdestination    */
          mini = i; minj = j;                /*    Select it                 */
          minpot = -1.0;
       )
       if (bgrid[i][j] < minpot) (           /* Find cell with min potential */
          if (((i != estx) || (j != esty)) && ((i != robx) || (j != roby)) &&
               (CgPt->getCg(i,j) < n_tho)) (
             mini = i; minj = j;
             minpot = bgrid[i][j];
          )
       )
```

```
    )
 )
 i1 = mini - estx;
 j1 = minj - esty;
 if (minpot == 10.0) {                          /* if no cell selected */
    DcPt->prtMsg("*SELECTMOVE:  Blocked at",mini,minj);
    stopMe();
    tgrid[mini][minj] = tgrid[mini][minj] + TRAVEL_VALUE;
    return(base_angl + 180.0);
 }
 if ( !atSDest(mini,minj) ) {
/*****************************************************************************/
/* Determine empty N2 neighbor with lowest badness potential. Only cells     */
/* which have not been previously eliminated and are in N1 of the cell just  */
/* found are considered.                                                     */
/*****************************************************************************/
    m1i = mini;
    m1j = minj;
    minpot = 10.0;
    if ((m1i - estx) != 0) {
       i = m1i + (m1i - estx);
       for (j = m1j-1; j <= m1j+1; j++) {
          if (atSDest(i,j) == TRUE) {       /* If cell is destination       */
             mini = i; minj = j;            /*    Select it                 */
             minpot = -1.0;
          }
          if ((bgrid[i][j] < minpot)&&(elimCur[i-estx+2][j-esty+2] == FALSE)) {
            if (CgPt->getCg(i,j) < n_tho) {
                mini = i;
                minj = j;
                minpot = bgrid[i][j];
            }
          }
       }
    }
    if ((m1j - esty) != 0) {
       j =  m1j + (m1j - esty);
       for (i = m1i-1; i <= m1i+1; i++) {
          if (atSDest(i,j) == TRUE) {       /* If cell is destination       */
             mini = i; minj = j;            /*    Select it                 */
             minpot = -1.0;
          }
          if ((bgrid[i][j] < minpot)&&(elimCur[i-estx+2][j-esty+2] == FALSE)) {
            if (CgPt->getCg(i,j) < n_tho) {
                mini = i; minj = j;
                minpot = bgrid[i][j];
            }
          }
       }
    }
 }
 i = mini - estx;
 j = minj - esty;
 elimNex[i+2][j+2] = FALSE;             /* Selected cell is not eliminated */
/*****************************************************************************/
/* Determine angle to local destination selected and make adjustments for N1 */
/* cells in the way.                                                         */
/*****************************************************************************/
 if ((i > 0) && (j >= 0)) {             /* First quadrant */
    tanang = (float)((j * CELL) - esoy) / (float)((i * CELL) - esox);
    deg = atan(tanang);
    deg = radToDegree(deg);
    if ((j == 2) && (i == 2)) {
       if (CgPt->getCg(estx+1,esty) > n_tho) deg = deg + 5.0;
       if (CgPt->getCg(estx,esty+1) > n_tho) deg = deg + 5.0;
    } else if ((j == 1) && (i == 2)) {
       if (CgPt->getCg(estx+1,esty+1) > n_tho) deg = deg - 5.0;
       if (CgPt->getCg(estx+1,esty)   > n_tho) deg = deg + 5.0;
    } else if ((j == 2) && (i == 1)) {
       if (CgPt->getCg(estx+1,esty+1) > n_tho) deg = deg + 5.0;
       if (CgPt->getCg(estx,esty+1)   > n_tho) deg = deg - 5.0;
    } else if ((j == 0) && (i == 2)) {
       if (CgPt->getCg(estx+1,esty+1) > n_tho) deg = deg - 5.0;
       if (CgPt->getCg(estx+1,esty-1) > n_tho) deg = deg + 5.0;
       if (CgPt->getCg(estx+1,esty)   > n_tho) deg = deg + (j1 * 50.0);
    }
 } else if ((j >= 0) && (i < 0)) {      /* 2nd quadrant */
    tanang = (float)((j * CELL) - esoy) / (float)((-i * CELL) + esox);
    deg = atan(tanang);
    deg = 180.0 - radToDegree(deg);
    if ((i == -2) && (j == 2)) {
       if (CgPt->getCg(estx-1,esty) > n_tho) deg = deg - 5.0;
       if (CgPt->getCg(estx,esty+1) > n_tho) deg = deg + 5.0;
    } else if ((j == 1) && (i == -2)) {
       if (CgPt->getCg(estx-1,esty+1) > n_tho) deg = deg + 5.0;
       if (CgPt->getCg(estx-1,esty)   > n_tho) deg = deg - 5.0;
    } else if ((j == 2) && (i == 1)) {
       if (CgPt->getCg(estx-1,esty+1) > n_tho) deg = deg - 5.0;
       if (CgPt->getCg(estx,esty+1)   > n_tho) deg = deg + 5.0;
    } else if ((j == 0) && (i == -2)) {
       if (CgPt->getCg(estx-1,esty+1) > n_tho) deg = deg + 5.0;
       if (CgPt->getCg(estx-1,esty-1) > n_tho) deg = deg - 5.0;
       if (CgPt->getCg(estx-1,esty)   > n_tho) deg = deg - (j1 * 50.0);
    }
 } else if ((i < 0) && (j <= 0)) {      /* 3rd quadrant */
    tanang = (float)((-j * CELL) + esoy) / (float)((-i * CELL) + esox);
    deg = atan(tanang);
    deg = -180.0 + radToDegree(deg);
    if ((j == -2) && (i == -2)) {
       if (CgPt->getCg(estx-1,esty) > n_tho) deg = deg + 5.0;
       if (CgPt->getCg(estx,esty-1) > n_tho) deg = deg - 5.0;
    } else if ((j == -1) && (i == -2)) {
       if (CgPt->getCg(estx-1,esty-1) > n_tho) deg = deg - 5.0;
       if (CgPt->getCg(estx-1,esty)   > n_tho) deg = deg + 5.0;
    } else if ((j == -2) && (i == -1)) {
       if (CgPt->getCg(estx-1,esty-1) > n_tho) deg = deg + 5.0;
       if (CgPt->getCg(estx,esty-1)   > n_tho) deg = deg - 5.0;
    }
 } else if ((i > 0) && (j <= 0)) {      /* 4th quadrant */
    tanang = (float)((-j * CELL) + esoy) / (float)((i * CELL) - esox);
    deg = atan(tanang);
    deg = -radToDegree(deg);
    if ((j == -2) && (i == 2)) {
       if (CgPt->getCg(estx+1,esty) > n_tho) deg = deg - 5.0;
       if (CgPt->getCg(estx,esty-1) > n_tho) deg = deg + 5.0;
    } else if ((j == -1) && (i == 2)) {
       if (CgPt->getCg(estx+1,esty-1) > n_tho) deg = deg + 5.0;
       if (CgPt->getCg(estx+1,esty)   > n_tho) deg = deg - 5.0;
    } else if ((j == -2) && (i == 1)) {
       if (CgPt->getCg(estx+1,esty-1) > n_tho) deg = deg - 5.0;
       if (CgPt->getCg(estx,esty-1)   > n_tho) deg = deg + 5.0;
    }
 } else if (i == 0) {
    if (esox == 0) {
       if (j > 0) deg = 90.0;
       else deg = -90.0;
    } else {
       tanang = (float)((j * CELL) + esoy) / (float)esox;
```

```
      deg = atan(tanang);
      if ((esox > 0) && (j > 0)) deg = 180.0 - radToDegree(deg);
      else if ((esox > 0) && (j < 0)) deg = -180.0 - radToDegree(deg);
      else deg = -1.0 * radToDegree(deg);
   )
   if (j == -2) {
      if (CgPt->getCg(estx+1,esty-1) > n_tho) deg = deg - 5.0;
      if (CgPt->getCg(estx-1,esty-1) > n_tho) deg = deg + 5.0;
      if (CgPt->getCg(estx,esty-1)   > n_tho) deg = deg + (i1 * 50.0);
   ) else if (j == 2) {
      if (CgPt->getCg(estx+1,esty+1) > n_tho) deg = deg + 5.0;
      if (CgPt->getCg(estx-1,esty+1) > n_tho) deg = deg - 5.0;
      if (CgPt->getCg(estx,esty+1)   > n_tho) deg = deg - (i1 * 50.0);
   )
)
DcPt->prtLne("SELECTMOVE: deg, base, estbase ");
DcPt->prtFlt(deg,base_angl); DcPt->prtFlt(base_angl+slew+slew);/*debug*/
return(deg);
)  /** end selectMove() **/
```

```
/*****************************************************************************/
/* Procedure: shiftTravel                                              Done */
/* Called by: centerRobot                                                    */
/* Calls:     <none>                                                         */
/* Globals:  tgrid, egrid (Read Write)                                       */
/*   Shifts the travel and dead end grids to compensate for the robot        */
/*   movement passed as arguments. The robot is restored to the center of    */
/*   the grid.  Values which are shifted off the grid are lost while new     */
/*   new cells are given a zero value.                                       */
/*****************************************************************************/
void shiftTravel(
   int dx, int dy)       /* X & Y movement of robot in grid cells            */
{
 int x,y;
 if (dx >= 0) {
    if (dy >= 0) {                    /* Robot has moved forward and up     */
       for (x=0; x<SZE; x++) {
          for (y=0; y<SZE; y++) {
             if ((x+dx >= SZE) || (y+dy >= SZE)) {
                tgrid[x][y] = 0.0;  egrid[x][y] = 0;
             ) else {
                tgrid[x][y] = tgrid[x+dx][y+dy];
                egrid[x][y] = egrid[x+dx][y+dy];
       )  )  )
    ) else {
       for (x=0; x<SZE; x++) {
          for (y=SZE-1; y>=0; y--) {   /* Robot has moved forward and down   */
             if ((x+dx >= SZE) || (y+dy < 0)) {
                tgrid[x][y] = 0.0;  egrid[x][y] = 0;
             ) else {
                tgrid[x][y] = tgrid[x+dx][y+dy];
                egrid[x][y] = egrid[x+dx][y+dy];
    )  )  )  )
 ) else {
    if (dy >= 0) {                    /* Robot has moved backward and up    */
       for (x=SZE-1; x>=0; x--) {
          for (y=0; y<SZE; y++) {
             if ((x+dx < 0) || (y+dy >= SZE)) {
                tgrid[x][y] = 0.0;  egrid[x][y] = 0;
             ) else {
                tgrid[x][y] = tgrid[x+dx][y+dy];
                egrid[x][y] = egrid[x+dx][y+dy];
             )
          )
       )
    ) else {
       for (x=SZE-1; x>=0; x--) {      /* Robot has moved backward and down */
          for (y=SZE-1; y>=0; y--) {
             if ((x+dx < 0) || (y+dy < 0)) {
                tgrid[x][y] = 0.0;  egrid[x][y] = 0;
             ) else {
                tgrid[x][y] = tgrid[x+dx][y+dy];
                egrid[x][y] = egrid[x+dx][y+dy];
             )
          )
       )
    )
 )
)  /** end shiftTravel() **/
```

```
/*****************************************************************************/
/* Procedure: stopMe     stops robot motion, and gets position          Done */
/* Called by: centerRobot, travel, selectMove                                */
/* Calls:     (display)drawCgrid  - Redraw the display                       */
/*            (display)prtData    - Displays current robot information       */
/*            (RWITim)halt        - Stops robot movement                     */
/*            (RWITim)quryVoltage - Gets the robot's current voltage         */
/*            fzbgrd              - Fuzz the badness grid                    */
/*            mkbgrd              - Make a new badness grid                  */
/*            newCell             - Perform tasks needed when new cell entere*/
/*            update              - find new  distance, angle and position   */
/* Globals:   spdfac, slew, volts (Write only); nwcell (Read,Write)          */
/*    Stops the robot movement and rotation and sets the speed and slew to 0.*/
/*    Also queries the voltage and recalculates the badness grid while the   */
/*    robot is halted. If the incremental navigator is being used then the   */
/*    direction grid is also recalculated.                                   */
/*****************************************************************************/
void stopMe(void)
{
 if ((slew != 0.0) || (spdfac != 0)) {
    DcPt->prtMsg("STOPME: ****** ",robx,roby);
    RwitPt->halt();
    nwcell = update(0.0,-1.0) || nwcell;
    spdfac = 0;
    slew = 0.0;
    if (nwcell) newCell();
    if (n_nav == N_INCREMENT) {   /* Redo the direction grid while stopped */
       mkdgrd(robx,roby,xdest,ydest,n_path,TRUE);
    )
    mkbgrd();                          /* Do some saved time consuming stuff   */
    fzbgrd();
    DxPt->drawCgrid(1);
 ) else DcPt->prtMsg("STOPME: ***** Not moving ",robx,roby);
 volts = RwitPt->queryVoltage();
 DcPt->prtData();
)  /** end stopMe() **/
```

```
/*****************************************************************************/
/* Procedure: subDest                                                   Done */
/* Called by: travelAuxBack, travelAuxPath                                   */
/* Calls:     (destack)queryStart  - Checks if current node is the start node*/
/*            (destack)getDx,getDy - Read current location from dest. stack  */
/*            (cgrid)  getCg       - Read certainty grid value               */
/*            mkdgrd               - Makes a new direction grid              */
/*            substart             - Find new start node instead of subdest  */
/* Returns:   short - TRUE if a node was added to the stack, FALSE if not    */
/* Globals:   bgrid, cgrid, dgrid (Read only);  mark (Write only)            */
/*    Does a breadth first traverse of all empty cells reachable from the    */
/*    the robot. Saves the one with the lowest dgrid which has not been      */
/*    marked as a dead end, and pushes it onto the destination stack.        */
/*    The cell found must have a lower direction value than the current cell */
/*    or a new subdestination is not pushed onto the stack.                  */
/*****************************************************************************/
short subDest(
    float pathvalue)     /* Maximum certainty allowed for path from dest.    */
{
 Cell *botpt;       /* pointer to bottom of list                 */
 Cell *curpt;       /* pointer to current node of list           */
 Cell *newpt;       /* pointer to node being added to list       */
 Cell *toppt;       /* pointer to top of list                    */
 int   xrob,yrob;   /* current robot cell                        */
 int   nborx,nbory; /* indices of cells next to current cell     */
 int   x,y,i;       /* loop indices, indices of current cell     */
 float mindis;      /* lowest empty dgrid, less than start cell  */
 int   minx,miny;   /* Coordinates of above cell                 */
 xrob = DsPt->getDx(); yrob = DsPt->getDy();
 mkdgrd(xrob,yrob,xdest,ydest,pathvalue,FALSE);  /* must precede unmarking   */
 for (x=0; x<SZE; x++) {                         /* unmark all cells         */
    for (y=0; y<SZE; y++) mark[x][y] = FALSE;
 )
 egrid[xdest][ydest] = 0;                    /* Ultimate dest. is not dead end*/
 minx = xrob; miny = yrob;                   /* Set default subdestination    */
 mindis = dgrid[minx][miny];
 toppt = new(Cell);                          /* Store robot position in list  */
 botpt = toppt;
 toppt->x = xrob;
 toppt->y = yrob;
 toppt->npt = NULL;
 mark[xrob][yrob] = TRUE;
 DcPt->prtMsg("SUBDEST",xrob,yrob);
 while (toppt != NULL) {                     /* While list not empty */
    curpt = toppt;                           /*    pop top node      */
    toppt = toppt->npt;
    x = curpt->x;
    y = curpt->y;
    if ((dgrid[x][y] < mindis) && (egrid[x][y] == 0)) {
       minx = x;
       miny = y;
       mindis = dgrid[x][y];
    )
    for (i = 0; i<= 3; i++) {                /* for the 4 adjacent neighbors */
       nborx = x + XNBOR[i];
       nbory = y + YNBOR[i];
       if ((nbory >= 0) && (nbory < SZE) && (nborx >= 0) && (nborx < SZE) &&
                          (!mark[nborx][nbory])) {
          mark[nborx][nbory] = TRUE;
          if (CgPt->getCg(nborx,nbory) <= n_the) {
             if (curpt != NULL) {            /* Reuse current node    */
                newpt = curpt;
                curpt = NULL;
```

```
            if (toppt == NULL) toppt = newpt;
         ) else newpt = new(Cell);      /* Allocate new node     */
         newpt->x = nborx;              /* Add to bottom of list */
         newpt->y = nbory;
         newpt->npt = NULL;
         if (botpt != newpt) botpt->npt = newpt;
         botpt = newpt;
       )
     )
   )
   if (curpt != NULL) delete(curpt);  /* Delete current node if not reused */
 )
 DcPt->prtMsg("   min cell and value is ",minx,miny);
 DcPt->prtFlt(mindis,CgPt->getCg(minx,miny));      /*debug*/
/******* Do we need a new start node? ************************************/
 if (n_nav == N_BACKTRACK) {
   if ((xrob == minx) && (yrob == miny) &&
      (DsPt->queryStart()) && (minx == robx) && (miny == roby)) {
      subStart();
      return(FALSE);  /* we did add one but want to act like we didn't */
   )
 )
/************ Do we add another destination node ***********************/
 if ((xrob != minx) || (yrob != miny)) {
   DsPt->pushDS(xrob,yrob,minx,miny,D_SUBDEST);
   DcPt->prtMsg("   New subdest pushed",minx,miny);
   DcPt->prtInt(DsPt->getDx(),DsPt->getDy());
   return(TRUE);
 )
 return(FALSE);
}  /** end subDest() **/
```

```
/*****************************************************************************/
/* Procedure: subStart         Determines new start node                Done */
/* Called by: subDest                                                        */
/* Calls:     (cgrid)  queryAtBgnd - Checks if a cell is at background       */
/*            (destack)newStart    - Creates a new start node in destination */
/*            (destack)queryStart  - Checks if current node is the start node*/
/*            (destack)getDx,getDy - Read current location from dest. stack  */
/*            (cgrid)  getBackgd   - Get background value                    */
/*            (cgrid)  getCg       - Read certainty grid value               */
/* Globals:   bgrid, cgrid, dgrid (Read only);  mark (Write only)            */
/*    Does a breadth first traverse of all empty cells reachable from the    */
/*    the robot searching for non dead end cells to use as a new start cell. */
/*    3 types are collected. 1) the cell near the background value which is  */
/*    closest to the destination (likely to lead to an unmapped area),       */
/*    2) the empty cell closest to the destination (the one for a normal     */
/*    subdestination must be closer than the current checkpoint), & 3) the   */
/*    cell with the lowest certainty value below the background value (might */
/*    lead to a passage upon closer inspection).  They will be prefered in   */
/*    the order given. If no cell meeting the requirements is found then the */
/*    current node is changed to a D_QUIT node.                              */
/*****************************************************************************/
void  subStart(void)
{
 Cell *botpt;       /* pointer to bottom of list                  */
 Cell *curpt;       /* pointer to current node of list            */
 Cell *newpt;       /* pointer to node being added to list        */
 Cell *toppt;       /* pointer to top of list                     */
 int   xrob,yrob;   /* current robot cell                         */
 int   nborx,nbory; /* indices of cells next to current cell      */
 int   x,y,i;       /* loop indices, indices of current cell      */
 float memdis;      /* lowest empty dgrid of reachable cell       */
 int   memx,memy;   /* coordinates of above cell                  */
 float mbgdis;      /* Reachable background cell with lowest dgrid */
 int   mbgx,mbgy;   /* Coordinates of above cell.                 */
 float mincer;      /* Lowest certainty below background found    */
 int   mcrx,mcry;   /* Coordinates of above cell                  */
 VoicePt->say("new.start");
 xrob = DsPt->getDx(); yrob = DsPt->getDy();
 for (x=0; x<SZE; x++) {                    /* unmark all cells        */
    for (y=0; y<SZE; y++) mark[x][y] = FALSE;
 }
 egrid[xdest][ydest] = 0;                   /* Ultimate dest. is not dead end*/
 mbgx = xrob; mbgy = yrob;                  /* Default background cell      */
 mbgdis = 10.0;
 memx = xrob; memy = yrob;                  /* Default empty cell           */
 memdis = 10.0;
 mcrx = xrob; mcry = yrob;                  /* Default low certainty cell   */
 mincer = CgPt->getBackgd();
 toppt = new(Cell);                         /* Store robot position in list */
 botpt = toppt;
 toppt->x = xrob;
 toppt->y = yrob;
 toppt->npt = NULL;
 mark[xrob][yrob] = TRUE;
 DcPt->prtMsg("SUBSTART",xrob,yrob);
 while (toppt != NULL) {            /* While list not empty */
    curpt = toppt;                  /*    pop top node      */
    toppt = toppt->npt;
    x = curpt->x;
    y = curpt->y;
    if ((dgrid[x][y] < memdis) && (egrid[x][y] == 0)) {
       memx = x;
       memy = y;
```

```
      memdis = dgrid[x][y];
   )
   for (i = 0; i<= 3; i++) {            /* for the 4 adjacent neighbors */
      nborx = x + XNBOR[i];
      nbory = y + YNBOR[i];
      if ((nbory >= 0) && (nbory < SZE) && (nborx >= 0) && (nborx < SZE) &&
                  (!mark[nborx][nbory])) {
         mark[nborx][nbory] = TRUE;
         if (CgPt->queryAtBgnd(nborx,nbory) && (egrid[x][y] == 0)) {
            if (dgrid[nborx][nbory] < mbgdis) {
               mbgdis = dgrid[nborx][nbory];
               mbgx = nborx;
               mbgy = nbory;
            )
         )
         if ((CgPt->getCg(nborx,nbory) < mincer) && (egrid[x][y] == 0)) {
            mincer = CgPt->getCg(nborx,nbory);
            mcrx = nborx;
            mcry = nbory;
         )
         if (CgPt->getCg(nborx,nbory) <= n_the) {
            if (curpt != NULL) {           /* Reuse current node    */
               newpt = curpt;
               curpt = NULL;
               if (toppt == NULL) toppt = newpt;
            } else newpt = new(Cell);      /* Allocate new node     */
            newpt->x = nborx;              /* Add to bottom of list */
            newpt->y = nbory;
            newpt->npt = NULL;
            if (botpt != newpt) botpt->npt = newpt;
            botpt = newpt;
         )
      )
   )
   if (curpt != NULL) delete(curpt);  /* Delete current node if not reused */
)
DcPt->prtMsg("   min backgound cell is ",mbgx,mbgy);
DcPt->prtFlt(mbgdis,CgPt->getCg(mbgx,mbgy));      /*debug*/
DcPt->prtMsg("   min empty cell is     ",memx,memy);
DcPt->prtFlt(memdis,CgPt->getCg(memx,memy));      /*debug*/
DcPt->prtMsg("   low certainty cell is ",mcrx,mcry);
DcPt->prtFlt(CgPt->getCg(mcrx,mcry));
/**** if different make a D_START dnode, else change to a  D_QUIT node *****/
if ((mbgx != xrob) || (mbgy != yrob)) {
   DsPt->newStart(mbgx,mbgy,D_START);
   DcPt->prtMsg("   New bg start node at",mbgx,mbgy);
) else if ((memx != xrob) || (memy != yrob)) {
   DsPt->newStart(memx,memy,D_START);
   DcPt->prtMsg("   New em start node at",memx,memy);
) else if ((mcrx != xrob) || (mcry != yrob)) {
   DsPt->newStart(memx,memy,D_START);
   DcPt->prtMsg("   New cr start node at",memx,memy);
) else {
   DsPt->changeDt(D_QUIT);
   DcPt->prtLne("   Changed checkpoint to quit");
)
) /** end subStart() **/
```

```
/*****************************************************************************/
/* Procedure: turn     Turns the robot and reads the sonars              Done */
/* Called by: moveMe                                                         */
/* Calls:     RwitPt->executeTurn - turns robot a set amount                 */
/*            CgPt->sonarScan - read the sonars                              */
/* Globals:   base_angl (Read only)                                          */
/*    Turns the robot a set number of degrees using RwitPt->executeTurn.     */
/*    Stops every 15 degrees to read the sonars. This routine should not be  */
/*    used when the robot is moving since it is very time consuming.         */
/*****************************************************************************/
void turn(
   float angle,          /* Number of degrees to turn                      */
   short waitFlag)       /* Flag for wait for turn to complete (1) or not (0)*/
(
 float robang;           /* Angle of robot when sonar readings taken       */
 float left;             /* Number of degrees left to turn                 */
 DcPt->prtLne("TURN: Turning to new angle. Change is "); DcPt->prtFlt(angle);
 left = angle;
 robang = base_angl;
 if (angle > 180.0) left = angle - 360.0;
 else if (angle < -180.0) left = angle + 360.0;
 if (CgPt->getSonarType() != S_OFF) {    /* if sonars on, scan while turning */
    while (left >= 15.0) {
       RwitPt->executeTurn(-15.0,10.0,TRUE); /* turn in positive direction*/
       left = left - 15.0;
       robang = robang + 15.0;
       CgPt->sonarScan(robang,robx,roby,offx,offy);
    )
    while (left <= -15.0) {
       RwitPt->executeTurn(15.0,10.0,TRUE);  /* turn in negative direction*/
       left = left + 15.0;
       robang =  robang - 15.0;
       CgPt->sonarScan(robang,robx,roby,offx,offy);
    )
 )
 if ((left > 1.0) || (left < -1.0)) {
    if (abs(left) <= 5.0) RwitPt->executeTurn(-left,2.0,waitFlag);
    else  RwitPt->executeTurn(-left,10.0,waitFlag);
 )
) /** end turn() **/
```

```
/*************************************************************************/
/* Procedure: travel              Main navigator routine             Done */
/* Called by: driver-test1                                                */
/* Calls:     (cgrid)getSonarType - see if sonars are on                  */
/*            (cgrid)sonarScan    - read the sonars                       */
/*            (display)drawCgrid  - displays latest cgrid and bgrid       */
/*            (Display)prtData    - displays latest robot data            */
/*            (Display)prtMsg     - print message on terminal             */
/*            (Display)say        - voice routine                         */
/*            atSDest    - checks to see if cell is robot subdestination  */
/*            collision  - checks current cell for obstacle               */
/*            elimInit   - initializes elimination arrays                 */
/*            fzbgrd     - fuzzes local badness grid                      */
/*            mkbgrd     - calculates badness grid                        */
/*            mkdgrd     - calculates direction grid                      */
/*            moveMe     - handles robot movement                         */
/*            newCell    - Performs tasks needed when new cell entered    */
/*            stopMe     - halts robot, and determines location           */
/*            travelAuxBack - determines next subdestination              */
/*            travelAuxPath - determines next subdestination              */
/*            travelAuxLocal - initializes local navigator                */
/*            update     - updates robot's distance, angle and position   */
/* Returns:   int - OK (destination reached), ERR (user abort), 2 (quitting) */
/* Globals:   base_angl,abortFlg,xdest,ydest,roby,roby,offx,offy,tgrid (Read)*/
/*            nwcell (Read,Write)                                         */
/* I/O:       Writes to terminal using curses                             */
/*    An inner loop controls the navigation to the sub-destination. Checks */
/*    for success, quitting, collisions and arranges for display of the map, */
/*    reading the sonars, and moving.                                     */
/*       An outer loop controls the navigation to ultimate destination. It */
/*    determines the next subdestination and whether to quit or not.      */
/*       When using the N_LOCAL or N_INCREMENT navigators the outer loop  */
/*    just initializes the local navigator.                               */
/*************************************************************************/
int travel(void)
{
 short rdsonar = FALSE;           /* Flag for sonars read last iteration  */
 short quit    = FALSE;           /* Flag for navigator quitting          */
 short doneyet = FALSE;           /* Flag for destination reached         */
 VoicePt->say("i.obey");
 elimInit();                       /* initialize elimination arrays */
 while ((!atUDest(robx,roby)) && (!abortFlg) && (quit < 2)) {
    if (n_nav == N_BACKTRACK)       travelAuxBack();
    else if (n_nav == N_INCREMENT)  travelAuxIncrement();
    else if (n_nav == N_PATH)       travelAuxPath();
    else if (n_nav == N_LOCAL)      travelAuxLocal();
    quit = quitOk();
    while ((!atSDest(robx,roby)) && (!abortFlg) && (!quit)) {
       quit = quitOk();
       collision();
       if (!quit) quit = destCheck();
       moveMe();
       nwcell = update(0.0,-1.0) || nwcell;
       if (nwcell && rdsonar) {         /* Read the sonars or decay the grid */
          newCell();                    /* but not both                      */
          rdsonar = FALSE;
       } else {
          CgPt->sonarScan(base_angl,robx,roby,offx,offy);
          DcPt->prtLne("Sonars read");
          rdsonar = TRUE;
          if (CgPt->getSonarType() != S_OFF)
             timer = timer + 0.66;  /* Sim. time for sonar read OK 10/31*/
       }
       nwcell = update(0.0,-1.0) || nwcell;
       fzbgrd();
       DxPt->drawCgrid(0);
       DcPt->prtData();
    }
    if ((!abortFlg) && (!quit)) {
       stopMe();
       VoicePt->say("checkpoint");
       DcPt->prtMsg("*TRAVEL: Robot has reached sub-destination ",xsubd,ysubd);
    }
 }
 if (abortFlg) {
    RwitPt->kill();
    nwcell = update(0.0, -1.0) || nwcell;
    spdfac = 0;
    slew = 0;
    DcPt->prtMsg("*Sub-destination changed to current location. ",robx,roby);
    DsPt->changeDS(robx,roby);
    xsubd = robx;
    ysubd = roby;
    DcPt->prtData();
 }
 mkbgrd();                                  /* make final display accurate */
 DxPt->drawCgrid(1);
 if (abortFlg) return(ERR);
 else if (quit) {
    VoicePt->say("i.quit");
    return(2);
 } else {
    VoicePt->say("destination");
    DcPt->prtMsg("*TRAVEL: Robot has reached destination ",xdest,ydest);
    return(OK);
 }
} /** end travel() **/
```

```
/*****************************************************************************/
/* Procedure: travelAuxBack   subdestinations & dead ends for backtrack  Done*/
/* Called by: travel                                                         */
/* Calls:     (cgrid)  circleScan - Circular sonar scan                      */
/*            (cgrid) sonarScan   - Read the sonars                          */
/*            (destack)(lots)     - Manipulate the destination stack         */
/*            (display)drawCgrid  - Update X display                         */
/*            (display)prtData    - Update the curses display                */
/*            fzbgrd     - Fuzz local badness grid                           */
/*            getAngle   - Get robot's current orientation                   */
/*            mkbgrd     - Make a new badness grid                           */
/*            mkdgrd     - Make a new direction grid                         */
/*            pathToDest - Check for possible existance of path to destinatn */
/*            subDest    - Find a subdestination                             */
/* Globals:   base_angl (Read, Write);  elmx, elmy (Write only);             */
/*            offx, offy, robx, roby (Read only)                             */
/*    Determines the next subdestination for the robot. When the procedure   */
/*    starts, the top dnode of the destination stack has the present location*/
/*    as the destination point. This is also the current sub destination.    */
/*    When the procedure exits a new subdestination will be on top of the    */
/*    stack. -- Uses subDest() to pick the next best subdestination, then    */
/*    determines whether to use it or whether reversing or backtracking is   */
/*    needed. Pops the stack or modifies the type field accordingly.         */
/*       Whenever a new sub-destination is farther from the ultimate dest.   */
/*    than the present one, then some cells are marked as dead ends.         */
/*       markDeadEndBox is used to set dead ends over a box from the last    */
/*    checkpoint to the current spot. This occurs during a reverse.          */
/*       markDeadEndAll is used to set all reachable cells between the       */
/*    current spot and the last checkpoint. This occurs during a backtrack.  */
/*    As long as the robot progresses toward the ultimate destination, only  */
/*    the actual route and neighbor cells of each checkpoint are marked as   */
/*    dead ends.                                                             */
/*****************************************************************************/
void travelAuxBack(void)
{
 int   i;                  /* loop index */
 short newnode = FALSE;    /* Flag indicating a new destination node on stack*/
 short newdest = FALSE;    /* Flag indicating a new subdestination is set    */
 short pathchecked = FALSE; /* Flag for pathToDest already run               */
 static int farx = -1;      /* X coordinate of checkpoint closest to  dest.  */
 static int fary = -1;      /* Y coordinate of checkpoint closest to dest.   */
/**** Mark the previous dead end if it was a backtrack ******/
 if (DsPt->getDt() == D_BACKTRACK)
    markDeadEndAll(DsPt->getDx(),DsPt->getDy(),farx,fary);
/**** Do some looking to ensure some local map is present ****/
 look();
 if (atUDest(xsubd,ysubd)) return;             /* Done */
 if (DsPt->getDt() != D_REVERSE) {
    farx = robx;
    fary = roby;
 }
/**** Do subdestination determinations ****/
/****** Dont' consider neighbors as next destination unless ultimate dest */
 for (i=0; i<=7; i++) { egrid[robx+XNBOR[i]][roby+YNBOR[i]] = 1; }
 egrid[robx][roby] = 1;
 egrid[xdest][ydest] = 0;
 newnode = subDest(n_path);
 newdest = FALSE;
 while (!newdest) {
    for (i=0; i<=7; i++) { egrid[robx+XNBOR[i]][roby+YNBOR[i]] = 1; }
    egrid[robx][roby] = 1;
    egrid[xdest][ydest] = 0;
    if (!newnode) {
       markDeadEndBox(DsPt->getSx(),DsPt->getSy(),DsPt->getDx(),DsPt->getDy());
       DcPt->prtMsg("popping ",DsPt->getDx(),DsPt->getDy());
       DsPt->popDS();
       if ( subDest(n_path) ) {
          if (distance(DsPt->getSx(),DsPt->getSy(),farx,fary) >
              distance(DsPt->getDx(),DsPt->getDy(),farx,fary)) {
             DsPt->changeDt(D_REVERSE);
             DcPt->prtMsg("reversing to",DsPt->getDx(),DsPt->getDy());
             VoicePt->say("reverse");
             if (!pathchecked) {
                if (pathToDest(DsPt->getDx(),DsPt->getDy(),xdest,ydest))
                   pathchecked = TRUE;
                else DsPt->changeDt(D_QUIT);
             }
          } else {
             DsPt->popDS();
             DcPt->prtMsg("reverse backtracking to",DsPt->getDx(),DsPt->getDy());
             VoicePt->say("backtrack");
             DsPt->changeDt(D_BACKTRACK);
             markDeadEndAll(DsPt->getDx(),DsPt->getDy(),farx,fary);
          }
       } else {
          DcPt->prtMsg("backtracking to",DsPt->getDx(),DsPt->getDy());
          VoicePt->say("backtrack");
          DsPt->changeDt(D_BACKTRACK);
          markDeadEndAll(DsPt->getDx(),DsPt->getDy(),farx,fary);
       }
    }
    newdest = TRUE;          /* at this point a new destination is ready */
/*****************************************************************************/
/* If the new destination is  close, it doesn't warrent a new checkpoint. */
/* When reversing it is not even necessary to go there. We mark it as if  */
/* we did and look for another checkpoint. For subdestinations closer to  */
/* the ultimate destination, the possibility of additional sonar data     */
/* requires the physical traverse so the destination is simply combined   */
/* with the previous one. This prevents a lot of close together           */
/* checkpoints which would complicate any backtracking routes.            */
/*****************************************************************************/
    if (nearTo(DsPt->getDx(), DsPt->getDy(),robx,roby)) {
       DcPt->prtLne("Sub destination is close");
       VoicePt->say("close");
       if  ((DsPt->getDt() == D_SUBDEST) &&
            (!atUDest(DsPt->getDx(),DsPt->getDy()))) {
          DcPt->prtLne("Popped and Added to previous subdestination");
          xsubd = DsPt->getDx();
          ysubd = DsPt->getDy();
          DsPt->popDS();
          DsPt->changeDS(xsubd,ysubd);
       } else if  (DsPt->getDt() == D_REVERSE) {
          newdest = FALSE;
          newnode = FALSE;
       }
    }
 }
 xsubd = DsPt->getDx();          /* read next subdestination from  */
 ysubd = DsPt->getDy();          /* top of destination list        */
 DcPt->prtMsg("New subdest and type are :",xsubd,ysubd);
 DcPt->prtInt((int)(DsPt->getDt()));
 if ((xsubd != robx) || (ysubd != roby)) {
    mktgrd();                                    /* initialize local travel */
    mkdgrd(robx,roby,xsubd,ysubd,n_the,FALSE); /* make new direction grid */
    mkbgrd();                                    /* make new badness grid   */
    fzbgrd();                                    /* fuzz local badness grid */
    DcPt->prtData();
```

```
    DxPt->drawCgrid(1);
 }
} /** end travelAuxBack() **/
```

```
/*****************************************************************************/
/* Procedure: travelAuxIncrement  Initialize Incremental navigator           */
/* Called by: travel                                                         */
/* Calls:     (destack)changeDS  - change subdestion on stack                */
/*            (display)prtData   - print current robot data                  */
/*            (display)drawCgrid - display latest certainty & badness maps   */
/*            fzbgrd - fuzz local badness grid                               */
/*            look   - uses sonars to generate some local map                */
/*            mkbgrd - recreate badness grid                                 */
/*            mkdgrd - recreate direction grid                               */
/*            mktgrd - reinitialize local travel grid                        */
/* Globals:   xsubd, ysubd (Write only)                                      */
/*    Does whatever initialization is needed when the incremental navigator  */
/*    is started or restarted. Recreates the direction, local travel and     */
/*    badness grids.                                                         */
/*****************************************************************************/
void travelAuxIncrement(void)
{
/**** Do some looking to ensure some local map is present ****/
 look();
 xsubd = xdest; ysubd = ydest;
 DsPt->changeDS(xdest,ydest);
 mktgrd();                                    /* reset local travel grid */
 mkdgrd(robx,roby,xdest,ydest,n_path,TRUE);  /* make new direction grid */
 mkbgrd();                                    /* make new badness grid   */
 fzbgrd();                                    /* fuzz local badness grid */
 DcPt->prtData();
 DxPt->drawCgrid(1);
}  /** end travelAuxIncrement() **/
/*****************************************************************************/
/* Procedure: travelAuxLocal       Initialize local navigator                */
/* Called by: travel                                                         */
/* Calls:     (destack)changeDS  - change subdestion on stack                */
/*            (display)prtData   - print current robot data                  */
/*            (display)drawCgrid - display latest certainty & badness maps   */
/*            eliminit   - initializes elimination arrays                    */
/*            fzbgrd - fuzz local badness grid                               */
/*            look   - uses sonars to generate some local map                */
/*            mkbgrd - recreate badness grid                                 */
/*            mkdgrd - recreate direction grid                               */
/*    Does whatever initialization is needed when the local navigator is     */
/*    started in global mode. Initializes the elimination arrays, destination*/
/*    and badness grids.                                                     */
/*****************************************************************************/
void travelAuxLocal(void)
{
/**** Do some looking to ensure some local map is present ****/
 look();
 elmx = robx; elmy = roby;          /* initialize elimination array center */
 elimInit();                        /* initialize elimination arrays */
 xsubd = xdest; ysubd = ydest;
 DsPt->changeDS(xdest,ydest);
 mkdgrd(robx,roby,xdest,ydest,n_path,TRUE);  /* make new direction grid */
 mkbgrd();                                    /* make new badness grid   */
 fzbgrd();                                    /* fuzz local badness grid */
 DcPt->prtData();
 DxPt->drawCgrid(1);
}
/*****************************************************************************/
/* Procedure: travelAuxPath        Determine next subdestination             */
/* Called by: travel                                                         */
/* Calls:     (cgrid)circlescan - Build a local map                          */
/*            (cgrid)sonarScan  - Read the sonars                            */
```

```
/*            (cgrid)getCg      - Access the certainty grid                 */
/*            (destack)(lots)   - Manipulate the destination stack          */
/*            (display)drawCgrid - Update X display                         */
/*            (display)prtData - Update the curses display                  */
/*            fzbgrd           - Fuzz local badness grid                    */
/*            getAngle         - Get robot's current orientation            */
/*            mkbgrd           - Make a new badness grid                    */
/*            mkdgrd           - Make a new direction grid                  */
/*    Determines the next subdestination for the PATH navigator and sets up  */
/*    the local navigator to head for it.                                    */
/*************************************************************************/
void travelAuxPath(void)
{
 short newnode = FALSE;    /* Flag indicating a new destination node on stack*/
/**** Do some looking to ensure some local map is present ****/
 look();
 if (atUDest(xsubd,ysubd)) return;             /* Done */
/**** Do subdestination determinations ****/
 newnode = subDest(n_path);
 if (!newnode) DsPt->changeDt(D_QUIT);
 xsubd = DsPt->getDx();          /* read next subdestination from  */
 ysubd = DsPt->getDy();          /* top of destination list        */
 DcPt->prtMsg("New subdest and type are :",xsubd,ysubd);
 DcPt->prtInt((int)(DsPt->getDt()));
 if ((xsubd != robx) || (ysubd != roby)) {
    mktgrd();                                  /* initialize local travel */
    mkdgrd(robx,roby,xsubd,ysubd,n_the,FALSE); /* make new direction grid */
    mkbgrd();                                  /* make new badness grid   */
    fzbgrd();                                  /* fuzz local badness grid */
    DcPt->prtData();
    DxPt->drawCgrid(1);
 }
}
  /** end travelAuxPath() **/
```

```
/*************************************************************************/
/* Procedure: update                                                Done */
/* Called by: collision, moveMe, travel, driver:test1, stopMe            */
/* Calls:     getAngle     -  Gets robots perception of angle             */
/*            getDistance  -  Gets robots perception of distance          */
/*            findPosition -  Determine which grid cell the robot is in   */
/* Returns:   int - TRUE if new cell entered, FALSE if not                */
/* Globals:   base_angl, base_dis, timer (Read, Write)                    */
/*    Updates the robot's distance, angle and location within the certainty */
/*    grid.  The new distance and angle are read from the robot itself unless*/
/*    passed as arguments. A passed distance of -1 indicates that they must */
/*    be read. They are fed into FINDPOSITION to determine the robot's new  */
/*    location.                                                           */
/*************************************************************************/
/*????? what if getDistance is bad??*/
short update(
   float newangle,       /* Robots orientation in degrees                 */
   float newdist)        /* Robot's distance in cm                        */
{
 int nwCell;             /* Flag for new cell entered                     */
 float dist;             /* Change in robot's distance                    */
 timer = timer + 0.77;                /* Simulated time for update (OK 10/31)*/
 if (newdist == -1.0) {               /* If distance & angle not passed     */
    if (spdfac == 0) {                /* If not translating, read dist first */
       newdist =  getDistance();
       newangle = getAngle();
    } else {                          /* else read angle before distance    */
       newangle = getAngle();
       newdist =  getDistance();
    }
 }
 dist = newdist - base_dis;
 if (abs(dist) > 0.2) {                          /* If distance has changed*/
    nwCell = findPosition(dist,newangle);        /*  find new grid position*/
    base_dis = newdist;                          /*  set new distance      */
 }
 if (newangle != base_angl) base_angl = newangle;  /* If rotation occurred   */
                                                   /*  store new orientation */
 timer = 0.0;                                      /* Reset simulator timer  */
 return(nwCell);
}  /** end update() **/
```

```
/*****************************************************************************/
/*       display.h   03/27/92   Timothy T. Good                              */
/*    Contains declarations for the four classes used in the display module. */
/*    DisplayC is for curses output, DisplayX is for X output and Voice is   */
/*    for aural output. Speech2 is the speech simulator.                     */
/*****************************************************************************/
#ifndef DISPLAYC_H
#define DISPLAYC_H

#include "cgrid.h"       /* certainty  grid declarations   */
#include "oman.h"        /* navigator  declarations        */
#include <Speech.h>      /* /pro/ai/robot/include/Speech.h */
#include <X11/Xlib.h>
#include <math.h>

 const int   NUM_COLORS      = 32;

/*****************************************************************************/
/*---------------------------------------------------------------------------*/
/*- Class: DisplayC           Handles curses output                         -*/
/*---------------------------------------------------------------------------*/
/*****************************************************************************/
class DisplayC {

 private:
      int   curline;            /* Current print line on screen            */
      FILE *fplog;              /* Log file pointer                        */

 public:
   DisplayC();                  /* Initializes curses, writes headings     */
   ~DisplayC();                 /* Closes the curses package               */
   void prtData(void);          /* Writes robot data                       */
   void prtFlt(float val);      /* Writes a floating point number          */
   void prtFlt(float val1,float val2); /* Writes two floating point numbers */
   void prtInt(int val);        /* Writes an integer                       */
   void prtInt(int val1, int val2); /* Writes two integers                 */
   void prtLne(char *str);      /* Changes output line & writes a string   */
   void prtMsg(char *,int x,int y); /* Writes string followed by two integers*/
   void prtParm(void);          /* Prints navigator parameters             */
   void prtScreen(void);        /* Rewrites the curses screen              */
 );
/*****************************************************************************/
/*---------------------------------------------------------------------------*/
/*- Class: DisplayX           Handles X output                              -*/
/*---------------------------------------------------------------------------*/
/*****************************************************************************/
class DisplayX {

 private:
      Display   *mydisplay;
      Window     win;
      int        screen;
      GC         gc;
      Colormap   xcolormap;
      unsigned long colorArray[ NUM_COLORS ];
      XColor     robotcolor,destcolor,subdcolor,occupcolor,deadcolor;
      inline int selectColor(float var) /* Determine grayness             */
               {  return((int)floor((var) * (NUM_COLORS - 1))); }
      void allocateColors(void);   /* (tht) Creates gray scale color table */
      void createWindow(void);     /* (tht) Creates window to hold display */
      void drawMapImage(void);     /* Puts part of cgrid, bgrid in window  */
      void drawNewMap(void);       /* (tht) Puts entire grids in window    */
      void setColors(void);        /* Assigns special colors               */
 public:
      DisplayX();
      ~DisplayX();
      void drawCgrid(int i);       /* Draws the grids in the X window      */
      void drawGridImage(int,int); /* Draws single cell                    */
      void drawHighLight(int,int); /* Highlights a single cell in color    */
      void drawRobot(int,int,float); /* Draws the robot                    */
      void drawDest(int x,int y,int xs,int xy); /* Puts detination in window*/
      void Refreshx(void);         /* Reprints the X display               */
);

/*****************************************************************************/
/*---------------------------------------------------------------------------*/
/*- Class: Speech2                                                          -*/
/*-   Declaration for the Speech2 class.  Speech2 is the speech simulator it */
/*-   uses /usr/demo/SOUND/play to have prerecorded sounds replayed on the   */
/*-   terminal speakers. Except for the name change to Speech2 the Speech.h  */
/*-   file in /pro/ai/robot/include could be used as a declaration.          */
/*---------------------------------------------------------------------------*/
/*****************************************************************************/
class Speech2 {

 public:
    Speech2(Comm *line);
    ~Speech2();
    void say(const char *, int wait = 0);
);

/*****************************************************************************/
/*---------------------------------------------------------------------------*/
/*- Class: Voice                                                            -*/
/*-    The Voice class is a wrapper around the Speech and Speech2 classes   -*/
/*-    to separate the choice of the speech option from the modules using it-*/
/*---------------------------------------------------------------------------*/
/*****************************************************************************/
class Voice {

 private:
    DisplayC *c_display;  /* Point to curses display class                 */
    int      p_line;      /* Print line for written output, -1,0 for aural */
    Speech  *s_pointer;   /* Pointer to Speech class (robot library)       */
    Speech2 *s2_pointer;  /* Pointer to Speech2 class (display)            */
 public:
          Voice (int pline, DisplayC *cdisplay);                  /* print */
          Voice (int pline, Comm *speechline, DisplayC *cdisplay); /* aural */
         ~Voice ();
    void  say(const char *);
);

#endif   /* DISPLAYC_H */
```

```
/*****************************************************************************/
/*    display.C  04/03/92  Timothy T. Good                                   */
/*  Contains routines to display the certainty grid using X-windows, print   */
/*  current robot information using the curses package and generate audio    */
/*  output using the robot speech generator or terminal simulator.           */
/*    Consists of 3 classes. DisplayC for curse output, DisplayX for X       */
/*  output and Voice for audio output. The Voice class uses the Speech class */
/*  in the robot library, the Speech2 class in this file or neither and      */
/*  outputs using DisplayC.                                                  */
/*     Because curses is used to position the cursor, all printed output     */
/*  should be though this interface or in the driver.                        */
/*                                                                           */
/*  Much of the code for the X-window display was originally written in C by */
/*  Tu-Hsin Tsai and modified by Tim Good. This includes the following:      */
/*    allocateColors(), RefreshX(), createWindow(), drawMapImage(),          */
/*    drawGridImage(), drawRobot()                                           */
/*  Modifications included changing names, sizes and colors; conversion to   */
/*  C++ and a class, and adding minimal documentation.                       */
/*****************************************************************************/
#include "cgrid.h"       /* certainty grid declarations    */
#include "oman.h"        /* navigator declarations         */
#include "destack.h"     /* destination stack declarations */
#include "display.h"
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include <stdio.h>
#include <curses.h>
#include <math.h>

 const int   PIXELS_PER_CELL = 4;
 const int   DEFAULT_WIDTH   = (SZE * PIXELS_PER_CELL);
 const int   DEFAULT_HEIGHT  = (SZE * PIXELS_PER_CELL);
 const int   BORDERW         = 4;
 const int   BASE            = 2048;

/*****************************************************************************/
/* Global Variables                                                          */
/*    These are values which are output which do not have access functions.  */
/*****************************************************************************/

extern float bgrid[SZE][SZE];    /*(oman) Combined grid                     */
extern short egrid[SZE][SZE];    /*(oman) Dead end grid                     */
extern int   robx,roby;          /*(oman) Robot's position in the grid      */
extern float slew;               /*(oman) Robot's turning rate              */
extern int   spdfac;             /*(oman) Robot's speed multiplier          */
extern int   xdest,ydest;        /*(oman) Robot's destination in the grid   */
extern Certgrid  *CgPt;          /*(driver) Certainty grid class            */
extern Destack   *DsPt;          /*(driver) Destination stack class         */
```

```
/*****************************************************************************/
/*---------------------------------------------------------------------------*/
/*- Class: DisplayC            Handles curses output                        -*/
/*---------------------------------------------------------------------------*/
/*****************************************************************************/
/*****************************************************************************/
/* Procedure: DisplayC     Constructor for DisplayC class                Done */
/* Called by: <user>                                                         */
/* Calls:     curses routines, display_init                                  */
/* Output:    Writes header to log file                                      */
/*    Initializes curses, opens the log file, and initializes the X display. */
/*****************************************************************************/
DisplayC::DisplayC(void)
{
 curline = 20;                   /* First print line for log file echo      */
 initscr();                      /* Initialize curses package               */
 nocbreak();
 erase();
 fplog = fopen("oman.log","w");
 fprintf(fplog,"\nLog file for robot navigation test.  Timothy T. Good ");
}
/*****************************************************************************/
/* Procedure: ~DisplayC      Destructor for DisplayC class               Done */
/* Called by: <user>                                                         */
/* Calls:     curses routines, display_off                                   */
/* Output:    closes the log file                                            */
/*    Closes the curses package, the log file and X display.                 */
/*****************************************************************************/
DisplayC::~DisplayC(void)
{
  endwin();
  fclose(fplog);
}
```

```
/*****************************************************************************/
/* Procedure: prtData                                                  Done */
/* Called by: <user>                                                        */
/* Calls:     curses routines, destination stack accessor functions         */
/*            navigator access functions                                    */
/* Globals:   robx,roby,spdfac,xdest,ydest  (Read)                          */
/*    Displays robot status on lines 3-5 of the terminal.                   */
/*****************************************************************************/
void DisplayC::prtData(void)
{
 int speed;                 /* Robot's speed (cm/sec)                       */
 speed = spdfac * SPEED_INCREMENT;
 mvprintw(3,0,"Position:  %3d,%3d;  Destination:%3d,%3d;  Subdest.:%3d,%3d  ",
   robx,roby,xdest,ydest,DsPt->getDx(),DsPt->getDy());
 printw("Checkpt:%3d,%3d",DsPt->getSx(),DsPt->getSy());
 mvprintw(4,0,"Bearing:  %6.1f     Distance: %7.1f     Voltage:  %4.1f",
         getBaseAngle(),getBaseDist(),getVolts());
 mvprintw(5,0,"Slew:      %5.1f     Speed:       %2d       ",
         slew,speed);
 switch( (int)DsPt->getDt() ) {
    case 0:
       printw("DestType: START    ");
    break;
    case 1:
       printw("Desttype: SUBDEST  ");
    break;
    case 2:
       printw("Desttype: REVERSE  ");
    break;
    case 3:
       printw("Desttype: BACKTRACK");
    break;
    case 4:
       printw("Desttype: QUIT     ");
    break;
    case 5:
       printw("Desttype: None     ");
    break;
    default:
       printw("Desttype: error    ");
    break;
 }
}  /** end prtData() **/
/*****************************************************************************/
/* Procedure: prtFlt     Print floating point number(s) using curses    Done */
/* Called by: <user>                                                        */
/* Calls:     curses routine                                                */
/*    Outputs the argument(s) to both the terminal and the log file.        */
/*****************************************************************************/
void DisplayC::prtFlt(
           float val)                /* number to be output */
{
 printw(" %6.2f ",val);
 fprintf(fplog," %6.2f ",val);
}

void DisplayC::prtFlt(
           float val1, float val2)    /* numbers to be output */
{
 printw(" %6.2f  %6.2f ",val1,val2);
 fprintf(fplog," %6.2f  %6.2f ",val1,val2);
}  /** end prtFlt() **/
```

```
/*****************************************************************************/
/* Procedure: prtInt    Print integer(s) using curses                  Done */
/* Called by: <user>                                                        */
/* Calls:     curses routine                                                */
/*    Outputs the argument(s) to both the terminal and the log file.        */
/*****************************************************************************/
void DisplayC::prtInt( int val)             /* integer to be ouput  */
{
 printw(" %d ",val);
 fprintf(fplog," %d ",val);
}

void DisplayC::prtInt( int val1, int val2)   /* integers to be ouput */
{
 printw(" %d  %d ",val1,val2);
 fprintf(fplog," %d  %d ",val1,val2);
}  /** end prtInt() **/

/*****************************************************************************/
/* Procedure: prtLne       Print a string using curses                  Done */
/* Called by: <user>                                                        */
/* Calls:     curses routines                                               */
/*    Positions the cursor at the next line of the display and outputs the  */
/*    string passed as an argument.  Also outputs the string to the log file.*/
/*****************************************************************************/
void DisplayC::prtLne(
           char *str)      /* string to be printed                          */
{
 move(curline+1,0);
 clrtoeol();
 mvprintw(curline,0,"%2d %s",curline,str);
 clrtoeol();
 curline++;
 refresh();
 if (curline==45) curline = 20;
 fprintf(fplog,"\n%s",str);
}  /** end prtLne() **/

/*****************************************************************************/
/* Procedure: prtMsg    Print a string and two integers using curses    Done */
/* Called by: <user>                                                        */
/* Calls:     curses routines                                               */
/*    Positions the cursor at the next line of the display and outputs the  */
/*    string and two integers to the terminal and the log file.             */
/*****************************************************************************/
void DisplayC::prtMsg(
    char *str,          /* String to be output                          */
    int x, int y)       /* Integers to be output, Usually a grid cell */
{
 move(curline+1,0);
 clrtoeol();
 mvprintw(curline,0,"%2d %s %d  %d ",curline,str,x,y);
 clrtoeol();
 curline++;
 refresh();
 if (curline==45) curline = 20;
 fprintf(fplog,"\n%s %d  %d ",str,x,y);
} /** end prtmsg() **/
```

```
/*****************************************************************************/
/* Procedure: prtParm      Print navigation parameters using curses          */
/* Called by: <user>                                                         */
/* Calls:     curses routines, navigator access functions,                   */
/*            (cgrid)getBackgd                                               */
/*    Prints the navigation parameters on lines 0 & 1 of the terminal.       */
/*****************************************************************************/
void DisplayC::prtParm(void)
{
 switch(CgPt->getSonarType()) {
    case 0:
       mvprintw(0,0,"Sonars:  OFF   ");
    break;
    case 1:
       mvprintw(0,0,"Sonars:  SQUARE");
    break;
    case 2:
       mvprintw(0,0,"Sonars:  TEST  ");
    break;
    case 3:
       mvprintw(0,0,"Sonars:  FRONT ");
    break;
    default:
       mvprintw(0,0,"Sonars:  ERROR ");
    break;
 }
 switch(getNavigator()) {
    case 0:
       printw("  Navigator:  LOCAL    ");
    break;
    case 1:
       printw("  Navigator:  BACKTRACK");
    break;
    case 2:
       printw("  Navigator:  PATH     ");
    break;
    case 3:
       printw("  Navigator:  INCREMENT");
    break;
    default:
       printw("  Navigator:  ERROR    ");
    break;
 }
 if (getDecayFlag() == 0) printw("  Decay:  OFF");
 else printw("  Decay:  ON ");
 mvprintw(1,0,"Empty:   %4.2f    Occupied:   %4.2f       Path value: %4.2f",
   getConstEmpty(),getConstOccup(),getConstPath());
 printw("  Background:  %4.2f",CgPt->getBackgd());
} /** end prtParm() **/

/*****************************************************************************/
/* Procedure: prtScreen                                                      */
/*    Reprints the entire curses display.                                    */
/*****************************************************************************/
void DisplayC::prtScreen(void)
{
 clearok(stdscr, TRUE);
}
```

```
/*****************************************************************************/
/*---------------------------------------------------------------------------*/
/*- Class: DisplayX          Handles X output                               -*/
/*---------------------------------------------------------------------------*/
/*****************************************************************************/
/*****************************************************************************/
/* Procedure: DisplayX          Constructor for DisplayX              Done */
/* Called by: <user>                                                       */
/* Calls:     createWindow   - make a window for the display               */
/*            allocateColors  - assign gray scale colors for grid values   */
/*            setColors      - assign colors for robot, destination, etc   */
/*    Creates the window and sets the colors.                              */
/*****************************************************************************/
DisplayX::DisplayX(void)
{
 createWindow();
 allocateColors();
 setColors();
} /** end DisplayX() **/

/*****************************************************************************/
/* Procedure: ~DisplayX        Destructor for DisplayX                 Done */
/* Called by: <user>                                                       */
/* Calls:     X  routines                                                  */
/*    Kills the window and closes the X display.                           */
/*****************************************************************************/
DisplayX::~DisplayX(void)
{
 XDestroyWindow(mydisplay,win);
 XCloseDisplay(mydisplay);
} /** end ~DisplayX() **/

/*********************************************************/
/* allocateColors    (Author Tu-Hsin Tsai)               */
/*                                                       */
/*  This routine allocates a color table for drawing     */
/*  the certainty grid cells. It uses 32 values on the   */
/*  gray scale.                                          */
/* Called by: DisplayX                                   */
/* Calls:     X routines                                 */
/*********************************************************/
void DisplayX::allocateColors(void)
{
    Status status;
    unsigned long plane_masks;
    XColor colorTemplate;
    unsigned int index,pvalue;

    status = XAllocColorCells (mydisplay, xcolormap, 0,
                               &plane_masks, 0, colorArray, NUM_COLORS);

    colorTemplate.flags = DoRed | DoGreen | DoBlue;

    for ( index = 0; index < NUM_COLORS; index++ )
    {   pvalue = (NUM_COLORS - index) * BASE - 1;
        colorTemplate.pixel = colorArray[ index ];
        colorTemplate.red = pvalue;
        colorTemplate.green = pvalue;
        colorTemplate.blue = pvalue;

        XStoreColor (mydisplay, xcolormap, &colorTemplate);
            }
}
```

```
/*******************************************************/
/* createWindow     (Author Tu-Hsin Tsai)              */
/*  Modified by ttg to include space for badness grid. */
/*                                                     */
/*  This routine creates a window large enough to show */
/*  the entire certainty grid.                         */
/* Called by: DisplayX                                 */
/* Calls:     X routines                               */
/*******************************************************/
void DisplayX::createWindow(void)
{
 unsigned int width, height;
 int    x = 0, y = 0;
 int    ac = 1;
 char   * av[4];
 XSizeHints size_hints;
 Pixmap     icon_pixmap;   /* this is used but not set, i don't know */
                           /* what it should be so I ignore it, ttg  */
 XGCValues  gcv;
 XEvent     report;

 if ((mydisplay=XOpenDisplay("")) == NULL)
 {      printf("basicwin: cannot connect to X server \n");
        exit( -1 );
 }

 screen = DefaultScreen(mydisplay);
 width = DEFAULT_WIDTH;
 height = DEFAULT_HEIGHT * 2;
 win = XCreateSimpleWindow(mydisplay, RootWindow(mydisplay,screen),
                           x, y, width, height, BORDERW,
                           BlackPixel(mydisplay,screen),
                           WhitePixel(mydisplay,screen));

 size_hints.flags = PPosition | PSize;
 size_hints.x = x;
 size_hints.y = y;
 size_hints.width = width;
 size_hints.height = height;
 av[0] = "y";
 av[1] = 0;
 XSetStandardProperties(mydisplay, win,"cgrid","cgrid",
                        icon_pixmap,av,ac,&size_hints);
 XSelectInput(mydisplay, win, ExposureMask | StructureNotifyMask );
 gc = XCreateGC( mydisplay, win, 0, &gcv);
 XSetForeground(mydisplay, gc, BlackPixel(mydisplay,screen));
 XMapWindow(mydisplay,win);
 XFlush(mydisplay);

 XNextEvent(mydisplay,&report);
 xcolormap = DefaultColormap( mydisplay, DefaultScreen( mydisplay ));
}
```

```
/*****************************************************************************/
/* Procedure: drawCgrid                                                 Done */
/* Called by: <user>                                                         */
/* Calls:     drawNewMap, drawMapImage, drawDest, drawRobot                  */
/*            destination stack access functions, X routines                 */
/* Globals:   robx, roby, xdest, ydest (Read only)                           */
/* I/O:       Redisplays the updated X window                                */
/*    Writes the certainty  and badness grids to the Xwindow and adds Luey's */
/*     position, the destination, and the subdestination.                    */
/*****************************************************************************/
void DisplayX::drawCgrid(int i)
{
 if (i==1) drawNewMap();
 else drawMapImage();
 drawDest(xdest,ydest,DsPt->getDx(),DsPt->getDy());
 drawRobot(robx,roby,getBaseAngle());
 XFlush(mydisplay);   /* Refreshx() */
}

/*****************************************************************************/
/* Procedure: drawDest                                                  Done */
/* Called by: <user>, drawCgrid                                              */
/* Calls:     X routines                                                     */
/*    Draws the destination cell in pink if it is within the grid bounds.    */
/*    Draws the subdestination in red if it is within the grid bounds.       */
/*****************************************************************************/
void DisplayX::drawDest(int x, int y, int xs, int ys)
{
/***** Draw the ultimate destination     */
 if ((x<0) || (x>=SZE) || (y<0) || (y>=SZE)) ;
 else {
    XSetForeground(mydisplay,gc,destcolor.pixel);
    XFillRectangle(mydisplay,win,gc,x*PIXELS_PER_CELL,
       DEFAULT_HEIGHT-((y+1)*PIXELS_PER_CELL),
       PIXELS_PER_CELL, PIXELS_PER_CELL);
    XFillRectangle(mydisplay,win,gc,x*PIXELS_PER_CELL,
       DEFAULT_HEIGHT+DEFAULT_HEIGHT-((y+1)*PIXELS_PER_CELL),
       PIXELS_PER_CELL, PIXELS_PER_CELL);
 }
/***** Draw the current sub destination     */
 if ((xs<0) || (xs>=SZE) || (ys<0) || (ys>=SZE)) return;
 XSetForeground(mydisplay,gc,subdcolor.pixel);
 XFillRectangle(mydisplay,win,gc,xs*PIXELS_PER_CELL,
    DEFAULT_HEIGHT-((ys+1)*PIXELS_PER_CELL),
    PIXELS_PER_CELL-1, PIXELS_PER_CELL-1);
 XFillRectangle(mydisplay,win,gc,xs*PIXELS_PER_CELL,
    DEFAULT_HEIGHT+DEFAULT_HEIGHT-((ys+1)*PIXELS_PER_CELL),
    PIXELS_PER_CELL-1, PIXELS_PER_CELL-1);
}
```

```
/*************************************************************************/
/* drawGridImage     (Author  Tu-Hsin Tsai)                              */
/*  Modified by ttg to include the badness grid cell                     */
/*                                                                       */
/*  This routine draws a corresponding gray color to the specified cell */
/* Called by: <user>                                                     */
/* Calls:     certainty grid access functions, X routines                */
/* Globals:   bgrid, egrid (Read only)                                   */
/*************************************************************************/
void DisplayX::drawGridImage(int x,int y)
{
 int color;

  if ((x < 0) || (x>= SZE) || (y < 0) || (y >= SZE)) return;
  color = selectColor(CgPt->getCg(x,y));
  XSetForeground( mydisplay, gc, colorArray[color] );
  XFillRectangle( mydisplay, win, gc,
                  x*PIXELS_PER_CELL, DEFAULT_HEIGHT - ((y+1)*PIXELS_PER_CELL),
                  PIXELS_PER_CELL, PIXELS_PER_CELL);
  if (CgPt->getCg(x,y) > THO)
     XSetForeground( mydisplay, gc, occupcolor.pixel);
  else {
     if (bgrid[x][y] > 3.0) color = 31;
     else color = selectColor((bgrid[x][y]/3.0));
     XSetForeground( mydisplay, gc, colorArray[color] );
  }
  XFillRectangle( mydisplay, win, gc,
                  x*PIXELS_PER_CELL,
                  (2 * DEFAULT_HEIGHT) - ((y+1)*PIXELS_PER_CELL),
                  PIXELS_PER_CELL, PIXELS_PER_CELL);
  if (egrid[x][y]) {
     XSetForeground( mydisplay, gc, deadcolor.pixel);
     XFillRectangle( mydisplay, win, gc,
          x*PIXELS_PER_CELL + 1,
          (2 * DEFAULT_HEIGHT) - ((y+1)*PIXELS_PER_CELL) + 1,
          2, 2);
  }
} /** end drawGridImage() **/

/*************************************************************************/
/* Procedure: drawHighLight     Marks center of cell in red              */
/* Called by: <user>                                                     */
/* Calls:     X routines                                                 */
/*    Highlights a single cell by putting a red dot on it.               */
/*************************************************************************/
void DisplayX::drawHighLight(int x,int y)
{
  if ((x < 0) || (x>= SZE) || (y < 0) || (y >= SZE)) return;
  XSetForeground( mydisplay, gc, subdcolor.pixel);
  XFillRectangle( mydisplay, win, gc,
                  x*PIXELS_PER_CELL + 1,
                  DEFAULT_HEIGHT - ((y+1)*PIXELS_PER_CELL) + 1,
                  2, 2);
  XFillRectangle( mydisplay, win, gc,
                  x*PIXELS_PER_CELL + 1,
                  (2 * DEFAULT_HEIGHT) - ((y+1)*PIXELS_PER_CELL) + 1,
                  2, 2);
}  /** end drawHighLight() **/
```

```
/*********************************************************/
/* drawMapImage     (Author Tu-Hsin Tsai)                */
/*  Modified by ttg to display only part of the grid &   */
/*  to include the badness grid.                         */
/*                                                       */
/*  This routine draws the 41 by 41 cgrid and 9 by 9     */
/*  bgrid maps centered on the robot.                    */
/* Called by: <user>                                     */
/* Calls: certainty grid access functions, X routines    */
/*        navigator access functions.                    */
/* Globals:  bgrid,egrid,robx,roby (Read Only)           */
/*********************************************************/
void DisplayX::drawMapImage(void)
{
 int x,y;                  /* Array indices                          */
 int color;                /* Color to use for current grid square */
 int height;               /* Height of entire display               */
 const float OCCUP = getConstOccup();

 for (x=robx-20; x < robx+20; x++)
   if ((x>=0) && (x<SZE)) {
      for ( y = roby-20; y < roby+20; y++ ) {
         if ((y>=0) && (y<SZE)) {
            color = selectColor(CgPt->getCg(x,y));
            XSetForeground( mydisplay, gc, colorArray[color] );
            XFillRectangle( mydisplay, win, gc,
              x*PIXELS_PER_CELL, DEFAULT_HEIGHT- ((y+1)*PIXELS_PER_CELL),
              PIXELS_PER_CELL, PIXELS_PER_CELL);
         }
      }
   }
 height = 2 * DEFAULT_HEIGHT;
 for (x=robx-3; x <= robx+3; x++)
   for ( y = roby-3; y <= roby+3; y++ ) {
       if (CgPt->getCg(x,y) > OCCUP)
          XSetForeground( mydisplay, gc, occupcolor.pixel);
       else {
          if (bgrid[x][y] > 3.0) color = 31;
          else color = selectColor((bgrid[x][y]/3.0));
          XSetForeground( mydisplay, gc, colorArray[color] );
       }
       XFillRectangle( mydisplay, win, gc,
                    x*PIXELS_PER_CELL, height - ((y+1)*PIXELS_PER_CELL),
                       PIXELS_PER_CELL, PIXELS_PER_CELL);
       if (egrid[x][y]) {
          XSetForeground( mydisplay, gc, deadcolor.pixel);
          XFillRectangle( mydisplay, win, gc,
               x*PIXELS_PER_CELL + 1, height - ((y+1)*PIXELS_PER_CELL) + 1,
                 2, 2);
       }
   }
} /** end drawMapImage() **/
```

```
/*********************************************************/
/* drawNewMap        (Author Tu-Hsin Tsai)               */
/*  Modified by ttg to include the badness grid         */
/*                                                       */
/*  This routine draws the whole grid map               */
/* Called by: <user>                                     */
/* Calls: certainty grid access functions, X routines    */
/*        navigator access functions.                    */
/* Globals:  bgrid,egrid,robx,roby (Read Only)           */
/*********************************************************/
void DisplayX::drawNewMap(void)
{
 int x,y;                  /* Array indices                          */
 int color;                /* Color to use for current grid square */
 int height;               /* Height of entire display             */
 const float OCCUP = getConstOccup();

 height = 2 * DEFAULT_HEIGHT;
 for (x=0; x < SZE; x++ ) {
    for ( y = 0; y < SZE; y++ ) {
       color = selectColor(CgPt->getCg(x,y));
       XSetForeground( mydisplay, gc, colorArray[color] );
       XFillRectangle( mydisplay, win, gc,
                    x*PIXELS_PER_CELL, DEFAULT_HEIGHT- ((y+1)*PIXELS_PER_CELL),
                       PIXELS_PER_CELL, PIXELS_PER_CELL);
       if (CgPt->getCg(x,y) > OCCUP)
          XSetForeground( mydisplay, gc, occupcolor.pixel);
       else {
          if (bgrid[x][y] > 3.0) color = 31;
          else color = selectColor((bgrid[x][y]/3.0));
          XSetForeground( mydisplay, gc, colorArray[color] );
       }
       XFillRectangle( mydisplay, win, gc,
                    x*PIXELS_PER_CELL, height - ((y+1)*PIXELS_PER_CELL),
                       PIXELS_PER_CELL, PIXELS_PER_CELL);
       if (egrid[x][y]) {
          XSetForeground( mydisplay, gc, deadcolor.pixel);
          XFillRectangle( mydisplay, win, gc,
               x*PIXELS_PER_CELL + 1, height - ((y+1)*PIXELS_PER_CELL) + 1,
                 2, 2);
       }
    }
 }
}  /** end drawNewMap() **/
```

```
/*********************************************************************/
/* drawRobot        (Author Tu-Hsin Tsai)                            */
/*  Modified by ttg to change the robot dimensions                   */
/*                                                                   */
/*  This routine draws the robot in the current position             */
/*  Called by: <user>, drawCgrid                                     */
/*  Calls:     X routines                                            */
/*********************************************************************/
void DisplayX::drawRobot(int dx,int dy,float angle)
{
  XPoint pt[3];
  double orient;
  double dis_angle = 2.3562;
  int    x,y;

  if ((dx < 0) || (dx>= SZE) || (dy < 0) || (dy >= SZE)) return;
  orient = (double)(angle * 0.017453293);     /* angle * pi / 180.0 */
  /* redraw the robot in the new position */
  x = (dx * PIXELS_PER_CELL) + 3;
  y = DEFAULT_HEIGHT - ((dy+1) * PIXELS_PER_CELL) + 2;
  pt[0].x = x + (int)(6.0*cos(orient));
  pt[0].y = y - (int)(6.0*sin(orient));
  pt[1].x = x + (int)(5.0*cos(orient + dis_angle));
  pt[1].y = y - (int)(5.0*sin(orient + dis_angle));
  pt[2].x = x + (int)(5.0*cos(orient - dis_angle));
  pt[2].y = y - (int)(5.0*sin(orient - dis_angle));
  XSetForeground( mydisplay, gc, robotcolor.pixel );
  XFillPolygon( mydisplay, win, gc, pt, 3, Convex, CoordModeOrigin);
  pt[0].y = pt[0].y + DEFAULT_HEIGHT;
  pt[1].y = pt[1].y + DEFAULT_HEIGHT;
  pt[2].y = pt[2].y + DEFAULT_HEIGHT;
  XFillPolygon( mydisplay, win, gc, pt, 3, Convex, CoordModeOrigin);
}

/*********************************************************************/
/* Refreshx                                                          */
/*                                                                   */
/* This routine refresh the screen to show the current image         */
/*********************************************************************/
void DisplayX::Refreshx(void)
{
 XFlush(mydisplay);
}

/*********************************************************************************/
/* Procedure: setColors                                                          */
/* Called by: display_init                                                       */
/* Calls:     X routines                                                         */
/*    Sets up some colors to use in the display.  The robot is LimeGreen,        */
/*    the destination is HotPink, the subdestination is Red, the                 */
/*    occupied cells are DarkOrchid, and dead end cells are Orange.              */
/*********************************************************************************/
void DisplayX::setColors(void)
{
 XColor theRGBColor;

 XLookupColor(mydisplay,xcolormap,"LimeGreen",&theRGBColor,&robotcolor);
 XAllocColor(mydisplay,xcolormap,&robotcolor);
 XLookupColor(mydisplay,xcolormap,"HotPink",&theRGBColor,&destcolor);
 XAllocColor(mydisplay,xcolormap,&destcolor);
 XLookupColor(mydisplay,xcolormap,"red",&theRGBColor,&subdcolor);
 XAllocColor(mydisplay,xcolormap,&subdcolor);
 XLookupColor(mydisplay,xcolormap,"orange",&theRGBColor,&deadcolor);
```

```
 XAllocColor(mydisplay,xcolormap,&deadcolor);
 XLookupColor(mydisplay,xcolormap,"DarkOrchid",&theRGBColor,&occupcolor);
 XAllocColor(mydisplay,xcolormap,&occupcolor);
)
```

```
/*****************************************************************************/
/*---------------------------------------------------------------------------*/
/*- Class: Voice                                                            -*/
/*-    Controls which robot speech routine is used. (if any)                -*/
/*-    By using the Voice class as a wrapper around the Speech class the    -*/
/*-    rest of the program does not have to be concerned with whether the   -*/
/*-    robot speech, terminal simulator, or written output is used.         -*/
/*---------------------------------------------------------------------------*/
/*****************************************************************************/

/*****************************************************************************/
/* Procedure: Voice      Constructor for voice class                  Done */
/* Called by: (driver)                                                       */
/* Objects:   Speech  - robot or terminal simulator speech routine           */
/*    Instantiates the desired speech routine. Either the Speech class       */
/*    from /pro/ai/robot/bin or the one in simulat.C If the argument pline   */
/*    is non zero then neither is used and speech is simply printed on the   */
/*    indicated print line using curses.                                     */
/*****************************************************************************/
Voice::Voice(int pline, DisplayC *cdisplay)       /* curses output */
{
 p_line = pline;
 c_display = cdisplay;
}
Voice::Voice(int pline, Comm *speechLine, DisplayC *cdisplay)   /* sim & rob */
{
 p_line = pline;
 c_display = cdisplay;
 if (pline == 0) s2_pointer = new Speech2(speechLine);        /* simulator */
 if (pline == -1) s_pointer = new Speech(speechLine);         /* robot     */
}

/*****************************************************************************/
/* Procedure: ~Voice      Destructor for voice class                   Done */
/*    Frees the Speech class used.                                           */
/*****************************************************************************/
Voice::~Voice(void)
{
 if (p_line < 0) delete s_pointer;
 if (p_line < 0) delete s2_pointer;
}

/*****************************************************************************/
/* Procedure: say                                                            */
/* Calls:     Speech(2)::say - have robot or terminal say the word           */
/* I/O:       Writes to terminal using curses                                */
/*    Passes the string to be spoken to the speech routine or outputs it on  */
/*    the terminal.                                                          */
/*****************************************************************************/
void Voice::say(const char *str)
{
 char *str2;
 str2 = (char *)str;
 if (p_line > 0) {
    mvprintw(p_line+2,0,"Voice: %s\007",str);
    clrtoeol();
 } else if (p_line == 0) s2_pointer->say(str);
 else s_pointer->say(str);
 c_display->prtLne(str2);
}
```

```
/*************************************************************************/
/*-----------------------------------------------------------------------*/
/*- Speech2 class - contains routines needed for simulated robot talking -*/
/*-     Except for the change in the class name this has the same interface -*/
/*-     as the Speech class in /pro/ai/robot/   This is not portable, it -*/
/*-     is dependent on the play routine in /usr/demo/SOUND              -*/
/*-----------------------------------------------------------------------*/
/*************************************************************************/

/*************************************************************************/
/* Procedure: Speech2::Speech2                                           */
/* Argument:  (Comm *line) not used - present for compatibility with Speech  */
/*    Constructor for Speech2  class. Sets the volume level and turns the    */
/*    speaker on.  For independent control   /usr/demo/SOUND/x_gaintool&     */
/*************************************************************************/
Speech2::Speech2(Comm *)
{
 system("/cs/bin/hack/ainfo -v 50");
 system("/cs/bin/hack/ainfo -s");
}
/*************************************************************************/
/* Procedure: Speech2::~Speech2    Destructor for Speech2 class, does nothing*/
/*************************************************************************/
Speech2::~Speech2()
{}
/*************************************************************************/
/* Procedure: Speech2::say                                               */
/* Argument:  (int wait) not used  -  present for compatibility with Speech  */
/*    Uses a system call is to have the terminal make funny noises or say    */
/*    prerecorded messages.  If the specific word has not been recorded      */
/*    The files must be present for the sounds to work.                      */
/*************************************************************************/
void Speech2::say(const char *str, int)
{
char *str2;
str2 = (char *)str;
if (str[4] == 'k')   /* checkpoint */
   system("/usr/demo/SOUND/play /u/ttg/omandir/sounddir/checkpoint.au &");
else if (str[3] == 'm')   /* boom */
   system("/usr/demo/SOUND/play /usr/demo/SOUND/sounds/crash.au &");
else if (str[1] == 'l')   /* close */
   system("/usr/demo/SOUND/play /u/ttg/omandir/sounddir/close.au &");
else if (str[0] == 'r')   /* reverse */
   system("/usr/demo/SOUND/play /u/ttg/omandir/sounddir/reverse.au &");
else if (str[5] == 'y')   /* i.obey */
   system("/usr/demo/SOUND/play /u/ttg/omandir/sounddir/i.obey.au &");
else if (str[0] == 'd')   /* destination */
   system("/usr/demo/SOUND/play /u/ttg/omandir/sounddir/destination.au &");
else if (str[3] == 'k')   /* backtrack */
   system("/usr/demo/SOUND/play /u/ttg/omandir/sounddir/backtrack.au &");
else if (str[0] == 'n')   /* newstart */
   system("/usr/demo/SOUND/play /u/ttg/omandir/sounddir/newstart.au &");
else if (str[2] == 'g')   /* i.quit */
   system("/usr/demo/SOUND/play /u/ttg/omandir/sounddir/i.quit.au &");
else if (str[4] == 'l')   /* wallleft */
   system("/usr/demo/SOUND/play /u/ttg/omandir/sounddir/wallleft.au &");
else if (str[6] == 'g')   /* wall right */
   system("/usr/demo/SOUND/play /u/ttg/omandir/sounddir/wallright.au &");
else    system("/usr/demo/SOUND/play /usr/demo/SOUND/sounds/drip.au &");
}
```

```
/*****************************************************************************/
/* driver.C  04/07/92   Timothy T. Good                                      */
/*    Driver for mobile robot blank map navigator using certainty grids.     */
/*    Uses cgrid.C for certainty grid map, oman.C for navigator, display.C   */
/*    for output, destack.C for destination stack, RWITim.C for robot        */
/*    routines and the robot libraries in /pro/ai/robot .  simulat.C is used */
/*    to simulate the robot libraries for non robot testing.                 */
/*****************************************************************************/
/* #define SIMULATE  should be done by compiler if driver is for simulator   */
#include <stdio.h>
#include <iostream.h>
#include <curses.h>      /* Screen IO              */
#include <signal.h>      /* For interrupts         */
#include "cgrid.h"       /* Certainty grid module  */
#include "oman.h"        /* Navigator module       */
#include "display.h"     /* Display module         */
#include "destack.h"     /* Destination stack      */
#include "RWITim.h"      /* Robot base commands    */
#include <Sonar.h>       /* Robot sonar commands   */

/*****************************************************************************/
/* Global data                                                               */
/*****************************************************************************/

extern float dgrid[SZE][SZE];/*(oman) Directional grid                        */
extern float tgrid[SZE][SZE];/*(oman) Travel grid                            */
extern float bgrid[SZE][SZE];/*(oman) Badness grid                           */
extern short egrid[SZE][SZE];/*(oman) Dead end grid                          */
extern short abortFlg;       /*(oman) Flag for abort processing in effect    */
extern int   offx,offy;      /*(oman) Offsets of robot center from cell cent.*/
extern int   robx,roby;      /*(oman) Robot's position in grid cells         */
extern float slew;           /*(oman) Robot's turn rate in degrees (+ is ccw)*/
extern int   spdfac;         /*(oman) Robot's speed factor                   */
extern int   xdest,ydest;    /*(oman) Ultimate destination coordinates       */
extern int   xsubd,ysubd;    /*(oman) Sub destination  coordinates           */
       int   decflg = 1;     /* Decay on                                     */
       Certgrid  *CgPt;      /* Pointer to certainty grid class              */
       Destack   *DsPt;      /* Pointer to destination stack class           */
       DisplayC  *DcPt;      /* Pointer to curses display class              */
       DisplayX  *DxPt;      /* Pointer to X display class                   */
       Sonar     *SonarPt;   /* Pointer to robot sonar class                 */
       Voice     *VoicePt;   /* Pointer to robot speech classes              */
       RWITim    *RwitPt;    /* Pointer to robot base commands class         */

/*****************************************************************************/
/* Procedures                                                                */
/*****************************************************************************/
            main();
 static void  changeBkgd(int line);     /* Change the background certainty   */
 static void  menu(int line_number);    /* Print the menu                    */
       void   robotAbrt(int sig);       /* Interrupt routine                 */
 static void  test1(Comm *speechLine);  /* Case on user input                */
 static short navigate(int line, short needd);/* Set up navigator call       */
 static void  positionLuey(int line);   /* Change robot starting position    */
 static void  queryGrids(int line);     /* See grid values                   */
 static void  setDestination(int line); /* Set a new destination             */
 static void  setGridValue(int line);   /* Set certainty and confidence      */
 static void  setParms(int line);       /* Set navigator parmeters           */
 static void  setSonarValue(int line);  /* Set a value in sonar map          */
```

```
/*****************************************************************************/
/* Procedure: main                                                      Done */
/* Calls:     (cgrid)  CgPt->getBackgd - Retrieve background value           */
/*            (display)drawCgrid - Display the certainty and badness grids   */
/*            (display)prtData   - Display robot information                 */
/*            (display)prtParm   - Display navigator settings                */
/*            (oman)   omanInit  - Initialize the navigator                  */
/*            (oman)   setVolts  - Store the voltage                         */
/*            (RWITim) queryVoltage - Get robot voltage (from robot)         */
/*            test1              - Case on user input                        */
/* Objects:   Certgrid  - Certainty grid                                     */
/*            Destack   - Destination stack                                  */
/*            DisplayC  - Curses display                                     */
/*            DisplayX  - X display                                          */
/*            SerialIO  - Robot communication line                           */
/*            MpxIO     - Multiplex communication lines                      */
/*            RWITim    - Robot base                                         */
/*            Sonar     - Robot sonar                                        */
/*            Voice     - Robot speech                                       */
/* Globals:   robx, roby, RwitPt, SonarPt, CgPt, DsPt (write only)           */
/* I/O:       Initializes curses and X displays                              */
/*    Main routine.     Initializes the communication lines, navigator,      */
/*    mapper, stack and display then runs a menu procedure.                  */
/*****************************************************************************/
main()
{
 SerialIO *mxline = 0;    /* Multiplex line for robot communications */
 Comm *baseLine = 0;      /* Serial line to robot base               */
 Comm *sonarLine = 0;     /* Serial line to sonar controller         */
 Comm *speechLine = 0;    /* Serial line to speech controller        */
 /***** Set up communication lines to the robot *****/
    mxline = new SerialIO ("/dev/ttyb",SerialIO::baud9600);
    baseLine = new MpxIO (*mxline,"RWI");
    sonarLine = new MpxIO (*mxline,"SONAR");
    speechLine = new MpxIO (*mxline, "SPEECH");
 RwitPt = new RWITim(baseLine);                /* Initialize robot base     */
 SonarPt = new Sonar(sonarLine,8);             /* Initialize sonars         */
 CgPt = new Certgrid(SonarPt,RwitPt,CENX,CENY); /* Initialize certainty grid */
 DsPt = new Destack();                        /* Initialize destination stack */
 DcPt = new DisplayC();                       /* Initialize curses display */
 DxPt = new DisplayX();                       /* Initialize X display      */
 VoicePt = new Voice(15,speechLine,DcPt);     /* Initialize speech (off)   */
 robx = CENX; roby = CENY;
 setVolts( RwitPt->queryVoltage());
 omanInit(CENX,CENY,TRUE);  /* Initialize navigator         */
 DxPt->drawCgrid(1);        /* Display cgrid in X window    */
 DcPt->prtParm();           /* Display navigator parameters */
 DcPt->prtData();           /* Display current robot data   */
 test1(speechLine);         /* Start driver menu routine    */
 delete DcPt;
 delete DxPt;
 delete SonarPt;
 delete VoicePt;
 delete RwitPt;
 delete CgPt;
 delete DsPt;
 delete speechLine;
 delete sonarLine;
 delete baseLine;
 delete mxline;
}
```

```
/*****************************************************************************/
/* Procedure: robotAbrt                                                      */
/* Called by: user interrupt  ^C                                             */
/* Calls:     update  - determine robot location                             */
/*            newCell - decay grid, check for recentering                    */
/* Globals:   abortFlg, spdfac, slew (Write only)                            */
/*    Interrupt Handler   ^C will halt robot and set the abort flag so       */
/*    control will return to the menu following the current travel cycle.    */
/*    Menu option 'continue' will allow the robot to continue.  *Caution:    */
/*    this may have unpredictable results and cause a program abort.  Also   */
/*    no recovery is possible once robot communication has been lost.        */
/*****************************************************************************/
void robotAbrt(int sig)
{
  RwitPt->kill();
  if (update(0.0,-1.0) == TRUE) newCell();
  spdfac = 0;  slew = 0.0;
  VoicePt->say("abort");
  DcPt->prtLne("***** Robot halted by ctl C. *****");
  abortFlg = TRUE;
  if (sig);   /* dummy statement to get rid of the sig not used warning */
}

/*****************************************************************************/
/* Procedure: menu                                                    Done */
/* Called by: test1                                                          */
/*    Displays the user menu using curses.                                   */
/*****************************************************************************/
void menu(int lno)              /* lno is line number of first line of menu */
{
   mvaddstr(lno,0,"What do you want to do?");
   mvaddstr(lno+1,0,
    " 0: Halt Program    1: Set Parameters  2: Find Background 3: Read Sonars");
   addstr(
 "\n 4: Circle Scan     5: Rotate Luey     6: Translate Luey  7: Set Cgrid");
   addstr(
 "\n 8: Read Cgrid      9: Read Sonar Map 10: Query Grids    11: Set Destination ");
   addstr(
 "\n12: Travel         13: Continue       14: Reset          15: Update");
   addstr(
 "\n16: Save Cgrid     17: Position Luey  18: Test Speech    19: Set Speech");
#ifdef SIMULATE
   addstr(
 "\n20: Set Sonar Map");
#endif
}
```

```
/*****************************************************************************/
/* Procedure: test1                                                    Done */
/* Called by: main                                                           */
/* Calls:     <just about everything>                                        */
/*            (cgrid)   cgridInit     - initialize the certainty grid        */
/*            (cgrid)   cgridRead     - read certainty values from ext. file */
/*            (cgrid)   circleScan    - read the sonars while rotating       */
/*            (cgrid)   getSonarType  - retrieve the sonar arrangement       */
/*            (cgrid)   saveMap       - save certainty values in ext. file   */
/*            (cgrid)   setCgrid      - set a single certainty value         */
/*            (cgrid)   sonarScan     - read the sonars                      */
/*            (display) drawCgrid     - draw and display grids using X       */
/*            (display) prtData       - display robot data using curses      */
/*            (oman)    getAngle      - get robot angle (from robot)         */
/*            (oman)    getBaseAngle  - get robot angle (from navigator)     */
/*            (oman)    mkbgrd        - recalculate the badness potential    */
/*            (oman)    newCell       - do new cell entered activities,      */
/*            (oman)    omanInit      - initialize the navigator             */
/*            (oman)    setBaseAngle  - Store the robot's baseangle          */
/*            (oman)    setVolts      - Store the robot voltage              */
/*            (oman)    update        - determine robot location             */
/*            (RWITim)executeTurn     - turn robot                           */
/*            (RWITim)executeMove     - move robot                           */
/*            (RWITim)queryVoltage    - get robot voltage                    */
/*            (simulat)readSonarMap   - read sonar map from external file    */
/*            (simulat)sonarReset     - reset simulator angle                */
/*            changeBkgd              - allow user to change background value*/
/*            navigate                - handle call to main navigator routine*/
/*            positionLuey            - allow user to teleport robot in grid */
/*            queryGrids              - allow user to look at grid values    */
/*            setDestination          - allow user to input a destination    */
/*            setGridValue            - allow user to set certainty values   */
/*            setParms                - change sonar arrangment, decay on/off*/
/*            setSonarValue           - allow user to set values in sonar map*/
/* Globals:   everything (Read, Write)                                       */
/* I/O:       reads and writes to terminal using curses and X                */
/*    Displays a menu and cases on user input to make the robot do its tricks*/
/*****************************************************************************/
void test1(Comm *speechLine)
{
 int   ans;                     /* User input, case variable                 */
 char  cvar[15];                /* String variable for user responses        */
 int   dx;                      /* Array coordinates                         */
 float fvar;                    /* Float variable for user responses         */
 int   ivar;                    /* Integer variable for user responses       */
 int   line = 15;               /* Keeps track of current screen output line */
 short needd = TRUE;            /* Flag for needs a destination              */
 signal(SIGINT,&robotAbrt);     /* Initialize ^C interrupt                   */
 menu(7);
 DcPt->prtData();
 ans = -1;
 while (ans != 0) {
    ans = -1;
    while (ans < 0) {
       move(7,25);                      /* put cursor after menu */
       clrtoeol();
       refresh();
       scanw("%d",&ans);
    }
    move(line,0);   clrtoeol();         /* Clear display area */
    move(line+1,0); clrtoeol();
    move(line+2,0); clrtoeol();
    move(line+3,0); clrtoeol();
```

```
switch(ans) {

case 0:
  ans = -1;
  mvprintw(line,0,"Enter 0 again if you really meant to quit :");
  refresh();
  scanw("%d",&ans);
  if (ans == 0) DcPt->prtLne("Program terminated");
 break;
case 1:  /****************************************** set parameters *****/
   setParms(line);
  break;
case 2:  /********************************** Find background value *****/
   changeBkgd(line);
 break;
case 3: /************************************************ Read sonars *****/
   if (CgPt->getSonarType() == S_OFF) {
      mvprintw(line,0,"\007Sonars are turned off");
   } else {
      mvprintw(line,0,"Reading sonars");
      refresh();
      CgPt->sonarScan(getBaseAngle(),robx,roby,offx,offy);
      mkbgrd();
      DxPt->drawCgrid(1);
      DcPt->prtData();
   }
 break;
case 4:  /************************************************** cscan *****/
   if (CgPt->getSonarType() == S_OFF)
      mvprintw(line,0,"\007cscan: Sonars are turned off");
   else {
      dx = -1;
      while (dx == -1) {
         mvprintw(line,0,"cscan: Enter increment (deg) & number :");
         refresh();
         scanw("%4d %4d",&ivar,&dx);
      }
      if (dx <= 0)
         mvprintw(line+1,0,"\007Number must be greater than 0");
      else {
         CgPt->circleScan(getBaseAngle(),ivar,dx,robx,roby,offx,offy,1);
         mkbgrd();
         DxPt->drawCgrid(1);
      }
   }
 break;
case 5:  /************************************************** Rotate *****/
   fvar = -361.0;
   while (fvar == -361.0) {
      mvprintw(line,0,"rotate:  Enter angle (+=ccw) in (float) degrees :");
      clrtoeol();
      refresh();
      scanw("%f",&fvar);
   }
   while (fvar >= 360.0) { fvar = fvar - 360.0; }
   while (fvar <= -360.0) { fvar = fvar + 360.0; }
   if (fvar != 0.0) {
      RwitPt->executeTurn(-fvar,10.0,1);
      setBaseAngle(getAngle());
      DxPt->drawCgrid(0);
      DcPt->prtData();
   }
 break;
```

```
    case 6:  /********************************************** Translate *****/
       fvar = -72.0;
       while (fvar == -72.0) {
         mvprintw(line,0,"translate: Enter distance in (float) centimeters :");
          clrtoeol();
          refresh();
          scanw("%f",&fvar);
       }
       RwitPt->executeMove(fvar,5.0,1);
       if (update(0.0,-1.0) == TRUE) newCell();
       DxPt->drawCgrid(0);
       DcPt->prtData();
     break;
    case 7:  /********************************************** Set cgrid *****/
       setGridValue(line);
     break;
    case 8:  /********************************************** Read cgrid *****/
       mvprintw(line,0,"Read Cgrid File: (Return to exit, 0 to clear map)");
       mvprintw(line+1,0,"Sample file is ailab.map, Enter file name - ");
       refresh();
       cvar[0] = '\0';
       scanw("%s",cvar);
       while (cvar[0] != '\0') {
          if (CgPt->cgridRead(cvar) == FALSE) {
             mvprintw(line+1,0,
              "CGRIDREAD: Bad file name '%s'     \007Map unchanged. ",cvar);
             clrtoeol();
          } else {
             if (cvar[0] == '0') mvprintw(line+1,0,"CGRIDREAD: Map cleared");
             else mvprintw(line+1,0,"CGRIDREAD: Map '%s' loaded",cvar);
             clrtoeol();
             CgPt->setCgrid(robx,roby,0.0,-1.0);  /* Robot cell is empty  */
             mkbgrd();
             DxPt->drawCgrid(1);
          }
          mvprintw(line+2,0,"Sample file is ailab.map, Enter file name - ");
          clrtoeol();
          refresh();
          cvar[0] = '\0';       /* set loop exit condition */
          scanw("%s",cvar);
       }
     break;
    case 9: /********************************************** Read Sonar File *****/
#ifdef SIMULATE
       mvprintw(line,0,"Read Sonar Map: (Return to exit, 0 to clear map)");
       mvprintw(line+1,0,"Sample file is ailab.map. Enter file name - ");
       refresh();
       cvar[0] = '\0';
       scanw("%s",cvar);
       while (cvar[0] != '\0') {
          if (readSonarMap(cvar) == FALSE) {
             mvprintw(line+1,0,"RDSGRD: Bad file name '%s'   \007Map unchanged"
             ,cvar);
             clrtoeol();
          } else {
             if (cvar[0] == '0') mvprintw(line+1,0,"RDSGRD: Map cleared");
             else mvprintw(line+1,0,"RDSGRD: Map '%s' loaded",cvar);
             clrtoeol();
          }
          mvprintw(line+2,0,"Sample file is ailab.map, Enter file name - ");
          clrtoeol();
          refresh();
          cvar[0] = '\0';           /* set loop exit condition */
          scanw("%s",cvar);
       }
#else
       mvprintw(line,0,"This option only available with simulator");
#endif
     break;
    case 10:  /********************************************** Query *****/
       queryGrids(line);
     break;
    case 11: /********************************************** Set Destination *****/
       mvprintw(line,0,"Set Destination");
       setDestination(line);
       DcPt->prtData();
       DxPt->drawCgrid(1);
       needd = FALSE;                          /* turn off need dest flag */
     break;
    case 12: /********************************************** Travel *****/
       if (abortFlg)  mvprintw(line,0,"Abort in effect. \007 Use option 13");
       else  needd = navigate(line,needd);
     break;
    case 13:  /********************************************** Continue *****/
       if (!abortFlg)  mvprintw(line,0,"No abort in effect. \007 Use option 12");
       else {
          abortFlg = FALSE;
          needd = navigate(line,FALSE);
       }
     break;
    case 14:  /********************************************** Reset *****/
       mvprintw(line,0,"Resetting to original state");
       DcPt->prtLne("DRIVER: reset");
#ifdef SIMULATE
       sonarReset(0.0);
#endif
       needd = TRUE;
       abortFlg = FALSE;
       robx = CENX; roby = CENY;
       CgPt->cgridInit(CENX,CENY,0.0,TRUE);
       omanInit(CENX,CENY,TRUE);
       DxPt->drawCgrid(1);
       DcPt->prtData();
     break;
    case 15: /********************************************** Update *****/
       mvprintw(line,0,"Update ");
       if  (update(0.0,-1.0) == TRUE) newCell();
       setVolts( RwitPt->queryVoltage());
       DcPt->prtData();
       DxPt->drawCgrid(1);
      break;
    case 16:  /********************************************** Save grid *****/
       mvprintw(line,0,"Save Cgrid in file: Enter 0 to return and do nothing");
       mvprintw(line+1,0,"Enter file name - ");
       refresh();
       scanw("%s",cvar);
       if (cvar[0] != '0') {
          if (CgPt->saveMap(cvar) == FALSE)
             mvprintw(line+2,0,"\007SAVEMAP: File could not be opened");
       }
     break;
    case 17:  /********************************************** Position Luey */
       positionLuey(line);
     break;
    case 18: /********************************************** Test Speech */
       mvprintw(line,0,"Test Speech,  Enter string (14 char max) - ");
       refresh();
       scanw("%s",cvar);
```

```
      VoicePt->say(cvar);
     break;
    case 19: /********************************************** Set Speech  ***/
      mvprintw(line,0,"Enter type of speech desired. ");
#ifdef SIMULATE
      addstr("0 - curses, 1 - simulator :");
      refresh();
      scanw("%d",&ivar);
      if ((ivar == 0) || (ivar == 1)) {
         delete VoicePt;
         if (ivar == 0) VoicePt = new Voice(line,DcPt);
         if (ivar == 1) VoicePt = new Voice(0,speechLine,DcPt);
      }
#else
      addstr("0 - curses, 1 - simulator, 2 - robot :");
      refresh();
      scanw("%d",&ivar);
      if ((ivar == 0) || (ivar == 1) || (ivar == 2)) {
         delete VoicePt;
         if (ivar == 0) VoicePt = new Voice(line,DcPt);
         if (ivar == 1) VoicePt = new Voice(0,speechLine,DcPt);
         if (ivar == 2) VoicePt = new Voice(-1,speechLine,DcPt);
      }
#endif
    break;
#ifdef SIMULATE
   case 20: /********************************************** Set sgrid *****/
      setSonarValue(line);
    break;
#endif
  default:  /********************************************** Default *****/
      mvprintw(line,0,"Enter number of your choice");
      DcPt->prtScreen();
      break;
   }
  }
}
```

```
/*****************************************************************************/
/* Procedure: changeBkgd                                                Done */
/* Called by: test1                                                          */
/* Calls:     (cgrid)getBackgd - Retrieve background value                   */
/*            (cgrid)findBg    - Determine and/or set background value       */
/* I/O:       Reads and writes to terminal using curses, X                   */
/*    Sets the background to either a user defined value or one calculated   */
/*    in findBg.                                                             */
/*****************************************************************************/
void changeBkgd(int line)
{
 float oldbkgd;                    /* current background */
 float newbkgd;                    /* new background     */
 oldbkgd = CgPt->getBackgd();
 mvprintw(line,0,"Current background is %5.3f",oldbkgd);
 mvprintw(line+1,0,"Enter 1.0 to return, -1.0 to have Luey calculate the new");
 mvprintw(line+2,0," background, or the background value you wish.  : ");
 refresh();
 newbkgd = -2.0;                          /* default to a bad value */
 scanw("%f",&newbkgd);
 if (newbkgd != 1.0) {
    if (((newbkgd < 0.0) || (newbkgd > 1.0)) && (newbkgd != -1.0))
       mvprintw(line+3,0,"\007Background must be between 0.0 and 1.0");

    else {
       if (newbkgd == -1.0) {
          mvprintw(line+1,0,"Calculating background value ");
          clrtoeol();
          move(line+2,0); clrtoeol(); refresh();
          newbkgd = CgPt->findBg(getBaseAngle(),-1.0,robx,roby,offx,offy);
       } else {
          mvprintw(line+1,0,"Storing new background value ");
          clrtoeol();
          move(line+2,0); clrtoeol(); refresh();
          newbkgd = CgPt->findBg(getBaseAngle(),newbkgd,robx,roby,offx,offy);
       }
       mvprintw(line+1,29,"- New value is : %5.3f",newbkgd);
        if (newbkgd >= THO)
          mvprintw(line+2,0,"Warning: Background above occupied level.");
       if (newbkgd <= THE)
          mvprintw(line+2,0,"Warning: Background below empty level.");
       DcPt->prtParm();
       mkbgrd();
       DxPt->drawCgrid(1);
    }
 }
) /** end changeBkgd() **/
```

```
/*****************************************************************************/
/* Procedure  navigate                                                 Done */
/* Called by: test1                                                          */
/* Calls:     (cgrid)  getSonarType - find sonar arrangement                 */
/*            (display)drawCgrid    - draw certainty and badness grids       */
/*            (oman)   travel       - navigate to destination                */
/*            RwitPt->queryVoltage - get robot battery voltage               */
/* Globals:   xdest,ydest,robx,roby (read only)                              */
/* I/O:       Reads and writes to terminal using curses, X                   */
/*    After making sure that a destination exists, calls the main navigator */
/*    routine. Prints appropriate messages based on return code.             */
/*****************************************************************************/
short navigate(int line, short needd)
{
 short retcode;              /* return code from travel procedure */
 if (needd) {
    mvprintw(line,0,"\007Destination is needed.");
    setDestination(line);
    move(line+1,0); clrtoeol();
    move(line+2,0); clrtoeol();
    DcPt->prtData();
    DxPt->drawCgrid(1);
 )
 if (CgPt->getSonarType() == S_OFF)
    mvprintw(line+1,0,"\007Warning: Sonars are off.");
 if (RwitPt->queryVoltage() <= 11.8)
    mvprintw(line+2,0,"\007Warning: Battery voltage is low.");
 if ((xdest!=robx) || (ydest!=roby)) {
    mvprintw(line,0,"Luey is on his own. Use <control c> to interrupt");
    refresh();
    retcode = travel();
    if (retcode == OK) mvprintw(line,0,"Luey has reached destination");
    if (retcode == ERR) {
       needd = FALSE;
       mvprintw(line,0,"User abort, Option 13 to continue");
    }
    if (retcode == 2)
       mvprintw(line,0,"Luey is stuck. No path to destination exits");
 ) else mvprintw(line,0,"Already at destination\007.");
 clrtoeol();
 if (getVolts() <= 11.8)
    mvprintw(line+2,0,"Warning\007: Battery voltage is low.");
 return(needd);
) /** end navigate() **/
```

```
/*****************************************************************************/
/* Procedure: positionLuey                                                   */
/* Called by: test1                                                          */
/* Calls:     (cgrid)cgridInit   - Reinitialize the certainty grid           */
/*            (destack)changeDS  - Put new location into destination stack   */
/*            (display)drawCgrid       - Draw cert., badness grids using X   */
/*            (display)drawGridImage   - Draw a single cell                  */
/*            (display)drawRobot       - Draw the robot                      */
/*            (display)prtData         - Display robot information using curs*/
/*            (display)Refreshx        - Display the X window                */
/*            checkSides      - Reinitialize side obstacle check            */
/*            omanInit        - Reinitialize the navigator                  */
/* Globals:   robx,roby (Read Write), xsubd,ysubd (Write only)               */
/* I/O:       reads and writes to the terminal using curses                  */
/*    Allows the user to place the robot in a different location in the grid.*/
/*    The base angle and distance are reset and a limited reinitialization   */
/*    is done.  The certainty grid and destination are not changed.          */
/*****************************************************************************/
void positionLuey(int line)
{
 int   dx,dy;                  /* new position for Luey */
 char  c = 'n';                /* hold user response    */
 int   oldx,oldy;              /* old position of Luey  */
 oldx = robx; oldy = roby;
 while (c != 'y') {
    dy = -72;
    while (dy == -72) {
       mvprintw(line,0,
       "Reposition: Luey is at (%3d,%3d).Enter x and y coords :",oldx,oldy);
       clrtoeol();
       refresh();
       scanw("%3d %3d",&dx,&dy);
    )
    if ((dx < 0) || (dy < 0) || (dx >= SZE) || (dy >= SZE)) {
       mvprintw(line+1,0,
          "\007Grid indices out of bounds. Position unchanged.      ");
       clrtoeol();
    } else {
       DxPt->drawRobot(dx,dy,0.0);
       mvprintw(line+1,0,"Moved to (%3d,%3d). Is this ok (y/n) :",dx,dy);
       clrtoeol();
       refresh();
       DxPt->Refreshx();
       c = getch();
       while (getch() != '\n') ;
       if  (c != 'y') DxPt->drawGridImage(dx,dy);
    )
 )
 robx = dx; roby = dy;
 DsPt->changeDS(robx,roby);            /* put new location into dest. stack */
 xsubd = robx; ysubd = roby;
#ifdef SIMULATE
 sonarReset(0.0);                      /* reset simulator angle             */
#endif
 omanInit(dx,dy,FALSE);                /* omanInit before cgridInit         */
 CgPt->cgridInit(dx,dy,getBaseDist(),FALSE);/* to set new base_dis          */
 checkSides(getBaseAngle(),0.0);       /* Reinitialize side obstacle check  */
 DxPt->drawCgrid(1);
 DcPt->prtData();
) /** end positionLuey() **/
```

```
/*****************************************************************************/
/* Procedure: queryGrids                                              Done */
/* Called by: test1                                                        */
/* Calls:     (cgrid)getCg, getConfi, querySonarMap      ,                 */
/*            (display)drawGridImage, drawHighLight, Refreshx              */
/* Globals:   bgrid, dgrid, tgrid,egrid (Read only)                        */
/* I/O:       reads and writes to the terminal using curses , X            */
/*    Allows the user to query for the certainty, confidence, badness,     */
/*    travel and deadend values.  If SIMULATE is defined also includes the */
/*    sonar map.                                                           */
/*****************************************************************************/
void queryGrids(int line)
{
 int   dx = 1,dy = 1;       /* cell to query     */
 int   px = 0, py = 0;       /* last cell queried */
       while ((dx != 0) && (dy != 0)) {
          dy = -72;
          while (dy == -72) {
             mvprintw(line,0,"query: (0 0 to exit). Enter x and y coords :");
             clrtoeol();
             refresh();
             scanw("%3d %3d",&dx,&dy);
          )
          if ((dx < 0) || (dy < 0) || (dx >= SZE) || (dy >= SZE)) {
             mvprintw(line+1,0,"\007Grid indices out of bounds. ");
             printw("Must be from 0 to %3d.",SZE-1);
             clrtoeol();
          ) else {
             mvprintw(line+1,0,"Grid values for [%3d][%3d]",dx,dy);
             clrtoeol();
             mvprintw(line+2,0,"Certainty:%7.4f  Confidence:%7.4f",
                 CgPt->getCg(dx,dy),CgPt->getConfi(dx,dy));
             printw("  Direction:%7.4f  Travel:   %7.4f",
                 dgrid[dx][dy],tgrid[dx][dy]);
             mvprintw(line+3,0,"Badness:  %7.4f  Deadend:    %1d       ",
                 bgrid[dx][dy],egrid[dx][dy]);
#ifdef SIMULATE
             printw("Sonar:      %1d",querySonarMap(dx,dy));
#endif
             DxPt->drawGridImage(px,py);
             DxPt->drawHighLight(dx,dy);
             DxPt->Refreshx();
             px = dx;  py = dy;
          )
       )
       DxPt->drawGridImage(px,py);
       DxPt->Refreshx();
) /** end queryGrids() **/
```

```
/*****************************************************************************/
/* Procedure: setDestination                                          Done */
/* Called by: test1                                                        */
/* Calls:     (display)drawDest       - show destination on X output       */
/*            (display)drawGridImage - draw a single cell                  */
/*            (display)RefreshX      - refresh the X display               */
/*            getDest  - tell navigator what it is                         */
/* I/O:       reads and writes to terminal using curses, X                 */
/*    Allows user to input a destination for the robot. Passes it to       */
/*    getDest to initialize the destination stack.                         */
/*****************************************************************************/
void setDestination(int line)
{
 int xd,yd;                  /* X & Y coordinates of robots destination     */
 char c = 'n';               /* holds character input from terminal         */
 while (c != 'y') {
    mvprintw(line+1,0,"Input destination in grid coordinates ? ");
    clrtoeol();
    refresh();
    scanw("%d %d",&xd,&yd);
    if ((xd >= 0) && (xd < SZE) && (yd >= 0) && (yd < SZE)) {
       DxPt->drawDest(xd,yd,-1,-1);    /* Draw in X display, no subdestn  */
       DxPt->Refreshx();
    )
    mvprintw(line+2,0,"New destination is (%4d,%4d). Is this ok? ",xd,yd);
    clrtoeol();
    refresh();
    c = getch();
    while (getchar() != '\n');
    if (c != 'y') {
       if ((xd >= 0) && (xd < SZE) && (yd >= 0) && (yd < SZE))
          DxPt->drawGridImage(xd,yd);
    )
 )
 getDest(xd,yd);
) /** end setDestination() **/
```

```
/*****************************************************************************/
/* Procedure: setGridValue                                            Done */
/* Called by: test1                                                        */
/* Calls:     (cgrid)setGrid, getCg                                        */
/*            (display)drawCgrid     -  draw cert. and badness grids in X  */
/*            (display)drawGridImage -  draw a single cell                 */
/*            (display)drawHighLight  - mark a single cell with red dot    */
/* Globals:   bgrid (Write only), tgrid, dgrid (Read only)                 */
/* I/O:       reads and writes to the terminal using curses, X             */
/*    Allows the user to set certainty and confidence values.              */
/*****************************************************************************/
void setGridValue(int line)
{
 int   dx,dy;             /* Cell indices     */
 int   px= -1, py = -1;   /* Previous cell    */
 float cert;              /* Certainty value  */
 float conf;              /* Confidence value */
      mvprintw(line,0,"Set: (0 0 to exit). ");
      addstr("Enter X, Y, (flt)Certainty, (flt)Confidence : ");
      refresh();
      conf = 7.0;                                /* default is bad value */
      scanw("%d %d %f %f",&dx,&dy,&cert,&conf);
      while ((dx != 0) || (dy != 0)) {
         if ((dx < 0) || (dy < 0) || (dx >= SZE) || (dy >= SZE)) {
            mvprintw(line+1,0,"\007Grid indices out of bounds.  ");
            printw("Must be from 0 to %3d.",SZE-1);
         } else if ((cert< 0.0)||(cert> 1.0) || (conf < -6.0)||(conf > 6.0)) {
            mvprintw(line+1,0,"\007Bad certainty or confidence value:  ");
            addstr("Ranges are (0.0 - 1.0) and (-6.0 - 6.0).");
         } else {
            CgPt->setGrid(dx,dy,cert,conf);
            bgrid[dx][dy] = CgPt->getCg(dx,dy)+ tgrid[dx][dy] + dgrid[dx][dy];
            mvprintw(line+1,0,"cgrid[%3d][%3d] is %6.3f",dx,dy,
             CgPt->getCg(dx,dy));
            printw("  confi[%3d][%3d] is %6.3f",dx,dy,CgPt->getConfi(dx,dy));
            DxPt->drawGridImage(px,py);
            DxPt->drawGridImage(dx,dy);
            DxPt->drawHighLight(dx,dy);
            px = dx; py = dy;
            DxPt->Refreshx();
         }
         clrtoeol();
         move(line,66); clrtoeol();
         refresh();
         conf = 7.0;                             /* default is bad value */
         scanw("%d %d %f %f",&dx,&dy,&cert,&conf);
      }
      DxPt->drawGridImage(px,py);
      DxPt->Refreshx();
```

```
/*****************************************************************************/
/* Procedure: setParms                                                Done */
/* Called by: test1                                                        */
/* Calls:     (cgrid)getBackgd                                             */
/*            (cgrid)getSonarType,   (cgrid)setSonarType                   */
/*            getConstEmpty,         setConstEmpty                         */
/*            getConstOccup,         setConstOccup                         */
/*            getConstPath,          setConstPath                          */
/*            getNavigator,          setNavigator                          */
/*            getDecayFlag,          setDecayFlag                          */
/* Globals:   decflg (Read, Write)                                         */
/* I/O:       Reads and writes to the terminal using curses                */
/*    Allows the user to set the navigator parameters. These are the sonar */
/*    arrangement, navigator technique, grid decay, empty threshold, occupied*/
/*    threshold, and path threshold. The order of presentation is important */
/*    because changing earlier parameters may reset later ones.            */
/*****************************************************************************/
void setParms(int line)
{
 int   ivar;              /* hold user integer input */
 float fvar;              /* hold user float input   */
      mvprintw(line+1,0,"Enter the following parameters ");
      addstr(" Hit return to leave a value unchanged.");
      mvprintw(line+2,0,"Sonars(3):  0-off, 1-square, 2-test, 3-front ? ");
      refresh();
      ivar = -1;
      scanw("%1d",&ivar);
      if (ivar != -1) {
         if ((ivar<0) || (ivar>3))
            mvprintw(line+3,0,"\007Bad sonar option");
         else {
            CgPt->setSonarType((sonarType)ivar);
            if (((sonarType)ivar == S_OFF) || ((sonarType)ivar == S_TEST))
               setDecayFlag(FALSE);
         }
      }
      DcPt->prtParm();
      mvprintw(line+2,0,
        "Navigator type(1): 0-Local, 1-Backtrack, 2-Path, 3-Incremental? ");
      clrtoeol();
      refresh();
      ivar = -1;
      scanw("%1d",&ivar);
      if (ivar != -1) {
         if ((ivar<0) || (ivar>3))
            mvprintw(line+3,0,"Bad navigator\007 option.");
         else setNavigator((NavMeth)ivar);
         DcPt->prtParm();
      }
      mvprintw(line+2,0,"Certainty grid decay(1):  0-off, 1-on ? ");
      clrtoeol();
      refresh();
      ivar = -1;
      scanw("%1d",&ivar);
      if (ivar != -1) {
         if ((ivar<0) || (ivar>1))
            mvprintw(line+3,0,"Bad\007 decay option");
         else {
            setDecayFlag(ivar);
            setConstPath(getConstPath() + 0.1);
         }
      }
      DcPt->prtParm();
```

```
      mvprintw(line+2,0,"Empty threshold: ? ");
      clrtoeol();
      refresh();
      fvar = -1.0;
      scanw("%f",&fvar);
      if (fvar != -1.0) {
         if ((fvar < 0.0) || (fvar > 0.5))
            mvprintw(line+3,0,"Bad empty threshhold\007. Range 0.0 - 0.5 ");
         else setConstEmpty(fvar);
      }
      DcPt->prtParm();
      mvprintw(line+2,0,"Occupied threshold: ? ");
      clrtoeol();
      refresh();
      fvar = -1.0;
      scanw("%f",&fvar);
      if (fvar != -1.0) {
         if ((fvar < 0.5) || (fvar > 1.0))
            mvprintw(line+3,0,"Bad occupied threshhold\007. Range 0.5 - 1.0 ");
         else setConstOccup(fvar);
      }
      DcPt->prtParm();
      mvprintw(line+2,0,"Path threshold: ? ");
      clrtoeol();
      refresh();
      fvar = -1.0;
      scanw("%f",&fvar);
      if (fvar != -1.0) {
         if ((fvar < 0.0) || (fvar > 1.0))
            mvprintw(line+3,0,"Bad path threshhold\007. Range 0.0 - 1.0 ");
         else setConstPath(fvar);
      }
      DcPt->prtParm();
      refresh();
}
```

```
#ifdef SIMULATE
/*****************************************************************************/
/* Procedure: setSonarValue                                                  */
/* Called by: test1                                                          */
/* Calls:     setSonarMap     - Set a value in the sonar map                 */
/*            querySonarMap   - Get a value from the sonar map               */
/* I/O:       reads and writes to the terminal using curses                  */
/*    Allows the user to set individual cells in the sonar map.  This        */
/*    is only available during simulator mode.                               */
/*****************************************************************************/
void setSonarValue(int line)
{
 int dx,dy;            /* Cell indices  */
 int px = -1, py = -1; /* previous cell */
 int ivar;             /* Sonar value   */
       mvprintw(line,0,"Set sonar: (0 0 to exit). ");
       addstr("Enter X, Y, sonar value (0 or 1): ");
       refresh();
       ivar= 2;                                  /* default to bad value */
       scanw("%d %d %1d",&dx,&dy,&ivar);
       while ((dx != 0) || (dy != 0)) {
          if ((dx < 0) || (dy < 0) || (dx >= SZE) || (dy >= SZE)) {
             mvprintw(line+1,0,"\007Grid indices out of bounds.  ");
             printw("Must be from 0 to %3d.",SZE-1);
          } else if ((ivar == 0) || (ivar == 1)) {
             setSonarMap(dx,dy,(short)ivar);
             mvprintw(line+1,0,"sgrid[%3d][%3d] is %1d",dx,dy,
                (int)querySonarMap(dx,dy));
             DxPt->drawGridImage(px,py);
             DxPt->drawHighLight(dx,dy);
             px = dx; py = dy;
             DxPt->Refreshx();
          } else mvprintw(line+1,0,"\007Bad sonar value, must be 0 or 1 ");
          clrtoeol();
          move(line,60); clrtoeol();
          refresh();
          ivar= 2;                                 /* default to bad value */
          scanw("%d %d %1d",&dx,&dy,&ivar);
       }
       DxPt->drawHighLight(dx,dy);
       DxPt->Refreshx();
}
#endif /* SIMULATE */
```

```
/*****************************************************************************/
/*   destack.h   03/27/92  Timothy T. Good                                   */
/*      Contains declarations for a Destack class to handle the destinations */
/*      and checkpoints used by the navigator. The robot's current           */
/*      sub destination is kept on the top of the stack, previous sub        */
/*      destinations are kept in the stack as checkpoints or removed after   */
/*      they are determined to be dead ends.                                 */
/*****************************************************************************/

static enum destType
   ( D_START, D_SUBDEST, D_REVERSE, D_BACKTRACK, D_QUIT, D_ERROR );

class Destack {

 public:

            Destack(void);              /* Constuctor, sets stack to NULL     */
            ~Destack(void);             /* Destructor, erases stack           */
    int     getDx(void);                /* Retrieve destination point X value */
    int     getDy(void);                /* Retrieve destination point Y value */
    int     getSx(void);                /* Retrieve source point X value      */
    int     getSy(void);                /* Retrieve source point Y value      */
    destType getDt(void);               /* Retrieve destination type          */
    void    pushDS(int dx,int dy,int sx,int sy,destType dtype); /*add node  */
    short   popDS(void);                /* Remove top node                    */
    void    newStart(int,int,destType);/* Add a new node to bottom of stack */
    void    changeDt(destType dtype); /* Change dest. type of top node      */
    void    changeDS(int dx, int dy); /* Change destination pt. of top node */
    void    shiftDS(int dx, int dy);  /* Shift all destinations and sources */
    void    clearDS(void);            /* Erase the stack                    */
    short   queryStart(void);         /* Returns TRUE if only one node left */

 private:

    typedef struct Dnode {          /* for linked lists of destination cells */
      int sx;                       /* source X value                        */
      int sy;                       /* source Y value                        */
      int dx;                       /* destination X value                   */
      int dy;                       /* destination Y value                   */
      destType dtype;               /* destination type                      */
      struct Dnode *npt;            /* pointer to next list element          */
    };
    struct Dnode *dest_stack;       /* destination stack                     */
};
```

```
/*****************************************************************************/
/*  destack.C  03/27/92  Timothy T. Good                                     */
/*     This file contains the procedures for creating and handling the       */
/*     destination stack used by the navigator in oman.C . It is implemented */
/*     as a class. The declarations are in destack.h .                       */
/*****************************************************************************/

#include "destack.h"
#define OK  1
#define ERR 0
const int   NULL  = 0;
const short TRUE  = 1;
const short FALSE = 0;
const int   GET_ERR = -1;

/*****************************************************************************/
/* Procedure: Destack                                                        */
/*    Constructor for Destack. Initializes the stack to NULL.                */
/*****************************************************************************/
Destack::Destack()
{
 dest_stack = NULL;
}
/*****************************************************************************/
/* Procedure: ~Destack      Destructor for Destack. Clears the stack.        */
/*****************************************************************************/
Destack::~Destack()
{
 clearDS();
}
/*****************************************************************************/
/* Procedure: getDx                                                          */
/*    Returns the X coordinate of the destination in the top Dnode of the    */
/*    destination stack.                                                     */
/*****************************************************************************/
int Destack::getDx(void)
{
 if (dest_stack != NULL) return(dest_stack->dx);
 else return(GET_ERR);
}
/*****************************************************************************/
/* Procedure: getDy                                                          */
/*    Returns the Y coordinate of the destination in the top Dnode of the    */
/*    destination stack.                                                     */
/*****************************************************************************/
int Destack::getDy(void)
{
 if (dest_stack != NULL) return(dest_stack->dy);
 else return(GET_ERR);
}
/*****************************************************************************/
/* Procedure: getSx                                                          */
/*    Returns the X coordinate of the source point in the top Dnode of the   */
/*    destination stack.                                                     */
/*****************************************************************************/
int Destack::getSx(void)
{
 if (dest_stack != NULL) return(dest_stack->sx);
 else return(GET_ERR);
}

/*****************************************************************************/
/* Procedure: getSy                                                          */
/*    Returns the Y coordinate of the source point in the top Dnode of the   */
/*    destination stack.                                                     */
/*****************************************************************************/
int Destack::getSy(void)
{
 if (dest_stack != NULL) return(dest_stack->sy);
 else return(GET_ERR);
}

/*****************************************************************************/
/* Procedure: getDt                                                          */
/*   Returns the destination type of the top Dnode in the destination stack. */
/*****************************************************************************/
destType Destack::getDt(void)
{
 if (dest_stack != NULL) return(dest_stack->dtype);
 else return(D_ERROR);
}

/*****************************************************************************/
/* Procedure: pushDS                                                         */
/*    Creates a new Dnode and pushes it onto the top of the destination stack*/
/*****************************************************************************/
void Destack::pushDS( int sx, int sy, int dx, int dy, destType dtype)
{
 Dnode *temp;
 temp = dest_stack;
 dest_stack = new(Dnode);
 dest_stack->sx = sx;
 dest_stack->sy = sy;
 dest_stack->dx = dx;
 dest_stack->dy = dy;
 dest_stack->dtype = dtype;
 dest_stack->npt = temp;
}

/*****************************************************************************/
/* Procedure: popDS                                                          */
/*    Pops the top node (if any) off of the destination stack & frees it.    */
/*    The last Dnode (start Dnode) in the stack cannot be popped.            */
/*****************************************************************************/
short Destack::popDS()
{
 Dnode *temp;
 if (dest_stack != NULL) {
    if (dest_stack->npt != NULL) {    /* check if last node */
       temp = dest_stack;
       dest_stack = dest_stack->npt;
       delete(temp);
       return(OK);
    }
 }
 return(ERR);
}
```

```
/*****************************************************************************/
/* Procedure: newStart                                                       */
/*   Creates a new start Dnode .  We add the new Dnode under the previous    */
/*   start so that it is reached by backtracking. This allows Luey to move   */
/*   away from the destination, he thinks he is just backtracking to a       */
/*   previous checkpoint.  A new start is needed when the following          */
/*   conditions are met.                                                     */
/*   1. Only a single checkpoint is left. 2. All known paths lead to dead    */
/*   ends. 3. All possible path of decreasing potential are known. 4. Luey   */
/*   is at the cell specified by the old start node.                         */
/*****************************************************************************/
void Destack::newStart(int x, int y, destType dtype)
{
 Dnode *temp;
 temp = new(Dnode);
 temp->dx = x;              /* Put current location as destination */
 temp->dy = y;
 temp->sx = x;              /* Source is not appicable             */
 temp->sy = y;
 temp->dtype = dtype;
 temp->npt = NULL;
 dest_stack->sx = x;        /* Current loc is source for old start  */
 dest_stack->sy = y;
 dest_stack->dtype = D_BACKTRACK;
 dest_stack->npt = temp;
}

/*****************************************************************************/
/* Procedure: changeDt                                                       */
/*    Changes the dtype field in the top Dnode in the destination stack.  A  */
/*    D_QUIT type cannot be changed.                                         */
/*****************************************************************************/
void Destack::changeDt(destType dtype)
{
 if (dest_stack != NULL) {
    if (dest_stack->dtype != D_QUIT) dest_stack->dtype = dtype;
 }
}

/*****************************************************************************/
/* Procedure: changeDS                                                       */
/*    Changes the destination coordinates in the top Dnode in the destination*/
/*    stack.                                                                 */
/*****************************************************************************/
void Destack::changeDS(int dx, int dy)
{
 if (dest_stack != NULL) {
    dest_stack->dx = dx;
    dest_stack->dy = dy;
 }
}
```

```
/*****************************************************************************/
/* Procedure: shiftDS                                                        */
/*    Shifts the destinations stored in the destination stack.               */
/*****************************************************************************/
void Destack::shiftDS(int x, int y)
{
 Dnode *temp;
 temp = dest_stack;
 while (temp != NULL) {
    temp->dx = temp->dx - x;
    temp->dy = temp->dy - y;
    temp->sx = temp->sx - x;
    temp->sy = temp->sy - y;
    temp = temp->npt;
 }
}

/*****************************************************************************/
/* Procedure: clearDS                                                        */
/*    Deletes the destination stack.                                         */
/*****************************************************************************/
void Destack::clearDS(void)
{
 Dnode *temp;
 while (dest_stack != NULL) {
    temp = dest_stack;
    dest_stack = dest_stack->npt;
    delete(temp);
 }
}

/*****************************************************************************/
/* Procedure: queryStart                                                     */
/*    Returns TRUE if there is only one node left in the destination stack.  */
/*****************************************************************************/
short Destack::queryStart(void)
{
 if (dest_stack != NULL) {
    if (dest_stack->npt == NULL) return(TRUE);
 }
 return(FALSE);
}
```

```
#ifndef RWITIM_H
#define RWITIM_H

// Original Name     :  /pro/ai/robot/include/RWISimple.h
// Original Authors  :  Jak Kirman, Ken Basye
// Date              :  7 Nov 1990
// Last modification : 28 Jun 1991
// Modifications by Tim Good:
//    name changed from RWISimple.h to RWITim.h
//    added executeTurn, executeMove
//    simulator routines: readSonarMap, querySonar, setSonarValue
// Last modification : 26 Mar 1992
//
// RWISimple is a simple interface to RWIComplete, to drive the Real
// World Interface robot base.  It is intended for use when all that
// is necessary is the ability to modify the rates of translation and
// rotation, for example to implement a joystick.  Some query functions
// have been added also.

#include <RWIComplete.h>

class RWITim : private RWIComplete {
private:
  float _curSpeed;
  float _curSlew;
  long _baseAngle;
  long _baseDistance;
public:
  RWITim (Comm *comm);
  ~RWITim ();
  float queryVoltage ();                // return the voltage (in Volts)
  float queryAngle ();                  // return the current angle in deg
  float querySpeed ();                  // return the current speed in cms/sec
  float querySlew ();                   // return the current slew rate
  float queryDistance ();               // return the current dist in cms
  void setSpeed (float cmspersec);      // set the speed
  void setSlew (float degpersec);       // set the slew rate
  void setAcceleration (float cmspersecpersec); // set acceleration
  void setSlewAcceleration (float degpersecpersec); // set slew acceleration
  void zeroDistance ();                 // zero distance counter
  void zeroAngle ();                    // zero angle counter;
  void kill ();                         // kill the base
  void halt ();                         // halt the base
  void executeTurn(float degrees, float degpersec, short waitFlag);
                                        // turn fixed amount
  void executeMove(float cms, float cmspersec, short waitFlag);
                                        // move a fixed amount
};
/*****************************************************************************/
/* If the simulator is being run instead of the robot then SIMULATE should   */
/* be defined when compiling the driver.  Instead of RWITim.C and Sonar.C    */
/* the file  simulat.C is used.  The following are routines to  manipulate   */
/* the sonar map.                                                            */
/*****************************************************************************/
#ifdef SIMULATE
    extern short readSonarMap(char *fn);     /* Reads in a new sonar map     */
    extern short querySonarMap(int x,int y); /* Queries the sonar map        */
    extern void  setSonarMap(int x,int y,short val); /* Sets the sonar map   */
    extern void  sonarReset(float newAngle); /* Reset simulator angle        */
#endif  /* SIMULATE */
#endif  /* RWITIM_H */
```

```
// RWITim.C         3/27/92    Timothy T. Good
//
// Original Name     :  /pro/ai/robot/src/rwiBase/RWISimple.C
// Original Authors  :  Jak Kirman, Ken Basye
// Attributions      :
// Date              :  7 Nov 1990
// Last modification : 30 Jul 1991

// Modifications by Tim Good:
//     name changed from RWISimple.C to RWITim.C
//     added executeTurn, added executeMove
//     halt calls translateVelocity and rotateVelocity directly
//     translateConstantPositive & rotatateConstantPositive assumed unless
//       speed, slew negative
// Last modification : 27 Mar 1992
//
#include "RWITim.h"
#include <iostream.h>
#include <math.h>


float normalize (float x)
{
  if (x >= 0)
    x = (x - int (x / 360) * 360);
  else
    x = x - int (x / 360) * 360 + 360;

  return x;
}


RWITim::RWITim (Comm *comm)
: RWIComplete (comm)
{
  translateHalt ();
  rotateHalt ();
  units (ENCODER);
  _baseAngle = long (rotateWhere ());
  _baseDistance = long (translateWhere ());
  translateVelocity (0);
  rotateVelocity (0);
  translateConstantPositive ();
  rotateConstantPositive ();
  _curSpeed = 0.0;
  _curSlew = 0.0;
}

RWITim::~RWITim ()
{
  kill ();
}

/*****************************************************************************/
float RWITim::queryVoltage ()
{
  return batteryVoltage () / 10.0;
}

/*****************************************************************************/
float RWITim::queryAngle ()
{
  return (normalize ((long (rotateWhere ()) - _baseAngle)/countsPerDegree));
}

/*****************************************************************************/
float RWITim::queryDistance ()
{
  return ((long (translateWhere ()) - _baseDistance)/countsPerCentimeter);
}


/*****************************************************************************/
void RWITim::zeroDistance ()
{
  _baseDistance = long (translateWhere ());
}

/*****************************************************************************/
void RWITim::zeroAngle ()
{
  _baseAngle = long (rotateWhere ());
}

/*****************************************************************************/
void RWITim::setAcceleration (float cmsPerSecPerSec)
{
  translateAcceleration (int (countsPerCentimeter * cmsPerSecPerSec));
}

/*****************************************************************************/
void RWITim::setSlewAcceleration (float degPerSecPerSec)
{
  rotateAcceleration (int (countsPerDegree * degPerSecPerSec));
}

/*****************************************************************************/
void RWITim::setSpeed (float cmsPerSec)
{
  if (cmsPerSec >= 0) {
     if (_curSpeed < 0.0) translateConstantPositive();
     translateVelocity (cmsPerSec * countsPerCentimeter);
  } else {
     if (_curSpeed >= 0.0) translateConstantNegative();
     translateVelocity (-cmsPerSec * countsPerCentimeter);
  }
  _curSpeed = cmsPerSec;
}

/*****************************************************************************/
void RWITim::setSlew (float degPerSec)
{
  if (degPerSec >= 0) {
     if (_curSlew < 0.0) rotateConstantPositive ();
     rotateVelocity (degPerSec * countsPerDegree);
  } else {
     if (_curSlew >= 0.0) rotateConstantNegative();
     rotateVelocity (-degPerSec * countsPerDegree);
  }
  _curSlew = degPerSec;
}

/*****************************************************************************/
void RWITim::halt ()
{
  translateVelocity(0);
  rotateVelocity(0);
  if (_curSpeed < 0.0) translateConstantPositive();
```

```
  if (_curSlew  < 0.0) rotateConstantPositive();
  _curSpeed = 0.0;
  _curSlew = 0.0;
}

/*****************************************************************************/
void RWITim::kill ()
{
  RWIComplete::kill ();
  if (_curSpeed < 0.0) translateConstantPositive();
  if (_curSlew  < 0.0) rotateConstantPositive();
  _curSpeed = 0;
  _curSlew = 0;
}

/*****************************************************************************/
float RWITim::querySpeed ()
{
  return _curSpeed;
}

/*****************************************************************************/
float RWITim::querySlew ()
{
  return _curSlew;
}
/*****************************************************************************/
void RWITim::executeTurn (float degrees, float degPerSec, short waitFlag)
{
 float degp,dpsp;           /* absolute value of degrees, degPerSec     */
 degp = abs(degrees);
 dpsp = abs(degPerSec);
 if ((degrees * degPerSec) >= 0.0) {
    rotateRelativeForward( (unsigned long)(degp * countsPerDegree));
    rotateVelocity( (unsigned long)(dpsp * countsPerDegree));
   _curSlew = dpsp * countsPerDegree;
 } else {
    rotateRelativeBackward( (unsigned long)(degp * countsPerDegree));
    rotateVelocity( (unsigned long)(dpsp * countsPerDegree));
    _curSlew = -dpsp * countsPerDegree;
 }
 if (waitFlag == 1) {
    waitEvent(RotateStop);
    _curSlew = 0.0;
    rotateVelocity(0);
    rotateConstantPositive();
 }
}

/*****************************************************************************/
void RWITim::executeMove (float cms, float cmsPerSec, short waitFlag)
{
 float cmsp,cpsp;           /* absolute value of cms, cmsPerSec     */
 cmsp = abs(cms);
 cpsp = abs(cmsPerSec);
 if ((cms * cmsPerSec) >= 0.0) {
    translateRelativeForward( (unsigned long)(cmsp * countsPerCentimeter));
    translateVelocity( (unsigned long)(cpsp * countsPerCentimeter));
    _curSpeed = cpsp * countsPerCentimeter;
 } else {
    translateRelativeBackward( (unsigned long)(cmsp * countsPerCentimeter));
    translateVelocity( (unsigned long)(cpsp * countsPerCentimeter));
    _curSpeed = -cpsp * countsPerCentimeter;
 }
 if (waitFlag == 1) {
    waitEvent(TranslateStop);
    _curSpeed = 0.0;
    translateVelocity(0);
    translateConstantPositive();
 }
}
```

```
/*****************************************************************************/
/* simulat.C  04/02/92  Timothy T. Good                                      */
/*    Replaces the files RWITim.C and Sonar.C  Contains routines to simulate */
/*    the robot base and sonar routines to allow testing without using the   */
/*    robot.  Distances and angles are calculated by using an estimation of  */
/*    the elapsed time.  Time estimates were determined empirically during   */
/*    testing with the robot.                                                */
/*****************************************************************************/

#include "cgrid.h"                /* Certainty grid module  */
#include "oman.h"                 /* Navigator module       */
#include "display.h"              /* Display module         */
#include "RWITim.h"               /* Robot base routines    */
#include <Speech.h>               /* Robot speech routines  */
#include <Sonar.h>                /* Robot sonar routines   */
#include <math.h>
#include <curses.h>

/*****************************************************************************/
/* Global data                                                               */
/*****************************************************************************/

extern Certgrid *CgPt;    /*(driver) Certainty grid                         */
extern int   offx,offy;   /*(oman) Offset of robot from cell center         */
extern int   robx,roby;   /*(oman) Robots current cell                      */
extern int   spdfac;      /*(oman) Multiplier for speed increment           */
extern float slew;        /*(oman) Robot's turn rate in degrees. + is ccw   */
extern float timer;       /*(oman) Estimate of elapsed time since last update*/

static float simAngle;    /*       Base angle in simulator (robot dir.is cw) */
static float simDist;     /*       Base distance in simultor               */
static short sgrid[SZE][SZE]; /* map for sonar simulation                  */

/*****************************************************************************/
/* Procedures:                                                               */
/*    Most are declared in the .h files  RWITim.h, Sonar.h                   */
/*    The following are additional ones needed for the sonar simulator.      */
/*****************************************************************************/

inline int   ROUND(float fval);                    /* Rounds float to int   */
       short querySonarMap(int x,int y);           /* Queries the sonar map */
       void  setSonarMap(int x,int y,short val);   /* Sets the sonar map    */
       void  sonarReset(float newangle);           /* Reset simulator angle */
       short readSonarMap(char *fname);            /* Reads a new sonar map */
static int   scanSonarMap(float,float,int,int,float); /* Sonar simulator     */
static int   cksgrd(int offset, int x, int y, int olddist);

/*****************************************************************************/
/* Procedure: ROUND                                                          */
/* Calls:     <none>                                                         */
/* Returns:   int - argument rounded to nearest integer                      */
/*****************************************************************************/
inline int ROUND(float fval)
{
 return   (  (fval >= 0)  ? (int)((fval) + 0.5)  : (int)((fval) - 0.5)  );
}
```

```
/*****************************************************************************/
/*---------------------------------------------------------------------------*/
/*- RWITim class - contains robot control routines                          -*/
/*-    This is dummy version of the robot base routines in RWITim.C which   -*/
/*-    is itself based on /pro/ai/robot/src/rwiBase/RWISimple               -*/
/*-    Keeps track of the  speed, turn rate, angle and distance traveled.   -*/
/*---------------------------------------------------------------------------*/
/*****************************************************************************/
/*****************************************************************************/
/* Procedure: RWITim         Constructor for RWITim class              Done */
/* Globals:  (within class) _curSlew, _curSpeed   (write only)               */
/*           (exo class)    simAngle, simDist     (write only)               */
/*    Initializes a few things.                                              */
/*    We use _curSlew and _curSpeed to keep track of the robot turn rate     */
/*    and speed but don't use _baseAngle or _baseDistance because they are   */
/*    the wrong type and would require the use of countsPerDegree and        */
/*    countsPerCentimeter. We use two new globals instead.                   */
/*****************************************************************************/
RWITim::RWITim(Comm *comm) : RWIComplete(comm)
{
 _curSlew = 0.0;
 _curSpeed = 0.0;
  simAngle = 0.0;
  simDist = 0.0;
}

/*****************************************************************************/
/* Procedure: ~RWITim         Destructor for RWITim class. Does nothing.     */
/*****************************************************************************/
RWITim::~RWITim()
{}

/*****************************************************************************/
/* Procedure: queryAngle     Returns robot angle                             */
/* Globals:   simAngle (Read, Write);  timer (Read only)                     */
/*****************************************************************************/
float RWITim::queryAngle(void)
{
 simAngle = simAngle + (_curSlew * timer);
 if (simAngle >= 360.0) simAngle = simAngle - 360.0;
 if (simAngle <= -360.0) simAngle = simAngle + 360.0;
 return(simAngle);
}

/*****************************************************************************/
/* Procedure: queryDistance   Returns robot distance.                        */
/* Globals:   simDist (Read, Write);  timer (Read only)                      */
/*****************************************************************************/
float RWITim::queryDistance(void)
{
 simDist = simDist + ((float)spdfac * (float)SPEED_INCREMENT * timer);
 return(simDist);
}

/*****************************************************************************/
/* Procedure: setSpeed       Sets robot speed.                               */
/* Globals:   _curSpeed  (Write only)                                        */
/*****************************************************************************/
void RWITim::setSpeed(float cmpersec)
{  _curSpeed = cmpersec; }
```

```
/*************************************************************************/
/* Procedure: setSlew        Sets robot turn rate.                       */
/* Globals:   _curSlew    (Write only)                                   */
/*************************************************************************/
void RWITim::setSlew(float degpersec)
{  _curSlew = degpersec; }

/*************************************************************************/
/* Procedure: queryVoltage        Returns a dummy value for robot voltage */
/*************************************************************************/
float RWITim::queryVoltage(void)
{ return(12.0); }

/*************************************************************************/
/* Procedure: halt                                                       */
/*    Determines new base angle and distance then sets speed and slew to 0. */
/* Globals:   simAngle, simDist (Read, Write);  timer (Read only)        */
/*************************************************************************/
void RWITim::halt(void)
{
 simAngle =  simAngle + (_curSlew * timer);
 simDist = simDist + ((float)spdfac * (float)SPEED_INCREMENT * timer);
 _curSlew = 0.0;
 _curSpeed = 0.0;
}

/*************************************************************************/
/* Procedure: executeTurn                                                */
/* Globals:   simAngle (Read, Write)                                     */
/*    Changes the base angle by the amout passed as argument. The 2nd    */
/*    argument (float rate) and 3rd arg: (short waitFlag) are not used.  */
/*************************************************************************/
void RWITim::executeTurn(float degrees, float , short )
{  simAngle = simAngle + degrees; }

/*************************************************************************/
/* Procedure: RWITim::executeMove                                        */
/* Globals:   simDist (Read, Write)                                      */
/*    Increases the base distance by the amount passed as argument.      */
/*    2nd argument (float rate) and 3rd argument (short waitFlag) not used. */
/*************************************************************************/
void RWITim::executeMove(float cms, float, short)
{ simDist = simDist + cms; }

/*************************************************************************/
/* Procedure: kill                                                       */
/* Globals:   simAngle, simDist (Read, Write);  timer (Read only)        */
/*    Determines new base angle and distance then sets speed and slew to 0. */
/*************************************************************************/
void RWITim::kill()
{
 simAngle =  simAngle + (_curSlew * timer);
 simDist = simDist + ((float)spdfac * (float)SPEED_INCREMENT * timer);
 _curSlew = 0.0;
 _curSpeed = 0.0;
}
```

```
/*************************************************************************/
/*-----------------------------------------------------------------------*/
/*- RWIComplete class - needed because it is part of RWITim             -*/
/*-----------------------------------------------------------------------*/
/*************************************************************************/

/*************************************************************************/
/* Procedure: RWIComplete::RWIComplete                                   */
/*   argument (Comm *comm) not used                                      */
/*************************************************************************/
RWIComplete::RWIComplete(Comm *)
()
RWIComplete::~RWIComplete()
()
/*************************************************************************/
/* The following functions are needed because they are listed as virtual */
/* functions in RWIComplete.h  None of these functions are actually called */
/* by the simulator.                                                     */
/*************************************************************************/
unsigned long RWIComplete::send (const char *const )
{ return((unsigned long)0); }
void RWIComplete::send (const char *const, unsigned long, int)
()
void RWIComplete::sendCommand (const char *const)
()
void RWIComplete::sendWaitCommand(const char *const)
()
unsigned long RWIComplete::sendQuery(const char *const)
{ return(0); }
RWIComplete::RWIError RWIComplete::sendThing (const char *, char *)
{ return(Ok); }
void RWIComplete::unitConvert(unsigned long &, const char* )
()
```

```
/*****************************************************************************/
/*---------------------------------------------------------------------------*/
/*- Sonar class - contains the routines for using the sonars                -*/
/*---------------------------------------------------------------------------*/
/*****************************************************************************/

/*****************************************************************************/
/* Procedure: Sonar::Sonar                                                   */
/*    Constructor for sonar class. Initializes the sonar map to 0.   The     */
/*    arguments (Comm *line) and (int nSonars) are not used.                 */
/*****************************************************************************/
Sonar::Sonar(Comm *, int)
{ readSonarMap("0"); }
/*****************************************************************************/
/* Procedure: Sonar::~Sonar      Destructor for sonar class, does nothing.   */
/*****************************************************************************/
Sonar::~Sonar()
{}
/*****************************************************************************/
/* Procedure: read                                                           */
/* Calls:     scanSonarMap - get distance to closest object for single sonar */
/* Globals:   simAngle (Read only)                                           */
/*    Determines the location of the sonar relative to the center of the     */
/*    cell with with robot then uses scanSonarMap to get a distance value    */
/*    for each of the 8 sonars.                                              */
/*    Below is a table with the hardcoded values giving each sonars position */
/*    they are x & y distance (cm) from robot center, radial distance from   */
/*    the robot center and the angular difference between the radial line    */
/*    to the sonar and the sonars direction                                  */
/*    # location      dir          x     y    r     deg     rad              */
/*    1 Left rear     90   1.5708   8.2   9.5  12.5  +40.80  +0.7121         */
/*    7 Left forw     90   1.5708  15.8   9.5  18.4  -58.98  -1.0294         */
/*    0 Right rear   -90  -1.5708   8.2   8.5  11.8  -43.97  -0.7674         */
/*    6 Right forw   -90  -1.5708  15.8   8.5  17.9  +61.72  +1.0773         */
/*    5 Front left     0   0.0     13.3  12.0  17.9  +42.06  +0.7341         */
/*    4 Front right    0   0.0     13.3  11.0  17.3  -39.59  -0.6910         */
/*    3 Angle left    45   0.7853  11.6  11.2  16.1   -1.00  -0.0175         */
/*    2 Angle right  -45  -0.7853  11.6  10.2  15.4   +3.67  +0.0641         */
/*    2 Back left    180   3.1416   5.7  12.0  13.3  -64.59  -1.1273         */
/*    3 Back right   180   3.1416   5.7  11.0  12.4  +62.61  +1.0927         */
/*****************************************************************************/
void Sonar::read(int *where)
{
 float angle;        /* Robot base angle(radians) */
 float angle2;       /* Angle(radians) from cell center to sonar */
 int   dx,dy;        /* Distance (mm) from cell center to sonar */
 angle = -simAngle * 0.017453293;         /* convert to radians */

 angle2 = angle + 1.5708 + 0.7121;
 dx = offx + (int)(125.0 * cos(angle2));
 dy = offy + (int)(125.0 * sin(angle2));
 where[1] = scanSonarMap(1.5708,angle,dx,dy,0.140);  /* Left rear */
 angle2 = angle + 1.5708 - 1.0294;
 dx = offx + (int)(184.0 * cos(angle2));
 dy = offy + (int)(184.0 * sin(angle2));
 where[7] = scanSonarMap(1.5708,angle,dx,dy,0.140);  /* Left forward */

 angle2 = angle - 1.5708 - 0.7674;
 dx = offx + (int)(118.0 * cos(angle2));
 dy = offy + (int)(118.0 * sin(angle2));
 where[0] = scanSonarMap(-1.5708,angle,dx,dy,0.140); /* Right rear */
 angle2 = angle - 1.5708 + 1.0773;
 dx = offx + (int)(179.0 * cos(angle2));
 dy = offy + (int)(179.0 * sin(angle2));
 where[6] = scanSonarMap(-1.5708,angle,dx,dy,0.140); /* Right forward */

 angle2 = angle + 0.7341;
 dx = offx + (int)(179.0 * cos(angle2));
 dy = offy + (int)(179.0 * sin(angle2));
 where[5] = scanSonarMap(0.0,angle,dx,dy,0.140);     /* Front left  */
 angle2 = angle - 0.6910;
 dx = offx + (int)(173.0 * cos(angle2));
 dy = offy + (int)(173.0 * sin(angle2));
 where[4] = scanSonarMap(0.0,angle,dx,dy,0.140);     /* Front right */

 if (CgPt->getSonarType() == S_FRONT) {
    angle2 = angle + 0.785 - 0.0175;
    dx = offx + (int)(161.0 * cos(angle2));
    dy = offy + (int)(161.0 * sin(angle2));
    where[3] = scanSonarMap(0.7853,angle,dx,dy,0.140);   /* Angle left */
    angle2 = angle - 0.785 + 0.0641;
    dx = offx + (int)(154.0 * cos(angle2));
    dy = offy + (int)(154.0 * sin(angle2));
    where[2] = scanSonarMap(-0.7853,angle,dx,dy,0.140);  /* Angle right */
 }
 if (CgPt->getSonarType() == S_SQUARE) {
    angle2 = angle + 3.1416 - 1.1273;
    dx = offx + (int)(133.0 * cos(angle2));
    dy = offy + (int)(133.0 * sin(angle2));
    where[2] = scanSonarMap(3.1416,angle,dx,dy,0.140);  /* Back left */
    angle2 = angle + 3.1416 + 1.0927;
    dx = offx + (int)(124.0 * cos(angle2));
    dy = offy + (int)(124.0 * sin(angle2));
    where[3] = scanSonarMap(3.1416,angle,dx,dy,0.140);  /* Back right */
 }
}

/*****************************************************************************/
/*---------------------------------------------------------------------------*/
/*- Following are not part of the Sonar class but are needed for the sonar  -*/
/*- simulator     querySonarMap, setSonarMap, readSonarMap, scanSonarMap,   -*/
/*- cksgrd, sonarReset                                                      -*/
/*---------------------------------------------------------------------------*/
/*****************************************************************************/

/*****************************************************************************/
/* Procedure: querySonarMap                                                  */
/* Called by: (driver)test1, (driver)SetSonarValue                           */
/*    Queries the sonar map and returns the value at the specified cell.     */
/*****************************************************************************/
short querySonarMap(int x, int y)
{
 return(sgrid[x][y]);
}
```

```
/*****************************************************************************/
/* Procedure: setSonarMap                                                    */
/*    Sets and element of the sonar map.                                     */
/*****************************************************************************/
void setSonarMap(int x, int y, short val)
{
 if ((val == 1) || (val == 0)) sgrid[x][y] = val;
}

/*****************************************************************************/
/* Procedure: sonarReset                                                     */
/* Globals:   simAngle                                                       */
/*    Resets the simulator angle so it will match baseAngle used by the      */
/*    program. This is necessary because while query_angle will adjust the   */
/*    simulator angle, the sonar read routine will not.                      */
/*****************************************************************************/
void sonarReset(float angle)
{ simAngle = angle;
}

/*****************************************************************************/
/* Procedure: readSonarMap                                                   */
/* Called by: (driver)test1, Sonar::Sonar                                    */
/* Calls:     (oman)getConstOccup - retrieve occupancy threshold             */
/*            (oman)getConstEmpty - retrieve empty threshold                 */
/* Returns:   FALSE if file could not be opened, TRUE otherwise              */
/* Globals:   sgrid (Write only)                                             */
/* I/O:       reads external file                                            */
/*    Reads the file fname to add to the map used by the sonar simulator. If */
/*    the file name starts with 0 then the sonar map is set to 0. If the     */
/*    file could not be opened, then the map is unchanged.                   */
/*****************************************************************************/
short readSonarMap(char *fname)
{
 int   i,j;              /* Cell indices          */
 float cert,conf;        /* Certainty & confidence */
 FILE *fpin;             /* Input file pointer     */
 const float OCCUP = getConstOccup();
 const float EMPTY = getConstEmpty();
 if (fname[0] == '0') {             /* Initialize to 0, no file to read   */
    for (i=0; i<SZE; i++) {
       for (j=0; j<SZE; j++) {
          sgrid[i][j] = 0;
       }
    }
    return(TRUE);
 }
/***** Read in initialization values                                    *****/
 fpin = fopen(fname,"r");
 if (fpin == NULL) return(FALSE);
 fscanf(fpin,"%3d %3d %f %f",&i,&j,&cert,&conf);
 while (i != 999) {
    if ((i<SZE) && (j<SZE) && (i>=0) && (j>=0)) {
       if (cert >= OCCUP) sgrid[i][j] = 1;
       if (cert <= EMPTY) sgrid[i][j] = 0;
    }
    fscanf(fpin,"%3d %3d %f %f",&i,&j,&cert,&conf);
 }
 return(TRUE);
}
```

```
/*****************************************************************************/
/* Procedure: scanSonarMap                                            Done   */
/* Called by: sonar::read                                                    */
/* Calls:     cksgrd  - check the sonar grid to see if a cell is occupied    */
/*            sondist - determines distance from sonar to cell               */
/* Returns:   int - distance (mm) to nearest object in cone                  */
/* Globals:   robx,roby,sgrid (read)   cgrid (write)                         */
/*    Sonar simulator routine. Checks all cells in the cone defined by the   */
/*    sonar position & angle and the angular spread of the sonar. Cells are  */
/*    checked against those in the sonar simulator map sgrid. Returns the    */
/*    distance in mm to the nearest object found.  Much of the code is the   */
/*    same as or modified from scan1 in cgrid.C                              */
/*****************************************************************************/
int scanSonarMap(
   float   angle,        /* sonar angle(radians) wrt robot head             */
   float baseangle,      /* Current robot direction(radians)                */
   int   dx, int dy,     /* sonar position(mm) relative to robot cell center */
   float spread)         /* Half angle(radians) of sonar spread             */
{
 int   adjust;           /* adjusts distance for sonar position & cell edge  */
 int   dist;             /* distance(mm) to closest object                   */
 float dir1,dir2;        /* direction to object ends                         */
 int   xg,yg;            /* grid coordinates of object center                */
 int   xg1,yg1,xg2,yg2;  /* grid coordinates of object ends                  */
 int   xp1,yp1,xp2,yp2;  /* grid coordinates of unoccupied squares           */
 int   xc,yc;            /* X and Y distance to object in grid cells         */
 float x,y;              /* x and y distances to object center               */
 float x1,y1,x2,y2;      /* x and y distances to object ends                 */
 float px1,py1,px2,py2;  /* coordinates of unoccupied squares                */
 float slope1,slope2;    /* slope of lines to object ends                    */
 int   i,j;              /* Loop indices                                     */
 int   offcx,offcy;      /* Distance (cells) of sonar from cell center       */
/*****************************************************************************/
/* The initial distance found is from the center of the robot cell to the    */
/* center of the object cell.  It must be adusted to account for the         */
/* distance of the sonar from the cell center and the distance from the      */
/* object cell center to the edge.                                           */
/*****************************************************************************/
 adjust = (int)(offCenter(dx,dy,((angle+baseangle)*57.296)) + 100.0);
/*****************************************************************************/
/* First we find the endpoints of the maximum cone scanned by a sonar.  Using*/
/* the sonar range and angle we compute the coordinates.                     */
/*****************************************************************************/
 dist = SONAR_RANGE;
 dir1 = baseangle + angle + spread; /* Get angles to cone ends      */
 dir2 = baseangle + angle - spread;
 if (dx > 0)  offcx = (dx + 100)/CELL;   /* Determine if sonar is not in */
 else offcx = (dx - 100)/CELL;             /* same cell as robot center    */
 if (dy > 0)  offcy = (dy + 100)/CELL;
 else offcy = (dy - 100)/CELL;
 x1 = ((float)dist * cos(dir1));      /* Calculate X & Y components of dist. */
 y1 = ((float)dist * sin(dir1));      /* to cone max left from robot center  */
 xc = ROUND(x1/CELL);                 /* Convert to grid cells               */
 xg1 = xc + robx + offcx;
 yc = ROUND(y1/CELL);
 yg1 = yc + roby + offcy;
 x2 = ((float)dist * cos(dir2));      /* Calculate X & Y components of dist. */
 y2 = ((float)dist * sin(dir2));      /* to cone max right from robot center */
 xc = ROUND(x2/CELL);                 /* Convert to grid cells               */
 xg2 = xc + robx + offcx;
 yc = ROUND(y2/CELL);
 yg2 = yc + roby + offcy;
/*****************************************************************************/
```

```
/* Some housekeeping to determine the loop boundaries. We will increment from*/
/* the robot to the closer of the two endpoint X values, then again to the   */
/* closer of the Ys.                                                         */
/*****************************************************************************/
 if (abs(xg1 - robx) > abs(xg2 - robx)) xg = xg2;
 else (xg = xg1);
 if (abs(yg1 - roby) > abs(yg2 - roby)) yg = yg2;
 else (yg = yg1);
 x  = (x1 + x2)/2.0;
 y  = (y1 + y2)/2.0;
/*****************************************************************************/
/* Next we increment along all the cells in the cone, and save the distance  */
/* to the closest one which the sonar map says is occupied.                  */
/*****************************************************************************/
 if (fabs((double)x) > (double)CELL) { /* If x distance greater than a cell  */
    slope1 = y1 / x1;
    slope2 = y2 / x2;
    if ((robx+offcx) < xg) {        /* positive increment from robx to xg   */
       py1 = ((float)(roby + offcy)) - slope1;
       py2 = ((float)(roby + offcy)) - slope2;
       for (xp1= robx+offcx; xp1<xg;xp1++) { /* For each X along line of   */
          py1 = py1 + slope1;            /* sight, calculate Ys of cone edge */
          yp1 = ROUND(py1);
          py2 = py2 + slope2;            /* For each Y between cone edges,   */
          yp2 = ROUND(py2);              /* check sgrid for cell             */
          if (yp1 <= yp2) for (j=yp1; j<=yp2; j++) {
             dist = cksgrd(adjust,xp1,j,dist);
          }
          else for (j=yp2; j<=yp1; j++) {
             dist = cksgrd(adjust,xp1,j,dist);
          }
       }
    } else {                          /* negative increment from robx to xg   */
       py1 = ((float)(roby + offcy)) + slope1;
       py2 = ((float)(roby + offcy)) + slope2;
       for (xp1=robx+offcx; xp1>xg; xp1--) {
          py1 = py1 - slope1;
          yp1 = ROUND(py1);
          py2 = py2 -  slope2;
          yp2 = ROUND(py2);
          if (yp1 <= yp2) for (j=yp1; j<=yp2; j++) {
             dist = cksgrd(adjust,xp1,j,dist);
          }
          else for (j=yp2; j<=yp1; j++) {
             dist = cksgrd(adjust,xp1,j,dist);
          }
       }
    }
 }
 if (fabs((double)y) > (double)CELL) {   /* Repeat along y direction         */
    slope1 = x1 / y1;
    slope2 = x2 / y2;
    if (roby < yg) {
       px1 = ((float)(robx + offcx)) - slope1;
       px2 = ((float)(robx + offcx)) - slope2;
       for (yp1=roby+offcy; yp1<yg; yp1++) { /* For each Y along line of   */
          px1 = px1 + slope1;            /* sight, calculate Xs of cone edges*/
          xp1 = ROUND(px1);
          px2 = px2 + slope2;            /* For each X between cone edges,   */
          xp2 = ROUND(px2);              /* check sgrid for cell             */
          if (xp1 <= xp2) for (i=xp1; i<=xp2; i++) {
             dist = cksgrd(adjust,i,yp1,dist);
          }
          else for (i=xp2; i<=xp1; i++) {
             dist = cksgrd(adjust,i,yp1,dist);
          }
       }
    } else {
       px1 = ((float)(robx + offcx)) + slope1;
       px2 = ((float)(robx + offcx)) + slope2;
       for (yp1=roby+offcy; yp1>yg; yp1--) {
          px1 = px1 - slope1;
          xp1 = ROUND(px1);
          px2 = px2 - slope2;
          xp2 = ROUND(px2);
          if (xp1 <= xp2) for (i=xp1; i<=xp2; i++) {
             dist = cksgrd(adjust,i,yp1,dist);
          }
          else for (i=xp2; i<=xp1; i++) {
             dist = cksgrd(adjust,i,yp1,dist);
          }
       }
    }
 }
 if (dist < 0) dist = 0;
 if (dist == SONAR_RANGE) return(-1);
 else return(dist);
} /** end scanSonarMap() **/
/*****************************************************************************/
/* Procedure: cksgrd                                                         */
/* Called by: scanSonarMap                                                   */
/* Calls:     distance - find distance between two points                    */
/* Globals:   robx, roby, sgrid (Read only)                                  */
/*    If the indicated cell is marked in the sonar map, finds the distance   */
/*    from that cell to the sonar. First the distance from the cell to the   */
/*    robot's cell is found, then it is adjusted to account for the sonar    */
/*    location and cell edges. Returns the lower of this distance or the     */
/*    previous low distance.                                                 */
/*****************************************************************************/
int cksgrd(
   int adjust,          /* Distance adjustment to account for sonar position */
   int x, int y,        /* Cell to check on sonar map                        */
   int olddist)         /* Distance to closest occupied cell so far          */
{
 int dist;
 if (sgrid[x][y] == 1) {
    dist = (int)distance(robx,roby,x,y);
    dist = (CELL * dist) - adjust;
    if (dist <= olddist) return(dist);
 }
 return(olddist);
}  /** end cksgrd() **/
```