BROWN UNIVERSITY
Department of Computer Science
Master's Project
CS-89-M8

"The Feature Recognition Module of the LDP System for the Robot Huey"

by
Margaret J. Randazza

The Feature Recognition Module of the LDP System for the Robot Huey

by

Margaret J. Randazza

Sc.B., Brown University, 1986

Thesis

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

May 1990

This thesis by Margaret J. Randazza 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 10/27/89 Thomas L. Dean

Thomas L. Dean

Introduction

The LDP system for Huey

The system this paper describes runs on a mobile robot named Huey. Huey consists of a mobile base, PC AT, vision processing board, sonar ring and controlling hardware, and a camera mounted on a pan-tilt head. Presently the LDP system for Huey uses the sonar ring as its sole sensing device. Two rings of eight transducers placed one on top of the other combine to form a ring of 16 transducers. There are 22.5 degrees between transducers, giving Huey a 360° view. Each transducer has a maximum range of 25.5 feet. Huey's mobile base is 12 inches in diameter. One motor controls translation of the three wheels of the base, and another controls base rotation. This second motor rotates the top platform of the base along with the wheels, allowing Huey to turn in place. Shaft encoders for the two motors allow the base to keep track of how far it has translated and how much it has rotated.

The Locally Distinctive Place (LDP) system for Huey classifies interesting places Huey encounters as it moves through the corridors on a floor of a building. A Locally Distinctive Place is any juncture of corridors. For example, an LDP can be an intersection, T junction, or L junction of corridors. When Huey encounters an LDP, the system attempts to classify it as an intersection, T junction, L junction, or doorway. To do this, the LDP system looks for features that form the LDP. For example, convex corners and concave corners are features of intersections and L junctions. An intersection consists of four convex corners, and an L junction consists of a convex corner and a concave corner.

The LDP system for Huey is a base for a robot geographer. A robot geographer builds maps as it wanders through its surroundings. The geographer could request the LDP system to classify any locally distinctive place it has not encountered before. Once the LDP is classified, the geographer adds the junction to its map. This system is particularly useful in an environment that often changes, such as an office building with cubicles. The robot geographer eliminates the need to supply the robot with a new map each time the configuration of cubicles changes.

Related Work

The LDP system for Huey shares several ideas with [6] and [7]. In [7], Kuipers and Byun apply concepts from the TOUR model of [6] to introduce a spatial representation that does not rely on metrical information. Their topological representation of distinctive places connected by travel edges includes procedural knowledge to guide the robot in a local area. To identify an area as one already encountered, the system performs rough matching of metrical information. If there is more than one possible match, the system formulates routes to adjacent distinctive places and tests its hypotheses. This system does not deal directly with sensor error, or with unexpected obstacles in the environment.

In [3] and [9], Moravec and Elfes propose a grid-based spatial representation. Each cell of the grid contains a value marking it as probably occupied, probably empty, or unknown. To obtain the grid, Moravec and Elfes integrate many range readings taken at different points on the grid. The system deals directly with sonar error by estimating the probability that a given point in the sonar cone is empty or occupied. Consistent evidence reinforces a cell's probability, while conflicting evidence weakens it. In [3], Elfes proposes a navigation system based on the occupancy grid maps. He defines three axes of representation for sonar maps; the abstraction, geographical, and resolution.

Crowley introduces a spatial representation based on line segments in [2]. He converts range data to a global coordinate system and uses recursive line fitting to produce a set of line segments from the data. These line segments form a sensor map that he integrates with a composite local map. A previously learned global map also contributes to the evolution of the composite local map. Similar to [9], Crowley reinforces confidence in a line segment when there is consistent data, and weakens confidence when there is conflicting data. Navigation in this model is based on convex regions cut from the global map. A local path planning module attempts to circumvent unexpected obstacles.

In [8], Miller proposes a model for spatial representation based on polygonal regions. He defines a j-F region as a region eliminating j

degrees of freedom from the positioning of the robot. If a position consists of x and y coordinates and an orientation theta, a 3-F region allows positioning of the robot with respect to all three components. Path planning involves identifying the regions to traverse, calculating the Voronoi diagram of the resulting polygon, and searching the graph of the diagram for the shortest path. This system assumes the existence of an accurate global map, and makes no provision for unexpected obstacles in the robot's environment.

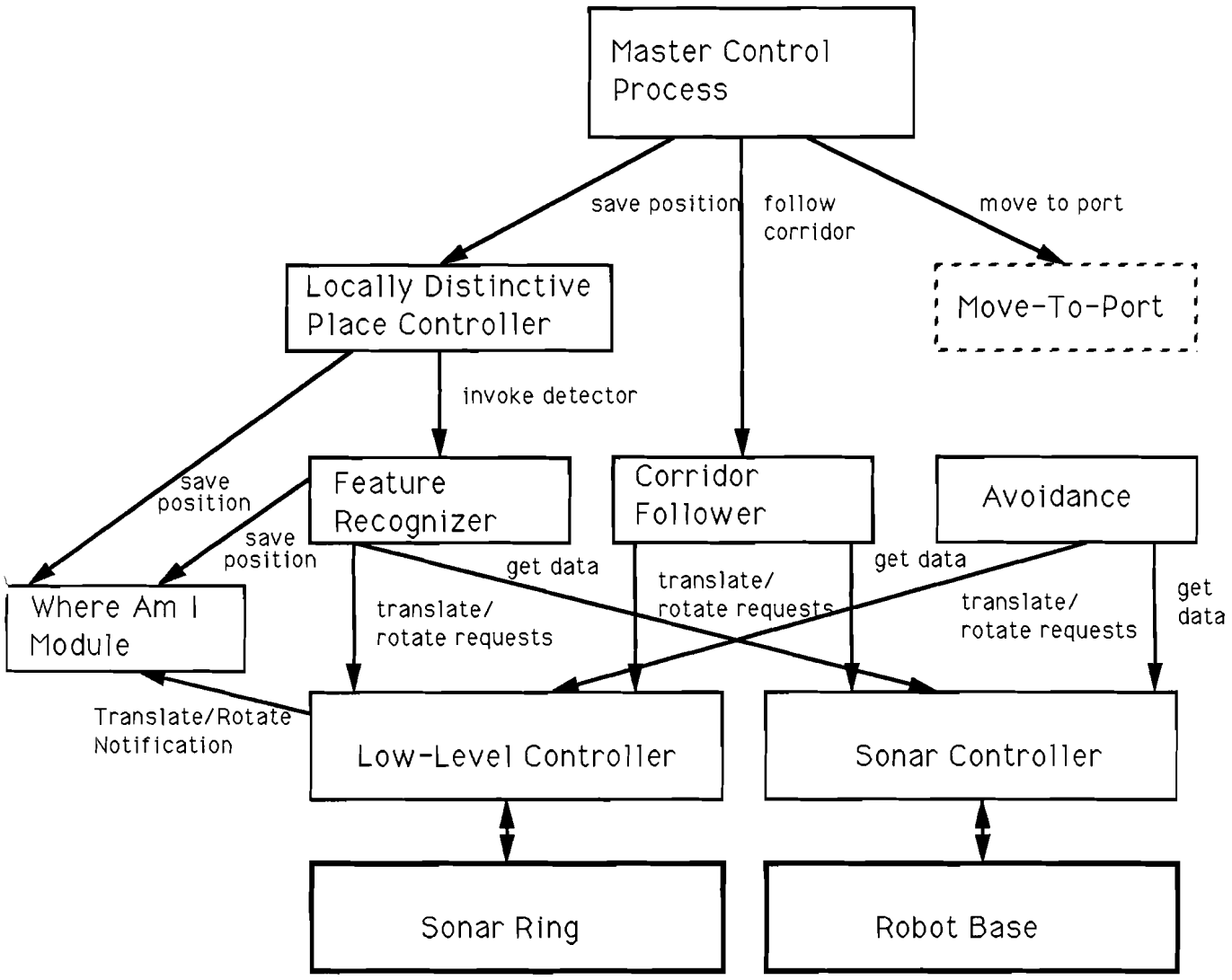
Overview of the System

There are eight processes in the LDP system for Huey. Figure 1 on the next page illustrates their interconnection. The Sonar Controller (SC) and Low-Level Controller (LLC) are the only two processes that have direct access to the robot's hardware. Any other process wanting to gain access to the sonars or the robot base must send a request to the Sonar Controller or the Low-Level Controller respectively. The Where-Am-I (WAI) process maintains a sense of position for the robot. The Low-Level-Controller sends messages to the WAI informing it of changes in the robot's motion. The Master Control (MC) process controls the action of the Locally Distinctive Place (LDP) process, the Corridor Following Module (CFM) process, and the Move-To-Port utility. The LDP process directs exploration of an LDP by requesting the Feature Recognition Module (FRM) process to invoke feature detectors that look for features forming the LDP. The CFM moves the robot down a corridor to the next LDP. The Move-To-Port utility positions the robot for the CFM by moving the robot into one of the corridors of the LDP. The final process of the system, the Avoidance Module (AM) process, prevents collisions of the robot with obstacles in its environment.

IPC and Name Server Modules

The eight processes of the system communicate via a message passing scheme provided by the Interprocess Communication Module (IPC) and the Name Server Module. The Name Server Module associates a unique name, or client ID, with each process in the system. The IPC module uses the unique client ID to deliver messages between the processes.

Before a process can receive messages from other processes in the system, it must first register itself with the name server module. At registration, the process supplies a name for itself and a process priority. The name server issues a client ID for the process and associates the process name and priority with the new ID. Other processes will use the name to get the client ID of the process in order to send messages to it. Some processes will use the priority



Overview of the System

Process

Utility

Hardware

Figure 1

to arbitrate between processes when more than one requests service at a time.

To send a request for service to another process, the sending process needs to know the client ID of the process it wants to send the request to. A request to the name server provides the sending process with this information. The sending process can then call the IPC module to send its request. The IPC module has an address for each client ID. It delivers the message to the address of the specified client. Once delivered, the message waits until the receiving process requests to see its messages.

When a process is ready to receive a message, it calls the IPC module, specifying a timeout value. If there are any messages pending, the IPC module delivers the oldest one to the receiving process. If there are no messages, the IPC module will suspend the receiving process. The process wakes up again when a message comes in for it or the timeout expires, whichever happens first.

Low-Level Controller

The Low-Level Controller (LLC) process manages access to the robot base. Any process wishing to move the robot must send a request to the LLC. Of utmost concern to the Low-Level controller is arbitration between processes wanting to control the base at the same time. For example, the Feature Recognition Module (FRM) may have control of the base and request the LLC to move the robot forward one meter. Someone is walking down the corridor in the direction of the robot. The Avoidance module senses this and requests the LLC to move the robot to one side. The LLC must override the FRM's request because the Avoidance module has a higher priority.

To gain control of the base, a process has to request a transaction with the LLC. Once the transaction is granted, the process has control of the base until it closes the transaction, or a higher priority process request control. When a higher priority process interrupts a transaction, the LLC notifies the original process of the override. The original process has two courses or action at this point. It can surrender by closing its transaction, or it can hang on and listen to warning messages issued by the LLC. As long as the

original process does not close its transaction, the LLC will send warning messages to it, keeping it up to date with requests the overriding process makes. When the overriding process yields control of the base, the original process can try to recover and pick up whatever it was doing before the override.

Sonar Controller

The Sonar Controller (SC) process handles access to sonar data collected by the Denning sonar system. Any process wishing to read the sonar data must issue a request to the SC. The primary purpose of the SC is to avoid collision of messages on the serial line to the Denning sonar system. This could happen if two processes try to read the sonar data directly from the Denning system at the same time.

The Sonar Controller periodically requests data from the Denning sonar system and stores the data in a buffer. When a process requests sonar data, the SC copies the current sonar readings into a response message and sends the message off. The Sonar Controller services requests on a FIFO basis.

Where-Am-I-Module

The Where-Am-I (WAI) Module is a process that maintains a sense of where the robot is by keeping track of the robot's motion. The Low-Level Controller notifies the WAI Module of changes in the robot's rotational and translational velocities. The WAI module uses this information to monitor the robot's position via dead reckoning.

At any time, a process can request the WAI module to record the robot's current position. The position includes the cartesian coordinates of the robot, its orientation, and a 3 by 3 covariance matrix for the estimated position. The WAI Module returns a handle to the position it just recorded. This is useful when a module wants to move the robot to gather data and then return the robot to its starting position. The module can record the starting position, gather its data, and call the library routine "Move-To-Point" to return the robot to its initial position.

Move-To-Point takes a position that the WAI module recorded, and returns the robot to that position. In order to do this, Move-To-Point needs to know the coordinates and orientation of the new position with respect to the robot's current position. A request to the WAI module supplies Move-To-Point with this information.

Avoidance Module

The Avoidance Module is a process that constantly monitors the existence of obstacles in the robot's environment and attempts to move the robot if necessary to avoid collision. The Avoidance Module has a higher priority than any other process in the system. This is to ensure that the Avoidance Module can always override another process' transaction with the LLC to avert a collision.

To avoid collision, the Avoidance Module keeps track of obstacles that appear to be moving closer to the robot. If the robot itself is moving, the Avoidance module may simply request the LLC to reduce the robot's translational velocity when it gets too close to an object. If the robot is not moving, Avoidance may issue a series of requests to the LLC to move the robot out of the way.

Corridor Following Module

The Corridor Following Module (CFM) moves the robot from one Locally Distinctive Place (LDP) to another. The Master Control process (MC) requests the CFM to follow a corridor after the LDP process classifies the current LDP. The Corridor Following Module moves the robot down the corridor until the sonar data indicates the presence of open space to the left or right side of the robot.

When the CFM receives a request to follow a corridor, it assumes the robot is already at the start of the corridor, facing in the right direction. The CFM module attempts to position the robot in the middle of the corridor. As the robot makes its way down the corridor, the CFM periodically adjusts the robot's orientation to keep it moving in a path parallel to the corridor's walls. Also periodically, the CFM checks the sonar data for evidence of open space to one side of the robot.

The CFM halts the robot when it detects open space to the left or right of the robot. Presumably, the open space indicates that the robot is in the vicinity of an LDP. The CFM positions the robot at the end of the corridor (see figure 2), and sends a response message to the MC process. The position the CFM leaves the robot in is the starting position during exploration of the LDP. The LDP process will return the robot to the starting position after invocation of each feature detector.

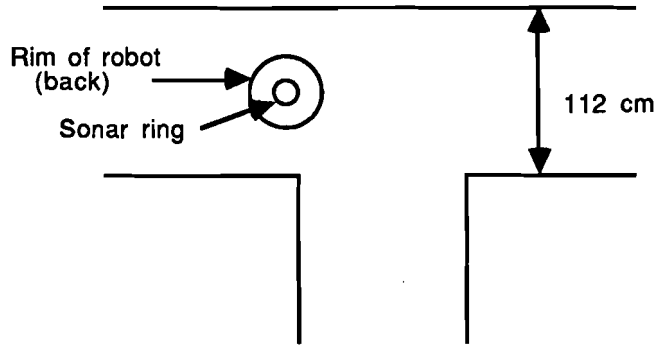


Figure 2

Feature Recognition Module

The Locally Distinctive Place process (LDP) directs exploration of an LDP by requesting the Feature Recognition Module (FRM) to look for corners, walls, and edges in the vicinity of the LDP. Each LDP consists of several features. A feature can be a wall, convex corner, concave corner, or wall edge. A wall edge is a section of wall that ends without meeting another wall to form a corner. For each feature, the FRM has a detector that attempts to find the feature within an area specified by the LDP process.

To look for a feature, the LDP process requests the FRM process to execute one of the feature detectors. The request contains an expected position for the feature with respect to the robot's starting position. The feature detector attempts to move the robot to the expected position. If this attempt fails, the detector aborts and the FRM sends a response message to the LDP process. Otherwise, the detector tries to find the feature by tracking the existence of walls forming it. If at any time the detector fails to

detect a wall that should exist, or detects a wall that should not exist, the FRM sends a response message to the LDP process indicating failure.

Locally Distinctive Place Process

The LDP process utilizes an influence diagram to decide which feature detector to request next in order to classify the current LDP. Behind the LDP's decisions is a set of hypotheses, or possible classifications. Initially all of the hypotheses are equally likely. The LDP process chooses a feature detector that will rule out as many hypotheses as possible. Also behind the LDP process' decision is the cost of each feature detector. At the start, the LDP process can glean as much information from the execution of a less expensive detector as from an expensive one.

As the LDP process receives results from the feature detectors, it prunes the hypothesis space. With each result, the LDP process reevaluates the influence diagram to arrive at a new probability distribution for the hypotheses. A result of success or failure from a detector contributes to the influence diagram. Eventually the LDP process rules out all but one hypothesis. At this point, the LDP process has classified the current LDP.

Master Control Module

The Master Control (MC) process is the driver of the whole system. This process decides what the robot's next action will be in its quest to find and classify the locally distinctive places on a floor of a building. At the start, the Master Control Module assumes the robot is in a corridor. The MC requests the Corridor Following Module (CFM) process to move the robot to the first LDP. Once the robot is at an LDP, the MC can request the LDP process to classify the locally distinctive place.

After the LDP process classifies the current LDP, the Master Control Module invokes the Move-To-Port utility to position the robot in one of the ports of the LDP. A port is an exit point for the robot from the current LDP. Often an LDP has more than one port from which to exit. It is up to the MC to decide in which direction the robot should

go. Once the robot is in a port, the Master Control process can request the CFM to follow the corridor to the next LDP.

Feature Recognition Module

Features

Each locally distinctive place consists of several features. The number of features, type of each feature, and location of each feature determine what the LDP is. There are four distinct features: convex corner, concave corner, wall, and wall edge.

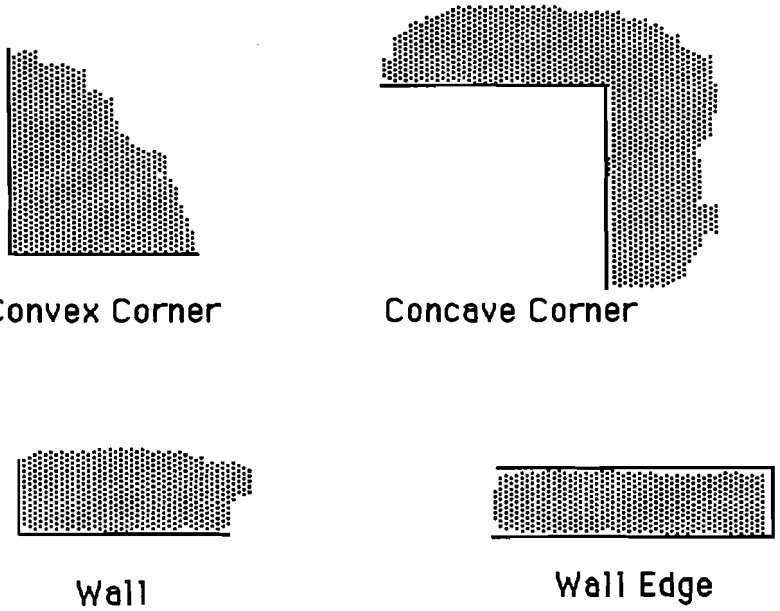
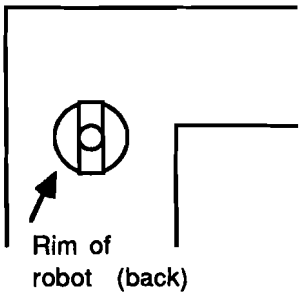


Figure 3

A wall edge is a section of wall that ends without meeting another wall, or a section of wall cut open by a doorway.

The decision to break locally distinctive places into the four features above was not an easy one. Another approach is to consider only wall segments and angles between them. In this scheme, scripts can describe each LDP. For example, the script for the LDP "L junction to right" might look as follows:



Right L Junction

```
1 Look for wall segment to
  to left.
2 Look for wall segment
  ahead.
3 Get angle between wall
  segments.
4 If angle is not 90 degrees
    ELIMINATE Right L
5         etc...
```

Script

Figure 4

The LDP process would have one script for each hypothesis it is considering for the current locally distinctive place. If a step of a script fails, the LDP process would eliminate its corresponding hypothesis. When only one script remains, or when a script successfully completes, its hypothesis becomes valid and the LDP process has classified the current locally distinctive place.

Scripts allow a higher granularity of decision making on the part of the LDP process than features do. The commitment the LDP process makes at each step is lower, and the total amount of information it receives is greater. For example, suppose the first step of the script for "L junction to the right" succeeds, but the second fails. As a result, the LDP process knows there is a wall segment to the left of the robot and no wall segment to the front. If instead the LDP process requests the FRM to execute the feature detector for the concave corner, it would simply receive an indication of failure for the whole feature. The LDP process would lose the information that there is a wall to the left.

There are several disadvantages to the script scheme. The script steps are not independent of one another. It makes no sense to measure the angle between two wall segments before establishing their existence. This makes it difficult to mix and match steps from several scripts in order to reduce the number of hypotheses as quickly as possible. It also makes it difficult to recognize when two scripts overlap, to avoid repetition of identical script steps. Repetition is also a factor when measuring the angle between two

wall segments. The robot has to track the two walls once to establish their existence, and again to measure the angle between them.

Breaking locally distinctive places up into features leads to a cleaner and more flexible design. The features in each hypothesis are independent of each other. The LDP process is free to mix and match features from several hypotheses in order to prune the hypothesis space quickly. Identical features in different hypotheses are easy to single out. Perhaps most importantly, the LDP process does not orchestrate the movement of the robot around the LDP. The FRM controls the movement of the robot, leaving the LDP process to concentrate on which feature to look for next.

Hypotheses

During exploration of an LDP, the LDP process has a collection of hypotheses it is considering for the current locally distinctive place. A hypothesis is a collection of features together with information about where each feature should be and how it should be oriented. Hypotheses also include information about exit points for the robot called ports. The hypothesis for the LDP "L junction to the right" is below.

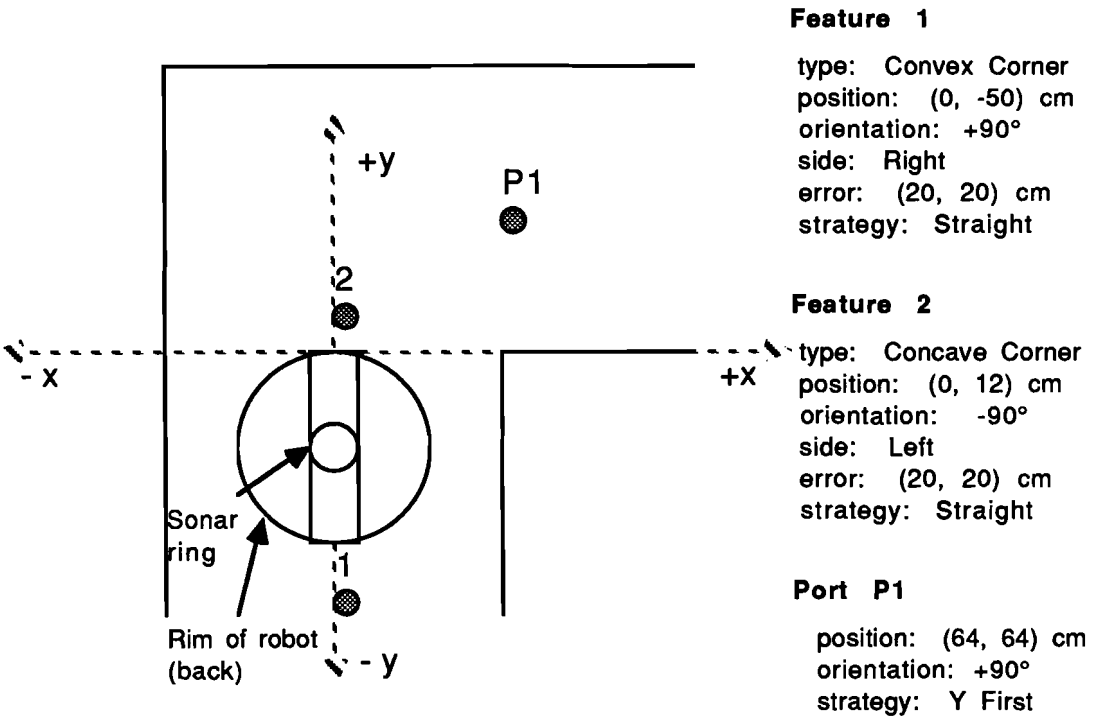


Figure 5

This hypothesis has two features; a convex corner to the right, and a concave corner to the left. There is also a port in the upper right-hand corner of the L junction (marked with the grey circle P1). If the LDP process classifies the current LDP as "L junction to the right", the robot will leave the L junction via port P1.

Start Position

The position of the robot in the diagram above is the start position for the hypothesis. The robot will return to the start position after execution of each feature detector. All other points in the hypothesis are relative to the start position. As the diagram indicates, points are cartesian coordinates with the positive X axis directly to the right of the robot, and the positive Y axis directly ahead of the robot. Orientations in the hypothesis are also relative to the start position. Orientations to the right of the robot are positive, and orientations to the left are negative.

Expected Position and Orientation

When the LDP process requests the feature recognition module to execute one of the feature detectors, it passes along all of the information it has about the feature. The first action the detector will take is move the robot to the expected position of the feature. This is the field "position" of the features in the hypothesis above. The expected position of a feature is a point in the vicinity of the feature, giving the robot an idea of where the feature should be. In order to get the robot to the expected position, the feature detector calls the library routine "Move-To-Point".

The routine "Move-To-Point" takes a parameter telling it the best way to get the robot to the desired position. This "strategy" parameter can take on one of three values: 'straight', 'Y first', or 'X first'. The "strategy" field of a feature in a hypothesis lets the feature detector know the best way to get the robot to the expected position of the feature. Sometimes the expected position is around the corner from the start position. In this case, it is best to move the robot the distance along the Y axis first, followed by the distance along the X axis. At other times the expected position is directly ahead or directly behind the robot. Moving the robot straight to the position is more efficient.

Once the robot is at the expected position, the feature detector turns the robot to the orientation specified in the "orientation" field of the feature. At this point, the robot should be facing a wall that makes up part of the feature. If there is no wall, the FRM will return an indication of failure to the LDP process. Otherwise, the feature detector will continue its endeavor to confirm or disconfirm the existence of the feature by attempting to move along the walls forming it.

Side Field

Suppose the robot is at the expected position of the convex corner in the hypothesis for "L junction to the right" (the grey circle labelled 1 in the diagram). The robot is facing the wall and has already detected it. The next step is to move along the walls forming the corner. How does the robot know which way to turn? The corner could be to the left or to the right. The "side" field of the feature

provides the robot with the information it needs. The value of the side field for the convex corner in the hypothesis above is 'Right'. This means the feature is to the right of the starting position in the hypothesis. Concave corners and wall edges also take a side field. Walls stretch to both sides of their expected position, rendering the side field unnecessary.

Error Field

The "error" field of a feature allows for error in the start position of the robot. If the robot is not exactly at the start position, it will not get to the exact location of the expected position. As the robot follows the wall forming part of a corner, it may reach the corner sooner or later than it expects. If the robot does not reach the corner within a distance bounded by the error field, the detector will fail and the FRM will relay the failure to the LDP process.

Ports

After the LDP process has classified the current LDP, the MC process will invoke the Move-To-Port utility to move the robot to one of the ports in the hypothesis corresponding to the LDP. Each port in a hypothesis has a position and a strategy for getting to the position. The Move-To-Port utility calls Move-To-Point to get the robot to the port position. Move-To-Port orients the robot so that it is heading down the corridor of the port, and moves the robot far enough down the corridor for the corridor following module to take over.

Calculation of Expected Positions

To calculate the expected positions for features in a hypothesis, there have to be some assumptions about the size of the robot and the width of the average corridor. The FRM module makes the following assumptions:

1. The stabilizing ring around the robot is 60 centimeters in diameter.
2. The sonar ring has a diameter of 16 centimeters.
3. Corridors are roughly 112 centimeters wide.

If the robot is in the middle of a corridor, the measurements are as follows:

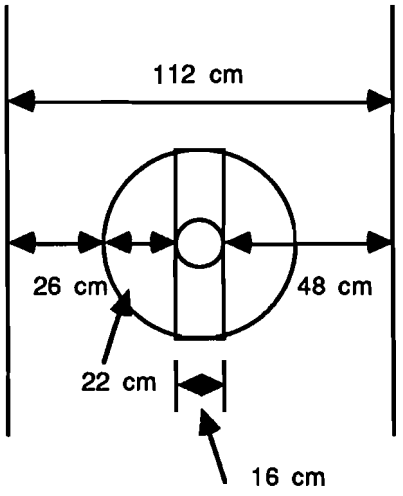


Figure 6

The coordinates of the expected position for a feature in a hypothesis depend on the feature type and the placement of the expected position with respect to the start position. Convex corners and wall edges have their expected positions so that the sonar at the front of the robot is 50 centimeters away from the corner. The expected position along concave corners is 100 centimeters between the front sonar and the corner. The expected position along a wall is 48 centimeters from the midpoint of the wall. The midpoint of a wall is the point directly opposite the midpoint of the corridor opposite the wall. The two LDPs below diagram the expected positions for each feature.

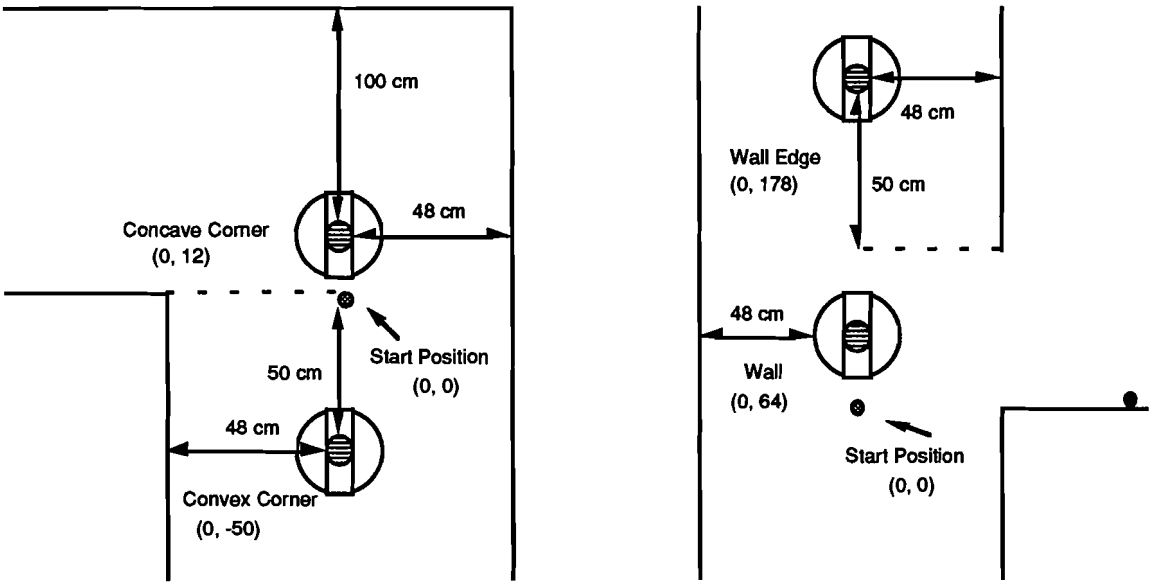


Figure 7

As an example, consider the convex corner of the L junction. At the start position, the sonar at the front of the robot should line up with the corner. In order for the front sonar to be 50 centimeters from the corner, the robot has to move -50 units along the Y axis. The left perpendicular sonar is already 48 centimeters from the wall to the left. No translation along the X axis is necessary. The coordinates of the expected position for the corner are (0, -50).

Feature Detectors

The feature detectors of the feature recognition module can either succeed, fail, or abort in an attempt to detect a feature. A detector fails when it cannot detect a wall that should form part of the feature, or when it detects a wall that should not form part of the feature. A detector aborts when there is an avoidance override. An abort neither confirms nor disconfirms the existence of a feature. When a detector succeeds, it returns points along the walls of the feature. Higher level processes can use the points to orient the feature with respect to other features of the current LDP.

Wall Detector

The wall detector attempts to track a wall as the detector guides the robot first to the right of the expected position and then to the left. If at any time the robot ceases to detect a wall, the wall detector will return with a failure. When it succeeds, the wall detector returns three points along the wall; the point at the expected position, and one point to either side, equidistant from the first.

Convex Corner Detector

The detector for convex corners starts at the expected position and attempts to move the robot around the corner. The detector calculates the maximum allowable distance between the expected position and the edge of the corner using the "position error" field of the corner. If the robot does not reach the corner edge within this distance, the detector will fail. Otherwise the detector turns the robot 90 degrees and attempts to follow the wall at the far side of the corner. If at any time the sonars fail to detect the walls that should make up the corner, the detector will fail. When it succeeds, the convex corner detector returns three points along the corner to the LDP process. One point is at the corner itself and the other two are to either side of the corner, equidistant from it.

Concave Corner Detector

The concave corner detector is similar to the detector for convex corners. The detector attempts to move the robot around the corner, tracking the walls forming it. The detector calculates the maximum allowable distance the robot should travel before coming within a certain distance of the opposite wall of the concave corner. If the robot exceeds this distance as it moves from the expected position towards the corner, the detector will fail. Otherwise the detector turns the robot 90 degrees and attempts to follow the opposite wall. If the sonars fail to detect the walls the robot tracks as it moves around the corner, the detector will fail. When the detector succeeds, it returns three points along the corner; the first at the corner itself, and the other two to either side, equidistant from the first.

Wall Edge Detector

The detector for wall edges is similar to the detectors for convex and concave corners. It calculates the maximum allowable distance between the expected position and the edge, and it attempts to move the robot around the edge as it tracks the walls forming it. This detector also returns three points along the wall edge when it succeeds. The challenge is lining the robot up with the midpoint of the edge. As wall edges usually form part of a doorway, the detector has to take extra caution in moving the robot around the edge. Attempting to line the robot up with the midpoint of the edge allows the detector to position the robot more accurately with respect to the edge. To position the robot at the midpoint of an edge, the detector slowly translates the robot forward until the sonar perpendicular to the edge detects it. The robot continues to move forward until the perpendicular sonar exceeds a threshold. The detector then translates the robot backwards a fixed distance, positioning it roughly at the center of the wall edge.

Calculation of Feature Costs

When deciding which feature of which hypothesis to look for next, the locally distinctive place process takes into account the cost of executing each feature detector. The LDP process calculates this cost from the distance between the expected position of the feature and the start position, and the cost of executing the detector for the feature. The cost of each feature detector is the average number of centimeters the robot would travel, added to the average number of degrees the robot would turn if the detector were successful.

As it turns out, the wall detector is the least costly. It does not have to turn any corners, reducing the number of degrees rotated by 180. The fact that there are no corners also reduces the amount of translation. The detector for concave corners is second in cost, followed closely by the detector for convex corners. The robot moves around the inside of a concave corner, slightly reducing the number of centimeters to travel. The detector for wall edges is by far the most costly. Lining the robot up with the midpoint of the edge itself accounts for the extra cost.

Experience With Sonar

Important Algorithms

There are several algorithms of great importance to the feature recognition module. Two of the algorithms detect walls. The first is for short range wall detection. The FRM uses this algorithm to track walls as the robot moves roughly parallel to them. The second algorithm is for longer range wall detection. The FRM uses this algorithm when the robot is at the expected position of a feature and is looking for a wall ahead of it. The third algorithm of note attempts to orient the robot parallel with a wall. The FRM uses this algorithm before following a wall to keep the robot from veering into the wall as it follows it.

Wall-Exist

The Wall-Exist algorithm is for short range wall detection. It needs to be fast because it is often called in a loop that should exit as soon as the algorithm fails to detect a wall. Wall-Exist takes one argument; the side of the robot on which to look. This reduces the number of sonar readings Wall-Exist has to consider. Wall-Exist only considers the five sonars most closely perpendicular to the side of the robot on which to look. If the side to look on is the left side, and the perpendicular sonar to the left of the robot is number 12, Wall-Exist will look at sonars 10 through 14.

Wall-Exist looks for a sequence of sonars that decrease in value and then increase. It does this by looping through the five sonar values, looking for the sonar with the smallest value. Wall-Exist then counts the number of increasing sonars to either side of the smallest value. If there are three or more sonars in the decreasing/increasing sequence, Wall-Exist succeeds.

Wall-In-Sight

Wall-In-Sight is for longer-range wall detection. This algorithm can take its time in deciding whether there is a wall or not.

Wall-In-Sight takes no arguments; it always looks for a wall to the front of the robot. The algorithm takes advantage of the time it has by rotating the robot to obtain a ring of sonar data with twice as many readings as transducers. This helps to reduce the effect of spurious readings and helps to make more evident the sonar signature of a wall when one exists.

Of the 32 sonar values available, Wall-In-Sight considers the 11 values that are closest to the front of the robot. If there is a sequence of four or more sonar values that fall within a fixed range of each other, and if the front sonar falls within a fixed range of the smallest sonar in the sequence, Wall-In-Sight succeeds. The fixed range allows for the increase in value of the sonars as they deviate from the perpendicular, and allows for small errors in the sonar readings. The stipulation that the front sonar be close in value to the smallest sonar in the sequence ensures that the wall stretches in front of the robot.

Orient-Parallel

Orient-Parallel attempts to align the robot parallel to a wall. The routine takes one argument; the side of the robot the wall is on. Orient-Parallel considers only those sonars to the given side of the robot. Accuracy is of primary concern to this routine. The more closely parallel the robot is to a wall, the less the chance is that the robot will veer into the wall when following it.

Orient-Parallel adjusts the robot's orientation in two phases. In the first phase, Orient-Parallel attempts to make the sonar perpendicular to the robot at the given side the sonar with the smallest value. It does this by first identifying the sonar with the smallest value. If the smallest sonar is more towards the back of the robot than the front, Orient-Parallel turns the front of the robot towards the wall five degrees. If the smallest sonar is more towards the front of the robot than the back, Orient-Parallel turns the front of the robot away from the wall five degrees. The routine continues turning the robot by five degrees until the sonar perpendicular to the robot at the given side has the smallest value.

In the second phase, Orient-Parallel attempts to balance the values of the sonars to either side of the smallest sonar. If the sonar

closer to the back of the robot is smaller than the one closer to the front, the front of the robot turns toward the wall one degree. If the sonar closer to the front of the robot is smaller, the front of the robot turns away from the wall one degree. This continues until the sonars to either side of the smallest sonar are equal in value, or until the robot has changed its direction of turn five times. The last stipulation keeps the robot from thrashing.

Using Sonar Data

The first step in coping with sonar error is to expect error by building into the system error-minimizing procedures. Taking the average or mode of several sonar readings before working with the data helps reduce the effect of spurious errors. The feature recognition module always works with data averaged over twelve readings. For further error filtering, a routine can work with several averaged rings of data before reaching a conclusion. As an example, both Wall-Exist and Wall-In-Sight try three times before concluding there is no wall. In the FRM it is usually better to conclude there is a wall when there is not than to conclude there is no wall when there is. Testing has shown Wall-Exist to fail on the first try but succeed (correctly) on the second.

Another approach in coping with sonar error is to leave room for error when working with sonar data. Often it is not important whether an object is 10 centimeters away or twenty centimeters away, but that it is around 15 centimeters away. For example, Wall-In-Sight looks for a sequence of four sonars that have roughly the same value. It does this by making sure that contiguous sonars fall within a certain range of each other. A constant, "ERR_THRESH", defines the range to be plus or minus 20 centimeters.

Sonar error is sometimes easy to avoid. The most reliable readings come from sonars perpendicular to the object to detect. There are times when an algorithm can count on certain sonars as being the most closely perpendicular ones. For example, when a feature detector wants to follow a wall, it will first call Orient-Parallel to orient the robot parallel to the wall it wants to follow. While the robot follows the wall, Wall-Exist monitors its presence. As the distance the robot travels is small, Wall-Exist can rely on certain

sonars as being the most closely perpendicular, and takes advantage of it by only considering the values of those sonars.

Trial and error is perhaps the best way to deal with sonar peculiarities. It is impossible to know what sonar data to expect from a particular environment. The Wall-Exist algorithm changed several times as testing revealed inadequacies in earlier versions. Testing also prompted change to the Wall-In-Sight algorithm. When the robot was standing in front of a corridor, slightly to the left of it, Wall-In-Sight picked up a convex corner to the left and concluded that there was a wall in front of the robot. This led to the stipulation that the front sonar have roughly the same value as the smallest sonar in the sequence of values Wall-In-Sight finds.

Enhancements

System-Wide Enhancements

There are several enhancements that would increase the performance of the LDP system for Huey as a whole. One rather simple enhancement is to form an initial list of hypotheses for the LDP process. In the present system, the LDP process considers all of the hypotheses to be equally likely at the start of LDP exploration. The CFM module could return an indication of where it detected open space at the end of a corridor; to the left of the robot, to the right, or to the left and right. If the LDP process had this information, it could cut its hypothesis space by two-thirds.

Another simple extension of the current system is elimination of the starting position from LDP exploration. Rather than returning to a starting position after execution of a feature detector, the robot could move directly to the expected position of the next feature to detect. This would involve some changes to the LDP module to calculate the position to move the robot to given the robot's current position and the expected position of the next feature.

Incorporation of partial results from the feature detectors is a more complex enhancement. This involves significant modification of the FRM and LDP processes. The FRM process could return partial results from a failed detector. For example, if the convex corner detector found the first wall of the alleged corner, but could not find the adjoining perpendicular wall, it could return the partial success. The LDP process would know that the first wall exists and could use this information in selecting the next detector.

A final system-wide enhancement is the addition of a heat sensor for detecting warm bodies. The avoidance module could distinguish between animate and inanimate obstacles. This would allow the FRM module to return a failure rather than an abort in some situations.

Enhancements to the FRM Module

If collection of points along features is not a priority, it is possible to significantly reduce the number of translations and rotations in the convex corner, concave corner, and wall edge detectors. At present, the detectors move the robot around the corner or edge twice. This allows the robot to position itself at a known point along the wall of the expected position in order to collect a point. If the points are not necessary, the detectors can simply move the robot around the corner or edge once. Another possibility is reducing the distance the robot travels around the corners and wall edges.

Recovery from an avoidance override is another possible enhancement to the FRM module. This requires the addition of a heat sensor to the robot so the FRM can distinguish between temporary and permanent obstacles. As an example, the expected position of a feature may be blocked by a wall because the feature does not exist. The FRM relies on the avoidance process to flag these situations. At present, the feature detector will return an indication of abort rather than failure because there is no way to tell whether the obstacle is temporary or permanent. If the detector knew an obstacle was temporary, it could attempt to recover from the avoidance override. The detector could return the robot to a known position along the feature and resume its activity from that point. If there is another avoidance override, the detector could give up and return an abort to the LDP process.

The addition of simultaneous translation and rotation to the feature detectors would reduce the cost of the detectors and make the robot appear more lifelike. This enhancement is not a simple one. In order for the robot to make the turn around a corner, the convex corner detector would have to detect the corridor formed by the corner sooner than it does as present. The concave corner detector would be the easiest to modify. The robot could detect and estimate the distance to the opposite wall of the corner in plenty of time to turn the robot.

Another possible enhancement is the addition of certainty measures to the results of the feature detectors. Instead of returning success or failure, a detector would return a number representing how certain it is that the feature exists. A low number would indicate that the feature most likely does not exist, and a high number would

indicate the opposite. The reliability of the wall detection algorithms could be measured through testing. When a detector fails because it could not detect a wall, it could take into account the reliability of the wall detection algorithm it used. When a detector fails because of an avoidance override due to a "permanent" obstacle, the detector could account for the probability that the obstacle was permanent.

In the current system the probability that a detector is wrong when it reports success is negligible. This is due to the fact that the robot moves around the perimeter of each feature, and the fact that the characteristics of each feature are entirely different. The probability of a false positive would increase, however, with a reduction in the number of translations and rotations in the detectors. For example, if the distance the robot travels around corners and edges is small enough, the robot could mistake a wall edge for a convex corner. The convex corner detector could account for this possibility in the measure of certainty it returns.

References

[1] Robert A. Chekaluk (1990)
Using Influence Diagrams in Recognizing Locally-Distinctive Places,
Master's Thesis, Brown University, Dept. of Computer Science.

[2] James L. Crowley (1985)
Dynamic World Modeling for an Intelligent Mobile Robot Using a Rotating Ultra-Sonic Ranging Device,
IEEE Proc. International Conference on Robotics and Automation,
pp. 128-135.

[3] Alberto Elfes (1986)
A Sonar-Based Mapping and Navigation System,
IEEE Proc. International Conference on Robotics and Automation,
pp. 1151-1156.

[4] R. Kuc and M. W. Siegel (1987)
Efficient Representation of Reflecting Structures for a Sonar Navigation Model,
IEEE Proc. International Conference on Robotics and Automation,
pp. 1916-1923.

[5] R. Kuc and B. Barshan (1989)
Navigating Vehicles Through an Unstructured Environment With Sonar,
IEEE Proc. International Conference on Robotics and Automation,
pp. 1422-1426.

[6] Benjamin J. Kuipers (1978)
Modeling Spatial Knowledge,
Cognitive Science, 2:pp. 129-153.

[7] Benjamin J. Kuipers and Yung-Tai Byun (1988)
A Robust, Qualitative Method for Robot Spatial Learning,
Proceedings of AAAI-88, pp. 774-779.

[8] David Miller (1985)
A Spatial Representation System for Mobile Robots,
IEEE Proc. International Conference on Robotics and Automation,
pp. 122-127.

[9] H. P. Moravec and A. Elfes (1985)
High Resolution Maps from Wide Angle Sonar,
IEEE Proc. International Conference on Robotics and Automation,
pp. 116-121.

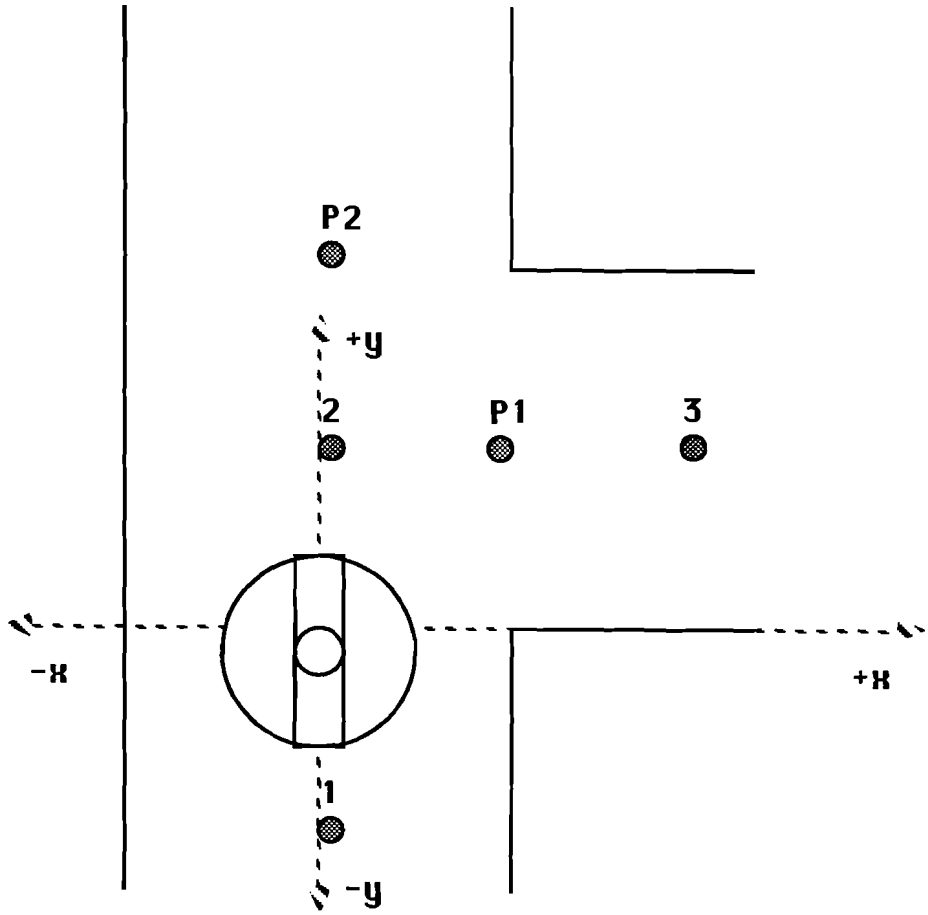
[10] Scott A. Walter (1987)
The Sonar Ring: Obstacle Detection for a Mobile Robot,
IEEE Proc. International Conference on Robotics and Automation,
pp. 1574-1579.

Appendices

Appendix A: Hypotheses

Appendix B: Code for the Feature Recognition Module

feature.c	-	the feature detectors.
frm.c	-	main loop for the FRM process.
frm_utils.c	-	routines interfacing with the LLC and WAI.
sonar.c	-	sonar data interpretation routines.
sonar_utils.c	-	code to preprocess sonar data.



Feature 1

type: Convex Corner
position: (0, -50) cm
orientation: +90°
side: Right
error: (20, 20) cm
strategy: Straight

Feature 2

type: Wall
position: (0, 64) cm
orientation: -90°
side: N/A
error: N/A
strategy: Straight

Feature 3

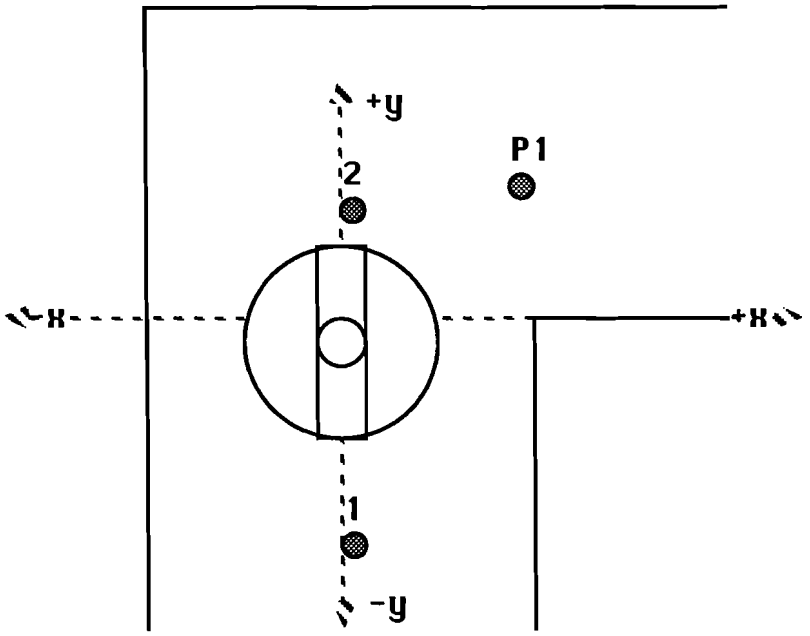
type: Convex Corner
position: (114, 64) cm
orientation: 0°
side: Right
error: (20, 20) cm
stategy: Y First

Port P1

position: (64, 64) cm
orientation: +90°
strategy: Y First

Port P2

position: (0, 128) cm
orientation: 0°
strategy: Straight



Feature 1

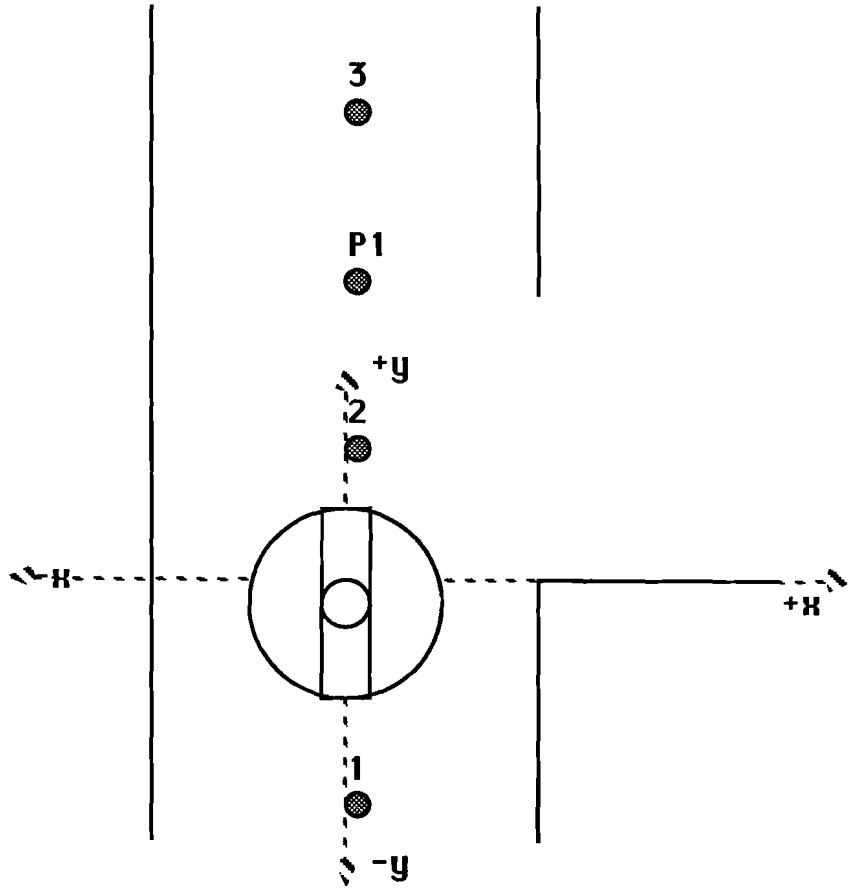
type: Convex Corner
position: (0, -50) cm
orientation: +90°
side: Right
error: (20, 20) cm
strategy: Straight

Port P1

position: (64, 64) cm
orientation: +90°
strategy: Y First

Feature 2

type: Concave Corner
position: (0, 12) cm
orientation: -90 °
side: Left
error: (20, 20) cm
strategy: Straight



Feature 1

type: Convex Corner
position: (0, -50) cm
orientation: +90°
side: Right
error: (20, 20) cm
strategy: Straight

Feature 2

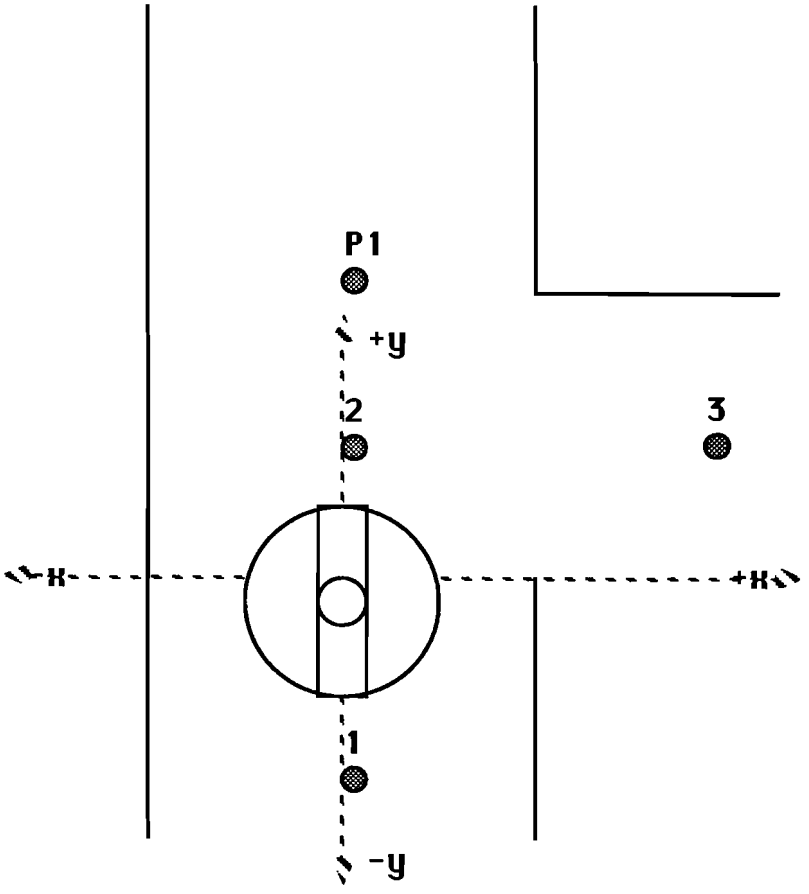
type: Wall
position: (0, 64) cm
orientation: -90°
side: N/A
error: N/A
strategy: Straight

Feature 3

type: Wall Edge
position: (0, 178) cm
orientation: +90°
side: Left
error: (20, 20) cm
stategy: Straight

Port P1

position: (0, 128) cm
orientation: 0°
strategy: Straight



Feature 1

type: Wall Edge
position: (0, -50) cm
orientation: +90°
side: Right
error: (20, 20) cm
strategy: Straight

Feature 2

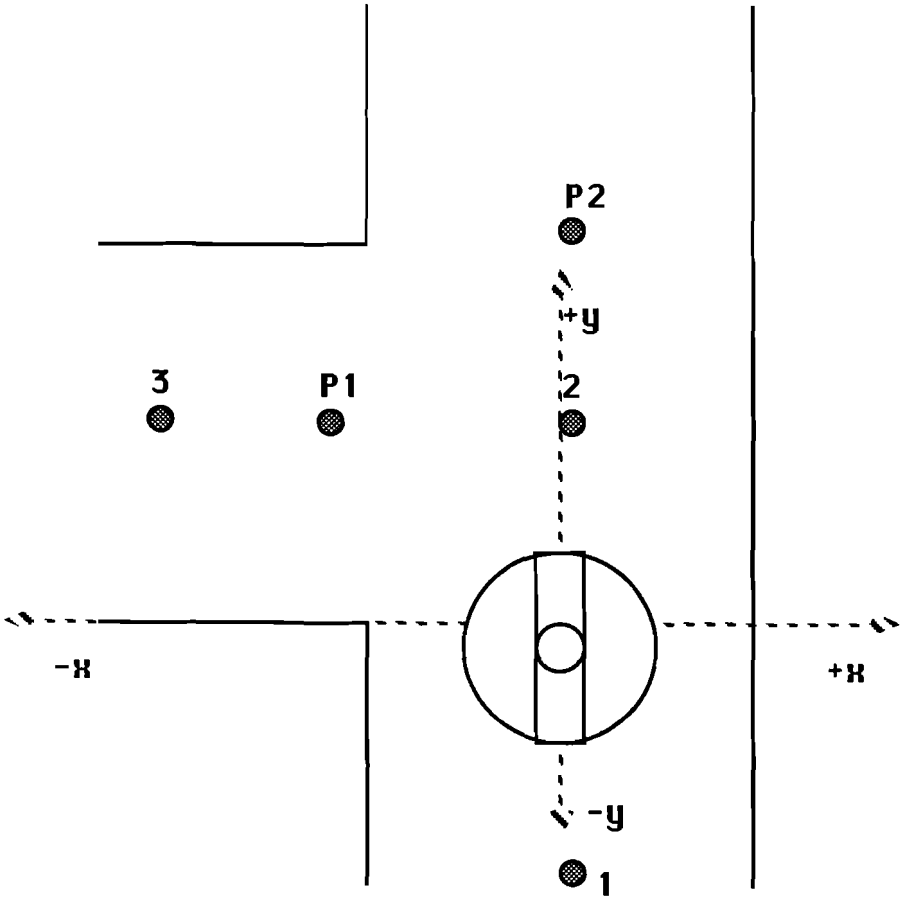
type: Wall
position: (0, 64) cm
orientation: -90°
side: N/A
error: N/A
strategy: Straight

Feature 3

type: Convex Corner
position: (114, 64) cm
orientation: 0°
side: Right
error: (20, 20) cm
stategy: Y First

Port P1

position: (0, 128) cm
orientation: 0°
strategy: Straight



Feature 1

type: Convex Corner
position: (0, -50) cm
orientation: -90°
side: Left
error: (20, 20) cm
strategy: Straight

Feature 2

type: Wall
position: (0, 64) cm
orientation: +90°
side: N/A
error: N/A
strategy: Straight

Feature 3

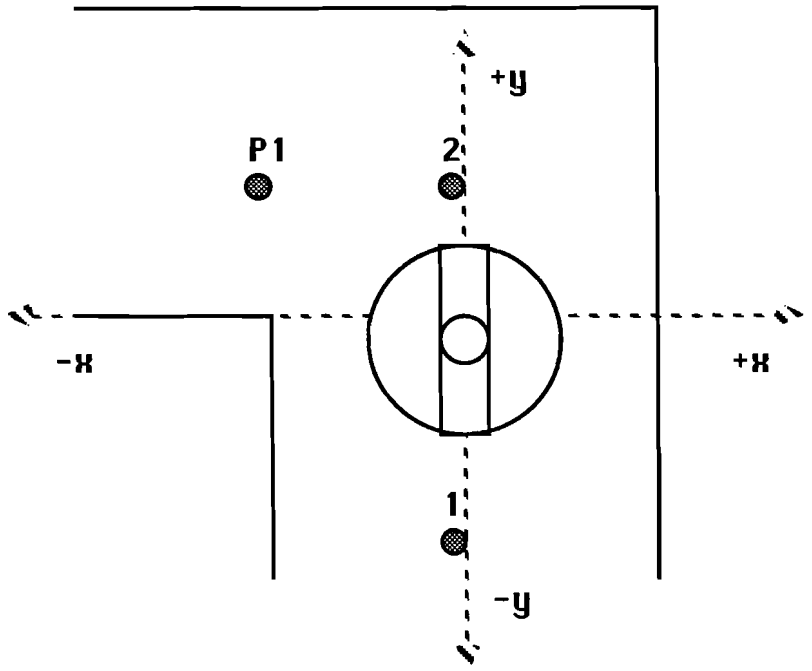
type: Convex Corner
position: (-114, 64) cm
orientation: 0°
side: Left
error: (20, 20) cm
stategy: Y First

Port P1

position: (-64, 64) cm
orientation: -90°
strategy: Y First

Port P2

position: (0, 128) cm
orientation: 0°
strategy: Straight



Feature 1

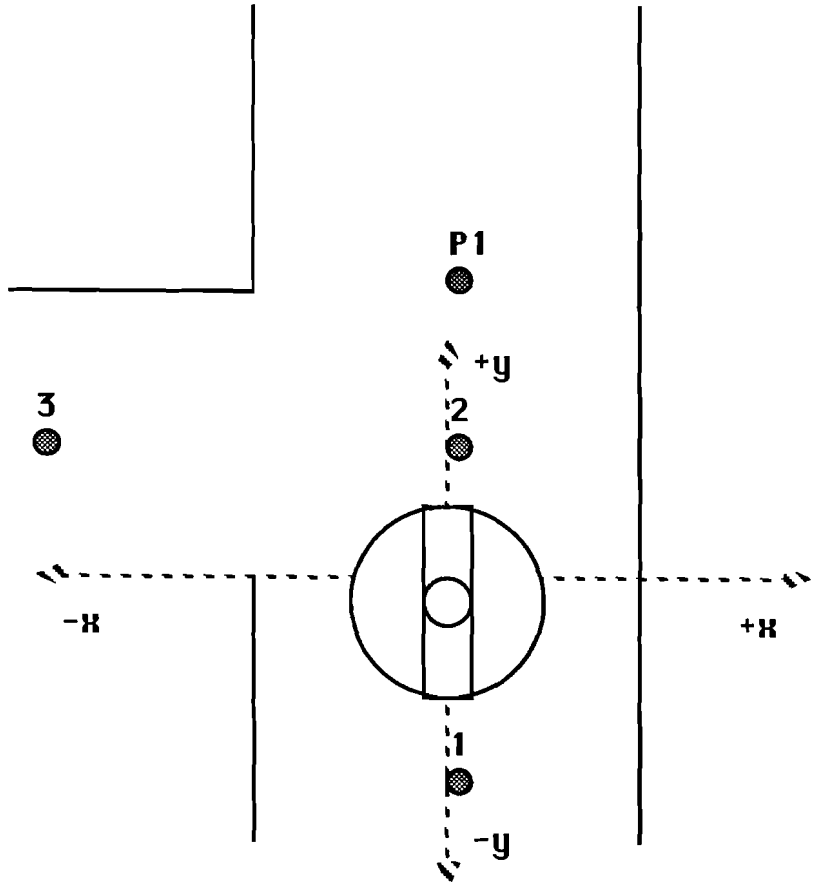
type: Convex Corner
position: (0, -50) cm
orientation: -90°
side: Left
error: (20, 20) cm
strategy: Straight

Port P1

position: (-64, 64) cm
orientation: -90°
strategy: Y First

Feature 2

type: Concave Corner
position: (0, 12) cm
orientation: +90°
side: Right
error: (20, 20) cm
strategy: Straight



Feature 1

type: Wall Edge
position: (0, -50) cm
orientation: -90°
side: Left
error: (20, 20) cm
strategy: Straight

Feature 2

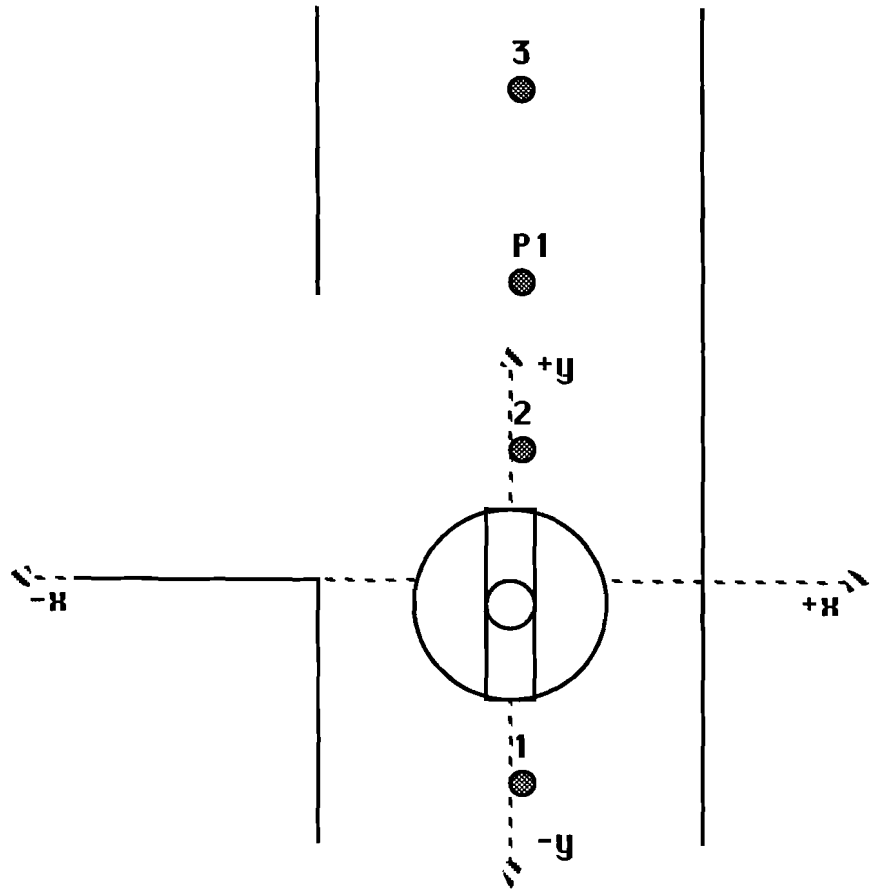
type: Wall
position: (0, 64) cm
orientation: +90°
side: N/A
error: N/A
strategy: Straight

Feature 3

type: Convex Corner
position: (-114, 64) cm
orientation: 0°
side: Left
error: (20, 20) cm
stategy: Y First

Port P1

position: (0, 128) cm
orientation: 0°
strategy: Straight



Feature 1

type: Convex Corner
position: (0, -50) cm
orientation: -90°
side: Left
error: (20, 20) cm
strategy: Straight

Feature 2

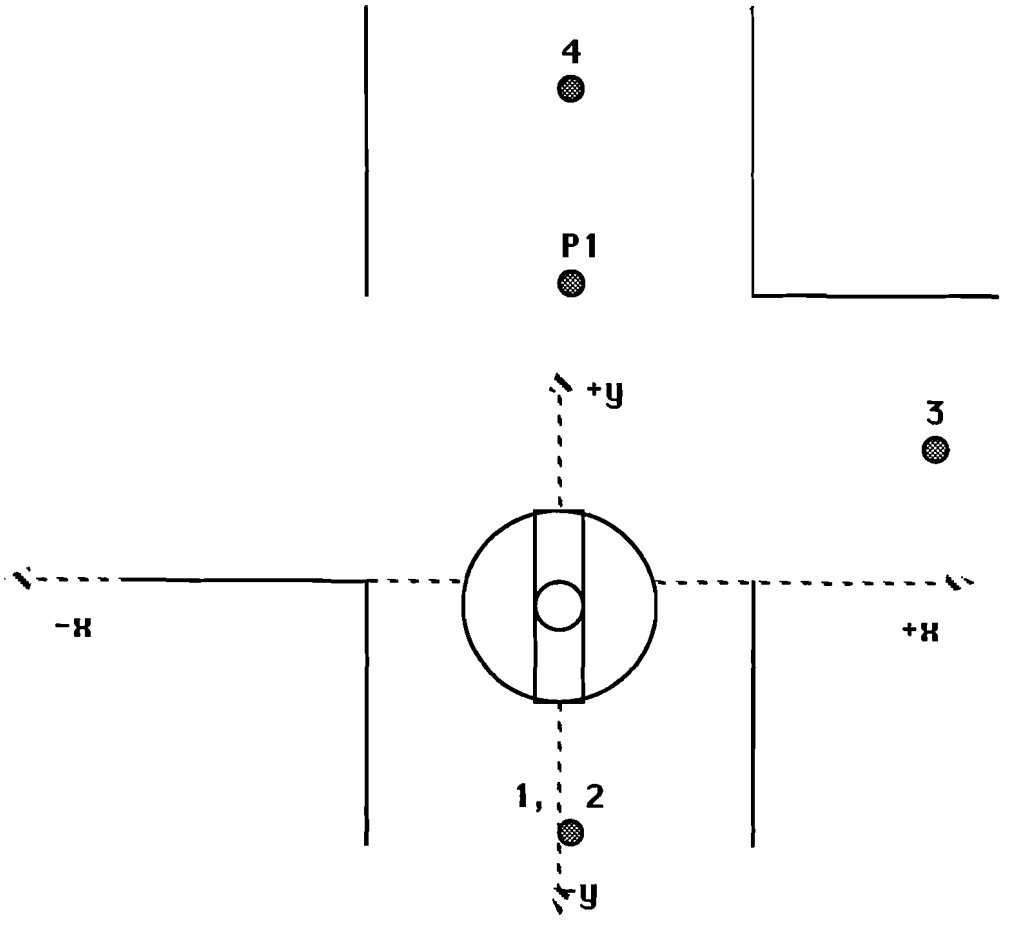
type: Wall
position: (0, 64) cm
orientation: +90°
side: N/A
error: N/A
strategy: Straight

Feature 3

type: Wall Edge
position: (0, 178) cm
orientation: -90°
side: Right
error: (20, 20) cm
stategy: Straight

Port P1

position: (0, 128) cm
orientation: 0°
strategy: Straight



Feature 1

type: Convex Corner
position: (0, -50) cm
orientation: -90°
side: Left
error: (20, 20) cm
strategy: Straight

Feature 3

type: Convex Corner
position: (114, 64) cm
orientation: 0°
side: Right
error: (20, 20) cm
stategy: Y First

Port P1

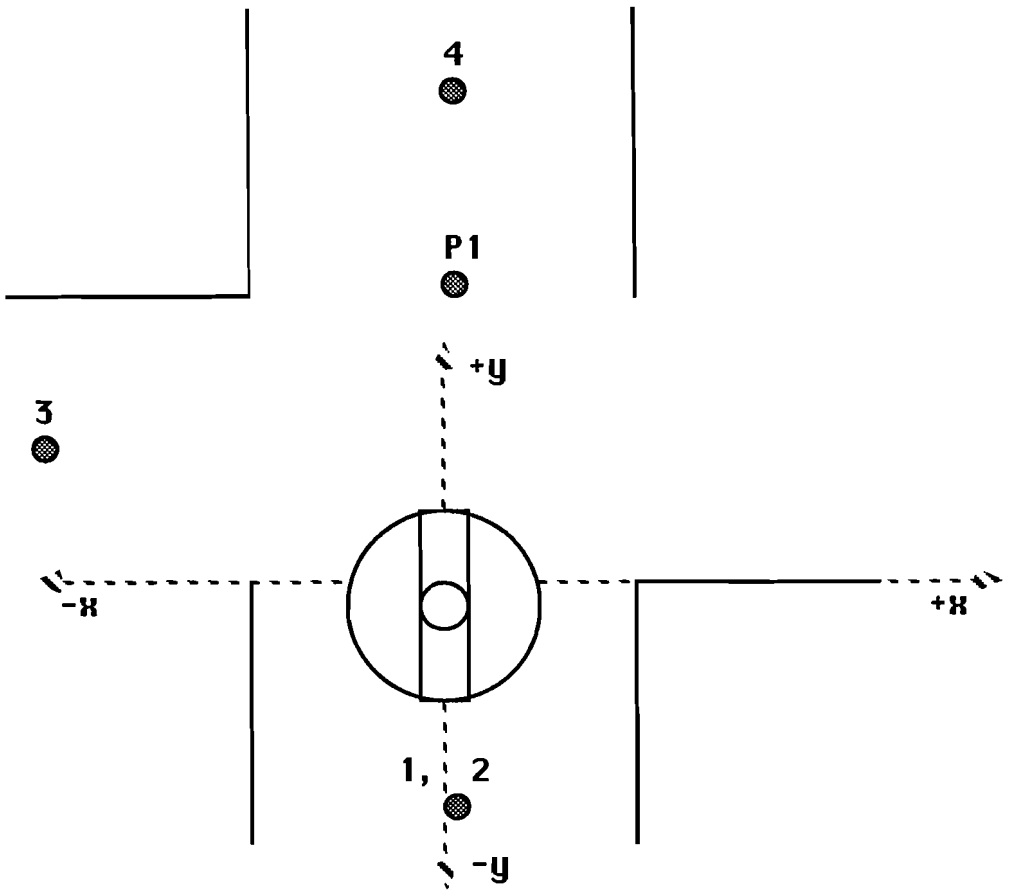
position: (0, 128) cm
orientation: 0°
strategy: Straight

Feature 2

type: Wall Edge
position: (0, -50) cm
orientation: +90°
side: Right
error: (20, 20)
strategy: Straight

Feature 4

type: Wall Edge
position: (0, 178) cm
orientation: -90°
side: Right
error: (20, 20) cm
stategy: Straight



Feature 1

type: Wall Edge
position: (0, -50) cm
orientation: -90°
side: Left
error: (20, 20) cm
strategy: Straight

Feature 3

type: Convex Corner
position: (-114, 64) cm
orientation: 0°
side: Left
error: (20, 20) cm
stategy: Y First

Port P1

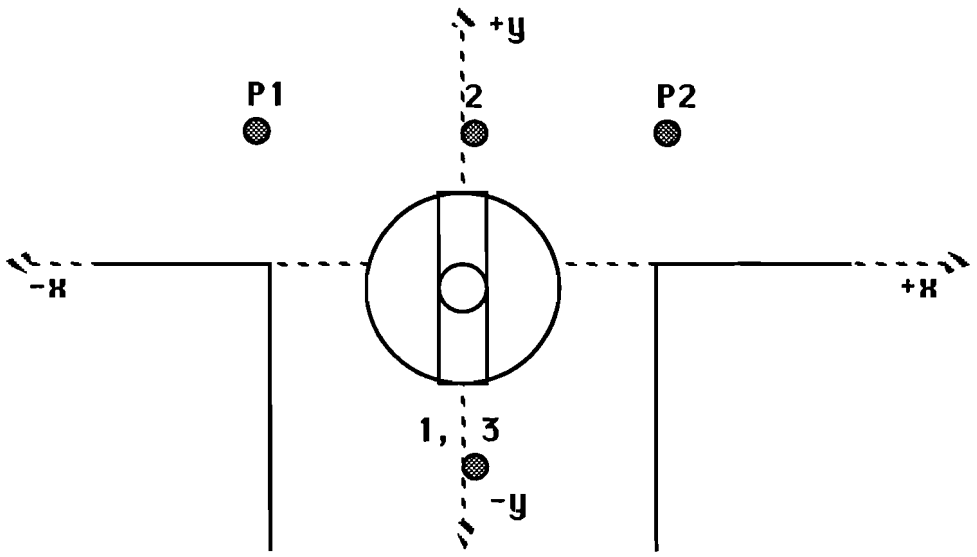
position: (0, 128) cm
orientation: 0°
strategy: Straight

Feature 2

type: Convex Corner
position: (0, -50) cm
orientation: +90°
side: Right
error: (20, 20)
strategy: Straight

Feature 4

type: Wall Edge
position: (0, 178) cm
orientation: +90°
side: Left
error: (20, 20) cm
stategy: Straight



Feature 1

type: Convex Corner
position: (0, -50) cm
orientation: -90°
side: Left
error: (20, 20) cm
strategy: Straight

Feature 2

type: Wall
position: (0, 64) cm
orientation: 0°
side: N/A
error: N/A
strategy: Straight

Feature 3

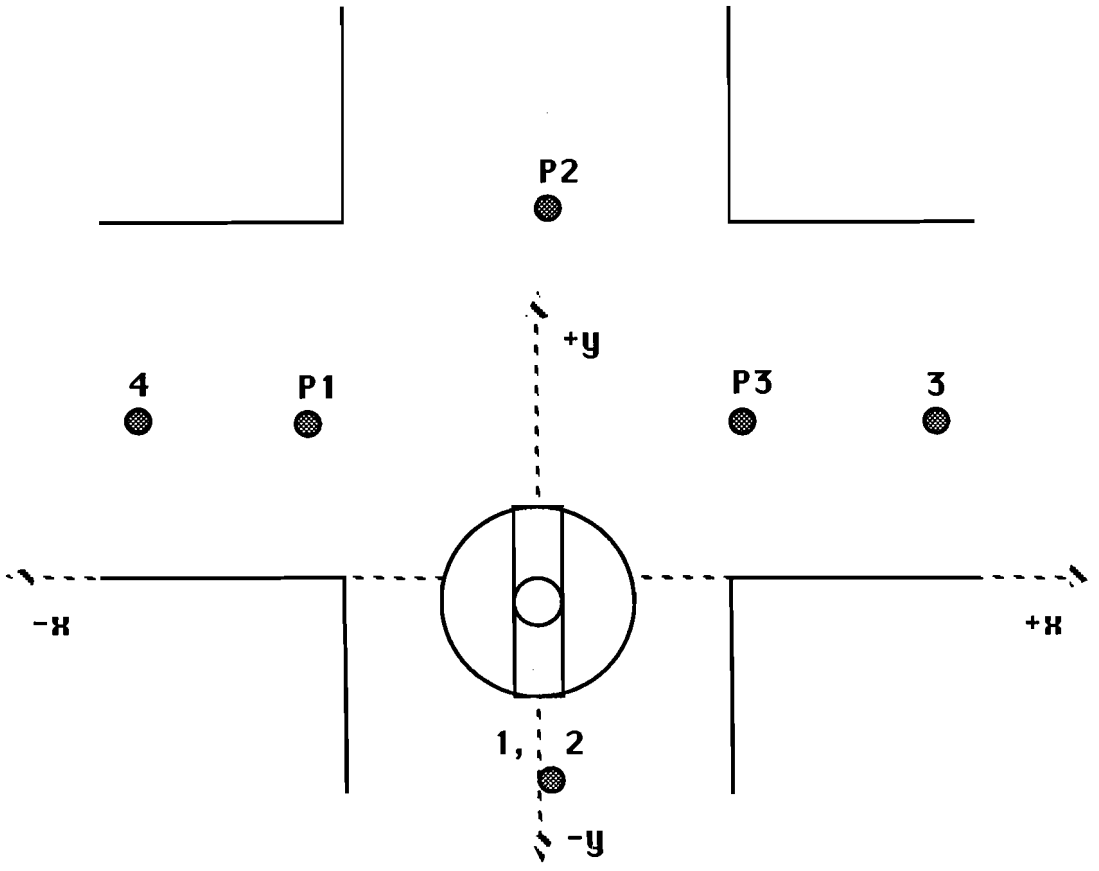
type: Convex Corner
position: (0, -50) cm
orientation: +90°
side: Right
error: (20, 20) cm
strategy: Straight

Port P1

position: (-64, 64) cm
orientation: -90°
strategy: Y First

Port P2

position: (64, 64) cm
orientation: 90°
strategy: Y First



Feature 1

type: Convex Corner
position: (0, -50) cm
orientation: -90°
side: Left
error: (20, 20) cm
strategy: Straight

Feature 2

type: Convex Corner
position: (0, -50) cm
orientation: +90°
side: Right
error: (20, 20)
strategy: Straight

Feature 3

type: Convex Corner
position: (114, 64) cm
orientation: 0°
side: Right
error: (20, 20) cm
stategy: Y First

Feature 4

type: Convex Corner
position: (-114, 64) cm
orientation: 0°
side: Left
error: (20, 20) cm
stategy: Y First

Port P1

position: (-64, 64) cm
orientation: -90°
strategy: Y First

Port P2

position: (0, 128) cm
orientation: 0°
strategy: Straight

Port P3

position: (64, 64) cm
orientation: +90°
strategy: Y First

```
/*
*
*  frm.h
*
*  All of the files of the feature recognition module include this
*  file.
*/

#define uint8  unsigned char
#define uint16 unsigned short

#define TRUE 1
#define FALSE 0

#define min(a,b)        ((a) < (b) ? (a) : (b))
#define sign(a)         ((a) ? (((a) >= 0) ? 1 : -1) : 0)

#define FRM_ERROR -1    /* Return code for error */
#define FRM_OK     1    /* Return code for success */

#define MY_PRIORITY 5   /* Priority of the FRM process */


/*
*  USE_LDP
*
*  Conditional compile flag for inclusion of code to interface with
*  the LDP module.  The LDP (Locally Distinctive Place) module chooses
*  which feature detector to invoke next in order to identify the
*  current location as a T junction, L junction, intersection etc.
*
*  To make the Menu program (Make_menu) the line below should be
*  commented out.
*/

#define USE_LDP 1



/*** Timeouts for communication via the IPC module ***/

#define HALF_SECOND 10
#define ONE_SECOND  20
#define TEN_SECONDS 200
#define ONE_MINUTE  1200
#define FOREVER    12000000


/*** Flags for debugging code ***/

#define DEBUG_FRM   1  /* Code in frm.c and feature.c */
#define DEBUG_MSGS  0  /* Code interfacing with other processes via messages */
#define DEBUG_SONAR 1  /* Sonar routines */

#define DISPLAY_SONAR 0  /* Flag for displaying sonar values graphically */
```

```
/*
*  FRMinterface.h
*
*  This file defines the interface to the Feature Recognition
*  Module (FRM).  The format of request messages is the same
*  for the different features with one exception.  Requests
*  to find corners and edges require a parameter to specify
*  whether the feature should be to the left or right of the
*  robot's starting position.
*
*  The format of response messages is uniform for corners,
*  walls, and edges.  The module returns three points located
*  along the feature.  The points are ordered so that the robot
*  would encounter each in turn as it moved along the feature,
*  keeping it to its right.
*
*/



/*
*  Request and Response Message Types
*/

#define FRM_WALL           64
#define FRM_CONVEX_CORNER  65
#define FRM_CONCAVE_CORNER 66
#define FRM_EDGE           67
#define FRM_PORT           68



/*
*  Request Message Format
*/


typedef struct FRM_POINT
{
  int x,
      y;
} FRM_POINT;


typedef struct FRM_REQUEST
{
  int type;              /* Feature to look for */
  FRM_POINT pos,         /* Expected coordinates of the feature  */
            pos_error;   /* Error bounds for feature coordinates */
  int theta,             /* Expected orientation of the feature  */
      side,              /* Side of the robot where the feature  */
      strategy;          /* Strategy for Move_To_Point:          */
} FRM_REQUEST;           /*    STRAIGHT, Y_FIRST, X_FIRST        */


/*
*  Valid values for the 'side' field of FRM_POSITION.
*/
```

```
#define FRM_LEFT  1
#define FRM_RIGHT 2



/*
*  Response Message Format
*/

typedef struct FRM_RESPONSE
{
  int type,         /* Feature looked for */
      result,       /* FRM_FEATURE_FOUND, FRM_FEATURE_NOT_FOUND, or FRM_ABORT */
      pt_a,         /* Data points along the feature        */
      pt_b,         /*   Points b and c not returned for ports */
      pt_c,
      theta,        /* Orientation       - returned for ports only */
      theta_error; /* Orientation error - returned for ports only */
} FRM_RESPONSE;


/*
*  Valid values for the 'result' field of response messages.
*/

#define FRM_FEATURE_FOUND     1  /* Feature was detected */
#define FRM_FEATURE_NOT_FOUND 2  /* Feature was not detected */
#define FRM_ABORT             3  /* Detector abort due to avoidance override */
```

feature.h


```
/*
*  feature.h
*
*  The feature detectors in feature.c use the constants below.
*
*/


#define MIN_CLEARANCE 40  /* Min distance between sonars and any obstacle */
                          /* (in centimeters)                             */

#define WALL_LEN 100              /* A wall must be at least 2 meters long */
#define HALF_WALL_LEN 50

#define MAX_DIST_TO_CORNER 80     /* Maximum distance from robot to corner */

#define RING_DIAMETER  16         /* Diameter of the sonar ring */
#define RING_RADIUS RING_DIAMETER/2 /* Radius of the sonar ring */
```

```
/*
*
*  sonar.h
*
*  Include file for code using sonar data.  (Sonar.c, Sonar_utils.c)
*
*/


#define NUM_READINGS 12          /* Number of readings to take to get average */

#define FIRST_VALID_SONAR 8      /* Denning controllers return 24 values, but */
                                 /*   only 16 are valid. */

#define NUM_SONARS 24            /* Number of values returned by Denning */
                                 /*   controllers. */

#define NUM_VALID_SONARS 16      /* Number of valid sonar values (number of */
                                 /*   transducers we have). */

#define LEFT  1                  /* Left side of robot */
#define RIGHT 2                  /* Right side of robot */
#define FRONT 3                  /* Front of robot */
#define BACK  4                  /* Back of robot */


#define FRONT_SONAR 0   /* Sonar directly at the front of the robot */
#define RIGHT_SONAR 4   /* Sonar directly to the right of the robot */
#define LEFT_SONAR 12   /* Sonar directly to the left of the robot  */
#define BACK_SONAR  8   /* Sonar directly to the back of the robot  */

#define SONAR_RANGE 777      /* Max sonar reading in centimeters */
#define HALF_SONAR_ANGLE 11 /* Half the angular distance between sonars */

#define SLOW_SPEED             8  /* A slow speed for translation */
#define EXTRA_SLOW_SPEED       2  /* An even slower speed for translation */

#define EXTRA_SLOW_TURN_SPEED 4  /* For rotations of 5 degrees or less */
#define SLOW_TURN_SPEED       8  /* For rotations between 5 and 90 degrees */
#define MEDIUM_TURN_SPEED     16 /* For rotations of 90 degrees or more */

#define TRANSLATION_ACCEL 218

#define ERR_THRESH 20      /* Allowance for sonar error, in centimeters */
#define NUM_TRIES  2       /* Number of times to try to find a wall */

#define MAX_5_DEG_TURNS 12       /* Max # of 5 deg turns in Orient_parallel */
#define MAX_1_DEG_TURNS 25       /* Max # of 1 deg turns in Orient_parallel */
#define MAX_5_CM_TRANS  6        /* Max # of 5 cm translations in */
                                 /*      Orient_between_doorposts */

#define MAX_DIST_TO_WALL 80  /* The sonars should be within approx. 80 cms */
                             /* of the wall the robot is following         */

#define DOORWAY_WIDTH 88     /* Width of a doorway */
#define DOORJAM_WIDTH 14     /* Width of a doorjam */

/*** Constants used by Orient_between_doorposts */
```

```
#define HALF_DOORJAM_WIDTH DOORJAM_WIDTH/2
#define DIST_TO_DOORJAM 31
#define DOORJAM_THRESH 38


#define WALL_SONAR_COUNT 4    /* Number of contiguous sonars with same value */
                              /* needed to flag presence of a wall           */

#define WALL_OVERSHOOT 14   /* Distance the sonars move beyond a wall before */
                            /* ceasing to detect it (in centimeters).        */

#define TOF_TO_CMS 3.048      /* Multiply to convert tenths of a foot to cms */

#define P_ERROR(err_msg) {fprintf(stderr, err_msg); return FRM_ERROR;}
```

```
#include <stdio.h>
#include <setjmp.h>
#include "/pro/ai/robot/software/huey/utils/src/utils.h"
#include "frm.h"
#include "sonar.h"
#include "feature.h"
#include "frm_interface.h"

/*
*  feature.c
*
*  Feature.c contains the feature detectors of the Feature Recognition
*  Module (FRM).  There is a feature detector for walls, convex corners,
*  concave corners, edges, and ports.  An edge is a section of wall that
*  ends without meeting another wall.  A port is a section of corridor
*  leading out of an intersection, T junction, or L junction of corridors.
*
*  Each feature detector takes as arguments an expected position for
*  a particular point along the feature, an error range for the expected
*  position, an orientation for the feature with respect to the robot's
*  starting position, and a strategy for dead reckoning to the expected
*  position.  The expected position marks a starting point for the robot.
*  Each detector uses dead reckoning to move to the expected position.
*  If the detector cannot find the feature within the given range allowed
*  for error, it returns an indication of failure.
*
*  The routines for external use are:
*
*    FRM_RESPONSE *FEATUREfind_wall(pos, pos_error, theta)
*    FRM_RESPONSE *FEATUREfind_convex_corner(pos, pos_error, theta, side)
*    FRM_RESPONSE *FEATUREfind_concave_corner(pos, pos_error, theta, side)
*    FRM_RESPONSE *FEATUREfind_edge(pos, pos_error, theta, side)
*    FRM_RESPONSE *FEATUREfind_port(pos, theta)
*/

jmp_buf errorHandler;   /* Context for exception handling */


static FRM_RESPONSE response;

/*
*  FEATUREnot_found
*
*  This routine returns an indication of failure to whichever routine
*  or module invoked the current detector.  If the LDP module is
*  driving this module, FEATUREnot_found returns a message.  Otherwise,
*  FEATUREnot_found returns the constant FRM_FEATURE_NOT_FOUND.
*/

#ifdef USE_LDP          /* Locally Distinctive Place module is active */

static FRM_RESPONSE *FEATUREnot_found(feature_type)
  int feature_type;
{
  response.type = feature_type;
  response.result = FRM_FEATURE_NOT_FOUND;
  return &response;
}
```

```
#else                   /* LDP module NOT active */

static int FEATUREnot_found(feature_type)
  int feature_type;
{
  return FRM_FEATURE_NOT_FOUND;
}

#endif                  /* End ifdef LDP */


/*
*  FEATUREabort
*
*  If an avoidance behavior overrides the current detector
*  and takes control of the robot, the detector aborts.
*  FEATUREabort returns an indication of the abort to whichever
*  module or routine invoked the detector.
*/

#ifdef USE_LDP          /* Locally Distinctive Place module is active */

static FRM_RESPONSE *FEATUREabort(feature_type)
  int feature_type;
{
  response.type = feature_type;   /* return a message */
  response.result = FRM_ABORT;
  return &response;
}

#else                   /* LDP module NOT active */

static int FEATUREabort(feature_type)
  int feature_type;
{
  return FRM_ABORT;             /* return a constant */
}

#endif                  /* End ifdef LDP */



/*
*  FEATUREfind_wall
*
*  FEATUREfind_wall looks for a wall centered at the given expected
*  position.  The wall must be at least WALL_LEN centimeters long, and
*  stretch out a distance of at least HALF_WALL_LEN centimeters to
*  either side of the expected position "pos".
*
*  The routine uses dead reckoning to position the robot at the expected
*  position and then orients the robot theta degrees from its initial
*  orientation.  If there is no wall in sight at this point, the detector
*  returns indicating a failure.  If FEATUREfind_wall detects the presence
*  of a wall, it attempts to move the robot along the wall, tracking its
*  presence.  If at any time the robot ceases to detect the wall,
*  FEATUREfind_wall returns with indication of error.
```

```
*
*  Paramters:  pos       - expected midpoint of the wall
*              pos_error - error bounds for expected position
*              theta     - orientation of wall to look for
*
*  Returns:    FRM_ABORT - if avoidance took control of the base
*              FRM_FEATURE_NOT_FOUND - if unable to find wall
*              FRM_FEATURE_FOUND     - if successfully found wall
*/

#ifdef USE_LDP
FRM_RESPONSE *FEATUREfind_wall(pos, pos_error, theta, strategy)
  FRM_POINT pos,        /* expected position */
            pos_error;  /* error bounds for expected position */
  int theta,            /* expected orientation */
      strategy;         /* strategy for move to point: STRAIGHT, or Y_FIRST */
#else
int FEATUREfind_wall()
#endif
{
  int pt1, pt2, pt3;
  int new_pos;

  if (setjmp(errorHandler) > 0)
    return FEATUREabort(FRM_WALL);
  else {
    if (DEBUG_FRM)
      printf("FEATUREfind_wall: Request to find a wall.\n");

#ifdef USE_LDP
    SONARorient_parallel(RIGHT);
    new_pos = record_point(pos.x, pos.y, theta);
    Move_To_Point(new_pos, strategy);
#endif

    if (!SONARwall_in_sight())
      return FEATUREnot_found(FRM_WALL);

    SONARorient_perpendicular(FRONT);
    SONARadjust_dist_to_object(MIN_CLEARANCE, SLOW_SPEED);

  /*** Record point on wall (PT. 2) ***/
    pt2 = record_point(0, MIN_CLEARANCE, 0);

    rotate_robot(90, MEDIUM_TURN_SPEED);
    SONARorient_parallel(LEFT);
    if (SONARfollow_wall(LEFT, HALF_WALL_LEN) < HALF_WALL_LEN)
      return FEATUREnot_found(FRM_WALL);

  /*** Record point on wall (PT. 1) ***/
    pt1 = record_point(-MIN_CLEARANCE, 0, 0);

    rotate_robot(180, MEDIUM_TURN_SPEED);
    SONARorient_parallel(RIGHT);
    if (SONARfollow_wall(RIGHT, WALL_LEN) < WALL_LEN)
      return FEATUREnot_found(FRM_WALL);

  /*** Record point on wall (PT. 3) ***/
```

```
    pt3 = record_point(MIN_CLEARANCE, 0, 0);

    rotate_robot(180, MEDIUM_TURN_SPEED);
    SONARorient_parallel(LEFT);
    translate_robot(HALF_WALL_LEN, SLOW_SPEED);

#ifdef USE_LDP          /*** Return pts. (1, 2, 3) to LDP module ***/
    response.type = FRM_WALL;
    response.result = FRM_FEATURE_FOUND;
    response.pt_a = pt1;
    response.pt_b = pt2;
    response.pt_c = pt3;
    return &response;
#else
    return FRM_FEATURE_FOUND;
#endif
  }
}


/*
*  FEATUREfind_convex_corner
*
*  This routine looks for a convex corner at the expected position
*  "pos".  The corner must consist of two perpendicular sections
*  of wall at least HALF_WALL_LEN in length.
*
*  FEATUREfind_convex_corner attempts to position the robot in the
*  vacinity of the corner by dead reckoning to the expected position.
*  The expected position is the position the robot would have if it
*  were to one side of the corner, a HALF_WALL_LEN distance away:
*
*   convex          -----------
*      corner ->    |********
*                   |*******
*                   |******
*                x  |****
*                  \
*                   expected position
*
*  Once at the expected position, the robot looks for the wall forming
*  one side of the corner.  If the robot fails to detect a wall, the
*  routine returns FRM_FEATURE_NOT_FOUND.  Otherwise, the robot attempts
*  to move around the corner, checking for the existance of the walls
*  making up the convex corner.  If the robot is unable to make the 90
*  degrees turn around the alleged corner, or fails to detect the walls
*  forming the alleged corner, the routine returns indicating failure.
*
*  Parameters:  pos         -  expected position of the corner
*               pos_error   -  error bound for the expected position
*               theta       -  orientation of alleged wall
*
*  Returns:     FRM_ABORT   -  if avoidance overrides the detector
*               FRM_FEATURE_NOT_FOUND  - if unable to find the corner
*               FRM_FEATURE_FOUND      - if successfully found corner
*/

#ifdef USE_LDP
```

```
FRM_RESPONSE *FEATUREfind_convex_corner(pos, pos_error, theta, side, strategy)
  FRM_POINT pos,        /* expected position */
            pos_error;  /* error bounds for expected position */
  int theta,            /* expected orientation */
      side,             /* side of robot feature is on */
      strategy;         /* strategy for move to point: STRAIGHT, or Y_FIRST */
#else
int FEATUREfind_convex_corner(side)
  int side;
#endif
{
  int direction,
      opposite_side;
  int pt1, pt2, pt3;
  int max_dist_to_corner;
  int new_pos,
      midpoint;

  if (setjmp(errorHandler) > 0)
    return FEATUREabort(FRM_CONVEX_CORNER);
  else
  {
    if (DEBUG_FRM)
      printf("FEATUREfind_convex_corner: Request to find a convex corner.\n");

    if (side == LEFT) {
      direction = 1;
      opposite_side = RIGHT;
    }
    else {
      direction = -1;
      opposite_side = LEFT;
    }

#ifdef USE_LDP
    SONARorient_parallel(RIGHT);
    new_pos = record_point(pos.x, pos.y, theta);
    Move_To_Point(new_pos, strategy);
#endif

    if (!SONARwall_in_sight())
      return FEATUREnot_found(FRM_CONVEX_CORNER);

    SONARorient_perpendicular(FRONT);
    SONARadjust_dist_to_object(MIN_CLEARANCE, SLOW_SPEED);
    rotate_robot(direction*90, MEDIUM_TURN_SPEED);

#ifdef USE_LDP
    max_dist_to_corner = HALF_WALL_LEN+RING_DIAMETER+WALL_OVERSHOOT+pos_error.y;
#else
    max_dist_to_corner = MAX_DIST_TO_CORNER;
#endif

    if (SONARfollow_wall(side, max_dist_to_corner) >= max_dist_to_corner)
      return FEATUREnot_found(FRM_CONVEX_CORNER);

    translate_robot((MIN_CLEARANCE-WALL_OVERSHOOT), SLOW_SPEED);
    midpoint = record_point(0, 0, direction*90);
```

```
    rotate_robot(-direction*90, MEDIUM_TURN_SPEED);
    translate_robot((MIN_CLEARANCE+RING_DIAMETER), SLOW_SPEED);
    SONARorient_parallel(side);
    if (SONARfollow_wall(side, HALF_WALL_LEN) < HALF_WALL_LEN)
      return FEATUREnot_found(FRM_CONVEX_CORNER);

    /*** RECORD POINT ON WALL (PT 1) ***/
    pt1 = record_point((-direction*MIN_CLEARANCE), 0, 0);

    rotate_robot(direction*180, MEDIUM_TURN_SPEED);
    SONARorient_parallel(opposite_side);
    translate_robot(HALF_WALL_LEN, SLOW_SPEED);

    /***  RECORD POINT AT CORNER (PT 2) ***/
    pt2 = record_point((direction*MIN_CLEARANCE), RING_RADIUS, 0);

    Move_To_Point(midpoint, STRAIGHT);
    rotate_robot(direction*90, MEDIUM_TURN_SPEED);
    translate_robot((MIN_CLEARANCE+RING_DIAMETER), SLOW_SPEED);
    SONARorient_parallel(opposite_side);
    if (SONARfollow_wall(opposite_side, HALF_WALL_LEN) < HALF_WALL_LEN)
      return FEATUREnot_found(FRM_CONVEX_CORNER);

    /*** RECORD POINT ON WALL (PT 3) ***/
    pt3 = record_point((direction*MIN_CLEARANCE), 0, 0);

    rotate_robot(-direction*180, MEDIUM_TURN_SPEED);

#if USE_LDP
    response.type = FRM_CONVEX_CORNER;
    response.result = FRM_FEATURE_FOUND;
    if (side == LEFT) {          /*** Return pts. (1, 2, 3) ***/
      response.pt_a = pt1;
      response.pt_b = pt2;
      response.pt_c = pt3;
    }
    else {                       /*** Return pts. (3, 2, 1) ***/
      response.pt_a = pt3;
      response.pt_b = pt2;
      response.pt_c = pt1;
    }
    return &response;
#else
    return FRM_FEATURE_FOUND;
#endif
  }
}


/*
*  FEATUREfind_concave_corner
*
*  FEATUREfind_concave_corner looks for a concave corner in the vacinity
*  of the expected position "pos".  The concave corner must consist of
*  two sections of wall, each at least WALL_LEN centimeters long,
*  meeting at a 90 degree angle.
*
*  The routine uses dead reckoning to move the robot to the expected
```

```
 *  position of the wall.  The expected position is the position the
 *  robot would have if it were to one side of the corner, WALL_LEN
 *  centimeters away:
 *
 *
 *           ********
 *         ************
 *        ****-----------
 *       *****|
 *       *****|
 *        ****|                  <- concave corner
 *         ***| x
 *              \
 *               expected position
 *
 *  Once at the expected position, the robot looks for the wall forming
 *  one side of the corner.  If the robot fails to detect a wall, the
 *  routine returns FRM_FEATURE_NOT_FOUND.  Otherwise, the robot attempts
 *  to move around the corner, checking for the existance of the walls
 *  making up the concave corner.  If the robot is unable to detect the
 *  wall opposite the one it is following, as it approaches the alleged
 *  corner, the routine returns indicating failure.
 *
 *  Parameters:  pos       - expected position of the corner
 *               pos_error - error bounds for the expected position
 *               theta     - orientation of the alleged corner
 *
 *  Returns:     FRM_ABORT - if avoidance overrides the detector
 *               FRM_FEATURE_NOT_FOUND - if unable to find corner
 *               FRM_FEATURE_FOUND     - if successfully found corner
 */

#ifdef USE_LDP
FRM_RESPONSE *FEATUREfind_concave_corner(pos, pos_error, theta, side, strategy)
  FRM_POINT pos,         /* expected position */
            pos_error;   /* error bounds for expected position */
  int theta,             /* expected orientation */
      side,              /* side of robot feature is on */
      strategy;          /* strategy for move to point: STRAIGHT, or Y_FIRST */
#else
int FEATUREfind_concave_corner(side)
  int side;
#endif
{
  int direction,
      opposite_side;
  int pt1, pt2, pt3;
  int max_dist_to_corner;
  int dist_to_follow = WALL_LEN - MIN_CLEARANCE;
  int new_pos;

  if (setjmp(errorHandler) > 0)
    return FEATUREabort(FRM_CONCAVE_CORNER);
  else {
    if (DEBUG_FRM)
      printf("FEATUREfind_concave_corner: Request to find a concave corner\n");

    if (side == LEFT) {
```

```
      direction = 1;
      opposite_side = RIGHT;
    }
    else {
      direction = -1;
      opposite_side = LEFT;
    }

#ifdef USE_LDP
    SONARorient_parallel(RIGHT);
    new_pos = record_point(pos.x, pos.y, theta);
    Move_To_Point(new_pos, strategy);
#endif

    if (!SONARwall_in_sight())
      return FEATUREnot_found(FRM_CONCAVE_CORNER);

    SONARorient_perpendicular(FRONT);
    SONARadjust_dist_to_object(MIN_CLEARANCE, SLOW_SPEED);
    rotate_robot(direction*90, MEDIUM_TURN_SPEED);

#ifdef USE_LDP
    max_dist_to_corner = dist_to_follow + pos_error.y;
#else
    max_dist_to_corner = MAX_DIST_TO_CORNER;
#endif

    if (SONARfollow_to_corner(side, max_dist_to_corner, MIN_CLEARANCE) == FRM_ERROR)
      return FEATUREnot_found(FRM_CONCAVE_CORNER);

    rotate_robot(direction*90, MEDIUM_TURN_SPEED);
    SONARorient_parallel(side);
    if (SONARfollow_wall(side, dist_to_follow) < dist_to_follow)
      return FEATUREnot_found(FRM_CONCAVE_CORNER);

    /*** RECORD POINT ON WALL (PT 1) ***/
    pt1 = record_point((-direction*MIN_CLEARANCE), 0, 0);


    rotate_robot(direction*180, MEDIUM_TURN_SPEED);
    SONARorient_parallel(opposite_side);
    translate_robot(dist_to_follow, SLOW_SPEED);

    /*** RECORD POINT AT CORNER (PT 2) ***/

    pt2 = record_point((direction*MIN_CLEARANCE), MIN_CLEARANCE, 0);

    rotate_robot(-direction*90, MEDIUM_TURN_SPEED);
    SONARorient_parallel(opposite_side);
    if (SONARfollow_wall(opposite_side, dist_to_follow) < dist_to_follow)
      return FEATUREnot_found(FRM_CONCAVE_CORNER);

    /*** RECORD POINT ON WALL (PT 3) ***/

    pt3 = record_point((direction*MIN_CLEARANCE), 0, 0);

    rotate_robot(-direction*180, MEDIUM_TURN_SPEED);
    SONARorient_parallel(side);
```

```
    translate_robot(dist_to_follow, SLOW_SPEED);
    rotate_robot(direction*90, MEDIUM_TURN_SPEED);

#if USE_LDP
    response.type = FRM_CONCAVE_CORNER;
    response.result = FRM_FEATURE_FOUND;
    if (side == LEFT) {          /*** Return pts. (1, 2, 3) ***/
      response.pt_a = pt1;
      response.pt_b = pt2;
      response.pt_c = pt3;
    }
    else {                       /*** Return pts. (3, 2, 1) ***/
      response.pt_a = pt3;
      response.pt_b = pt2;
      response.pt_c = pt1;
    }
    return &response;
#else
    return FRM_FEATURE_FOUND;
#endif
  }
}


/*
*  FEATUREfind_edge
*
*  FEATUREfind_edge looks for an edge in the vacinity of the expected
*  position "pos".  The edge must consist of a section of wall at
*  least HALF_WALL_LEN centimeters long, and roughly DOORJAM_WIDTH
*  centimeters thick.
*
*  The routine uses dead reckoning to move the robot to the expected
*  position.  The expected position is the position the robot would
*  have if it were on one side of the section of wall, and HALF_WALL_LEN
*  centimeters away from the edge (the end of the wall):
*
*         -----------------|
*         ****************|       <- edge
*         -----------------|
*             x
*              \
*                expected position
*
*  Once at the expected position, the robot looks for the wall forming
*  the edge.  If the robot fails to detect a wall, the routine returns
*  FRM_FEATURE_NOT_FOUND.  Otherwise the robot attempts to move around
*  the edge, checking for the existance of the wall forming it.  If the
*  robot is unable to move around the alleged edge, or if the robot fails
*  to detect the wall forming the edge as it moves around it, the routine
*  returns FRM_FEATURE_NOT_FOUND.
*
*  Parameters:  pos       - expected position of the edge
*               pos_error - error bounds for the expected position
*               theta     - orientation of the alleged edge
*
*  Returns:     FRM_ABORT - if avoidance overrides the detector
*               FRM_FEATURE_NOT_FOUND - if unable to find edge
```

```
*               FRM_FEATURE_FOUND     - if successfully found edge

*/

#ifdef USE_LDP
FRM_RESPONSE *FEATUREfind_edge(pos, pos_error, theta, side, strategy)
  FRM_POINT pos,         /* expected position */
            pos_error;   /* error bounds for expected position */
  int theta,             /* expected orientation */
      side,              /* side of robot feature is on */
      strategy;          /* strategy for move to point: STRAIGHT, or Y_FIRST */
#else
int FEATUREfind_edge(side)
  int side;
#endif
{
  int direction,
      opposite_side;
  int pt1, pt2, pt3;
  int max_dist_to_corner;
  int new_pos,
      midpoint;

  if (setjmp(errorHandler) > 0)
    return FEATUREabort(FRM_EDGE);
  else {
    if (DEBUG_FRM)
      printf("FEATUREfind_edge: Request to find an edge.\n");

    if (side == LEFT) {
      direction = 1;
      opposite_side = RIGHT;
    }
    else {
      direction = -1;
      opposite_side = LEFT;
    }

#ifdef USE_LDP
    SONARorient_parallel(RIGHT);
    new_pos = record_point(pos.x, pos.y, theta);
    Move_To_Point(new_pos, strategy);
#endif

    if (!SONARwall_in_sight())
      return FEATUREnot_found(FRM_EDGE);

    SONARorient_perpendicular(FRONT);
    SONARadjust_dist_to_object(MIN_CLEARANCE, SLOW_SPEED);
    rotate_robot(direction*90, MEDIUM_TURN_SPEED);

#ifdef USE_LDP
    max_dist_to_corner = HALF_WALL_LEN+RING_DIAMETER+WALL_OVERSHOOT+pos_error.y;
#else
    max_dist_to_corner = MAX_DIST_TO_CORNER;
#endif

    if (SONARfollow_wall(side, max_dist_to_corner) >= max_dist_to_corner)
```

```
      return FEATUREnot_found(FRM_EDGE);

    translate_robot((MIN_CLEARANCE-WALL_OVERSHOOT), SLOW_SPEED);
    rotate_robot(-direction*90, MEDIUM_TURN_SPEED);
    translate_robot((MIN_CLEARANCE+WALL_OVERSHOOT), EXTRA_SLOW_SPEED);
    if (SONARwall_exist(side))  /* Make sure we are not looking at a corner */
      return FEATUREnot_found(FRM_EDGE);
    translate_robot(-WALL_OVERSHOOT, EXTRA_SLOW_SPEED);
    SONARorient_between_doorposts(side);

    translate_robot((MIN_CLEARANCE+DOORJAM_WIDTH), EXTRA_SLOW_SPEED);
    midpoint = record_point(0, 0, direction*90);
    rotate_robot(-direction*90, MEDIUM_TURN_SPEED);
    translate_robot(DIST_TO_DOORJAM+RING_DIAMETER, SLOW_SPEED);
    SONARorient_parallel(side);
    if (SONARfollow_wall(side, HALF_WALL_LEN) < HALF_WALL_LEN)
      return FEATUREnot_found(FRM_EDGE);

  /*** RECORD POINT ON WALL (PT 1) ***/
    pt1 = record_point((-direction*MIN_CLEARANCE), 0, 0);

    rotate_robot(direction*180, MEDIUM_TURN_SPEED);
    SONARorient_parallel(opposite_side);
    Move_To_Point(midpoint, STRAIGHT);
    rotate_robot(direction*90, MEDIUM_TURN_SPEED);
    translate_robot(MIN_CLEARANCE, EXTRA_SLOW_SPEED);
    SONARorient_between_doorposts(opposite_side);

  /*** RECORD POINT AT EDGE (PT 2) ***/
    pt2 = record_point((direction*DIST_TO_DOORJAM), 0, 0);

    translate_robot((MIN_CLEARANCE+DOORJAM_WIDTH), EXTRA_SLOW_SPEED);
    rotate_robot(direction*90, MEDIUM_TURN_SPEED);
    translate_robot(DIST_TO_DOORJAM+RING_DIAMETER, SLOW_SPEED);
    SONARorient_parallel(opposite_side);
    if (SONARfollow_wall(opposite_side, HALF_WALL_LEN) < HALF_WALL_LEN)
      return FEATUREnot_found(FRM_EDGE);

  /*** RECORD POINT ON WALL (PT 3) ***/
    pt3 = record_point((direction*MIN_CLEARANCE), 0, 0);

    rotate_robot(-direction*180, MEDIUM_TURN_SPEED);

#if USE_LDP
    response.type = FRM_EDGE;
    response.result = FRM_FEATURE_FOUND;
    if (side == LEFT) {          /*** Return pts. (1, 2, 3) ***/
      response.pt_a = pt1;
      response.pt_b = pt2;
      response.pt_c = pt3;
    }
    else {                       /*** Return pts. (3, 2, 1) ***/
      response.pt_a = pt3;
      response.pt_b = pt2;
      response.pt_c = pt1;
    }
    return &response;
#else
```

```
    return FRM_FEATURE_FOUND;
#endif
  }
}


/*
*  FEATUREfind_port
*
*  FEATUREfind_port starts the robot on its way out of the
*  current LDP.  At this point, the Locally Distinctive Place
*  Module should have correctly identified the current junction
*  as a T, L, intersection, etc.
*
*  A port is simply a way to exit the current LDP:
*
*
*                 ____________________
*
*                      <- ports ->
*
*                 _______  _______
*                        |  X   |
*                       / |      |
*                robot   |      |
*
*  All this routine has to do is use dead reckoning to move the
*  robot far enough through one of the ports so that the Corridor
*  Following Module (CFM) can take over.
*
*  Parameters:   pos   -  position to move to
*                theta -  orientation for heading out through port
*
*  Returns:      FRM_FEATURE_FOUND - this routine should never fail!
*/

FRM_RESPONSE *FEATUREfind_port(pos, theta, strategy)
  FRM_POINT pos;        /* position of port */
  int theta,            /* orientation for moving out of LDP via port */
      strategy;         /* strategy for Move_To_Point: STRAIGHT or Y_FIRST */
{
  int new_pos;

  if (DEBUG_MSGS) printf("FEATUREfind_port: Request to find a port.\n");

  SONARorient_parallel(RIGHT);
  new_pos = record_point(pos.x, pos.y, theta);
  Move_To_Point(new_pos, strategy);
  translate_robot(RING_DIAMETER, SLOW_SPEED);
  SONARorient_parallel(LEFT);
  response.type = FRM_PORT;
  response.result = FRM_FEATURE_FOUND;
  return &response;
}
```

```
#include <stdio.h>
#include "/pro/ai/robot/software/ipc/src/ipc.h"
#include "frm.h"
#include "sonar.h"
#include "frm_interface.h"

/*
*
*  frm.c
*
*  Frm.c contains the main routine for the feature recognition
*  module.  Main communicates with the locally distinctive place
*  process (LDP) via message passing.
*/

        /* Client IDs of other processes in the system */

clientId my_id;
clientId LDP_id;
clientId WAI_id;
clientId LLC_id;
clientId SC_id;

static FRM_REQUEST req; /* Request to FRM from LDP */


        /*  Feature Detectors */

extern FRM_RESPONSE *FEATUREfind_wall();
extern FRM_RESPONSE *FEATUREfind_convex_corner();
extern FRM_RESPONSE *FEATUREfind_concave_corner();
extern FRM_RESPONSE *FEATUREfind_edge();
extern FRM_RESPONSE *FEATUREfind_port();


/*
*  FRMinit
*
*  FRMinit registers the FRM process with the name server.
*  It then requests the client IDs of the other processes
*  in the system.
*/

FRMinit()
{
  my_id = NSregisterSelf("FRM", MY_PRIORITY);
  while ((LDP_id = NSgetClient("LDP")) == NULL) sleep(1);
  if (DEBUG_FRM) printf("FRMinit: Got LDP id.\n");
  while ((WAI_id = NSgetClient("WAI")) == NULL) sleep(1);
  if (DEBUG_FRM) printf("FRMinit: Got WAI id.\n");
  while ((LLC_id = NSgetClient("LLC")) == NULL) sleep(1);
  if (DEBUG_FRM) printf("FRMinit: Got LLC id.\n");
  while ((SC_id = NSgetClient( "SC" )) == NULL) sleep(1);
  if (DEBUG_FRM) printf("FRMinit: Got SC id.\n");

  if (DISPLAY_SONAR)
    init_sd(16, 255, 5.0, NULL);
}
```

```
/*
*  main
*
*  Main calls FRMinit and then enters an infinite loop to
*  wait for and process requests from the LDP process.
*  If there are no errors in a received request, main calls
*  the appropriate feature detector to carry out the request.
*  The feature detector forms the response message.  Main just
*  has to send it back to the LDP process with a call to
*  IPCsendMessage.
*/


main()
{
  int req_len = sizeof(FRM_REQUEST);
  clientId client;
  FRM_RESPONSE *response;

  FRMinit();
  while( 1 ) {
    if (DEBUG_FRM) printf("main: Calling IPCrecvMessage...\n");
    client = IPCrecvMessage(&req, &req_len, FOREVER);
    if (DEBUG_FRM) printf("main: Returned from IPCrecvMessage.\n");
    if ((client == LDP_id) && (req_len == sizeof(FRM_REQUEST))) {
      if (DEBUG_FRM)
        printf("main: Message received from LDPcontrol with type %d.\n",
                                                              req.type);
      switch(req.type) {
        case FRM_WALL:
          response = FEATUREfind_wall(req.pos, req.pos_error, req.theta,
                                                             req.strategy);
          break;

        case FRM_CONVEX_CORNER:
          response = FEATUREfind_convex_corner(req.pos, req.pos_error,
                                    req.theta, req.side, req.strategy);
          break;

        case FRM_CONCAVE_CORNER:
          response = FEATUREfind_concave_corner(req.pos, req.pos_error,
                                    req.theta, req.side, req.strategy);
          break;

        case FRM_EDGE:
          response = FEATUREfind_edge(req.pos, req.pos_error, req.theta,
                                          req.side, req.strategy);
          break;

        case FRM_PORT:
          response = FEATUREfind_port(req.pos, req.theta, req.strategy);
          break;

        default:
          fprintf(stderr, "FRM: Unknown feature type: %d.\n", req.type);
```

```
      }
      IPCsendMessage(response, sizeof(FRM_RESPONSE), LDP_id);
      if (DEBUG_FRM) {
        if (response->result == FRM_FEATURE_NOT_FOUND)
          printf("main: FEATURE_NOT_FOUND.\n");
        else if (response->result == FRM_ABORT)
          printf("main: FEATURE_ABORT.\n");
        else if (response->result == FRM_FEATURE_FOUND)
          printf("main: FEATURE FOUND.\n");
        else
          printf("main: UNKNOWN RESULT FROM FEATURE DETECTOR.\n");
      }
    }
    else if (client != LDP_id)
      fprintf(stderr, "FRM: Message from unknown client %s.\n",
                                              NSgetName(client));
    else {
      fprintf(stderr, "FRM: Invalid message size of %d.\n", req_len);
      req_len = sizeof(FRM_REQUEST);
    }
  }
}
```

```
#include <stdio.h>
#include <setjmp.h>
#include "frm.h"
#include "sonar.h"
#include "/pro/ai/robot/software/ipc/src/ipc.h"
#include "/pro/ai/robot/software/huey/llc/src/llc.h"
#include "/u/ssh/research/ControlArchitecture/whereami/src/WAIinterface.h"

extern jmp_buf errorHandler;


/*
*  Frm_utils.c
*
*  This file contains routines that communicate via message passing
*  with the Low Level Controller (LLC).  The LLC module provides access
*  to the robot base.  Anytime the FRM wants control of the base, it
*  has to open a transaction with the LLC.
*
*  There is one other routine in this file that communicates via
*  message passing with the WAI module.  This routine requests the
*  WAI module to record the current position of the robot.  The
*  caller receives a handle to the newly recorded postion.
*
*  The routines defined are:
*
*   translate_robot(distance, speed)
*   rotate_robot(degrees, speed)
*   get_current_distance()
*   get_current_angle()
*   get_current_voltage()
*   get_current_current()
*   start_robot_translation(speed)
*   stop_robot_translation()
*   record_point(delta_x, delta_y, delta_theta)
*
*/


extern clientId LLC_id;  /* Client ID for the Low Level Control module */

LLCmessage request_llc;  /* Request message for the LLC module */
LLCmessage response_llc; /* Response message from the LLC module */

static int in_transaction = FALSE; /* TRUE during transaction with LLC */


/*
*  translate_robot
*
*  Translate_robot issues a request to move the robot a fixed distance
*  at a fixed speed.  To do this, the FRM has to gain control of the base
*  by opening a transaction with the LLC.  Translate_robot sends four
*  messages to the LLC.  The first sets the translational velocity of the
*  base to the specified speed.  The remaining messages open a transaction,
*  request the translation, and close the transaction.  If any one of
*  these requests fail, wait_for_LLC_response will return control to
*  the feature detector which called translate_robot.
```

```
*
*  Parameters:  distance - distance in centimeters to translate.
*               speed    - velocity at which to translate.
*
*  Returns:     FRM_OK   - if requests to LLC are successful.
*/

int translate_robot(distance, speed)
  int distance;          /* distance, in centimeters, to move */
  unsigned speed;        /* speed at which to move */
{
  int direction;         /* direction to translate (backwards or forwards) */

  direction = sign(distance);   /* less than zero is backwards */
  speed = speed*direction;

  if (DEBUG_MSGS)
    printf("Translate_robot: request to translate robot %d cms.\n", distance);
  request_llc.type = LLCsetTranslationVel;
  request_llc.data = speed;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCsetTranslationVel, ONE_SECOND);
  if (DEBUG_MSGS)
    printf("Translate_robot: Received ack of LLCsetTranslationVel.\n");

  request_llc.type = LLCbeginTransaction;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCbeginTransaction, ONE_SECOND);
  in_transaction = TRUE;
  if (DEBUG_MSGS)
    printf("Translate_robot: Received ack of LLCbeginTransaction.\n");

  request_llc.type = LLCtranslateRelative;
  request_llc.data = distance;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCtranslateRelative, ONE_MINUTE);
  if (DEBUG_MSGS)
    printf("Translate_robot: Received ack of LLCtranslateRelative.\n");

  request_llc.type = LLCendTransaction;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCendTransaction, ONE_SECOND);
  in_transaction = FALSE;
  if (DEBUG_MSGS)
    printf("Translate_robot: Received ack of LLCendTransaction.\n");

  return FRM_OK;
}


/*
*  rotate_robot
*
*  Rotate_robot issues a request to rotate the robot a fixed amount
*  at a fixed speed.  To do this, the FRM has to gain control of the base
*  by opening a transaction with the LLC.  Rotate_robot sends four
*  messages to the LLC.  The first sets the rotational velocity of the
*  base to the specified speed.  The remaining messages open a transaction,
```

```
*  request the rotation, and close the transaction.  If any one of
*  these requests fail, wait_for_LLC_response will return control to
*  the feature detector which called rotate_robot.
*
*  Parameters:  degrees  - number of degrees to rotate.
*               speed    - velocity at which to rotate.
*
*  Returns:     FRM_OK   - if requests to LLC are successful.
*/

int rotate_robot(degrees, speed)
  int degrees;          /* amount to rotate */
  unsigned speed;       /* velocity at which to rotate */
{
  int direction;        /* direction to rotate (left or right) */

  direction = sign(degrees);    /* less than zero is left */
  speed = speed*direction;

  if (DEBUG_MSGS)
    printf("Rotate_robot: request to rotate robot by %d degrees.\n", degrees);
  request_llc.type = LLCsetRotationVel;
  request_llc.data = speed;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCsetRotationVel, ONE_SECOND);
  if (DEBUG_MSGS)
    printf("Rotate_robot: Received ack of LLCsetRotationVel.\n");

  request_llc.type = LLCbeginTransaction;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCbeginTransaction, ONE_SECOND);
  in_transaction = TRUE;
  if (DEBUG_MSGS)
    printf("Rotate_robot: Received ack of LLCbeginTransaction.\n");

  request_llc.type = LLCrotateRelative;
  request_llc.data = degrees;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCrotateRelative, ONE_MINUTE);
  if (DEBUG_MSGS)
    printf("Rotate_robot: Received ack of LLCrotateRelative.\n");

  request_llc.type = LLCendTransaction;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCendTransaction, ONE_SECOND);
  in_transaction = FALSE;
  if (DEBUG_MSGS)
    printf("Rotate_robot: Received ack of LLCendTransaction.\n");

  return FRM_OK;
}


/*
*  get_current_distance
*
*  Get_current_distance requests the LLC to read the robot's
*  odometer.  A negative reading indicates the robot is behind
```

```
 *  its initial position.  No transaction with the LLC is necessary
 *  to read the current distance.
 *
 *  Returns:  Distance in centimeters the robot has traveled.
 */

int get_current_distance()
{

  if (DEBUG_MSGS)
    printf("Get_current_distance: sending LLCqueryDistance msg to LLC.\n");
  request_llc.type = LLCqueryDistance;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCqueryDistance, ONE_SECOND);
  if (DEBUG_MSGS)
    printf("Get_current_distance: Recieved resp to LLCqueryDistance msg, dist= %d\n",
                                                        response_llc.data);
  return response_llc.data;
}


/*
*  get_current_angle
*
*  Get_current_angle requests the LLC to read the robot's current
*  angle.  The angle is the difference in degrees from the initial
*  position of the robot.  A negative reading indicates the robot is
*  to the left of its start position.  No transaction with the LLC
*  is necessary.
*
*  Returns:  Current angle of the robot in degrees.
*/

int get_current_angle()
{

  if (DEBUG_MSGS)
    printf("Get_current_angle: sending LLCqueryAngle msg to LLC.\n");
  request_llc.type = LLCqueryAngle;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCqueryAngle, ONE_SECOND);
  if (DEBUG_MSGS)
    printf("Get_current_angle: Recieved resp to LLCqueryAngle, dist= %d\n",
                                                      response_llc.data);
  return response_llc.data;
}


/*
*  get_current_voltage
*
*  Get_current_voltage requests the LLC to read the voltage of the
*  batteries on the robot base.  No transaction with the LLC is
*  necessary.
*
*  Returns: Voltage of the batteries on the robot base.
*/
```

```
int get_current_voltage()
{

  if (DEBUG_MSGS)
    printf("Get_current_voltage: sending LLCqueryVoltage msg to LLC.\n");
  request_llc.type = LLCqueryVoltage;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCqueryVoltage, ONE_SECOND);
  if (DEBUG_MSGS)
    printf("Get_current_voltage: Recvd resp to LLCqueryVoltage, dist= %d\n",
                                                        response_llc.data);
    return response_llc.data;
}


/*
*  get_current_current
*
*  Get_current_current requests the LLC to read the current of the
*  batteries on the robot base.
*
*  Returns: Current of batteries on the base.
*/

int get_current_current()
{

  if (DEBUG_MSGS)
    printf("Get_current_current: sending LLCqueryCurrent msg to LLC.\n");
  request_llc.type = LLCqueryCurrent;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCqueryCurrent, ONE_SECOND);
  if (DEBUG_MSGS)
    printf("Get_current_current: Rcvd resp to LLCqueryCurrent, dist= %d\n",
                                                        response_llc.data);
  return response_llc.data;
}


/*
*  start_robot_translation
*
*  Start_robot_translation requests the LLC to start the base
*  translating at a given velocity.  The routine sends three
*  messages to the LLC.  The first sets the translational velocity
*  of the base to the specified speed.  The next message opens a
*  transaction with the LLC, giving the FRM control of the robot
*  base.  The final message starts the base translating.  If any
*  of the requests to the LLC fail, the routine wait_for_LLC_reponse
*  returns control to the feature detector that called this routine.
*
*  Parameters:  speed - velocity at which to translate the base.
*
*  Returns:     FRM_OK - if the requests to the LLC are successful.
*/

int start_robot_translation(speed)
  int speed;
```

```
{

  if (DEBUG_MSGS)
    printf("Start_robot_translation: sending SetTranslationVel msg to LLC.\n");
  request_llc.type = LLCsetTranslationVel;
  request_llc.data = speed;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCsetTranslationVel, ONE_SECOND);
  if (DEBUG_MSGS)
    printf("Start_robot_translation: Recvd ack of LLCsetTranslationVel.\n");

  request_llc.type = LLCbeginTransaction;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCbeginTransaction, ONE_SECOND);
  in_transaction = TRUE;
  if (DEBUG_MSGS)
    printf("Start_robot_translation: Recvd ack of LLCbeginTransaction.\n");

  request_llc.type = LLCtranslateStart;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCtranslateStart, ONE_SECOND);
  if (DEBUG_MSGS)
    printf("Start_robot_translation: Recvd ack of LLCtranslateStart.\n");

  return FRM_OK;
}


/*
*  stop_robot_translation
*
*  Stop_robot_translation requests the LLC to stop translating
*  the base.  The routine sends two messages to the LLC.
*  The first stops translation of the base and the second closes
*  the transaction that start_robot_translation opened.  If any of
*  the requests to the LLC fail, the routine wait_for_LLC_reponse
*  returns control to the feature detector that called this routine.
*
*  Returns:     FRM_OK - if the requests to the LLC are successful.
*/

int stop_robot_translation()
{

  if (DEBUG_MSGS)
    printf("Stop_robot_translation: Sending LLCtranslateEnd msg to LLC.\n");
  request_llc.type = LLCtranslateEnd;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCtranslateEnd, ONE_SECOND);
  if (DEBUG_MSGS)
    printf("Stop_robot_translation: Received ack of LLCtranslateEnd.\n");

  request_llc.type = LLCendTransaction;
  IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
  wait_for_LLC_response(LLCendTransaction, ONE_SECOND);
  in_transaction = FALSE;
  if (DEBUG_MSGS)
    printf("Stop_robot_translation: Received ack of LLCendTransaction.\n");
```

```
  return FRM_OK;
}


/*
*  wait_for_LLC_response
*
*  This routine waits for the response to an LLC request.  If
*  the specified timeout expires before a reponse comes in,
*  or if the reponse is malformed, or if the LLC sends a warning
*  of an avoidance override, a call to longjmp() will blast
*  control back to the currently active feature detector.
*
*  Parameters:  message_type - type of message to expect
*               timeout - maximum time to wait for a response
*
*  Returns:     FRM_OK - if reponse was OK
*/

static int wait_for_LLC_response(message_type, timeout)
  LLCmessageType message_type;  /* message type to expect */
  int timeout;                  /* how long to wait for message */
{
  int msg_length;                       /* length of request to LLC */
  int resp_len = sizeof(LLCmessage);    /* length of reponse message */
  int still_waiting = TRUE;             /* TRUE while waiting for response */
  clientId client;                      /* client that send response msg */

  /*** Read messages until we get the response we want, an error, ***/
  /***   or a transaction override by a higher priority task.     ***/

  do {
    client = IPCrecvMessage(&response_llc, &resp_len, timeout);
    if (client == NULL) {
      fprintf(stderr, "wait_for_LLC_response: Timed out waiting for resp.\n");
      still_waiting = FALSE;
    }
    else if (client != LLC_id) {
      fprintf(stderr, "wait_for_LLC_response: Msg from unknown client.\n");
      still_waiting = FALSE;
    }
    else if (resp_len != sizeof(LLCmessage)) {
      fprintf(stderr, "wait_for_LLC_response: Response was wrong size: %d\n",
                                                    resp_len);
      still_waiting = FALSE;
    }
    else if (response_llc.type != message_type) {

  /*** Override by Avoidance module, so send an abort request to  LLC ***/

      if (response_llc.type == LLCwarningOverride) {
        request_llc.type = LLCabortRequest;
        IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
        msg_length = sizeof(LLCmessage);
        IPCrecvMessage(&response_llc, &msg_length, ONE_SECOND);
        in_transaction = FALSE;
        if (DEBUG_MSGS)
```

```
          printf("wait_for_LLC_response: Abort due to override.\n");
        still_waiting = FALSE;
      }
      else if (DEBUG_MSGS)
        printf("wait_for_LLC_response: Unexpected msg from LLC, type = %d\n",
                                                    response_llc.type);
    }
    else
      return FRM_OK;    /* Got what we wanted! */
  } while (still_waiting);

  if (in_transaction)   /* Error, so end current transaction with LLC */
  {
    request_llc.type = LLCendTransaction;
    IPCsendMessage(&request_llc, sizeof(LLCmessage), LLC_id);
    msg_length = sizeof(LLCmessage);
    IPCrecvMessage(&response_llc, &msg_length, ONE_SECOND);
    in_transaction = FALSE;
    if (DEBUG_MSGS)
      printf("wait_for_LLC_response: Sent req to end current transaction.\n");
  }

  /*** ERROR: blast back to feature detector ***/

  if (DEBUG_MSGS)
    printf("wait_for_LLC_response: longjmp back to feature detector!.\n");
  longjmp(errorHandler, 1);
}


extern clientId WAI_id;                 /* Client ID of WAI process */

static WAIrequestMsg request_wai;       /* request message to WAI */
static WAIreplyMsg   response_wai;      /* response message from WAI */


/*
*  record_point
*
*  This routine sends a request to the Where-Am-I process to save
*  the position at the given offsets from the robot's current position.
*  If record_point times out waiting for the response, or if the
*  response is malformed or has the wrong type, a call to longjmp() will
*  return control to the currently active feature detector.  Otherwise
*  record_point returns the saved position.
*
*  Parameters:  delta_x - offset along the X axis.
*               delta_y - offset along the Y axis.
*               delta_theta - offset in orientation.
*
*  Returns:     New position if successful.
*/

int record_point(delta_x, delta_y, delta_theta)
  int delta_x, delta_y,
      delta_theta;
{
  int resp_len = sizeof(WAIreplyMsg);
```

```
    clientId client;

    request_wai.type = typeFHMsavePositionDelta;
    request_wai.x = delta_x;
    request_wai.y = delta_y;
    request_wai.angle = delta_theta;
    IPCsendMessage(&request_wai, sizeof(WAIrequestMsg), WAI_id);

    if (DEBUG_MSGS)
      printf("Record_point: Waiting for response from WAI module...\n");

    client = IPCrecvMessage(&response_wai, &resp_len, ONE_SECOND);
    if (client == NULL)
      fprintf(stderr, "Record_point: Timed out waiting for resp from WAI.\n");
    else if (client != WAI_id)
      fprintf(stderr, "Record_point: Received msg from unknown client.\n");
    else if (resp_len != sizeof(WAIreplyMsg))
      fprintf(stderr, "Record_point: Resp recvd from WAI was wrong size: %d\n",
                                                        resp_len);
    else if (response_wai.type != typeFHMsavePositionDelta)
      fprintf(stderr, "Record_point: Resp recvd from WAI had wrong type: %d\n",
                                                        response_wai.type);
    else {
      if (DEBUG_MSGS)
        printf("Record_point: Response from WAI module OK. Pos = %d\n",
                                                        response_wai.position);
      return (response_wai.position);
    }

    if (DEBUG_MSGS)
      printf("record_point: longjmp back to feature detector!.\n");
    longjmp(errorHandler, 1);
}
```

```
#include <stdio.h>
#include <string.h>
#include "frm.h"
#include "sonar.h"

static uint16 ring[NUM_VALID_SONARS];           /* ring of sonar data */
static uint16 double_ring1[2*NUM_VALID_SONARS]; /* double ring of sonar data */
static uint16 double_ring2[2*NUM_VALID_SONARS]; /* double ring of sonar data */

/*
*  Sonar.c
*
*  Sonar.c contains any routines that interpret sonar data.
*  The routines do not use raw sonar data, but data that has
*  been averaged over several sensor readings.  The important
*  routines defined here are:
*
*  SONARorient_parallel(side)
*  SONARorient_perpendicular(side)
*  SONARfollow_wall(side, dist_to_follow)
*  SONARfollow_to_corner(side, max_travel, dist_to_wall)
*  SONARorient_between_doorposts(side)
*  SONARwall_exist(side)
*  SONARwall_in_sight()
*  SONARadjust_dist_to_object(dest_dist, speed)
*
*/



/*
*  Orient_parallel
*
*  Orient_parallel attempts to align the given side of the robot
*  parallel with the wall next to it.  This routine assumes there
*  is a wall (or other flat, vertical object) at the side of the
*  robot to align.
*
*  The routine works by repeatedly turning the robot 5 degrees
*  until the perpendicular sonar at the given side is closest
*  to the wall.  Orient_parallel then turns the robot in 1 degree
*  increments until the sonars to either side of the perpendicular
*  are balanced in value.  A count of change in turn direction prevents
*  the robot from thrashing back and forth.
*
*  Note that orienting the front of the robot parallel to a wall
*  is a special case because the sonar values are not contiguous
*  in the array.  SONARorient_front deals with this.
*
*  Parameter:   side - side of the robot to align with the wall.
*
*  Returns:     FRM_ERROR - if sonar data unreliable.
*               FRM_OK    - otherwise.
*/

int SONARorient_parallel(side)
  int side;     /* side of robot to face wall (LEFT, RIGHT, FRONT, BACK) */
{
```

```
int min_sonar = -1;      /* Sonar with smallest value */
int perpen_sonar = 0;    /* Sonar perpendicular to the given side */
int thrash_count = 0;    /* Number of times the robot changes direction */
int l_count = 0;         /* Number of left turns */
int r_count = 0;         /* Number of right turns */
int last_turn = -1;      /* Direction of last turn */

switch (side)
{
  case LEFT:
    perpen_sonar = LEFT_SONAR;
    break;
  case RIGHT:
    perpen_sonar = RIGHT_SONAR;
    break;
  case BACK:
    perpen_sonar = BACK_SONAR;
    break;
  case FRONT:                  /* FRONT is a special case. */
    SONARorient_front();
    return;
}

while ((min_sonar != perpen_sonar) && (l_count < MAX_5_DEG_TURNS) &&
                                          (r_count < MAX_5_DEG_TURNS))
{
   SONARget_average_data(ring);
   min_sonar = SONARside_min(ring, side);
   if (DEBUG_SONAR)
     printf("SONARorient_parallel: min sonar is %d with a value of %d.\n",
                                          min_sonar, ring[min_sonar]);
   if (min_sonar < perpen_sonar)
   {
     rotate_robot(-5, EXTRA_SLOW_TURN_SPEED); /* turn LEFT 5 degrees */
     l_count++;
   }
   else if (min_sonar > perpen_sonar)
   {
     rotate_robot(5, EXTRA_SLOW_TURN_SPEED);   /* turn RIGHT 5 degrees */
     r_count++;
   }
}

if ((l_count == MAX_5_DEG_TURNS) || (r_count == MAX_5_DEG_TURNS))
  P_ERROR("SONARorient_parallel: Sonar data unreliable.\n");


l_count = 0; r_count = 0;
do
{
  if (ring[perpen_sonar+1] > ring[perpen_sonar-1])
  {
    if (last_turn == RIGHT)
      thrash_count++;
    else
      l_count++;
    rotate_robot(-1, EXTRA_SLOW_TURN_SPEED);  /* turn LEFT one degree */
    last_turn = LEFT;
```

```
    }
    else if (ring[perpen_sonar-1] > ring[perpen_sonar+1])
    {
      if (last_turn == LEFT)
        thrash_count++;
      else
        r_count++;
      rotate_robot(1, EXTRA_SLOW_TURN_SPEED);   /* turn RIGHT one degree */
      last_turn = RIGHT;
    }
    SONARget_average_data(ring);
  }
  while((ring[perpen_sonar-1] != ring[perpen_sonar+1]) && (thrash_count < 5)
            && (l_count <= MAX_1_DEG_TURNS) && (r_count <= MAX_1_DEG_TURNS));

  if ((l_count > MAX_1_DEG_TURNS) || (r_count > MAX_1_DEG_TURNS))
    P_ERROR("SONARorient_parallel: Sonar data unreliable.\n");

  return FRM_OK;
}


/*
*  Orient_perpendicular
*
*  Calls orient_parallel to orient the front or back
*  of the robot parallel to a wall (thus orienting
*  the robot itself perpendicular to the wall).
*  Makes the code in feature.c more readable.
*
*  Parameter:   side - side of robot to face the wall.
*/

int SONARorient_perpendicular(side)
  int side;                     /* FRONT or BACK */
{
  return( SONARorient_parallel(side));
}


/*
*  Orient_front
*
*  This routine handles the special case of aligning the
*  front of the robot parallel with a wall.  Orient_front
*  assumes there is a wall to the front of the robot.
*
*  This routine works the same way as SONARorient_parallel,
*  but accounts for the values not being contiguous in
*  the array.
*
*  Returns:  FRM_ERROR - if the sonar data is unreliable.
*            FRM_OK        - otherwise.
*/

static int SONARorient_front()
{
  int min_sonar = -1;     /* Sonar with smallest value */
```

```
int thrash_count = 0;    /* Number of times the robot changes direction */
int l_count = 0;         /* Number of left turns */
int r_count = 0;         /* Number of right turns */
int last_turn = -1;      /* Direction of last turn */


while ( (min_sonar != FRONT_SONAR) && (l_count < MAX_5_DEG_TURNS) &&
                                        (r_count < MAX_5_DEG_TURNS) )
{
   SONARget_average_data(ring);
   min_sonar = SONARfront_min(ring);
   if (DEBUG_SONAR)
     printf("SONARorient_parallel: min sonar at front is %d with a value of %d.\n",
                                              min_sonar, ring[min_sonar]);
   if (min_sonar > 13)
   {
     rotate_robot(-5, EXTRA_SLOW_TURN_SPEED); /* turn LEFT 5 degrees */
     l_count++;
   }
   else if (min_sonar < 3)
   {
     rotate_robot(5, EXTRA_SLOW_TURN_SPEED);   /* turn RIGHT 5 degrees */
     r_count++;
   }
}

if ((l_count == MAX_5_DEG_TURNS) || (r_count == MAX_5_DEG_TURNS))
  P_ERROR("SONARorient_parallel: Sonar data unreliable.\n");

l_count = 0; r_count = 0;
do
{
  if (ring[1] > ring[15])
  {
    if (last_turn == RIGHT)
      thrash_count++;
    else
      l_count++;
    rotate_robot(-1, EXTRA_SLOW_TURN_SPEED);  /* turn LEFT one degree */
    last_turn = LEFT;
  }
  else if (ring[15] > ring[1])
  {
    if (last_turn == LEFT)
      thrash_count++;
    else
      r_count++;
    rotate_robot(1, EXTRA_SLOW_TURN_SPEED);   /* turn RIGHT one degree */
    last_turn = RIGHT;
  }
  SONARget_average_data(ring);
}
while((ring[1] != ring[15]) && (thrash_count < 5) &&
      (l_count <= MAX_1_DEG_TURNS) && (r_count <= MAX_1_DEG_TURNS));

if ((l_count > MAX_1_DEG_TURNS) || (r_count > MAX_1_DEG_TURNS))
  P_ERROR("SONARorient_parallel: Sonar data unreliable.\n");
```

```
  return FRM_OK;
}


/*
*  Side_min
*
*  Side_min returns the sonar at the given side with the
*  smallest value.  The smallest sonar to the front is a
*  special case handled by SONARfront_min.
*
*  Parameters:  ring - array of sonar data.
*               side - side of the ring for which to find
*                      the min value.
*
*  Returns:  The sonar at "side" with the smallest value.
*/

static int SONARside_min(ring, side)
  uint16 ring[];
  int side;     /* side of ring to look at (LEFT, RIGHT, BACK) */
{
  int i;
  int current_min_value;        /* Smallest sonar value so far */
  int current_min_sonar;        /* Sonar with smallest value so far */
  int lower, upper;             /* Bounds for loop */
  int perpen_sonar;             /* Sonar perpendicular to "side" */


  switch(side)
  {
    case LEFT:
      lower = 10,  upper = 15;
      perpen_sonar = LEFT_SONAR;
      break;
    case RIGHT:
      lower = 2,  upper = 7;
      perpen_sonar = RIGHT_SONAR;
      break;
    case BACK:
      lower = 6,  upper = 11;
      perpen_sonar = BACK_SONAR;
  }

  current_min_value = SONAR_RANGE+1;
  for (i=lower; i<upper; i++)
    if (current_min_value > ring[i])
    {
      current_min_value = ring[i];
      current_min_sonar = i;
    }

  if (ring[perpen_sonar] == current_min_value)
    return perpen_sonar;        /* Return middle sonar if it has min value */
  else
    return current_min_sonar;
}
```

```
/*
*  Front_min
*
*  Front_min handles the special case of finding the sonar
*  with the smallest value at the front of the robot.
*
*  Parameter:   ring -  array of sonar data.
*
*  Returns:     The sonar at the front with the smallest value.
*/

static int SONARfront_min(ring)
  uint16 ring[];
{
  int i;
  int current_min_value = 255;  /* Smallest sonar value so far */
  int current_min_sonar;        /* Sonar with the smallest value so far */

  for (i=14; i<16; i++)
    if (current_min_value > ring[i])
    {
      current_min_value = ring[i];
      current_min_sonar = i;
    }

  for (i=0; i<3; i++)
    if (current_min_value > ring[i])
    {
      current_min_value = ring[i];
      current_min_sonar = i;
    }

  if (ring[FRONT_SONAR] == current_min_value)
    return FRONT_SONAR; /* Return middle sonar if it has min value */
  else
    return current_min_sonar;
}


/*
*  Follow_wall
*
*  Follow_wall attempts to move along a wall to the given
*  side for the given distance.  If the routine fails to
*  detect a wall to "side", it will stop the robot and return.
*
*  Parameters:  side           -  side of the robot the wall is on.
*               dist_to_follow -  distance, in centimeters, to travel.
*
*  Returns:     Distance, in centimeters, that the robot travelled.
*/

int SONARfollow_wall(side, dist_to_follow)
  int side,             /* LEFT or RIGHT */
      dist_to_follow;   /* how far to follow the wall (in centimeters) */
{
  int start_dist,       /* distance the robot has travelled until now  */
```

```
        ramp_up_dist,     /* distance the robot travels while accelerating */
        ramp_down_dist,   /* distance the robot travels while decelerating */
        dest_dist,        /* distance at which to start deceleration */
        current_dist;     /* distance the robot has currently travelled */

    if (DEBUG_SONAR)
    {
      printf("SONARfollow_wall: Request to follow wall to ");
      if (side == LEFT)
        printf("left ");
      else
        printf("right ");
      printf("for %d centimeters.\n", dist_to_follow);
    }

    start_dist = get_current_distance();
    ramp_up_dist = (SLOW_SPEED * SLOW_SPEED) / (2 * TRANSLATION_ACCEL);
    ramp_down_dist = min( (dist_to_follow / 2), ramp_up_dist);
    dest_dist = start_dist + dist_to_follow - ramp_down_dist;

    if (DEBUG_SONAR)
      printf("SONARfollow_wall: ramp_up_dist = %d, ramp_down_dist = %d, dest_dist = %d\n",
                                ramp_up_dist, ramp_down_dist, dest_dist);
    if (SONARwall_exist(side) && (dist_to_follow > 0))
    {
      start_robot_translation(SLOW_SPEED);
      do
        current_dist = get_current_distance();
      while ( ((dest_dist - current_dist) > 0) && SONARwall_exist(side) );
      stop_robot_translation();
    }
    if (DEBUG_SONAR)
      printf("SONARfollow_wall: travelled %d centimeters.\n",
                                (get_current_distance() - start_dist));

    return( get_current_distance() - start_dist);
}


/*
*  Follow_to_corner
*
*  Follow_to_corner attempts to move along a wall to the given
*  side until the robot is within dist_to_wall cms of the perpendicular
*  wall.  If SONARwall_exist fails to detect a wall to the given side,
*  or if the robot travels max_travel centimeters before coming within
*  dist_to_wall cms of a perpendicular wall, the routine will stop
*  the robot and return FRM_ERROR.
*
*  Parameters:  side         -  side of the robot the wall to follow is on.
*               max_travel   -  max. distance, in centimeters, to travel.
*               dist_to_wall -  distance to leave between robot and wall.
*
*  Returns:     FRM_ERROR - if not able to move within dist_to_wall centimeters
*                       of the opposite wall.
*
*               FRM_OK    - otherwise.
*/
```

```
int SONARfollow_to_corner(side, max_travel, dist_to_wall)
  int side,              /* LEFT or RIGHT */
      max_travel,        /* maximum distance the robot should travel */
      dist_to_wall;      /* final dist between robot and opposite wall */
{
  int start_dist,        /* distance the robot has travelled until now  */
      ramp_up_dist,      /* distance the robot travels while accelerating */
      ramp_down_dist,    /* distance the robot travels while decelerating */
      dest_dist,         /* distance at which to start deceleration */
      current_dist;      /* distance the robot has currently travelled */

  if (DEBUG_SONAR)
  {
    printf("SONARfollow_to_corner: Request to follow wall to ");
    if (side == LEFT)
      printf("left ");
    else
      printf("right ");
    printf("for %d centimeters max.\n", max_travel);
  }

  start_dist = get_current_distance();
  ramp_up_dist = (SLOW_SPEED * SLOW_SPEED) / (2 * TRANSLATION_ACCEL);
  ramp_down_dist = min( (max_travel / 2), ramp_up_dist);
  dest_dist = start_dist + max_travel - ramp_down_dist;

  if (SONARwall_exist(side) && (max_travel > 0))
  {
    start_robot_translation(SLOW_SPEED);
    do
    {
      current_dist = get_current_distance();
      if (!SONARwall_exist(side))
      {
        stop_robot_translation();
        if (DEBUG_SONAR)
          printf("SONARfollow_to_corner: wall to given side not found.\n");
        return FRM_ERROR;
      }
      SONARget_average_data(ring);
      if (ring[FRONT_SONAR] <= dist_to_wall)
      {
        stop_robot_translation();
        if (DEBUG_SONAR)
          printf("SONARfollow_to_corner: opposite wall found.\n");
        return FRM_OK;
      }
    } while ((dest_dist - current_dist) > 0);
    stop_robot_translation();
  }
  if (DEBUG_SONAR)
    printf("SONARfollow_to_corner: travelled max dist of %d cms.\n", max_travel);
  return FRM_ERROR;
}

/*
```

```
 *  Orient_between_doorposts
 *
 *  This routine attempts to center the robot in a doorway.
 *  It assumes the robot is standing in front of the doorway,
 *  and that the doorpost without the door hinges is to the
 *  given side.
 *
 *  The routine starts by translating the robot in 5 cm increments
 *  until the sonar perpendicular to the given side detects
 *  the doorpost.  The routine continues to translate the robot
 *  until the perpendicular sonar loses track of the doorpost
 *  and then translates the robot backwards a fixed distance,
 *  centering it under the doorway.  Finally, the routine rotates
 *  the robot 90 degrees, centers it between the doorposts, and
 *  rotates back 90 degrees.
 *
 *  Parameters:  side - side of the doorway with the doorpost
 *                      without hinges.
 *
 *  Returns:     FRM_ERROR - if no doorpost was detected.
 *               FRM_OK    - otherwise.
 */

int SONARorient_between_doorposts(side)
  int side;
{
  int perpen_sonar;
  int fwd_count = 0;

  if (side == LEFT)
    perpen_sonar = LEFT_SONAR;
  else
    perpen_sonar = RIGHT_SONAR;

  do {
    translate_robot(5, EXTRA_SLOW_SPEED);              /* move fwd 5 centimeters */
    SONARget_average_data(ring);
  } while((ring[perpen_sonar] > DOORJAM_THRESH) &&
          (fwd_count++ < MAX_5_CM_TRANS));

  if (fwd_count == MAX_5_CM_TRANS)
    P_ERROR("SONARorient_between_doorposts: Can't find doorpost.\n");

  start_robot_translation(EXTRA_SLOW_SPEED);
  do
    SONARget_average_data(ring);
  while ( ring[perpen_sonar] <= DOORJAM_THRESH);
  stop_robot_translation();

  translate_robot((-(HALF_DOORJAM_WIDTH+WALL_OVERSHOOT)), EXTRA_SLOW_SPEED);
  rotate_robot(90, MEDIUM_TURN_SPEED);
  SONARadjust_dist_to_object(DIST_TO_DOORJAM, EXTRA_SLOW_SPEED);
  rotate_robot(-90, MEDIUM_TURN_SPEED);

  return FRM_OK;
}
```

```
/*
*  Wall_exist
*
*  Wall_exist attempts to locate a wall to the given side of
*  the robot.  The wall to detect must be within "MAX_DIST_TO_WALL"
*  centimeters from the robot.
*
*  The routine searches for a sequence of sonar values to the
*  given side of the robot that decreases in value and then
*  increases.  Wall_exist finds the sonar with the smallest
*  value first, then counts the number of sonars that form
*  an increasing sequence to either side.  If there are three
*  or more sonars in the sequence as a whole, Wall_exist returns
*  TRUE.
*
*  Parameter:   side - side of the robot on which to look.
*
*  Returns:     TRUE - if the sonars detect a wall to "side".
*               FALSE - otherwise.
*/


int SONARwall_exist(side)
  int side;             /* LEFT or RIGHT */
{
  int i, j,
      lower, upper,             /* lower and upper bounds for the sonar data */
      smallest,                 /* sonar with the smallest value */
      perpen_sonar;             /* sonar perpendicular to the given side */
  int increasing_left,          /* # of sonars in increasing seq to left */
      increasing_right;         /* # of sonars in increasing seq to right */
  int small_val;                /* value of the sonar "smallest" */


  if (side == LEFT)
  {
    lower = 10;
    upper = 15;
    perpen_sonar = LEFT_SONAR;
  }
  else
  {
    lower = 2;
    upper = 7;
    perpen_sonar = RIGHT_SONAR;
  }


  for (i=0; i<NUM_TRIES; i++)
  {
    SONARget_average_data(ring);
    small_val = SONAR_RANGE + 1;

    for (j=lower; j<upper; j++)         /* get smallest sonar */
    {
      if (ring[j] < small_val)
      {
        small_val = ring[j];
```

```
        smallest = j;
      }
    }

    increasing_left = 0, increasing_right = 0;

    for (j = (smallest-1); j >= lower; j--) /* count increasing seq to left */
      if (ring[j] >= ring[j+1])
        increasing_left++;
      else
        break;

    for (j = (smallest+1); j < upper; j++) /* count decreasing seq to right */
      if (ring[j] >= ring[j-1])
        increasing_right++;
      else
        break;

    if (((increasing_left + increasing_right + 1) >= 3) &&
         (ring[perpen_sonar] < MAX_DIST_TO_WALL))
    {
      if (DEBUG_SONAR)
      {
        printf("SONARwall_exist: wall found on ");
        if (side == LEFT)
          printf("left, ");
        else
          printf("right, ");
        printf("centered at sonar %d\n", smallest);
      }
      return TRUE;
    }
    else
      printf("SONARwall_exist: increasing_left = %d, increasing_right = %d.\n",
                                        increasing_left, increasing_right);
  }

  if (DEBUG_SONAR)
  {
    printf("SONARwall_exist: unable to find wall, side = %d, ring = \n", side);
    SONARprint_ring(ring);
  }
  return FALSE;
}



/*
*  Wall_in_sight
*
*  Wall_in_sight searches for a wall to the front of the robot.
*  The only restriction is that the wall be within the sonars'
*  range.
*
*  The routine collects an array of 32 sonar values and copies
*  the first and last five values of the array into another
*  array.  Once the ten sonar values at the front of the robot
*  are in a contiguous array, Wall_in_sight searches for a sequence
```

```
*  of WALL_SONAR_COUNT readings that are close in value.  If it finds
*  such a sequence, Wall_in_sight returns TRUE.
*
*  Returns:     TRUE -  if there is a wall within the sonars' range.
*               FALSE -  otherwise.
*/


int SONARwall_in_sight()
{
  int i, j, k,
      last,
      count,
      smallest;

  for (i=0; i < NUM_TRIES; i++)         /* Try more than once */
  {
    SONARget_average_data_double(double_ring1);

   /* Create a contiguous array of the sonar values */
   /* to the front of the robot.                    */

    for (j=0, k=27; k < 32; j++, k++)
      double_ring2[j] = double_ring1[k];
    for (j=5, k=0;  k < 6; j++, k++)
      double_ring2[j] = double_ring1[k];

    if (DEBUG_SONAR)
    {
      printf("SONARwall_in_sight:  ");
      for (j=0; j<11; j++)
        printf("%d ", double_ring2[j]);
      printf("\n\n");
    }

    /* Search for a sequence of WALL_SONAR_COUNT sonars */
    /* that are close in value.                        */

    last = -ERR_THRESH;
    count = 1;
    smallest = SONAR_RANGE + 1;
    for (j=0; j < 11; j++)
    {
      if ((double_ring2[j] >= last-ERR_THRESH) &&
          (double_ring2[j] <= last+ERR_THRESH) &&
          (last < SONAR_RANGE-ERR_THRESH)) {
        count++;
        if (double_ring2[j] < smallest)
          smallest = double_ring2[j];
      }
      else {
        count = 1;
        smallest = SONAR_RANGE + 1;
      }
      if ((count == WALL_SONAR_COUNT) &&
          (double_ring1[FRONT_SONAR] >= smallest-ERR_THRESH) &&
          (double_ring1[FRONT_SONAR] <= smallest+ERR_THRESH))
      {
```

```
        if (DEBUG_SONAR)
          printf("SONARwall_in_sight: wall found. j = %d front_sonar = %d.\n",
                                              j, double_ring1[FRONT_SONAR]);
        return TRUE;
      }
      last = double_ring2[j];
    }
  }

  if (DEBUG_SONAR)
    printf("SONARwall_in_sight: no wall found.\n");
  return FALSE;
}


/*
*  Adjust_dist_to_object
*
*  Move the robot "dest_dist" centimeters away from the object ahead.
*  Adjust_dist_to_object assumes the robot is perpendicular to the object
*  ahead of it.
*
*  Parameter:   dest_dist -  distance to achieve between robot and object.
*/

SONARadjust_dist_to_object(dest_dist, speed)
  int dest_dist,        /* In centimeters */
      speed;            /* Velocity in cm/sec */
{
  int start_dist;       /* Distance between robot and object at start */

  SONARget_average_data(ring);
  start_dist = ring[FRONT_SONAR];
  if (DEBUG_SONAR)
    printf("SONARadjust_dist_to_object: request to move %d cms from object ahead.  Front sonar = %d.\n", dest_dist, ring[FRONT_SONAR]);

  translate_robot((start_dist - dest_dist), speed);
}
```

```
#include <stdio.h>
#include <math.h>
#include "frm.h"
#include "sonar.h"

#include "/pro/ai/robot/software/ipc/src/ipc.h"
#include "/pro/ai/robot/software/huey/sonar/src/sonar.h"


uint16 raw_ring[NUM_SONARS];          /* Ring of raw sonar values */
uint16 avg_ring[NUM_SONARS];          /* Ring of averaged sonar values */
uint16 distribution[NUM_SONARS][256]; /* For collecting mode data */
uint16 ordered_ring[NUM_SONARS];      /* Ordered ring of sonar values */
uint16 temp_ring[NUM_VALID_SONARS];

extern clientId SC_id;


/*
*  Sonar_utils.c
*
*  Sonar_utils.c contains routines to gather sonar data and order
*  the data into a ring of values.  The routines Get_average_data
*  and Get_mode_data use "NUM_READINGS" of raw sonar data to arrive
*  at the average or mode of the sonar data. Get_average_data_double
*  and Get_mode_data_double rotate the robot by "HALF_SONAR_ANGLE"
*  degrees to get an array of data with twice as many values
*  as physical sonars.  A complete list of available routines
*  follows:
*
*  SONARget_data(ring)
*  SONARget_average_data(ring)
*  SONARget_average_data_double(double_ring)
*  SONARget_mode_data(ring)
*  SONARget_mode_data_double(double_ring)
*  SONARprint_ring(ring)
*  SONARprint_ring_double(ring)
*/



/*
* Get_raw_data
*
* Requests raw sonar data from appropriate underlying module/library.
*
*  Parameter: ring - array to hold the ordered sonar data.
*/

static int get_raw_data(ring)
  uint16 ring[];
{
  static SCrequest myReq = { 24, 077777777 };
  SCresponse response;
  int length = sizeof(SCresponse);

  IPCsendMessage(&myReq, sizeof(SCrequest), SC_id);
  if (IPCrecvMessage(&response,&length,ONE_SECOND) != SC_id)
```

```
    return -1;
  else
  {
    memcpy((char *)ring, (char *)response.data,
           MAX_NUM_SENSORS * sizeof(uint16));
    return 0;
  }
}


/*
*  Get_data
*
*  Requests raw sonar data and calls SONARorder_ring
*  to order the data in a clockwise ring.
*
*  Parameter: ring - array to hold the ordered sonar data.
*/

int SONARget_data(ring)
  uint16 ring[];
{
  int i;

  if (get_raw_data(raw_ring) < 0)
    printf("SONARget_data: Request to read sonar failed.\n\n");
  else
  {
    SONARorder_ring(raw_ring, ring);
    if (DEBUG_SONAR)
    {
      printf("SONARget_data:  ");
      for (i=0; i < NUM_VALID_SONARS; i++)
        printf("%d ", ring[i]);
      printf("\n\n");
    }
  }
}


/*
*  Get_average_data
*
*  Reads in the raw sonar values "NUM_READINGS" times
*  and averages the data.
*
*  Parameter: ring - array to hold the averaged sonar data.
*/

int SONARget_average_data(ring)
  uint16 ring[];
{
  int i, j;

  for (i = FIRST_VALID_SONAR; i < NUM_SONARS; i++)
    avg_ring[i] = 0;

  for (i=0; i < NUM_READINGS; i++)
```

```
  {
     if (get_raw_data(raw_ring) < 0)
       printf("SONARget_average_data: Request to read sonar failed.\n");
     else
       for (j = FIRST_VALID_SONAR; j < NUM_SONARS; j++)
         avg_ring[j] = avg_ring[j] + raw_ring[j];
  }
  for (i = FIRST_VALID_SONAR; i < NUM_SONARS; i++)
    avg_ring[i] = (avg_ring[i] / NUM_READINGS);

  SONARorder_ring(avg_ring, ring);

  if (DEBUG_SONAR)
  {
    printf("SONARget_average_data:  ");
    SONARprint_ring(ring);
  }

  if (DISPLAY_SONAR)
    draw_sd(ring);
}


/*
*  Get_average_data_double
*
*  This routine rotates the robot to obtain a ring
*  with twice as many values as there are physical
*  sonars.  Get_average_data_double returns the robot
*  to its initial orientation.
*
*  Parameter: double_ring - array to hold the averaged data.
*/

int SONARget_average_data_double(double_ring)
  uint16 double_ring[];
{
  int i;

  SONARget_average_data(temp_ring);
  for(i=0; i < NUM_VALID_SONARS; i++)
    double_ring[i*2] = temp_ring[i];

  rotate_robot(HALF_SONAR_ANGLE, SLOW_TURN_SPEED);

  SONARget_average_data(temp_ring);
  for(i=0; i < NUM_VALID_SONARS; i++)
    double_ring[(i*2)+1] = temp_ring[i];

  rotate_robot(-HALF_SONAR_ANGLE, SLOW_TURN_SPEED);

  if (DEBUG_SONAR)
  {
    printf("SONARget_average_data_double:  ");
    SONARprint_ring_double(double_ring);
  }
}
```

```
/*
*  Get_mode_data
*
*  Get_mode_data reads in the raw sonar data "NUM_READINGS"
*  times, and returns a ring of the most frequently occurring
*  values.
*
*  Parameter: ring - array to hold the mode of the sonar data.
*/

int SONARget_mode_data(ring)
  uint16 ring[];
{
  int i, j,
      peak,             /* Number of times the most frequent value appeared */
      max_value;        /* Value that appeared most frequently */

  for (i = FIRST_VALID_SONAR; i < NUM_SONARS; i++)
  {
    avg_ring[i] = 0;
    for (j = 0; j < 256; j++)
      distribution[i][j] = 0;
  }

  for (i=0; i < NUM_READINGS; i++)
  {
    if (get_raw_data(raw_ring) < 0)
      printf("SONARget_mode_data: Request to read sonar failed.\n");
    else
      for (j = FIRST_VALID_SONAR; j < NUM_SONARS; j++)
        distribution[j] [raw_ring[j]]++;
  }

  for (i = FIRST_VALID_SONAR; i < NUM_SONARS; i++)
  {
    peak = 0;
    max_value = 0;

    for (j = 0; j < 256; j++)
      if (distribution[i][j] > peak)
      {
        peak = distribution[i][j];
        max_value = j;
      }
    avg_ring[i] = max_value;
  }

  SONARorder_ring(avg_ring, ring);

  if (DEBUG_SONAR)
  {
    printf("SONARget_mode_data:  ");
    SONARprint_ring(ring);
  }

  if (DISPLAY_SONAR)
    draw_sd(ring);
```

```
}


/*
*  Get_mode_data_double
*
*  This routine rotates the robot to obtain a ring
*  with twice as many values as there are physical
*  sonars.  Get_mode_data_double returns the robot
*  to its initial orientation.
*
*  Parameter: double_ring - array to hold mode data.
*/

int SONARget_mode_data_double(double_ring)
  uint16 double_ring[];
{
  int i;

  SONARget_mode_data(temp_ring);
  for (i=0; i<NUM_VALID_SONARS; i++)
    double_ring[i*2] = temp_ring[i];

  rotate_robot(HALF_SONAR_ANGLE, SLOW_TURN_SPEED);

  SONARget_mode_data(temp_ring);
  for (i=0; i<NUM_VALID_SONARS; i++)
    double_ring[(i*2)+1] = temp_ring[i];

  rotate_robot(-HALF_SONAR_ANGLE, SLOW_TURN_SPEED);

  if (DEBUG_SONAR)
  {
    printf("SONARget_mode_data_double:  ");
    SONARprint_ring_double(double_ring);
  }
}


/*
*  Order_ring
*
*  Get_raw_data returns the raw sonar data out of order.
*  Order_ring orders the values returned by get_raw_data
*  into a clockwise ring starting at the front of the robot.
*
*  Parameters: unordered - array of unordered sonar data.
*              ordered   - array to hold the ordered data.
*/

static int SONARorder_ring(unordered, ordered)
  uint16 unordered[];
  uint16  ordered[];
{
  ordered[0] = (uint16)(TOF_TO_CMS*unordered[18]);
  ordered[1] = (uint16)(TOF_TO_CMS*unordered[14]);
  ordered[2] = (uint16)(TOF_TO_CMS*unordered[17]);
  ordered[3] = (uint16)(TOF_TO_CMS*unordered[15]);
```

```
   ordered[4] = (uint16)(TOF_TO_CMS*unordered[16]);
   ordered[5] = (uint16)(TOF_TO_CMS*unordered[8]);
   ordered[6] = (uint16)(TOF_TO_CMS*unordered[23]);
   ordered[7] = (uint16)(TOF_TO_CMS*unordered[9]);
   ordered[8] = (uint16)(TOF_TO_CMS*unordered[22]);
   ordered[9] = (uint16)(TOF_TO_CMS*unordered[10]);
   ordered[10] = (uint16)(TOF_TO_CMS*unordered[21]);
   ordered[11] = (uint16)(TOF_TO_CMS*unordered[11]);
   ordered[12] = (uint16)(TOF_TO_CMS*unordered[20]);
   ordered[13] = (uint16)(TOF_TO_CMS*unordered[12]);
   ordered[14] = (uint16)(TOF_TO_CMS*unordered[19]);
   ordered[15] = (uint16)(TOF_TO_CMS*unordered[13]);
}


/*
*  Print_ring
*
*  Prints the given ring of sonar data for
*  debugging purposes.
*
*  Parameter: ring - ring to print.
*/

int SONARprint_ring(ring)
  uint16 ring[];
{
  int i;

  printf("    ");
  for (i=0; i<NUM_VALID_SONARS; i++)
    printf("%d ", ring[i]);
  printf("\n");
}


/*
*  Print_ring_double
*
*  Prints a double ring of sonar data for
*  debugging puposes.
*
*  Parameter: double_ring - ring to print.
*/

int SONARprint_ring_double(ring)
  uint16 ring[];
{
  int i;

  printf("    ");
  for (i=0; i < (2*NUM_VALID_SONARS); i++)
    printf("%d ", ring[i]);
  printf("\n");
}
```