BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-90-M1

"Handling Uncertainties in Classifying Junctions"

by
Seungseok Hyun

Handling Uncertainties in Classifying Junctions

by

Seungseok Hyun

B.S., Korea University, 1988

Department of Computer Science
Brown University

Thesis

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

February, 1990

This thesis by Seungseok Hyun 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 2/20/90

Thomas L. Dean
Advisor

Handling Uncertainties in Classifying Junctions of Corridors

Seungseok Hyun

February 20, 1990

Abstract

This paper describes some extensions to the Geographer Module of Huey. Keeping track of and handling motion errors and incorporating the uncertainty of a feature detector into the decision procedure are explained. We also introduce a new set of geometric features and add a new action for reducing positional errors. These extensions provide efficiency and robustness to the Geographer Module.

Contents

List of Figures

1 Introduction

A robot working in the real world has to face many uncertainties. For example, its sensors are erroneous and sometimes misleading, and its effectors are not accurate, therefore, its beliefs about the environment are not always true. In order to survive in the real world, a robot must be able to handle these uncertainties.

Our major concern is to devise methods to handle positional and sensory uncertainties of the robot and to choose the best action under such uncertain circumstances in a control system for a practical task. Huey is a real mobile robot with sonars as its sensors and three motor driven wheels as its effectors. The *Geographer Module* of Huey is a module for exploring unknown environments. This paper describes some of the components for this module.

The task of the Geographer Module is to build an indoor map consisting of corridors and junctions of corridors. The map can be represented as a graph whose arcs correspond to corridors, and vertices correspond to junctions. We adopt the concept of a *locally distinctive place* (LDP henceforth) and regard each junction as an LDP [7]. These LDP's can be partitioned into *equivalent classes* which are based on the geometric relationships (e.g. cross junctions, T junctions, and L junctions) of the corridors which join at these LDP's. Figure 1 shows an example of such a map.

The *Corridor Following Module* and the *Junction Classifying Module* are two of the sub-modules of the Geographer Module. The Corridor Following Module moves the robot from one junction to another, and the Junction Classifying Module determines the equivalent class of a junction. Therefore, the overall map building task will include some interleaved sequence of corridor following and junction classifying tasks.

The corridor following task is done with a simple strategy which moves Huey along the center line of a corridor, watching the sonar data for an abrupt change which, presumably, indicates that Huey is in the vicinity of a junction. After detecting such an abrupt change, Huey is positioned at the *entry position* of the junction which is at the end of the traveled corridor.

As the Corridor Following Module finishes positioning Huey at the entry position of a junction, the Junction Classifying module takes control over Huey and tries to classify the junction into an equivalent class. The *feature*,

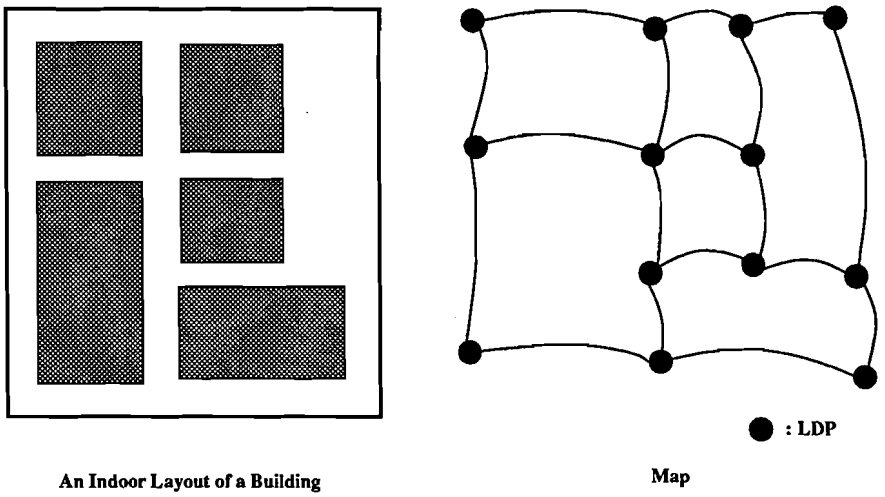

Figure 1: Example of a Map

a geometric relationship between a wall and the entry position, provides a clue for the classification. The junction classifying module employs *feature detectors* which detect such a relationship by composite sensing and movement operations. The Junction Classification Module decides the equivalent class of the junction by issuing appropriate feature detectors.

The task decomposition is roughly parallel to the control architecture of the Geographer Module. Each module of the architecture is responsible for a specific task, and a message-passing mechanism is used for the communication between modules. The control architecture is shown in Figure 2. The WHEREAMI(it reads "where am i") module keeps track of positions and positional errors. The Low-Level Controller controls the mobile base of Huey, and the Sonar Controller handles the sonar ring.

The previous version of the Geographer Module was implemented by Chekaluk and Randazza [3, 10]. Although it worked quite well, there were some points which could be improved:

- The system keeps the expected motion errors(expected positional errors) but does not use them.

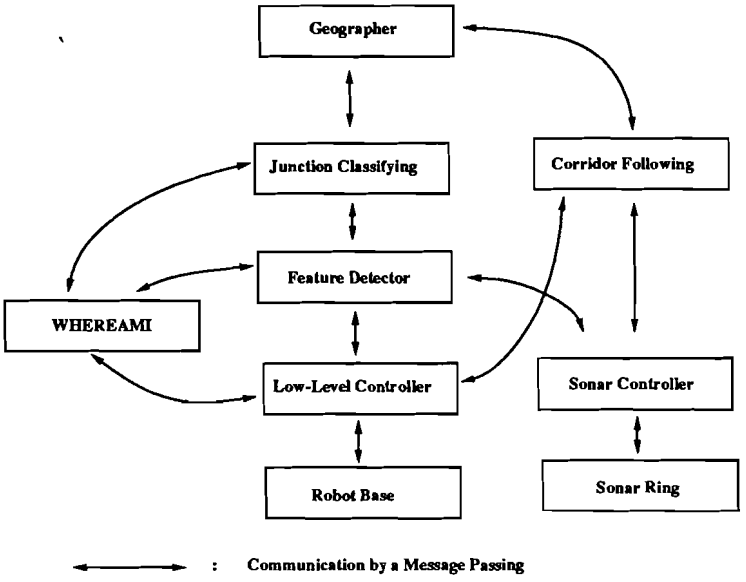

Figure 2: Control Architecture of the Geographer Module

- The Junction Classifying Module assumes that the feature detectors are always 100% correct.
- The system does not take advantage of any partial results from the feature detectors.
- Positional errors keep growing, but the system has no method of reducing them.

In the following sections, we describe our approaches to improve the previous version of Geographer Module. In section 2, we explain how to keep track of positions and expected motion errors. Incorporating partial results and positional uncertainties in the Feature Detector Module is explained in section 3. Section 4 presents the decision model that chooses the best action under uncertain circumstances in the Junction Classifying Module.

2 WHEREAMI Module

Huey's measurement of the distance translated or the angle rotated has some unavoidable errors due to the limited precision of the motors and the shaft encoders. It is common for the motors to coast while trying to stop, since Huey is quite massive and has no brakes. It is also quite possible that Huey might slide on a wet or waxed floor. Each single error may be small enough to be ignored, but the accumulated error is significant. Therefore, we need to keep track of these motion errors and to consider them in the decision procedures of the Huey Geographer.

When Huey wants to move to a nearby position that it has visited before, the relative coordinates of the previous position with respect to its current position are required. There may be many such positions that Huey will wish to re-visit later. Thus, it is necessary that Huey has a method to store these positions and to retrieve relative coordinates between such positions efficiently.

These requirements are handled by the WHEREAMI Module. It stores motion errors and such positions that Huey may wish to re-visit later. This module provides the relative relationships between any two positions in order to meet the requirements. The WHEREAMI Module is based upon Smith and Cheeseman's work on the representation and estimation of spatial uncertainty [13]. Their work offers a general method for estimating the nominal relationship and expected error between two coordinate frames representing the relative locations of objects. In our domain, the objects corresponds to the positions Huey has visited before and Huey's current position. The nominal relationship and expected error are determined by translations and rotations of Huey. Since the robot moves from one position to another in a single step, we can represent the relationships between positions using a list structure where new positions are appended in a chronological order.

Smith and Cheeseman's work does not use an absolute coordinate system; every position is the origin of its own frame of reference. As a result, there is no difference between positions and frames of reference. The relationship between positions(or frames) is represented by an *approximate transformation* (AT henceforth) from one position(or frame) to another. An AT consists of an estimated mean relation of one position relative to another and a covari-

ance matrix that expresses the uncertainty of the estimate. An estimated mean is a triple (X, Y, θ) where X and Y are the values of a Cartesian coordinate system which represent a movement between two positions, and θ is the difference of the orientations between the final position and the initial position. The covariance matrix C is a 3×3 matrix which represents the covariances of X, Y and θ :

$$C = \begin{pmatrix} \sigma_x^2 & \rho_{x,y}\sigma_x\sigma_y & \rho_{x,\theta}\sigma_x\sigma_\theta \\ \rho_{x,y}\sigma_x\sigma_y & \sigma_y^2 & \rho_{y,\theta}\sigma_y\sigma_\theta \\ \rho_{x,\theta}\sigma_x\sigma_\theta & \rho_{y,\theta}\sigma_y\sigma_\theta & \sigma_\theta^2 \end{pmatrix}$$

σ_i^2 is the variance of i, and σ_i is the standard deviation of i, where $i \in \{x, y, \theta\}$. $\rho_{i,j}$ is the correlation coefficient for i and j. For example, after Huey moves 30 cm forward, the AT from the initial position and the final position will consist of the mean relation $(0, 30, 0)$ and the covariance matrix determined by a simulation.

Every time Huey moves, the simulation generates a fixed number of random samples and computes covariances from the samples. We assume a simple model of motion errors, and the simulation generates random samples according to this model. The model assumes that the error of a translation is proportional to the distance traveled, and the error can not exceed a certain amount determined by the distance traveled. In the current implementation, each simulation generates 50 random samples, and the model assumes that a translational error lies within 3% of the distance translated and a rotational error within 1% of the degrees rotated. The model also assumes that a translation(rotation) does not affect Huey's orientation(position) at all. In the previous example, the simulator generates x_i, y_i, θ_i, where $0 \leq i < 50$, $x_i = 0$, $y_i = 30 + 0.03 \times 30 \times rnd(i)$, and $\theta_i = 0$.[1] And the covariance matrix is computed from these x_i, y_i, and θ_i. Although this model does not consider the translation speed, which might seem crucial in some cases, it works well in our domain, simply because the Geographer Module only uses a set of relatively slow speeds. If Huey were to travel at much a faster speed, this model can be easily extended to accommodate faster translation speeds. Rotation is treated in a similar way.

[1] $rnd(i)$ is a function which generates a random number between -1 and 1.

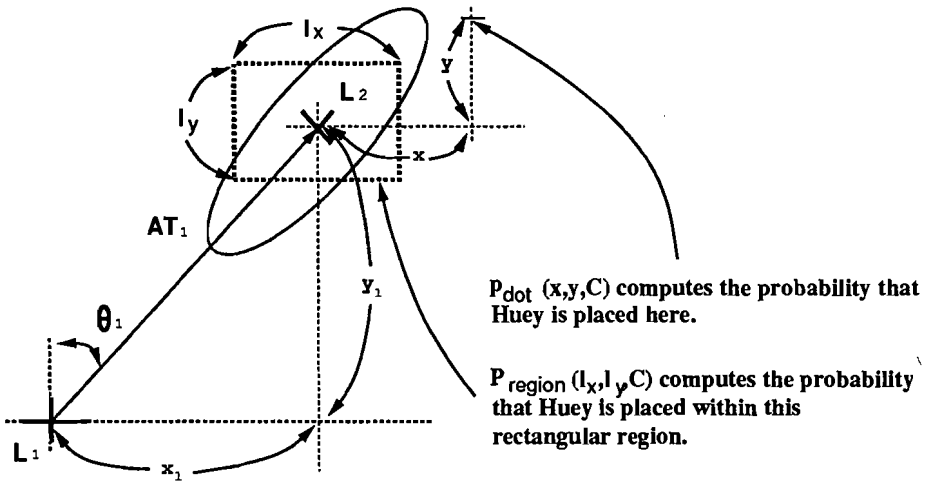

Figure 3: Positional Uncertainty Between Two Positions

In order to retrieve a relation between two arbitrary positions, we use the *compound operator* and the *inverse operator.*[2] The compound operator compounds two AT's into one. If there are three positions L_1, L_2 and L_3, and we know both the AT's from L_1 to L_2 and from L_2 to L_3 , the compound operator will produce the AT from L_1 to L_3 . In the case when the system knows the AT from L_1 to L_2 , the inversed AT can be obtained by the inverse operator. By applying appropriate sequence of compound and inverse operators, any relationship between two positions can be calculated easily.

Positional uncertainties can be estimated by the covariance matrix of an AT, assuming that the probability distribution of our knowledge of motions is a multivariate (x, y, θ) Gaussian distribution. Figure 3 shows a positional uncertainty between two positions, L_1 and L_2, after Huey has moved from L_1 to L_2. AT_1 is the approximate transformation for the movement. The points of equal positional uncertainty form an ellipse which has the expected position as its center. Using AT_1, which consists of a mean triple (x_1, y_1, θ_1) and the 3×3 covariance matrix C, we can obtain the probability that Huey is placed at the position which is (x, y) away from L_2 in the frame of reference

[2]The Smith and Cheeseman's paper contains the actual derivations of these operators[13].

of L_1 as follows [13].

$$p_{dot}(x, y, C) =$$

$$\frac{1}{2\pi\sqrt{\sigma_x^2\sigma_y^2(1-\rho_{x,y})^2}} \exp -\frac{1}{2(1-\rho_{x,y}^2)}[\frac{x^2}{\sigma_x^2} + \frac{2\rho_{x,y}xy}{\sigma_x\sigma_y} + \frac{y^2}{\sigma_y^2}], \quad (1)$$

$$C = \begin{pmatrix} \sigma_x^2 & \rho_{x,y}\sigma_x\sigma_y & \rho_{x,\theta}\sigma_x\sigma_\theta \\ \rho_{x,y}\sigma_x\sigma_y & \sigma_y^2 & \rho_{y,\theta}\sigma_y\sigma_\theta \\ \rho_{x,\theta}\sigma_x\sigma_\theta & \rho_{y,\theta}\sigma_y\sigma_\theta & \sigma_\theta^2 \end{pmatrix}$$

where ρ is the correlation coefficient for x and y, and $p_{dot}(x, y, C)$ is a variation of the bivariate Gaussian distribution. In our domain, the probability that Huey is placed within a region offers a good estimate of the positional certainty. The probability function $P_{region}(l_x, l_y, C)$ is written as

$$P_{region}(l_x, l_y, C) = \int_{x=-\frac{l_x}{2}}^{\frac{l_x}{2}} \int_{y=-\frac{l_y}{2}}^{\frac{l_y}{2}} p_{dot}(x, y, C)dydx \quad (2)$$

This computes the probability that the robot lies within a rectangular region whose center is L_2, and the length of whose x-axis side is l_x, and the length of whose y-axis side is l_y in the frame of reference of L_1. We chose a rectangular region in the equation, because a positional certainty measurement by a rectangular shaped region has a better correlation with the certainty of a feature detector in our domain, since a rectangular shape is dominant (i.e. corners, corridors.)

Equation 2 is computed using the *rectangle rule* [4]. In order to make the computation fast, the step size of the current version is 2 cm which may be too big to provide an accurate integration result. We could get more accurate integration result by using a smaller step size or other numerical integration method, but it is not clear how much the accuracy of computing this integration will affect the overall decision procedure.

3 Feature Detector Module

The Junction Classifying Module assumes that the number of equivalent classes of junctions are limited to a typical indoor environment (e.g. buildings.) The complete list of such junctions is shown in Figure 4. We numbered

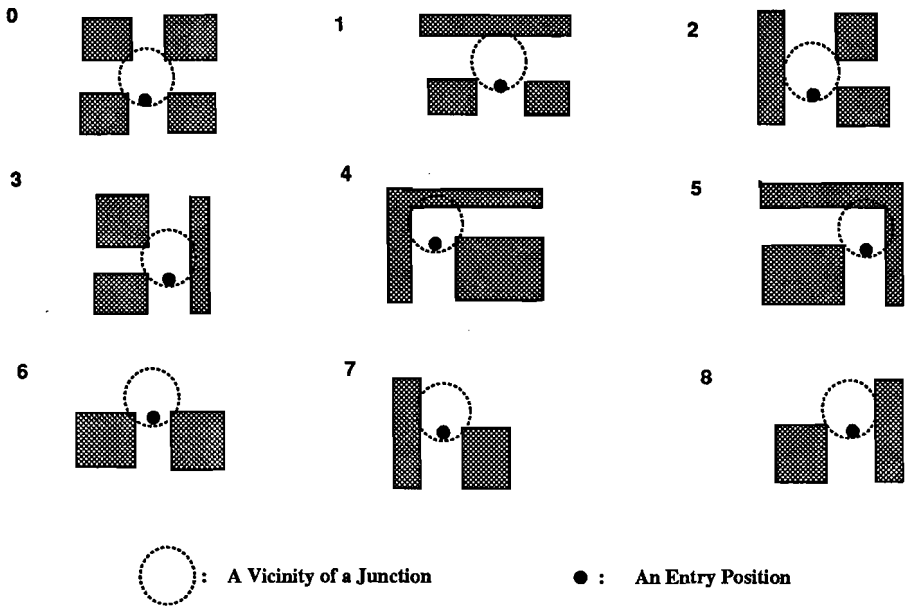

Figure 4: List of Equivalent Classes of Junctions

each equivalent class for convenience and use the number when we refer to it. For example, Class 0 is the equivalent class for a cross junction. In order to classify a junction, the Junction Classifying Module examines the features of the junction using the Feature Detector Module which reports whether a certain feature exist or not.

In the previous version of the Feature Detector Module, features such as walls, convex corners, and concave corners were recognized. A detector for such features prunes the possible equivalent classes of a junction when the feature detector is totally successful in recognizing an expected feature. But it does not take advantage of partial results that may occur from the detector. For example, Figure 5 shows a situation where the robot failed to recognize the top-right convex corner in a Class 3 junction. If we just accept that there is no such corner, we can only eliminate Class 0 and Class 2 from the possible equivalent classes of the junction. But if we interpret the partial result as the indication that Wall 1 exists, we can then eliminate Class 1, Class 4, Class 5, Class 6, and Class 7 from the possible equivalent classes, making it more efficient.

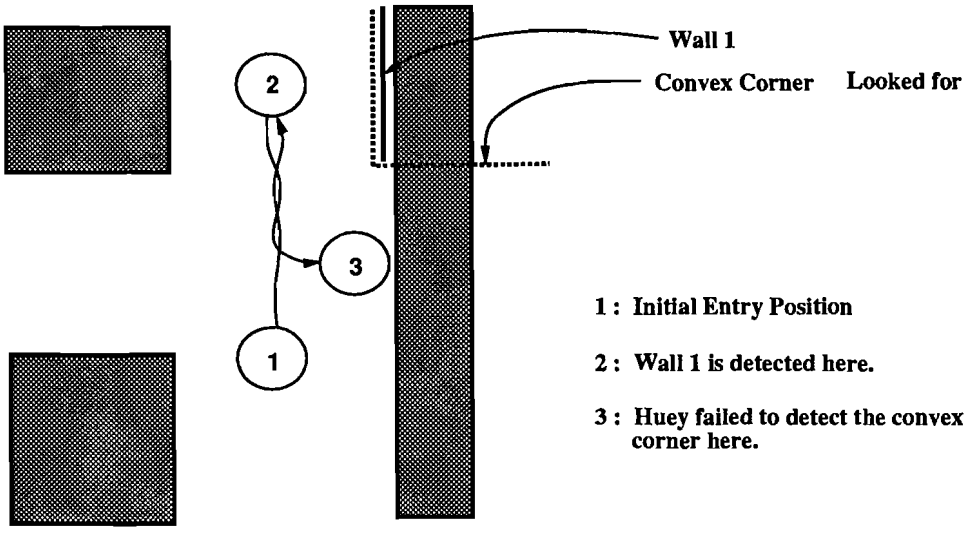

Figure 5: Example of a Partial Result

Features of finer granularity such as a wall rather than a corner help incorporate a partial result, and also the detectors for such features are much simpler. We introduce six such features which are shown in Figure 6. They are also numbered; we refer to the top-left wall as Feature 0. A Class 1 junction has Feature 2, Feature 3, Feature 4, and Feature 5 according to this numbering. This set of features does not contain all possible features such as the wall between Feature 2 and Feature 3 in a Class 1 junction. But we argue that existence(non-existence) of the wall can be determined by non-existence(existence) of Feature 0 or Feature 1, and that such omission does not seriously affect the efficiency of the junction classification procedure. Also the similarity between the six features makes it simple to implement the detectors. The Feature Detector Module has six detectors for each of these features.

Randazza [10] devised and implemented a robust wall detection algorithm which only assumes that the orientation to the *target wall*, the wall a detector is looking for, is known beforehand. This assumption is a reasonable one, because a feature detector is supposed to start at the entry position of a junction. This algorithm exploits the spatial arrangement of the sonar transducers in the sonar ring in order to detect a wall. The Feature Detector

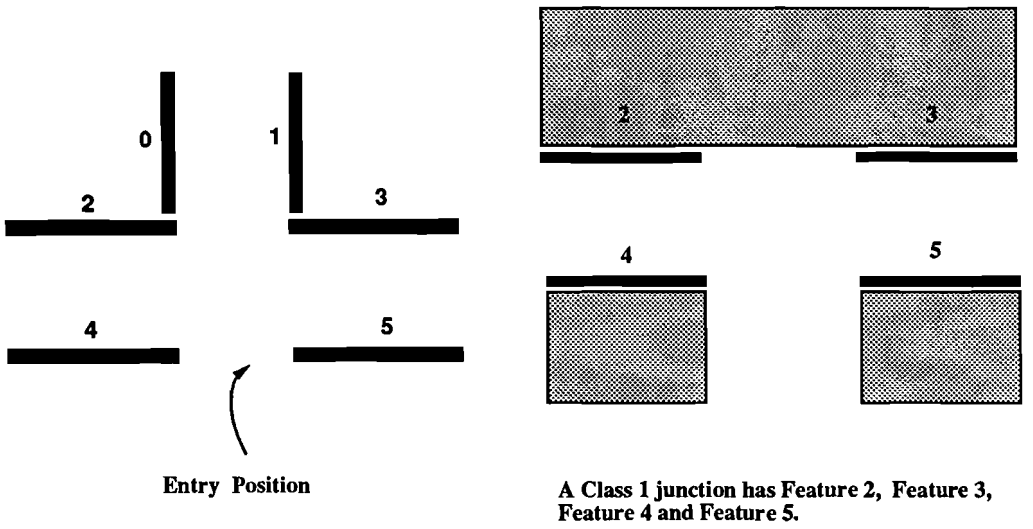

Figure 6: List of Features

Module adopts this algorithm in measuring the certainty that a wall exists. We define a wall as a vertical planar surface which is longer than ℓ. A feature detector first moves Huey to the position where the wall detection algorithm is expected to detect a target wall and then starts to traverse the target wall, running the wall detection algorithm periodically. If the wall detection algorithm reports that the target wall dose not exist, or that the distance traversed so far is longer than ℓ, the feature detector will stop traversing the target wall and return to the entry position. The certainty that the target wall exists is determined by a model which assumes that the certainty is proportional to the distance traversed. Thus, the probability that the target wall exists given the distance traversed is

$$P_{wall}(d) = \begin{cases} \frac{d}{\ell} & \text{if } d < \ell \\ 1 & \text{otherwise} \end{cases} \tag{3}$$

where d is the distance traversed.

Equation 3 assumes that Huey starts a feature detector at the entry position of a *target junction*, the junction being classified, and that the detector starts to traverse a target wall where the wall is expected to be detected.

But this assumption can not always hold, since the positional certainty may decrease as Huey moves around. Experiments show that a feature detector often fails to detect a target wall correctly if it starts at a position which was far from the entry position. Therefore, we should consider the positional certainty of the start position of a feature detector when we decide the certainty that a target wall exists. For that purpose, we introduce the notion of *region of tolerance* (RT henceforth). The feature detectors are guaranteed to work properly if they start within an RT, and the entry position of a target junction is the center of an RT. The size of an RT is represented by τ_x and τ_y which are the length of the x-axis side and the length of the y-axis side respectively. The actual numbers of these lengths were determined from experiments.

Let the AT from an entry position to the start position of a feature detector consist of a mean triple (x^*, y^*, θ^*) and a covariance matrix C^*. The mean triple values are all zeros, because the feature detector starts at the position which it believes to be the entry position, therefore, the probability that Huey is placed within the RT when the feature detector starts is

$$c^* = P_{region}(\tau_x, \tau_y, C^*) \tag{4}$$

where $P_{region}(\tau_x, \tau_y, C^*)$ is from Equation 2.

We use c^*, the positional certainty of a start position relative to an entry position, to degrade the probability that a target wall exists. Since all the features are walls in reality, we will use the term *target feature* instead of target wall. The degraded probability that a target feature exists is then

$$P_{feature}(d, c^*) = P_{wall}(d) + K_1(0.5 - P_{wall}(d))(1 - c^*), \tag{5}$$

$$0 \leq K_1 \leq 1$$

where K_1 is a constant which denotes the degree of degrading and is adjusted by experiments, and $P_{wall}(d)$ is from Equation 3. Equation 5 offers a simple model of a feature detection. For example, in the case that d is 50 cm and c^* is 0.9, the probability that a target feature exists is $P_{feature}(50, 0.9)$, and the probability that it does not exist is $1 - P_{feature}(50, 0.9)$. A feature detector uses this model in deciding the certainty of its result. The result of a feature detector has the form of an ordered pair (α, ϵ), where $\alpha \in \{0, 1\}$ and $\epsilon \in \Re$ such that $0 \leq \epsilon \leq 1$. $\alpha = 1$ means that a target feature exists,

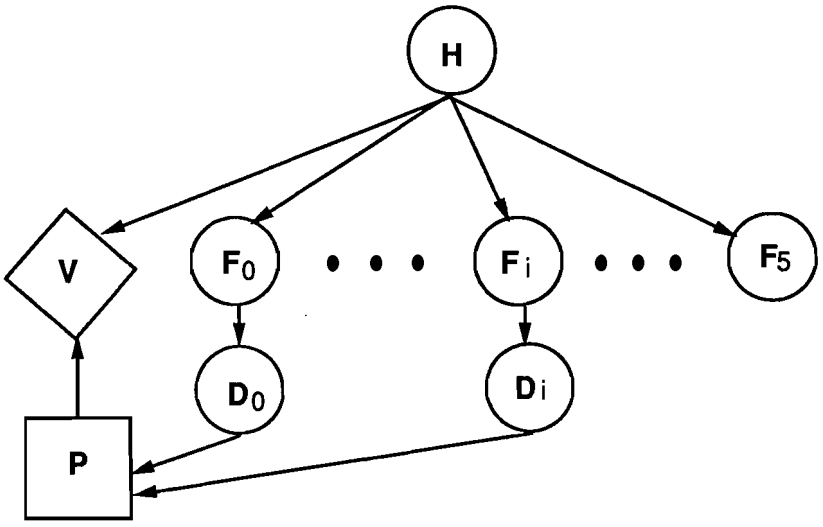

Figure 7: Extended Influence Diagram

and ϵ is the certainty of this answer. In the previous example, the result will be $(1, P_{feature}(50, 0.9))$ or $(0, 1 - P_{feature}(50, 0.9))$, and both results have the same meaning and effect.

4 Decision Model

Among available feature detectors, the Junction Classifying module selects the next detector and assimilates the result of the detection. Since each feature detection is a relatively expensive operation which contains many movements and sonar readings, the Junction Classifying Module optimizes the sequence of detections in order to minimize the number of operations, using the technique by Bayesian decision theory[2]. The *influence diagram* technique is used to represent the model of the detector selection and result assimilation[12, 11]. We have extended the influence diagram used in the previous version of the Junction Classifying Module in order to deal with the uncertainty of the result of a feature detector. The extended influence diagram is shown in Figure 7.

The node H represents the hypothesis about the equivalent class of a

target junction. $H = 0$ means that the target junction is a Class 0 junction. The nodes F_i represent whether a Feature i exists (i.e. $F_i = 1$) or not (i.e. $F_i = 0$). We added the nodes D_i to the previous influence diagram which represent the result of the Feature i detection. $D_i = \alpha$ means that the Feature i detector *has* returned the result (α, ϵ). The decision node P represents the decision to select the next feature detector, and the value node V represents the utility of such a decision.

The influence diagram has three prior distributions, $P(H)$, $P(F_i|H)$, and $P(D_i|F_i)$. $P(H = i)$ is the belief that the target junction is a Class i junction. The conditional probability distribution $P(F_i|H = j)$ gives the probability of the existence of Feature i in a Class j junction using the definitions of the feature and the equivalent class. For example, $P(F_1 = 1|H = 0) = 1$, and $P(F_1 = 0|H = 0) = 0$.[3]

The certainty of a feature detection is represented by the conditional distribution $P(D_i|F_i)$. $P(D_i = \alpha|F_i = \alpha)$ is the probability that the Feature i detection is correct, and $P(D_i = \alpha|F_i = \neg\alpha)$ is the probability that the detection is wrong.[4] Given that the Feature i detection returns (α, ϵ), the distribution is

$$\begin{aligned} P(D_i = \alpha|F_i = \alpha) &= \epsilon \\ P(D_i = \alpha|F_i = \neg\alpha) &= 1 - \epsilon \end{aligned}$$

Since the conditional distribution $P(D_i|F_i)$ can be determined only after the feature detector has finished the detection for Feature i, we add the influence diagram dynamically the node D_i and the arc from F_i to D_i. Figure 7 shows the situation where it has finished two feature detections for Feature 0 and Feature i.

The hypothesis about the equivalent class of a target junction has the initial distribution $P(H = i) = \frac{1}{|H|}$, where $0 \leq i < |H|$.[5] A new result of a feature detection is assimilated by updating the distribution $P(H)$ to $P'(H)$

[3] See Figure 4 and Figure 6.

[4] $\neg 0 = 1$, and $\neg 1 = 0$.

[5] $|H|$ is the size of the hypothesis space, i.e. 9 in our domain.

as follows.

$$
\begin{aligned}
P'(H=i) &= P(H=i|D_j=\alpha) \\
&= \frac{P(D_j=\alpha|H=i)P(H=i)}{P(D_j=\alpha)} \\
&= \frac{P(H=i)\sum_{f=0}^{1}[P(D_j=\alpha|F_j=f)P(F_j=f|H=i)]}{\sum_{h=0}^{|H|-1}\sum_{f=0}^{1}[P(D_j=\alpha|F_j=f)P(F_j=f|H=h)P(H=h)]} \\
&= \frac{P(H=i)[\epsilon\cdot P(F_j=\alpha|H=i)+(1-\epsilon)P(F_j=\neg\alpha|H=i)]}{\sum_{h=0}^{|H|-1}[\epsilon\cdot P(F_j=\alpha|H=h)P(H=h)+(1-\epsilon)P(F_j=\neg\alpha|H=h)P(H=h)]}
\end{aligned}
\tag{6}
$$

where the result of the Feature j detection is (α, ϵ), and $0 \leq i < |H|$. $P'(H)$ is the new belief about the equivalent class of a target junction after the detection of Feature j.

We use Equation 6 to calculate the posterior distribution of the hypothesis instead of the inference engine for influence diagrams[3, 11], because we can predict which kinds of queries about the diagram are necessary, and such queries are fixed. Such direct computation saves running time and space of the Junction Classifying Module.

In order to facilitate the analysis of the utility of a feature detector, we rephrase Equation 6 by introducing a probability function $P_h(i, j, l, c^*)$. This function computes the same posterior distribution as $P'(H = i)$, given that the feature detector for Feature j has traversed the target wall l cm, and that the positional certainty of the start position relative to the entry position was c^*.

$$
P_h(i,j,l,c^*) =
$$

$$
\frac{P(H=i)[P_{feature}(l,c^*)P(F_j=1|H=i)+(1-P_{feature}(l,c^*))P(F_j=0|H=i)]}{\sum_{h=0}^{|H|-1}[P_{feature}(l,c^*)P(F_j=1|H=h)P(H=h)+(1-P_{feature}(l,c^*))P(F_j=0|H=h)P(H=h)]} \tag{7}
$$

$P_{feature}(l, c^*)$ is from Equation 5, and c^* was calculated using Equation 4. $P_h(i, j, l, c^*)$ provides a microscopic view on how the posterior distribution is determined, which is necessary for the analysis.

The feature detector selection requires the utility of each feature detector; the detector which has the best utility will be selected. A major component

of the utility is the *discrimination function.* This computes the expectation on how much a feature detector changes the hypothesis about the equivalent class of a target junction[2, 3].

The discrimination function measures the difference between the current hypothesis and the expected posterior hypothesis. Let c^* be the positional certainty of the start position relative to the entry position from Equation 4. Then, the discrimination function of the Feature i detector is

$$Discrim(i, c^*) = \sum_{l \in \{0, \ell\}} P(d = l) \sum_{h=0}^{|H|-1} |P_h(h, i, l, c^*) - P(H = h)| \quad (8)$$

where $P(d)$ is the distribution which represents the expectation on the distance traversed in the Feature i detection. We assume that $P(d = \ell) = P(F_i = 1)$, and $P(d = 0) = P(F_i = 0)$ in the current implementation. The value of the $Discrim(i, c^*)$ is the expectation on how much the Feature i detector will alter the hypothesis. If we consider the cost of the Feature i detector, the overall utility of the Feature i detector will be

$$U_{detector}(i, c^*) = K_2 \times Discrim(i, c^*) - K_3 \times C_{detector}(i) \quad (9)$$

$C_{detector}(i)$ is the cost function which is based upon the time consumed by the Feature i detector. K_2 and K_3 are constants used for weighting and were adjusted through experiments. From Equation 9, the Feature d^* detector is the best detector which is determined as follows.

$$d^* = \max_{d \in D} U_{detector}(d, c^*) \quad (10)$$

where D is the set of the features which have not yet been examined.

It is possible to lose the entry position of a target junction, as Huey continues to explore features, hence, we sometimes need a method for reducing the positional uncertainty. We adopt a new action called *reacquire action* which repeats the operations of the last stage of a corridor following and makes Huey reacquire the entry position of the target junction. The Junction Classifying Module decides when to execute the reacquire action. This decision is also based upon the utility of the reacquire action.

We can derive the utility of the reacquire action using the same idea used for the utility of a feature detector. The reacquire action does not

change the hypothesis about the equivalent class of a target junction. The positional certainty of the entry position will be reset to 1 as a side effect. From Equation 7, Equation 8, and Equation 9, the higher positional certainty yields better discrimination. Therefore, we use the difference between the discrimination of the best detector after the reacquire action and that of the best detector alone as a major component of the utility of the reacquire action. The difference is

$$Diff(d^*, c^*) = Discrim(d^*, 1) - Discrim(d^*, c^*)$$

Then the utility of the reacquire action is

$$U_{reacquire}(d^*, c^*) = K_4 \times Diff(d^*, c^*) - K_5 \times C_{reacquire} \qquad (11)$$

where $C_{reacquire}$ is a constant which is the cost of the reacquire action based upon time consumed by the action. K_4 and K_5 are arbitrary constants for weighting.

We have derived the utilities of the feature detector and the reacquire action. The Junction Classifying Module chooses the best action using these utilities. A list of available actions for the Junction Classifying Module includes the feature detectors not yet invoked, the reacquire action, and the *quitting* action which ends the junction classification task and reports the result to the Geographer Module Module. The Junction Classifying Module runs a simple decision algorithm as follows:

1. If the set of available feature detectors is empty, quit with reporting i such that $P(H = i) \geq P(H = j)$ where $0 \leq i < |H|$, $0 \leq j < |H|$, and $i \neq j$.

2. If there exists i such that $P(H = i) > \delta$ where $0 \leq i < |H|$, quit with reporting i.

3. Calculate c^* which is the positional certainty of the current position relative to the entry position—The current position will be the start position of a feature detector.

4. Select the best Feature d^* detector according to Equation 10.

5. If $U_{detector}(d^*, c^*) > U_{reacquire}(d^*, c^*)$, invoke the Feature d^* detector, otherwise execute the reacquire action first and then invoke the Feature d^* detector.

The Junction Classifying Module repeats the above algorithm until it exhausts the list of applicable feature detectors, or until a hypothesis for an equivalent class of the target junction is found with high probability.

5 Conclusion

We have described the extensions to the previous version of Huey Geographer. The extensions were keeping track of motion errors and incorporating such errors into a feature detector, extending the influence diagram to handle the uncertainty of a feature detector, introducing a new set of features which are finer in granularity, and adding new action for reducing positional uncertainty. All the extensions were implemented and tested on the robot Huey.[6]

Some experiments have demonstrated that the current version of Huey Geographer is more efficient than the previous version in the sense that it requires less number of wall traversals. For example, in a Class 3 junction, the previous version requires wall traversals which amount to six feature detections in the current version. The current version classifies a Class 3 junction only after three feature detections. Due to the limited length of the power line, we could not test exhaustively all the equivalent classes in our domain. However, the finer granularity of the new features helps to reduce the number of wall traversals, because the feature detectors of the previous version examine some wall segments (i.e. the features in the current version) in duplicate and can not recognize them in detection failures.

Incorporating motion errors and handling the uncertainty of feature detectors help avoid premature commitment to the equivalent class of a target junction based on uncertain feature detections. Adding the reacquire action offers a method to keep the positional uncertainty under control, making the Junction Classifying Module more robust.

[6]The Appendix has the list of source codes.

The weakest point of the current version of the Geographer Module is the assumption that the domain has a limited number of equivalent classes of junctions. If Huey operates in a new environment which has equivalent classes not included in Figure 4, the Junction Classifying Module should be extended to include the new equivalent classes and features. The recent work by Leonard and Durrant-Whyte[9] could be applied to eliminate this assumption. Their work offers a method to build a map which is just a list of *target* locations, where targets are geometric features such as corners, planes, and cylinders. Their system keeps the interpretations about the targets and reduces them as a robot moves around and acquires more sonar data from its environments. We could devise some strategies of motion and sensing to prune the interpretations quickly and assign utilities to such strategies. The technique described in Section 4 could be used for choosing the best strategy. A junction and a corridor could be represented by the targets around it.

The obvious next step of the Geographer project is building an actual map[8]. A map shown in the Figure 1 can be built using the current Geographer Module, if we had a method to identify a previously visited LDP. Adding other kinds of sensors such as a camera would facilitate such an identification, but it is still interesting to build a map using only sonar sensors. In the case that the equivalent classes in Figure 4 contains all the equivalent classes of an environment, and that junctions of an equivalent class are not near to each other, it seems likely that the system can identify a previously visited LDP correctly using the result of the junction classification, the positional information from dead-reckoning, and the map built so far. It would also be interesting to analyze the identification certainty using the certainty of a junction classification, the motion error, and the information about the environment such as the distribution of equivalent classes, the average distances between two junctions of an equivalent class[1].

Great flexibility can be obtained if the feature detector can be interrupted at any point during its execution[6]. Then the time dependent planning technique can be applied to the Junction Classifying Module which optimizes the classification procedure according to the time constraint set by a module of higher level task[5]. The feature detectors should be redesigned so that it will be able to provide the probability of the existence of a feature at any time, and that new sensory data will be assimilated into the probability. It is not clear how much the time-dependent technique will increase the efficiency

of the Junction Classifying Module, but it is certain that the Junction Classifying Module with time constraint will facilitate the decision procedure of the module which allocates time to exploration and execution of a task.

6 Acknowledgment

First of all, I would like to thank Tom Dean for his advice throughout this research. I also like to thank the former "Control Architecture Team" members, Rob Chekaluk and Meg Randazza, for offering the basis of this research. Many thanks go to Moi Lejter for helping the implementation and experiments. Finally, I would like to thank Jin Joo Lee for her comments on this paper.

7 Reference

1. Basye, Kenneth and Dean, Thomas and Vitter, Jeffrey Scott, "Coping with Uncertainty in Map Learning", *Proceedings IJCAI 11, Detroit, Michigan*, pp. 663-668, IJCAI, 1989.

2. Cameron, Alec and Durrant-Whyte, Hugh, "A Bayesian Approach to Optimal Sensor Placement", *Technical Report*, Department of Engineering Science, University of Oxford, October 1988.

3. Chekaluk, Robert A.,"Using Influence Diagrams in Recognizing Locally-Distinctive Places", *M.Sc. Thesis*, Department of Computer Science, Brown University, 1989.

4. Conte, Samuel D. and Boor, Carl de, *Elementary Numerical Analysis: An Algorithmic Approach, 3rd Ed.*, McGraw-Hill, Inc., 1980.

5. Dean, Thomas and Boddy, Mark, "An Analysis of Time-Dependent Planning", *Proceedings AAAI-88*, pp. 49-54, AAAI, 1988.

6. Dean, Thomas, "Decision-Theoretic Control of Inference for Time-Critical Applications", *Technical Report* No. CS-89-44, Department of Computer Science, Brown University, November 1989.

7. Kuipers, Benjamin J., "Modeling Spatial Knowledge", *Cognitive Science*, 2: pp. 129-153, 1978.

8. Kuipers, Benjamin J. and Yung-Tai Byun, "A Robust, qualitative method for robot spatial reasoning", *Proceedings AAAI-88*, pp. 774-779, AAAI, 1988.

9. Leonard, John J. and Durrant-Whyte, Hugh F., "A Unified Approach to Mobile-Robot Navigation", *Technical Report*, Department of Engineering Science, University of Oxford, September 1989.

10. Randazza, Margaret J., "The Feature Recognition Module of the LDP System for the Robot Huey", *M.Sc. Thesis*, Department of Computer Science, Brown University, 1989.

11. Rege, Ashutosh and Agogino, Alice M.,"Topological Framework for Representing and Solving Probabilistic Inference Problems in Expert Systems", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 18, No. 3, May/June 1988.

12. Shachter, Ross D., "Evaluating Influence Diagrams", *Operations Research*, Vol. 34, No.6 November-December 1986.

13. Smith, Randall C. and Cheeseman, Peter, "On the Representation and Estimation of Spatial Uncertainty", *The International Journal of Robotics Research*, Vol. 5, No. 4, pp. 56-68, Winter 1986.

A List of Constants

The current implementation has some constants which were determined from experiments and intuitions. For those who might want to extend or just understand the current version of the Geographer Module, we give the values of all the constants used in the current version as follows:

- ℓ in Equation 3 is 60 cm.
- τ_x in Equation 4 is 60 cm.
- τ_y in Equation 4 is 40 cm.
- K_1 in Equation 5 is 1.
- K_2 in Equation 9 is 1.
- K_3 in Equation 9 is 0.1.
- $C_{detector}(i)$ in Equation 9 is 60, where $0 \leq i \leq 5$. We assume that all detectors consume same amount of time.
- K_4 in Equation 11 is 2.
- K_5 in Equation 11 is 0.1.
- $C_{require}$ in Equation 11 is 40.
- δ in the decision algorithm is 0.75.

B List of Source Codes

```
/*
*       CFMinterface.h
*
*     This file is the header file for interfacing with Corridor Follower
*     Module.
*/

/* type for request and reply messages */

typedef struct {
    int type;
    int returnMsg;
} CFMmsg;

/* type numbers for the type field */

#define typeCFMcorridor         1
#define typeCFMreacquire        2


/* type numbers for the returnMsg  */

#define CFM_OPEN                101    /* corridor following successful */
#define CFM_DEADEND             102    /* meets a dead end              */
#define CFM_FAILED              103    /* failed due to ???             */
#define CFM_LOST                104    /* lost ???                      */

/* interface

   CFMfollow
        ingoing : type
        outgoing: type, returnMsg

        returnMsg is CFM_OPEN : The robot found a new LDP. The robot is at
                                an entrance of the new LDP.
                                Everything is normal.

        returnMsg is CFM_DEADEND : The robot met a deadend while traversing a
                                   hallway. The robot returned to the
                                   entrance of the initial LDP with the opposite
                                   orientation to the initial one.

        returnMsg is CFM_FAILED : Some obstacles were playing too near to
                                  the robot while the robot was traversing a
                                  hallway. the robot gave up going on and
                                  returned to the entrance of the initial
                                  LDP with the opposite orientation to
                                  the initial one.

        returnMsg is CFM_LOST   : The robot doesn't know what to do, the robot
                                  is in a hallway, got lost its position.
                                  This is the worst case.

*/
```

```
#include <stdio.h>
#include "/pro/ai/robot/software/ipc/src/ipc.h"
#include "/pro/ai/robot/software/misc/src/error.h"
#include "CFMinterface.h"
#include <setjmp.h>

/* CFM.c
*
*  main program of  CFM module
*
*       author : Seungseok Hyun
*       date   : Aug. 21, 1989
*/

/* main
*
*       main gets requests by the ipc message passing mechanism,
*       calls procedures in corridor.c according to the requests, and
*       replies.
*
*/

jmp_buf errorHandler;
clientId WAI_id, LLC_id, SC_id;

main()
{

    clientId senderId, myOwnId;
    CFMmsg message;
    int requestLength;

    myOwnId = NSregisterSelf("CFM", 10);

    while((WAI_id = NSgetClient("WAI")) == nullClient)
          sleep(1);

    while((LLC_id = NSgetClient("LLC")) == nullClient)
          sleep(1);

    while((SC_id = NSgetClient("SC")) == nullClient)
          sleep(1);

    /* This program never halts. */

    while ( 1 ) {

        requestLength = sizeof(CFMmsg);

        do
            senderId = IPCrecvMessage(&message, &requestLength, 5);
        while (senderId == nullClient);

        switch(message.type) {
            case typeCFMcorridor:
                message.returnMsg = CFMcorridor();
                IPCsendMessage(&message, sizeof(CFMmsg), senderId);
                break;
            case typeCFMreacquire:
                message.returnMsg = CFMreacquire();
                IPCsendMessage(&message, sizeof(CFMmsg), senderId);
                break;
            default:
                fprintf(stderr,"*** CFM got wrong request.\n");
```

```
            }
        }
    }
```

```
#define TRUE       1
#define FALSE      0
#define NOT_KNOWN  -9991
#define DISTANCE_FOR_FIRST_MOVEMENT 8 /* 8 cm */
#define CENTERING_ERROR 4 /* 4 cm */

#define TRAVELING_SPEED          11       /* 11 cm/sec  */
#define HALF_TRAVELING_SPEED      5       /* 5 cm/sec   */

#define SMALL_SAFTY_RADIUS  30       /* 30 cm */
#define LARGE_SAFTY_RADIUS  40       /* 40 cm */

#define MAX_DISTANCE_TO_WALL 100      /* 100 cm */

#define CFM_TURN_SPEED            15       /* 15 degrees/sec */
#define CFM_SLOW_TURN_SPEED       10       /* 10 degrees/sec */
#define CFM_VERY_SLOW_TURN_SPEED  5        /*  5 degrees/sec */

#define CFM_SLOW_SPEED            5       /* 5 cm/sec */
#define CFM_VERY_SLOW_SPEED       3       /* 3 cm/sec */

#define CFM_NUMBER_OF_RETRY       2       /* 2 times */

#define CFM_ENTERING_BACK         40      /* 40 cm */
#define CFM_ENTERING_BACKUP       15      /* 15 cm */

#define CFM_MOVE_PARALLEL            30
#define CFM_MOVE_RIGHTWARD           31
#define CFM_MOVE_LEFTWARD            32

#define CFM_ADJUST_DEGREE            24      /* 24 degrees */
#define CFM_ERROR_COMP                2      /* 2 degrees */

#define CFM_MOVING_BACK               30      /* 30 cm */
```

```
#include "CFM.h"
#include "/pro/ai/robot/software/huey/frm/src/sonar.h"
#include "/pro/ai/robot/software/huey/frm/src/frm.h"

/* CFMcenter()
*
*    This functions places the robot at the center of a hallway.
*
*       author : Seungseok Hyun
*       Date   : September 15, 1989
*
* Credits
*
*     algorithm
*          author : Jin Joo Lee
*          date : September, 1989
*
*     functions
*        SONARget_average_data(), SONARorient_parallel(), rotate_robot(),
*        translate_robot()
*           author :  Margaret J. Randazza
*           date : September, 1989
*
*/

CFMcenter()
{
        short ring[NUM_SONARS];
        int   i, smallest, smallestIndex;
        int   shorterSide;

    /* check which side is nearer to the robot */

        SONARget_average_data(ring);
        smallest = SONAR_RANGE + 1;
        for (i = 0; i <= 15; i++)
            if (ring[i] < smallest){
                smallest = ring[i];
                smallestIndex = i;
            }
        if (smallestIndex >= FRONT_SONAR && smallestIndex <= BACK_SONAR)
            shorterSide = RIGHT;
        else
            shorterSide = LEFT;

    /* make the robot parallel to the nearer side */

        SONARorient_parallel(shorterSide);

    /* make the robot to move to the center of a hall way */

        rotate_robot(-90, CFM_TURN_SPEED);
        do{
            SONARget_average_data(ring);
            translate_robot(ring[FRONT_SONAR] - ring[BACK_SONAR],
                            CFM_VERY_SLOW_SPEED);
        } while (abs(ring[FRONT_SONAR] - ring[BACK_SONAR]) > CENTERING_ERROR);
        rotate_robot(90, CFM_TURN_SPEED);
}
```

```
#include "CFM.h"
#include "CFMinterface.h"
#include "/pro/ai/robot/software/huey/frm/src/sonar.h"
#include "/pro/ai/robot/software/huey/frm/src/frm.h"


/* CFMcorridor()
*
*    This function moves the robot to an entrance of a LDP in front of the
*  robot.
*
*       author : Seungseok Hyun
*       Date   : September 15, 1989
*
* Credits
*
*     algorithm
*           author : Jin Joo Lee
*           date : September, 1989
*
*     functions
*        start_robot_translation(), stop_robot_translation()
*        SONARget_average_date()
*           author :  Margaret J. Randazza
*           date : September, 1989
*
*/

int CFMcorridor()
{

        short ring[NUM_SONARS];
        int result, tempResult;
        int numberOfFailure, tempNumberOfFailure;

        extern int movingDirection;

        numberOfFailure = 0;

    /* place the robot at the center of a hallway */

        CFMcenter();

    /* traverse a hallway */

        result = CFMfollow();

        if (result != CFM_OPEN) {
            translate_robot(-CFM_MOVING_BACK, HALF_TRAVELING_SPEED);
            CFMcenter();
            tempResult = CFMfollow();
            while ((tempResult != CFM_OPEN) &&
                    numberOfFailure < CFM_NUMBER_OF_RETRY){
                numberOfFailure++;
                translate_robot(-CFM_MOVING_BACK, HALF_TRAVELING_SPEED);
                CFMcenter();
                tempResult = CFMfollow();
            }
            if (tempResult == CFM_OPEN)
                result = CFM_OPEN;
        }

    /* back to starting LDP */
```

```
    if (result == CFM_DEADEND){
        rotate_robot(180, CFM_SLOW_TURN_SPEED);
        translate_robot(CFM_MOVING_BACK, HALF_TRAVELING_SPEED);
        CFMcenter();
        tempResult = CFMfollow();
        tempNumberOfFailure = 0;
        while (tempResult != CFM_OPEN &&
            tempNumberOfFailure <= CFM_NUMBER_OF_RETRY){
            tempNumberOfFailure++;
            translate_robot(-CFM_MOVING_BACK, HALF_TRAVELING_SPEED);
            CFMcenter();
            tempResult = CFMfollow();
        }
        if (tempResult == CFM_DEADEND){
            return(CFM_LOST);
        }
    }

/* place the robot at an appropriate position for mjr's module */

    start_robot_translation(-HALF_TRAVELING_SPEED);
    switch (movingDirection) {
        case CFM_MOVE_PARALLEL:
            do {
                SONARget_average_data(ring);
            } while ((ring[3] > MAX_DIST_TO_WALL &&
                      ring[4] > MAX_DISTANCE_TO_WALL) ||
                     (ring[4] > MAX_DIST_TO_WALL &&
                      ring[5] > MAX_DISTANCE_TO_WALL) ||
                     (ring[11] > MAX_DIST_TO_WALL &&
                      ring[12] > MAX_DISTANCE_TO_WALL) ||
                     (ring[12] > MAX_DIST_TO_WALL &&
                      ring[13] > MAX_DISTANCE_TO_WALL));
            break;
        case CFM_MOVE_RIGHTWARD:
            do {
                SONARget_average_data(ring);
            } while ((ring[2] > MAX_DIST_TO_WALL &&
                      ring[3] > MAX_DISTANCE_TO_WALL) ||
                     (ring[3] > MAX_DIST_TO_WALL &&
                      ring[4] > MAX_DISTANCE_TO_WALL) ||
                     (ring[10] > MAX_DIST_TO_WALL &&
                      ring[11] > MAX_DISTANCE_TO_WALL) ||
                     (ring[11] > MAX_DIST_TO_WALL &&
                      ring[12] > MAX_DISTANCE_TO_WALL));
            break;
        case CFM_MOVE_LEFTWARD:
            do {
                SONARget_average_data(ring);
            } while ((ring[4] > MAX_DIST_TO_WALL &&
                      ring[5] > MAX_DISTANCE_TO_WALL) ||
                     (ring[5] > MAX_DIST_TO_WALL &&
                      ring[6] > MAX_DISTANCE_TO_WALL) ||
                     (ring[12] > MAX_DIST_TO_WALL &&
                      ring[13] > MAX_DISTANCE_TO_WALL) ||
                     (ring[13] > MAX_DIST_TO_WALL &&
                      ring[14] > MAX_DISTANCE_TO_WALL));
            break;
    }
    stop_robot_translation();
    switch (movingDirection) {
        case CFM_MOVE_RIGHTWARD:
            rotate_robot(-CFM_ADJUST_DEGREE, CFM_SLOW_TURN_SPEED);
            break;
```

```
            case CFM_MOVE_LEFTWARD:
                rotate_robot(CFM_ADJUST_DEGREE, CFM_SLOW_TURN_SPEED);
                break;
        }

        translate_robot(-CFM_ENTERING_BACK+CFM_ENTERING_BACKUP,
                        HALF_TRAVELING_SPEED);
        /* center the robot */
        CFMcenter();

        return(result);
}

int CFMreacquire()
{
    short ring[NUM_SONARS];
    int result;

    extern int movingDirection;

    /* move backward */

    translate_robot(-CFM_ENTERING_BACK, TRAVELING_SPEED);

    /* traverse a hallway */

    result = CFMfollow();

    /* place the robot at an appropriate position for mjr's module */

        start_robot_translation(-HALF_TRAVELING_SPEED);
        switch (movingDirection) {
            case CFM_MOVE_PARALLEL:
                do {
                    SONARget_average_data(ring);
                } while ((ring[3] > MAX_DIST_TO_WALL &&
                          ring[4] > MAX_DISTANCE_TO_WALL) ||
                         (ring[4] > MAX_DIST_TO_WALL &&
                          ring[5] > MAX_DISTANCE_TO_WALL) ||
                         (ring[11] > MAX_DIST_TO_WALL &&
                          ring[12] > MAX_DISTANCE_TO_WALL) ||
                         (ring[12] > MAX_DIST_TO_WALL &&
                          ring[13] > MAX_DISTANCE_TO_WALL));
                break;
            case CFM_MOVE_RIGHTWARD:
                do {
                    SONARget_average_data(ring);
                } while ((ring[2] > MAX_DIST_TO_WALL &&
                          ring[3] > MAX_DISTANCE_TO_WALL) ||
                         (ring[3] > MAX_DIST_TO_WALL &&
                          ring[4] > MAX_DISTANCE_TO_WALL) ||
                         (ring[10] > MAX_DIST_TO_WALL &&
                          ring[11] > MAX_DISTANCE_TO_WALL) ||
                         (ring[11] > MAX_DIST_TO_WALL &&
                          ring[12] > MAX_DISTANCE_TO_WALL));
                break;
            case CFM_MOVE_LEFTWARD:
                do {
                    SONARget_average_data(ring);
                } while ((ring[4] > MAX_DIST_TO_WALL &&
                          ring[5] > MAX_DISTANCE_TO_WALL) ||
                         (ring[5] > MAX_DIST_TO_WALL &&
                          ring[6] > MAX_DISTANCE_TO_WALL) ||
                         (ring[12] > MAX_DIST_TO_WALL &&
```

```
                        ring[13] > MAX_DISTANCE_TO_WALL) ||
                       (ring[13] > MAX_DIST_TO_WALL &&
                        ring[14] > MAX_DISTANCE_TO_WALL));
                break;
        }
        stop_robot_translation();
        switch (movingDirection) {
            case CFM_MOVE_RIGHTWARD:
                rotate_robot(-CFM_ADJUST_DEGREE, CFM_SLOW_TURN_SPEED);
                break;
            case CFM_MOVE_LEFTWARD:
                rotate_robot(CFM_ADJUST_DEGREE, CFM_SLOW_TURN_SPEED);
                break;
        }

        translate_robot(-CFM_ENTERING_BACK+CFM_ENTERING_BACKUP,
                        HALF_TRAVELING_SPEED);

    /* center the robot */

    CFMcenter();
    return(result);
}
```

```
#include "CFM.h"
#include "CFMinterface.h"
#include "/pro/ai/robot/software/huey/frm/src/sonar.h"
#include "/pro/ai/robot/software/huey/frm/src/frm.h"
#include <stdio.h>

/* CFMfollow()
*
*    This functions moves the robot along a hallway until the robot detects
*  an opening of the walls of the hallway, or the hallway is blocked.
*
*       author : Seungseok Hyun
*       date   : October 12, 1989
*
* Credits
*
*     algorithm
*          author : Jin Joo Lee
*          date : September, 1989
*
*     functions
*        start_robot_translation(), stop_robot_translation()
*        SONARget_average_data()
*           author :  Margaret J. Randazza
*           date : September, 1989
*
*/

int movingDirection;

int CFMfollow()
{

        short ring[NUM_SONARS];
        int i, smallest, smallestSonar;
        int smallestLeft, smallestRight;

        movingDirection = CFM_MOVE_PARALLEL;


        start_robot_translation(TRAVELING_SPEED);

        while (1) {

            SONARget_average_data(ring);

            /* check if  the hallway is blocked. */

            switch (movingDirection) {
                case CFM_MOVE_PARALLEL:
                    if (ring[15] < LARGE_SAFTY_RADIUS ||
                        ring[0]  < LARGE_SAFTY_RADIUS ||
                        ring[1]  < LARGE_SAFTY_RADIUS) {
                            stop_robot_translation();
fprintf(stderr,"CFM_DEADEND\n");
                            return(CFM_DEADEND);
                    }
                    break;
                case CFM_MOVE_RIGHTWARD:
                    if (ring[14] < LARGE_SAFTY_RADIUS ||
                        ring[15]  < LARGE_SAFTY_RADIUS ||
                        ring[0]  < LARGE_SAFTY_RADIUS) {
                            stop_robot_translation();
fprintf(stderr,"CFM_DEADEND\n");
```

```
                        return(CFM_DEADEND);
                    }
                    break;
                case CFM_MOVE_LEFTWARD:
                    if (ring[0] < LARGE_SAFTY_RADIUS ||
                        ring[1]  < LARGE_SAFTY_RADIUS ||
                        ring[2]  < LARGE_SAFTY_RADIUS) {
                            stop_robot_translation();
fprintf(stderr,"CFM_DEADEND\n");
                            return(CFM_DEADEND);
                    }
                    break;
            }

            /* check if there's an opening to a side */

            switch (movingDirection) {
                case CFM_MOVE_PARALLEL:
                    if ((ring[3] > MAX_DIST_TO_WALL &&
                         ring[4] > MAX_DISTANCE_TO_WALL) ||
                        (ring[4] > MAX_DIST_TO_WALL &&
                         ring[5] > MAX_DISTANCE_TO_WALL) ||
                        (ring[11] > MAX_DIST_TO_WALL &&
                         ring[12] > MAX_DISTANCE_TO_WALL) ||
                        (ring[12] > MAX_DIST_TO_WALL &&
                         ring[13] > MAX_DISTANCE_TO_WALL)) {
                        stop_robot_translation();
                        return(CFM_OPEN);
                    }
                    break;
                case CFM_MOVE_RIGHTWARD:
                    if ((ring[2] > MAX_DIST_TO_WALL &&
                         ring[3] > MAX_DISTANCE_TO_WALL) ||
                        (ring[3] > MAX_DIST_TO_WALL &&
                         ring[4] > MAX_DISTANCE_TO_WALL) ||
                        (ring[10] > MAX_DIST_TO_WALL &&
                         ring[11] > MAX_DISTANCE_TO_WALL) ||
                        (ring[11] > MAX_DIST_TO_WALL &&
                         ring[12] > MAX_DISTANCE_TO_WALL)) {
                        stop_robot_translation();
                        return(CFM_OPEN);
                    }
                    break;
                case CFM_MOVE_LEFTWARD:
                    if ((ring[4] > MAX_DIST_TO_WALL &&
                         ring[5] > MAX_DISTANCE_TO_WALL) ||
                        (ring[5] > MAX_DIST_TO_WALL &&
                         ring[6] > MAX_DISTANCE_TO_WALL) ||
                        (ring[12] > MAX_DIST_TO_WALL &&
                         ring[13] > MAX_DISTANCE_TO_WALL) ||
                        (ring[13] > MAX_DIST_TO_WALL &&
                         ring[14] > MAX_DISTANCE_TO_WALL)) {
                        stop_robot_translation();
                        return(CFM_OPEN);
                    }
                    break;
            }

            /* check if the robot is too near to walls */

            switch (movingDirection) {
                case CFM_MOVE_PARALLEL:
                    smallestLeft = SmallestOfThree(ring[11], ring[12],
                                                   ring[13]);
```

```
                    smallestRight = SmallestOfThree(ring[3], ring[4], ring[5]);
                    if (abs(smallestLeft - smallestRight) > 7*CENTERING_ERROR){
                        stop_robot_translation();
                        if (smallestLeft > smallestRight) {
                            movingDirection = CFM_MOVE_LEFTWARD;
                            rotate_robot(-CFM_ADJUST_DEGREE,
                                         CFM_SLOW_TURN_SPEED);
                        }
                        else {
                            movingDirection = CFM_MOVE_RIGHTWARD;
                            rotate_robot(CFM_ADJUST_DEGREE,
                                         CFM_SLOW_TURN_SPEED);
                        }
                        start_robot_translation(TRAVELING_SPEED);
                    }
                    break;
                case CFM_MOVE_RIGHTWARD:
                    smallestLeft = SmallestOfThree(ring[10], ring[11],
                                                   ring[12]);
                    smallestRight = SmallestOfThree(ring[2], ring[3], ring[4]);
                    if (smallestLeft > smallestRight) {
                        stop_robot_translation();
                        movingDirection = CFM_MOVE_PARALLEL;
                        rotate_robot(-(CFM_ADJUST_DEGREE - CFM_ERROR_COMP),
                                     CFM_SLOW_TURN_SPEED);
                        start_robot_translation(TRAVELING_SPEED);
                    }
                    break;
                case CFM_MOVE_LEFTWARD:
                    smallestLeft = SmallestOfThree(ring[12], ring[13],
                                                   ring[14]);
                    smallestRight = SmallestOfThree(ring[4], ring[5], ring[6]);
                    if (smallestLeft < smallestRight) {
                        stop_robot_translation();
                        movingDirection = CFM_MOVE_PARALLEL;
                        rotate_robot(CFM_ADJUST_DEGREE - CFM_ERROR_COMP,
                                     CFM_SLOW_TURN_SPEED);
                        start_robot_translation(TRAVELING_SPEED);
                    }
                    break;
            }
        }
}

int SmallestOfThree(i,j,k)
int i,j,k;
{
    if (i < j)
        if (i < k) return(i);
        else return(k);
    else
        if (j < k) return(j);
        else return(k);
}
```

```
/* FDMinterface.h
*
* header file for the interface with Feature Detection Module
*
*
*/

/*  message types  */

#define typeFDMdetectWallZero   10
#define typeFDMdetectWallOne    11
#define typeFDMdetectWallTwo    12
#define typeFDMdetectWallThree  13
#define typeFDMdetectWallFour   14
#define typeFDMdetectWallFive   15

/* type definition of the message */

typedef struct {
    int type;
    int entryPosition;
    float certainty;
} FDMmsg;
```

```
#include <stdio.h>
#include "/pro/ai/robot/software/ipc/src/ipc.h"
#include "/pro/ai/robot/software/misc/src/error.h"
#include "FDMinterface.h"

/* FDM.c
*
*  main program of  FDM module
*
*       author : Seungseok Hyun
*       date   : Jan. 19, 1990
*/

/* main
*
*       main gets requests by the ipc message passing mechanism,
*       calls procedures in fdm.c according to the requests, and
*       replies.
*
*/

clientId WAI_id, LLC_id, SC_id, myOwnId;

main()
{

    clientId senderId;
    FDMmsg message;
    int requestLength;
    float DetectWallZero(), DetectWallOne(), DetectWallTwo(),
          DetectWallThree(), DetectWallFour(), DetectWallFive();


    myOwnId = NSregisterSelf("FDM", 10);

    while((WAI_id = NSgetClient("WAI")) == nullClient)
          sleep(1);

    while((LLC_id = NSgetClient("LLC")) == nullClient)
          sleep(1);

    while((SC_id = NSgetClient("SC")) == nullClient)
          sleep(1);

    /* This program never halts. */

    while ( 1 ) {

        requestLength = sizeof(FDMmsg);

        do
            senderId = IPCrecvMessage(&message, &requestLength, 5);
        while (senderId == nullClient);

        switch(message.type) {
            case typeFDMdetectWallZero:
                message.certainty = DetectWallZero(message.entryPosition);
                IPCsendMessage(&message, sizeof(FDMmsg), senderId);
                break;
            case typeFDMdetectWallOne:
                message.certainty = DetectWallOne(message.entryPosition);
                IPCsendMessage(&message, sizeof(FDMmsg), senderId);
                break;
            case typeFDMdetectWallTwo:
```

```
                message.certainty = DetectWallTwo(message.entryPosition);
                IPCsendMessage(&message, sizeof(FDMmsg), senderId);
                break;
            case typeFDMdetectWallThree:
                message.certainty = DetectWallThree(message.entryPosition);
                IPCsendMessage(&message, sizeof(FDMmsg), senderId);
                break;
            case typeFDMdetectWallFour:
                message.certainty = DetectWallFour(message.entryPosition);
                IPCsendMessage(&message, sizeof(FDMmsg), senderId);
                break;
            case typeFDMdetectWallFive:
                message.certainty = DetectWallFive(message.entryPosition);
                IPCsendMessage(&message, sizeof(FDMmsg), senderId);
                break;
            default:
                fprintf(stderr, "FDM unknow type request \n");
                break;
        }
    }
}
```

```
/*
*  header file for Feature Detection Module
*
*
*/

#define TRUE 1
#define FALSE 0

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

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

#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 LEFT   1
#define RIGHT  2
#define FRONT  3
#define BACK   4

#define VERY_SLOW_SPEED       4  /* A very slow speed for translation */
#define SLOW_SPEED            10  /* A slow speed for translation */
#define SLOW_TURN_SPEED       16  /* For rotations between 5 and 90 degrees */

#define SAFTY_RADIUS            50              /* 40 cm  */
#define MAX_DIST_TO_WALL        100             /* 100 cm */
#define WALL_LENGTH             60              /*  60 cm */

#define WEIGHT_K             1.0
#define SONAR_ERROR             4               /* 4 cm */
```

```
#include <stdio.h>
#include "/pro/ai/robot/software/huey/wai/src/WAIinterface.h"
#include "/pro/ai/robot/software/misc/src/error.h"
#include "/pro/ai/robot/software/ipc/src/ipc.h"
#include "utils.h"
#include "fdm.h"

/*
*    Feature Detection Module
*
*
*    Author: Seungseok Hyun
*
*    Date: Jan. 19, 1990
*/

/* Wall numbering
*
*            0 |    | 1
*              |    |
*         2 ----    ---- 3
*
*         4 ----    ---- 5
*                x
*
*    x : entry position of an LDP
*
*/

/* external global variables for client Ids. These are set by  */
/* FDM.c                                                       */

extern clientId WAI_id, SC_id, LLC_id, myOwnId;

/* DetectWallZero()
*
*    DetectWallZero detects the front-left wall of an LDP,
* and returns the certainty of the detection.
*
*    It assumes that the robot is placed at the entry position of an
* LDP.
*
*/


float DetectWallZero(entryPosition)
int entryPosition;
{
    unsigned short ring[16];
    int distanceTravelled, distanceAtStart, distanceAtCurrent;
    int centerPosition, portPosition;
    float result, certaintyWall, certaintyPosition;
    float CertaintyWall();
    WAIreplyMsg *Get_Point_Data();
    int openRight, openLeft;

    certaintyPosition = Get_Point_Data(typeFHMcertaintyToCurrent,
                                       entryPosition)->certainty;

    portPosition = -1;

    /* move to the center of a junction */

    translate_robot(SAFTY_RADIUS, SLOW_SPEED);
```

```
    /* set open direction */

    openRight = FALSE;
    openLeft = FALSE;

    SONARget_average_data(ring);
    if ((ring[3] > MAX_DIST_TO_WALL && ring[4] > MAX_DIST_TO_WALL)  ||
        (ring[4] > MAX_DIST_TO_WALL && ring[5] > MAX_DIST_TO_WALL ))
        openRight = TRUE;
    if ((ring[11] > MAX_DIST_TO_WALL && ring[12] > MAX_DIST_TO_WALL) ||
        (ring[12] > MAX_DIST_TO_WALL && ring[13] > MAX_DIST_TO_WALL ))
        openLeft = TRUE;

fprintf(stderr, "openRight %d openLeft %d \n", openRight, openLeft);

    /* save CenterPosition */

    centerPosition = Save_Point();

    /* go forward until it meets a corridor, or it has moved more than     */
    /* the length of a corridor, or                                        */
    /* it meets an obstacle.                                               */

    distanceAtStart = distanceAtCurrent = get_current_distance();

    start_robot_translation(SLOW_SPEED);

    SONARget_average_data(ring);

    if (openLeft == TRUE)
        while (ring[15] > SAFTY_RADIUS && ring[0] > SAFTY_RADIUS &&
               ring[1] > SAFTY_RADIUS &&
               (ring[11] > MAX_DIST_TO_WALL || ring[12] > MAX_DIST_TO_WALL ||
                ring[13] > MAX_DIST_TO_WALL) &&
               (distanceAtCurrent - distanceAtStart) <= MAX_DIST_TO_WALL) {

            distanceAtCurrent = get_current_distance();
            SONARget_average_data(ring);
        }
    else if (openRight == TRUE)
        while (ring[15] > SAFTY_RADIUS && ring[0] > SAFTY_RADIUS &&
               ring[1] > SAFTY_RADIUS &&
               (ring[3] > MAX_DIST_TO_WALL || ring[4] > MAX_DIST_TO_WALL ||
                ring[5] > MAX_DIST_TO_WALL) &&
               (distanceAtCurrent - distanceAtStart) <= MAX_DIST_TO_WALL) {

            distanceAtCurrent = get_current_distance();
            SONARget_average_data(ring);
        }

    /* stop the robot */

    stop_robot_translation();

    distanceTravelled = 0;
    if (ring[15] <= SAFTY_RADIUS || ring[0] <= SAFTY_RADIUS ||
        ring[1] <= SAFTY_RADIUS) {
    }
    else if (ring[11] <= MAX_DIST_TO_WALL || ring[12] <= MAX_DIST_TO_WALL ||
             ring[13] <= MAX_DIST_TO_WALL) {


        /* make parallel to the right wall */
```

```
        SONARorient_parallel(LEFT);

        portPosition = Save_Point();

        /* set the robot 50 cm(safty radius) off from the wall */

        rotate_robot(-90, SLOW_TURN_SPEED);

        SONARget_average_data(ring);
        while (abs((int) ring[0] - SAFTY_RADIUS ) > SONAR_ERROR){
            translate_robot((int) ring[0] - SAFTY_RADIUS, VERY_SLOW_SPEED);
            SONARget_average_data(ring);
        }

        rotate_robot(90, SLOW_TURN_SPEED);

        distanceAtStart = distanceAtCurrent = get_current_distance();

        /* start to move forward */

        start_robot_translation(SLOW_SPEED);

        SONARget_average_data(ring);

        while (ring[15] > SAFTY_RADIUS && ring[0] > SAFTY_RADIUS &&
               ring[1] > SAFTY_RADIUS &&
               SONARwall_exist(LEFT) &&
               (distanceAtCurrent - distanceAtStart) <= WALL_LENGTH){

            distanceAtCurrent = get_current_distance();
            SONARget_average_data(ring);
        }
        distanceTravelled = distanceAtCurrent - distanceAtStart;

        /* stop the robot */

        stop_robot_translation();
    }

    /*go back to the entry position */

    if (portPosition != -1) Move_To_Point(portPosition, STRAIGHT);
    Move_To_Point(centerPosition, Y_FIRST);
    Move_To_Point(entryPosition,Y_FIRST);

    /* compute the probability */

    certaintyWall = CertaintyWall(distanceTravelled);

    result = certaintyWall + (0.5 - certaintyWall) *
                             (1.0 - certaintyPosition) *
                            WEIGHT_K;
    fprintf(stderr, " certaintyWall %f, certaintyPosition %f, result %f\n",
            certaintyWall, certaintyPosition, result);
    return(result);
}

/* DetectWallOne()
*
*    DetectWallOne detects the front-right wall of an LDP,
* and returns the certainty of the detection.
*
*    It assumes that the robot is placed at the entry position of an
* LDP.
```

```
*
*/


float DetectWallOne(entryPosition)
int entryPosition;
{
    unsigned short ring[16];
    int distanceTravelled, distanceAtStart, distanceAtCurrent;
    int centerPosition, portPosition;
    float result, certaintyWall, certaintyPosition;
    float CertaintyWall();
    WAIreplyMsg *Get_Point_Data();
    int openRight, openLeft;

    certaintyPosition = Get_Point_Data(typeFHMcertaintyToCurrent,
                                       entryPosition)->certainty;

    portPosition = -1;

    /* move to the center of a junction */

    translate_robot(SAFTY_RADIUS, SLOW_SPEED);

    /* set open direction */

    openRight = FALSE;
    openLeft = FALSE;
    SONARget_average_data(ring);
    if ((ring[3] > MAX_DIST_TO_WALL && ring[4] > MAX_DIST_TO_WALL) ||
        (ring[4] > MAX_DIST_TO_WALL && ring[5] > MAX_DIST_TO_WALL ))
        openRight = TRUE;
    if ((ring[11] > MAX_DIST_TO_WALL && ring[12] > MAX_DIST_TO_WALL) ||
        (ring[12] > MAX_DIST_TO_WALL && ring[13] > MAX_DIST_TO_WALL ))
        openLeft = TRUE;

fprintf(stderr, "openRight %d openLeft %d \n", openRight, openLeft);

    /* save CenterPosition */

    centerPosition = Save_Point();

    /* go forward until it meets a corridor, or it has moved more than     */
    /* the length of a corridor, or                                        */
    /* it meets an obstacle.                                               */

    distanceAtStart = distanceAtCurrent = get_current_distance();

    start_robot_translation(SLOW_SPEED);

    SONARget_average_data(ring);

    if (openRight == TRUE)
        while (ring[15] > SAFTY_RADIUS && ring[0] > SAFTY_RADIUS &&
               ring[1] > SAFTY_RADIUS &&
               (ring[3] > MAX_DIST_TO_WALL || ring[4] > MAX_DIST_TO_WALL ||
                ring[5] > MAX_DIST_TO_WALL) &&
               (distanceAtCurrent - distanceAtStart) <= MAX_DIST_TO_WALL) {

            distanceAtCurrent = get_current_distance();
            SONARget_average_data(ring);
        }
    else if (openLeft == TRUE)
        while (ring[15] > SAFTY_RADIUS && ring[0] > SAFTY_RADIUS &&
```

```
            ring[1] > SAFTY_RADIUS &&
            (ring[11] > MAX_DIST_TO_WALL || ring[12] > MAX_DIST_TO_WALL ||
             ring[13] > MAX_DIST_TO_WALL) &&
            (distanceAtCurrent - distanceAtStart) <= MAX_DIST_TO_WALL) {

        distanceAtCurrent = get_current_distance();
        SONARget_average_data(ring);
    }

/* stop the robot */

stop_robot_translation();


distanceTravelled = 0;
if (ring[15] <= SAFTY_RADIUS || ring[0] <= SAFTY_RADIUS ||
    ring[1] <= SAFTY_RADIUS) {
}
else if (ring[3] <= MAX_DIST_TO_WALL || ring[4] <= MAX_DIST_TO_WALL ||
         ring[5] <= MAX_DIST_TO_WALL) {


    /* make parallel to the right wall */
    SONARorient_parallel(RIGHT);

    portPosition = Save_Point();

    /* set the robot 50 cm(safty radius) off from the wall */

    rotate_robot(90, SLOW_TURN_SPEED);

    SONARget_average_data(ring);
    while (abs((int)ring[0] - SAFTY_RADIUS ) > SONAR_ERROR){
        translate_robot((int) ring[0] - SAFTY_RADIUS, VERY_SLOW_SPEED);
        SONARget_average_data(ring);
    }

    rotate_robot(-90, SLOW_TURN_SPEED);

    distanceAtStart = distanceAtCurrent = get_current_distance();

    /* start to move forward */

    start_robot_translation(SLOW_SPEED);

    SONARget_average_data(ring);

    while (ring[15] > SAFTY_RADIUS && ring[0] > SAFTY_RADIUS &&
           ring[1] > SAFTY_RADIUS &&
           SONARwall_exist(RIGHT) &&
           (distanceAtCurrent - distanceAtStart) <= WALL_LENGTH){

        distanceAtCurrent = get_current_distance();
        SONARget_average_data(ring);
    }
    distanceTravelled = distanceAtCurrent - distanceAtStart;

    /* stop the robot */

    stop_robot_translation();
}

/*go back to the entry position */
```

```
    if (portPosition != -1) Move_To_Point(portPosition, STRAIGHT);
    Move_To_Point(centerPosition, Y_FIRST);
    Move_To_Point(entryPosition, Y_FIRST);

    /* compute the probability */

    certaintyWall = CertaintyWall(distanceTravelled);

    result = certaintyWall + (0.5 - certaintyWall) *
                             (1.0 - certaintyPosition) *
                             WEIGHT_K;
    fprintf(stderr, " certaintyWall %f, certaintyPosition %f, result %f\n",
            certaintyWall, certaintyPosition, result);
    return(result);
}

/* DetectWallTwo()
*
*    DetectWallTwo detects the left-front wall of an LDP,
* and returns the certainty of the detection.
*
*    It assumes that the robot is placed at the entry position of an
* LDP.
*
*/

float DetectWallTwo(entryPosition)
int entryPosition;
{
    unsigned short ring[16];
    int distanceTravelled, distanceAtStart, distanceAtCurrent;
    int centerPosition, portPosition;
    float result, certaintyWall, certaintyPosition;
    float CertaintyWall();
    WAIreplyMsg *Get_Point_Data();

    portPosition = -1;

    certaintyPosition = Get_Point_Data(typeFHMcertaintyToCurrent,
                                       entryPosition)->certainty;

    /* move to the center of a junction */

    translate_robot(SAFTY_RADIUS, SLOW_SPEED);

    /* turn left 90 degree */

    rotate_robot(-90, SLOW_TURN_SPEED);

    /* save CenterPosition */

    centerPosition = Save_Point();

    /* go forward until it meets a corridor, or it has moved more than     */
    /* the length of a corridor, or                                        */
    /* it meets an obstacle.                                               */

    distanceAtStart = distanceAtCurrent = get_current_distance();

    start_robot_translation(SLOW_SPEED);

    SONARget_average_data(ring);

    while (ring[15] > SAFTY_RADIUS && ring[0] > SAFTY_RADIUS &&
```

```
        ring[1] > SAFTY_RADIUS &&
        (ring[11] > MAX_DIST_TO_WALL || ring[12] > MAX_DIST_TO_WALL ||
         ring[13] > MAX_DIST_TO_WALL) &&
        (distanceAtCurrent - distanceAtStart) <= MAX_DIST_TO_WALL) {

    distanceAtCurrent = get_current_distance();
    SONARget_average_data(ring);
}

/* stop the robot */

stop_robot_translation();



distanceTravelled = 0;
if (ring[15] <= SAFTY_RADIUS || ring[0] <= SAFTY_RADIUS ||
    ring[1] <= SAFTY_RADIUS) {
}
else if (distanceAtCurrent - distanceAtStart > MAX_DIST_TO_WALL) {
}
else if (ring[3] <= MAX_DIST_TO_WALL || ring[4] <= MAX_DIST_TO_WALL ||
         ring[5] <= MAX_DIST_TO_WALL) {


    /* make parallel to the right wall */
    SONARorient_parallel(RIGHT);

    portPosition = Save_Point();

    /* set the robot 50 cm(safty radius) off from the wall */

    rotate_robot(90, SLOW_TURN_SPEED);

    SONARget_average_data(ring);
    while (abs((int)ring[0] - SAFTY_RADIUS ) > SONAR_ERROR){
        translate_robot((int) ring[0] - SAFTY_RADIUS, VERY_SLOW_SPEED);
        SONARget_average_data(ring);
    }

    rotate_robot(-90, SLOW_TURN_SPEED);

    distanceAtStart = distanceAtCurrent = get_current_distance();

    /* start to move forward */

    start_robot_translation(SLOW_SPEED);

    SONARget_average_data(ring);

    while (ring[15] > SAFTY_RADIUS && ring[0] > SAFTY_RADIUS &&
           ring[1] > SAFTY_RADIUS &&
           SONARwall_exist(RIGHT) &&
           (distanceAtCurrent - distanceAtStart) <= WALL_LENGTH){

        distanceAtCurrent = get_current_distance();
        SONARget_average_data(ring);
    }
    distanceTravelled = distanceAtCurrent - distanceAtStart;

    /* stop the robot */

    stop_robot_translation();
}
```

```
    /*go back to the entry position */

    if (portPosition != -1) Move_To_Point(portPosition, STRAIGHT);
    Move_To_Point(centerPosition, Y_FIRST);
    Move_To_Point(entryPosition, Y_FIRST);

    /* compute the probability */

    certaintyWall = CertaintyWall(distanceTravelled);

    result = certaintyWall + (0.5 - certaintyWall) *
                             (1.0 - certaintyPosition) *
                             WEIGHT_K;
    fprintf(stderr, " certaintyWall %f, certaintyPosition %f, result %f\n",
            certaintyWall, certaintyPosition, result);
    return(result);
}

/* DetectWallThree()
*
*    DetectWallThree detects the right-front wall of an LDP,
* and returns the certainty of the detection.
*
*    It assumes that the robot is placed at the entry position of an
* LDP.
*
*/


float DetectWallThree(entryPosition)
int entryPosition;
{
    unsigned short ring[16];
    int distanceTravelled, distanceAtStart, distanceAtCurrent;
    int centerPosition, portPosition;
    float result, certaintyWall, certaintyPosition;
    float CertaintyWall();
    WAIreplyMsg *Get_Point_Data();

    portPosition = -1;

    certaintyPosition = Get_Point_Data(typeFHMcertaintyToCurrent,
                                       entryPosition)->certainty;

    /* move to the center of a junction */

    translate_robot(SAFTY_RADIUS, SLOW_SPEED);

    /* turn right 90 degree */

    rotate_robot(90, SLOW_TURN_SPEED);

    /* save CenterPosition */

    centerPosition = Save_Point();

    /* go forward until it meets a corridor, or it has moved more than     */
    /* the length of a corridor, or                                        */
    /* it meets an obstacle.                                               */

    distanceAtStart = distanceAtCurrent = get_current_distance();

    start_robot_translation(SLOW_SPEED);
```

```
SONARget_average_data(ring);

while (ring[15] > SAFTY_RADIUS && ring[0] > SAFTY_RADIUS &&
       ring[1] > SAFTY_RADIUS &&
       (ring[3] > MAX_DIST_TO_WALL || ring[4] > MAX_DIST_TO_WALL ||
        ring[5] > MAX_DIST_TO_WALL) &&
       (distanceAtCurrent - distanceAtStart) <= MAX_DIST_TO_WALL) {

    distanceAtCurrent = get_current_distance();
    SONARget_average_data(ring);
}

/* stop the robot */

stop_robot_translation();


distanceTravelled = 0;
if (ring[15] <= SAFTY_RADIUS || ring[0] <= SAFTY_RADIUS ||
    ring[1] <= SAFTY_RADIUS) {
}
else if (distanceAtCurrent - distanceAtStart > MAX_DIST_TO_WALL) {
}
else if (ring[11] <= MAX_DIST_TO_WALL || ring[12] <= MAX_DIST_TO_WALL ||
         ring[13] <= MAX_DIST_TO_WALL) {


    /* make parallel to the right wall */
    SONARorient_parallel(LEFT);

    portPosition = Save_Point();

    /* set the robot 50 cm(safty radius) off from the wall */

    rotate_robot(-90, SLOW_TURN_SPEED);

    SONARget_average_data(ring);
    while (abs((int)ring[0] - SAFTY_RADIUS ) > SONAR_ERROR){
        translate_robot((int) ring[0] - SAFTY_RADIUS, VERY_SLOW_SPEED);
        SONARget_average_data(ring);
    }

    rotate_robot(90, SLOW_TURN_SPEED);

    distanceAtStart = distanceAtCurrent = get_current_distance();

    /* start to move forward */

    start_robot_translation(SLOW_SPEED);

    SONARget_average_data(ring);

    while (ring[15] > SAFTY_RADIUS && ring[0] > SAFTY_RADIUS &&
           ring[1] > SAFTY_RADIUS &&
           SONARwall_exist(LEFT) &&
           (distanceAtCurrent - distanceAtStart) <= WALL_LENGTH){

        distanceAtCurrent = get_current_distance();
        SONARget_average_data(ring);
    }
    distanceTravelled = distanceAtCurrent - distanceAtStart;

    /* stop the robot */
```

```
        stop_robot_translation();
    }

    /*go back to the entry position */

    if (portPosition != -1) Move_To_Point(portPosition, STRAIGHT);
    Move_To_Point(centerPosition, Y_FIRST);
    Move_To_Point(entryPosition, Y_FIRST);

    /* compute the probability */

    certaintyWall = CertaintyWall(distanceTravelled);

    result = certaintyWall + (0.5 - certaintyWall) *
                             (1.0 - certaintyPosition) *
                            WEIGHT_K;
    fprintf(stderr, " certaintyWall %f, certaintyPosition %f, result %f\n",
            certaintyWall, certaintyPosition, result);
    return(result);
}

/* DetectWallFour()
*
*    DetectWallFour detects the left-back wall of an LDP,
* and returns the certainty of the detection.
*
*    It assumes that the robot is placed at the entry position of an
* LDP.
*
*/

float DetectWallFour(entryPosition)
int entryPosition;
{
    unsigned short ring[16];
    int distanceTravelled, distanceAtStart, distanceAtCurrent;
    int centerPosition, portPosition;
    float result, certaintyWall, certaintyPosition;
    float CertaintyWall();
    WAIreplyMsg *Get_Point_Data();

    portPosition = -1;

    certaintyPosition = Get_Point_Data(typeFHMcertaintyToCurrent,
                                       entryPosition)->certainty;

    /* move to the center of a junction */

    translate_robot(SAFTY_RADIUS, SLOW_SPEED);

    /* turn left 90 degree */

    rotate_robot(-90, SLOW_TURN_SPEED);

    /* save CenterPosition */

    centerPosition = Save_Point();

    /* go forward until it meets a corridor, or it has moved more than     */
    /* the length of a corridor, or                                        */
    /* it meets an obstacle.                                               */

    distanceAtStart = distanceAtCurrent = get_current_distance();
```

```
start_robot_translation(SLOW_SPEED);

SONARget_average_data(ring);

while (ring[15] > SAFTY_RADIUS && ring[0] > SAFTY_RADIUS &&
       ring[1] > SAFTY_RADIUS &&
       (ring[11] > MAX_DIST_TO_WALL || ring[12] > MAX_DIST_TO_WALL ||
        ring[13] > MAX_DIST_TO_WALL) &&
       (distanceAtCurrent - distanceAtStart) <= MAX_DIST_TO_WALL) {

    distanceAtCurrent = get_current_distance();
    SONARget_average_data(ring);
}

/* stop the robot */

stop_robot_translation();


distanceTravelled = 0;
if (ring[15] <= SAFTY_RADIUS || ring[0] <= SAFTY_RADIUS ||
    ring[1] <= SAFTY_RADIUS) {
}
else if (distanceAtCurrent - distanceAtStart > MAX_DIST_TO_WALL) {
}
else if (ring[11] <= MAX_DIST_TO_WALL || ring[12] <= MAX_DIST_TO_WALL ||
         ring[13] <= MAX_DIST_TO_WALL) {


    /* make parallel to the right wall */
    SONARorient_parallel(LEFT);

    portPosition = Save_Point();

    /* set the robot 50 cm(safty radius) off from the wall */

    rotate_robot(-90, SLOW_TURN_SPEED);

    SONARget_average_data(ring);
    while (abs((int)ring[0] - SAFTY_RADIUS ) > SONAR_ERROR){
        translate_robot((int) ring[0] - SAFTY_RADIUS, VERY_SLOW_SPEED);
        SONARget_average_data(ring);
    }

    rotate_robot(90, SLOW_TURN_SPEED);


    distanceAtStart = distanceAtCurrent = get_current_distance();

    /* start to move forward */

    start_robot_translation(SLOW_SPEED);

    SONARget_average_data(ring);

    while (ring[15] > SAFTY_RADIUS && ring[0] > SAFTY_RADIUS &&
           ring[1] > SAFTY_RADIUS &&
           SONARwall_exist(LEFT) &&
           (distanceAtCurrent - distanceAtStart) <= WALL_LENGTH){

        distanceAtCurrent = get_current_distance();
        SONARget_average_data(ring);
    }
```

```
        distanceTravelled = distanceAtCurrent - distanceAtStart;

        /* stop the robot */

        stop_robot_translation();
    }

    /*go back to the entry position */

    if (portPosition != -1) Move_To_Point(portPosition, STRAIGHT);
    Move_To_Point(centerPosition, Y_FIRST);
    Move_To_Point(entryPosition, Y_FIRST);

    /* compute the probability */

    certaintyWall = CertaintyWall(distanceTravelled);

    result = certaintyWall + (0.5 - certaintyWall) *
                             (1.0 - certaintyPosition) *
                             WEIGHT_K;
    fprintf(stderr, " certaintyWall %f, certaintyPosition %f, result %f\n",
            certaintyWall, certaintyPosition, result);
    return(result);
}

/* DetectWallFive()
*
*    DetectWallFive detects the right-back wall of an LDP,
* and returns the certainty of the detection.
*
*    It assumes that the robot is placed at the entry position of an
* LDP.
*
*/

float DetectWallFive(entryPosition)
int entryPosition;
{
    unsigned short ring[16];
    int distanceTravelled, distanceAtStart, distanceAtCurrent;
    int centerPosition, portPosition;
    float result, certaintyWall, certaintyPosition;
    float CertaintyWall();
    WAIreplyMsg *Get_Point_Data();

    portPosition = -1;

    certaintyPosition = Get_Point_Data(typeFHMcertaintyToCurrent,
                                       entryPosition)->certainty;

    /* move to the center of a junction */

    translate_robot(SAFTY_RADIUS, SLOW_SPEED);

    /* turn right 90 degree */

    rotate_robot(90, SLOW_TURN_SPEED);

    /* save CenterPosition */

    centerPosition = Save_Point();

    /* go forward until it meets a corridor, or it has moved more than     */
    /* the length of a corridor, or                                        */
```

```
/* it meets an obstacle.                                              */

distanceAtStart = distanceAtCurrent = get_current_distance();

start_robot_translation(SLOW_SPEED);

SONARget_average_data(ring);

while (ring[15] > SAFTY_RADIUS && ring[0] > SAFTY_RADIUS &&
       ring[1] > SAFTY_RADIUS &&
       (ring[3] > MAX_DIST_TO_WALL || ring[4] > MAX_DIST_TO_WALL ||
        ring[5] > MAX_DIST_TO_WALL) &&
       (distanceAtCurrent - distanceAtStart) <= MAX_DIST_TO_WALL) {

    distanceAtCurrent = get_current_distance();
    SONARget_average_data(ring);
}

/* stop the robot */

stop_robot_translation();


distanceTravelled = 0;
if (ring[15] <= SAFTY_RADIUS || ring[0] <= SAFTY_RADIUS ||
    ring[1] <= SAFTY_RADIUS) {
}
else if (distanceAtCurrent - distanceAtStart > MAX_DIST_TO_WALL) {
}
else if (ring[3] <= MAX_DIST_TO_WALL || ring[4] <= MAX_DIST_TO_WALL ||
         ring[5] <= MAX_DIST_TO_WALL) {


    /* make parallel to the right wall */
    SONARorient_parallel(RIGHT);

    portPosition = Save_Point();

    /* set the robot 50 cm(safty radius) off from the wall */

    rotate_robot(90, SLOW_TURN_SPEED);

    SONARget_average_data(ring);
    while (abs((int)ring[0] - SAFTY_RADIUS ) > SONAR_ERROR){
        translate_robot((int) ring[0] - SAFTY_RADIUS, VERY_SLOW_SPEED);
        SONARget_average_data(ring);
    }

    rotate_robot(-90, SLOW_TURN_SPEED);

    distanceAtStart = distanceAtCurrent = get_current_distance();

    /* start to move forward */

    start_robot_translation(SLOW_SPEED);

    SONARget_average_data(ring);

    while (ring[15] > SAFTY_RADIUS && ring[0] > SAFTY_RADIUS &&
           ring[1] > SAFTY_RADIUS &&
           SONARwall_exist(RIGHT) &&
           (distanceAtCurrent - distanceAtStart) <= WALL_LENGTH){

        distanceAtCurrent = get_current_distance();
```

```
            SONARget_average_data(ring);
        }
        distanceTravelled = distanceAtCurrent - distanceAtStart;

        /* stop the robot */

        stop_robot_translation();
    }

    /*go back to the entry position */

    if (portPosition != -1) Move_To_Point(portPosition, STRAIGHT);
    Move_To_Point(centerPosition, Y_FIRST);
    Move_To_Point(entryPosition, Y_FIRST);

    /* compute the probability */

    certaintyWall = CertaintyWall(distanceTravelled);

    result = certaintyWall + (0.5 - certaintyWall) *
                             (1.0 - certaintyPosition) * WEIGHT_K;
    fprintf(stderr, " certaintyWall %f, certaintyPosition %f, result %f\n",
            certaintyWall, certaintyPosition, result);
    return(result);
}

/* CertaintyWall()
*
*       This computes the certainty that a wall exists given the
*  distance traversed.
*
*/

float CertaintyWall(wallLength)
int wallLength;
{
        if (wallLength > WALL_LENGTH)
            return(1.0);
        else
            return ((float) wallLength / WALL_LENGTH);
}
```

```
/* JCMinterface.h
*
*  header file for the interface with Junction Recognizer Module
*
*
*/

#define typeJCMrecognizeJunction   10

/* message type for a request and a reply  */

typedef struct {
    int type;
    int entryPosition;
    int result;
} JCMmsg;

/* junction type constants  */

#define CROSS_JUNCTION                  0
#define T_JUNCTION                      1
#define T_RIGHT_JUNCTION                2
#define T_LEFT_JUNCTION                 3
#define L_RIGHT_JUNCTION                4
#define L_LEFT_JUNCTION                 5
#define OPNE_SPACE                      6
#define OPEN_SPACE_RIGHT                7
#define OPNE_SPACE_LEFT                 8
```

```
#include <stdio.h>
#include "/pro/ai/robot/software/ipc/src/ipc.h"
#include "/pro/ai/robot/software/misc/src/error.h"
#include "JCMinterface.h"

/* JCM.c
*
*  main program of Junction Recognizer Module
*
*       author : Seungseok Hyun
*       date   : Jan. 24, 1990
*/

/* main
*
*       main gets requests by the ipc message passing mechanism,
*       calls procedures in jcm.c according to the requests, and
*       replies.
*
*/

clientId myOwnId, WAI_id, FDM_id, CFM_id;

main()
{

    clientId senderId;
    JCMmsg message;
    int requestLength;

    myOwnId = NSregisterSelf("JCM", 10);

    while((WAI_id = NSgetClient("WAI")) == nullClient)
          sleep(1);

    while((FDM_id = NSgetClient("FDM")) == nullClient)
          sleep(1);

    while((CFM_id = NSgetClient("CFM")) == nullClient)
          sleep(1);

    /* This program never halts. */

    while ( 1 ) {

        requestLength = sizeof(JCMmsg);

        do
            senderId = IPCrecvMessage(&message, &requestLength, 5);
        while (senderId == nullClient);

        switch(message.type) {
            case typeJCMrecognizeJunction:
                message.result = RecognizeJunction(message.entryPosition);
                IPCsendMessage(&message, sizeof(JCMmsg), senderId);
                break;
            default:
                fprintf(stderr, "JCM unknow type request \n");
                break;
        }
    }
}
```

```
/* jcm.h
*
*       header file for jcm.c
*
*/

#define THRESHOLD_HYPOTHESIS    0.75    /* if any hypothesis exceeds this */
                                        /* stop detecting features        */

#define FALSE                   0
#define TRUE                    1


#define K1                      1.0
#define K2                      1.0
#define K3                      0.1
#define K4                      2.0
#define K5                      0.1

#define COST_DETECTOR           60
#define COST_REACQUIRE          40
```

```
#include <stdio.h>
#include "jcm.h"
#include "JCMinterface.h"
#include "/pro/ai/robot/software/huey/fdm/src/FDMinterface.h"
#include "/pro/ai/robot/software/huey/fdm/src/fdm.h"
#include "/pro/ai/robot/software/ipc/src/ipc.h"
#include "/pro/ai/robot/software/misc/src/error.h"
#include "/pro/ai/robot/software/huey/wai/src/WAIinterface.h"
#include "/pro/ai/robot/software/huey/cfm/src/CFMinterface.h"

/* jcm.c
*
*       functions for Junction Classifying Module
*
*/

/*
* external variables
*/

extern clientId myOwnId, WAI_id, FDM_id, CFM_id;

/* global variables for probability distributions
*
* The distributions we try to represent correspond to the following
* influence diagram.
*                                  (H)
*                                   |
*                        ---------------------
*                        |          |         |
*                        V          V         V
*                       (F0)      (F2)  ...  (F5)
*                        |          |         |
*                        V          V         V
*                       (D0)      (D2)  ...  (D5)
*
*  H  : Hypothesis Node (it has 9 states.)
*  Fn : Feature (wall) node(it has 2 states.)
*  Dn : Detector Node (it has 2 states.)
*
*/

float H[9];              /* Pr(H) */
float F[6][2][9];        /* Pr(F|H) */
float D[6][2][2];        /* Pr(D|H) , not used actually */

/* other global variables */

int   DetectorUsed[6];

/* RecognizeJunction()
*
*       input  : entry position (int)
*       output : junction type
*
*     RecognizeJunction() sends requests to FDM module, and decide
*  the junction type.
*/


int RecognizeJunction(entryPosition)
int entryPosition;
{
    int i;
```

```
int bestDetector;
float utilityBestDetector, utilityReacquiring;
float probabilityWall;

void UpdateHypothesis();
float DetectWall();
void ReacquireEntryPosition();
float UtilityReacquiring();
int ChooseBestDetector();
int NumberDetectorAvailable();
int BestHypothesis();
void InitializeDistributions();
float positionCertainty;
WAIrequestMsg waiRequestMsg;
WAIreplyMsg waiReplyMsg;
int msgLength;


/* initialize the a priori distributions */

InitializeDistributions();

/* other initialization */

for (i = 0; i <= 5; i++)
    DetectorUsed[i] = FALSE;

/* choose the next action until it runs out detectors, or */
/* a hypothesis exceeds THRESHOLD_HYPOTHESIS              */

while ( NumberDetectorAvailable() != 0 &&
    H[BestHypothesis()] < THRESHOLD_HYPOTHESIS){

    /* get the positional certainty from WAI */

    waiRequestMsg.type = typeFHMcertaintyToCurrent;
    waiRequestMsg.position1 = entryPosition;
    IPCsendMessage(&waiRequestMsg, sizeof(WAIrequestMsg), WAI_id);
    msgLength = sizeof(WAIreplyMsg);
    while (nullClient == IPCrecvMessage(&waiReplyMsg, &msgLength, 5));
    positionCertainty = waiReplyMsg.certainty;
    fprintf(stderr, " Current positional certainty is %f\n",
            positionCertainty);

    /* choose the best detector */

    bestDetector = ChooseBestDetector(&utilityBestDetector,
                                      positionCertainty);

    /* set this detector status 'used' */

    DetectorUsed[bestDetector] = TRUE;

    /* get the utility of reacquiring */

    utilityReacquiring = UtilityReacquiring(bestDetector,
                                            positionCertainty);

    /* if the utility of reacquiring is better, do it */
    /* before the detector */

    if (utilityReacquiring > utilityBestDetector){

        ReacquireEntryPosition();
```

```
            /* save new entry position */

            waiRequestMsg.type = typeFHMsavePosition;
            IPCsendMessage(&waiRequestMsg, sizeof(WAIrequestMsg), WAI_id);
            msgLength = sizeof(WAIreplyMsg);
            while (nullClient == IPCrecvMessage(&waiReplyMsg, &msgLength, 5));
            entryPosition = waiReplyMsg.position;
            fprintf(stderr," New entry position %d\n", entryPosition);
        }

        /* run the detector */

        probabilityWall = DetectWall(bestDetector, entryPosition);

        /* update the hypothesis */

        UpdateHypothesis(bestDetector, probabilityWall);
    }
    return(BestHypothesis());
}

/* NumberDetectorAvailable()
*
*       input : none (it uses global variables)
*       output: number of unused detectors
*
*   This function returns the number of detectors not yet invoked.
*
*/

int NumberDetectorAvailable()
{
    int result, i;

    result = 0;
    for (i = 0; i <= 5; i++)
        if (DetectorUsed[i] == FALSE) result++;
    fprintf(stderr," Number detector available %d\n", result);
    return(result);
}

/* BestHypothesis()
*
*       input :none (it uses global variables)
*       output:the best hypothesis among H[0] to H[8]
*
*    This function returns the hypothesis which has the greatest
*  probability.
*
*/

int BestHypothesis()
{
    int i;
    int result;

    result = 0;
    for (i=1; i <= 8; i++)
        if (H[i] > H[result]){
            result = i;
        }

    fprintf(stderr," Best Hypothesis is %d\n", result);
```

```
    return(result);
}

/* ChooseBestDetector()
*
*       input : positionCertainty
*       output: the best detector number,
*               the utility of the best one(by pointer parameter)
*
*     This function chooses the best detector to invoke next.
*
*/

int ChooseBestDetector(utility, positionCertainty)
float *utility;
float  positionCertainty;
{
    int i;
    int result;
    float temp;

    float Discrim();

    /* compute utility, select the best one */

    *utility = 0.0;           /* smallest utility */

    fprintf(stderr," Choosing Best Detector...\n");
    for (i = 0; i <= 5 ; i++)
        if (DetectorUsed[i] == FALSE){
            temp = K2 * Discrim(i, positionCertainty) - K3 * COST_DETECTOR;
            fprintf(stderr, " Utility of detector %d is %f\n", i, temp);
            if (temp > *utility){
                result = i;
                *utility = temp;
            }
        }
    fprintf(stderr," ... best one is %d\n", result);
    return(result);
}

/* Discrim()
*
*       input : detector number, positionCertainty
*       output: discrimination value of the detector
*
*    This function computes the discrimination value of a detector.
*
*/

float Discrim(detector, positionCertainty)
int detector;
float positionCertainty;
{
    int r, h;
    float result;
    float temp, temp2;

    float PrPosteriory(), PrF(), PrFeature();

    result = 0.0;
    for (r = 0; r <= WALL_LENGTH; r = r + WALL_LENGTH){
        temp = 0.0;
        for (h = 0; h <= 8; h++){
```

```
            temp2 =  PrPosteriory(h,detector,PrFeature(r, positionCertainty))
                    - H[h];
            if (temp2 < 0.0) temp2 = -temp2;

            temp = temp + temp2;
        }
        result = result + PrF(detector, r/WALL_LENGTH) * temp;
    }

    return(result);
}

/* PrF()
*
*       input : feature, state
*       output: Pr(F_feature = state)
*
*    This function computes the Pr(F_{feature} = state)
*
*/

float PrF(feature, state)
int feature;
int state;
{
    float result;
    int h;

    result = 0.0;

    for (h = 0; h <= 8; h++)
        result = result + F[feature][state][h] * H[h];

    return(result);
}

/* PrPosteriory()
*
*       input : hypothesis number, detector number, certainty of the detector
*       output: posteriory Pr(H|D)
*
*    This function computes the posterior distribution of P'(H=h),
*  given the distanced traversed and the certainty of the start position.
*
*/

float PrPosteriory(h, d, c)
int h;      /* hypothesis */
int d;      /* detector   */
float c;    /* certainty of the detector */
{
    float result;
    int i;
    float temp1, temp2;

    temp1 = H[h] * ( (1.0 - c) * F[d][0][h] + c * F[d][1][h] );

    temp2 = 0.0;
    for (i = 0; i <= 8; i++)
        temp2 = temp2 + H[i] * ((1.0 - c) * F[d][0][i] + c * F[d][1][i]);

    if (temp2 == 0.0){
        result = temp1;
```

```
    }
    else result = temp1 / temp2;

    return(result);
}

/* PrFeature()
*
*       input : distance, positional certainty
*       output: the certainty of a detecting a wall
*
*    This function computes the probability that the feature exists given
*  distanced traversed and the positional certainty of the start position.
*
*/

float PrFeature(d, p)
int d;    /* distance */
float p;  /* positional certainty */
{
      float wallCertainty;
      float result;

      if (d >= WALL_LENGTH)
          wallCertainty = 1.0;
      else
          wallCertainty = (float) d / (float) WALL_LENGTH;

      result = wallCertainty + (0.5 - wallCertainty) * (1.0 - p) * K1;

      return(result);
}


/* UtilityReacquiring()
*
*       input : best detector, positional certainty
*       output: utility of a reacquiring
*
*       This function computes the utility of the reacquire action.
*
*/

float UtilityReacquiring(d, p)
int d;     /* detector */
float p;   /* positional certainty */
{
    float result;
    float Discrim();

    result = K4 * (Discrim(d, 1.0) - Discrim(d, p)) -
             K5 * COST_REACQUIRE;

    fprintf(stderr, " Utility of Reacquiring is %f given certainty %f\n",
              result, p);

    return(result);
}

/* DetectWall()
*
*       input : detector number, entryPosition name
*       output: the certainty of the detecting the wall
*
```

```
*       This function calls the detector by the message-passing mechanism.
*
*/

float DetectWall(detector, entryPosition)
int detector, entryPosition;
{
   FDMmsg requestMsg, replyMsg;
   int    lengthMsg;
   float result;

   switch (detector) {
       case 0:
           requestMsg.type = typeFDMdetectWallZero;
           break;
       case 1:
           requestMsg.type = typeFDMdetectWallOne;
           break;
       case 2:
           requestMsg.type = typeFDMdetectWallTwo;
           break;
       case 3:
           requestMsg.type = typeFDMdetectWallThree;
           break;
       case 4:
           requestMsg.type = typeFDMdetectWallFour;
           break;
       case 5:
           requestMsg.type = typeFDMdetectWallFive;
           break;
        default:
           fprintf(stderr,"*** wrong detector number\n");
    }

    fprintf(stderr, " detect wall %d , entryposition is %d\n", detector,
                                    entryPosition);

    requestMsg.entryPosition = entryPosition;

    IPCsendMessage(&requestMsg, sizeof(requestMsg), FDM_id);

    lengthMsg = sizeof(FDMmsg);
    while (nullClient == IPCrecvMessage(&replyMsg,&lengthMsg, 100));

    result = replyMsg.certainty;

    return(result);
}

/* UpdateHypothesis()
*
*       input : detector number, certainty of the detecting the wall
*       output: void
*
*    This function updates the hypothesis about hte equivalent class of
* a target junction, after a feature detection.
*
*/

void
UpdateHypothesis(detector, certainty)
int detector;
float certainty;
{
```

```
    int h;

    fprintf(stderr, " Hypothesis Updating... \n");
    for (h = 0; h <= 8; h++){
        H[h] = PrPosteriory(h, detector,  certainty);
        fprintf(stderr," Hypothesis %d is %f\n", h, H[h]);
    }
    fprintf(stderr,"\n");
}

/* ReacquiringEntryPosition()
*
*       input : none
*       output: void
*
*    This function invokes the reacquire action(in CFM module) by
*  the message-passing mechanism.
*
*/

void
ReacquireEntryPosition()
{
    CFMmsg requestMsg, replyMsg;
    int msgLength;

    requestMsg.type = typeCFMreacquire;
    IPCsendMessage(&requestMsg, sizeof(CFMmsg), CFM_id);
    msgLength = sizeof(CFMmsg);
    while (nullClient == IPCrecvMessage(&replyMsg, &msgLength, 200));

    if (replyMsg.returnMsg == CFM_OPEN)
          fprintf(stderr," Reacquiring successful\n");
    else
          fprintf(stderr," Reacquiring failed %d\n", replyMsg.returnMsg);
}
```

```
#include <stdio.h>
#include "jcm.h"
#include "JCMinterface.h"

/* jcm_init.c
*
*       functions for initialize Junction Recognizer Module
*
*/

/* global variables for probability distributions
*
* The distributions we try to represent correspond to the following
* influence diagram.
*                                  (H)
*                                   |
*                        ---------------------
*                        |          |         |
*                        V          V         V
*                       (F0)       (F2) ...  (F5)
*                        |          |         |
*                        V          V         V
*                       (D0)       (D2) ...  (D5)
*
*  H  : Hypothesis Node (it has 9 states.)
*  Fn : Feature (wall) node(it has 2 states.)
*  Dn : Detector Node (it has 2 states.)
*/

extern float H[9];                /* Pr(H) */
extern float F[6][2][9];          /* Pr(F|H) */
extern float D[6][2][2];          /* Pr(D|H) */

/* other global variables */

extern int   DetectorUsed[6];

/* InitializeDistributions()
*
*  This functions initialize the a priori distributions.
*
*/

void
InitializeDistributions()
{
    int i,j,k;

    /* even distributions for H */

    for (i = 0; i <= 8; i++)
        H[i] = 1.0 / 9.0;

    /* initialize Pr(F|H)       */

    for (i = 0; i <= 5; i++)
        for (j = 0; j <= 1; j++)
            for (k = 0; k <= 8; k++)
                F[i][j][k] = 0.0;

    /* Pr(F0|H) */
    F[0][1][0] = 1.0;
    F[0][0][1] = 1.0;
    F[0][1][2] = 1.0;
```

```
    F[0][1][3] = 1.0;
    F[0][0][4] = 1.0;
    F[0][0][5] = 1.0;
    F[0][0][6] = 1.0;
    F[0][1][7] = 1.0;
    F[0][0][8] = 1.0;

    /* Pr(F1|H) */
    F[1][1][0] = 1.0;
    F[1][0][1] = 1.0;
    F[1][1][2] = 1.0;
    F[1][1][3] = 1.0;
    F[1][0][4] = 1.0;
    F[1][0][5] = 1.0;
    F[1][0][6] = 1.0;
    F[1][0][7] = 1.0;
    F[1][1][8] = 1.0;

    /* Pr(F2|H) */
    F[2][1][0] = 1.0;
    F[2][1][1] = 1.0;
    F[2][0][2] = 1.0;
    F[2][1][3] = 1.0;
    F[2][0][4] = 1.0;
    F[2][1][5] = 1.0;
    F[2][0][6] = 1.0;
    F[2][0][7] = 1.0;
    F[2][0][8] = 1.0;

    /* Pr(F3|H)  */
    F[3][1][0] = 1.0;
    F[3][1][1] = 1.0;
    F[3][1][2] = 1.0;
    F[3][0][3] = 1.0;
    F[3][1][4] = 1.0;
    F[3][0][5] = 1.0;
    F[3][0][6] = 1.0;
    F[3][0][7] = 1.0;
    F[3][0][8] = 1.0;

    /* Pr(F4|H)  */
    F[4][1][0] = 1.0;
    F[4][1][1] = 1.0;
    F[4][0][2] = 1.0;
    F[4][1][3] = 1.0;
    F[4][0][4] = 1.0;
    F[4][1][5] = 1.0;
    F[4][1][6] = 1.0;
    F[4][0][7] = 1.0;
    F[4][1][8] = 1.0;

    /* Pr(F5|H)  */
    F[5][1][0] = 1.0;
    F[5][1][1] = 1.0;
    F[5][1][2] = 1.0;
    F[5][0][3] = 1.0;
    F[5][1][4] = 1.0;
    F[5][0][5] = 1.0;
    F[5][1][6] = 1.0;
    F[5][1][7] = 1.0;
    F[5][0][8] = 1.0;

}
```

```
/*
* WAIinterface.h
*
*       header file for interfacing with WHEREAMI module
*
*
*/

/* type for a request to WHEREAMI module */

typedef struct {
    int type;
    int distance;
    int angle;
    int x;
    int y;
    int position1;
    int position2;
    int position3;
} WAIrequestMsg;

/* type for a reply from WHEREAMI module */

typedef struct {
    int type;
    int position;
    int x;
    int stdX;
    int y;
    int stdY;
    int angle;
    int stdAngle;
    int distance;
    int stdDistance;
    int errorMsg;
    float certainty;
} WAIreplyMsg;

/*
*    type numbers for the "type" field
*
*/

#define typeFHMsavePosition             1
#define typeFHMsavePositionDelta        2
#define typeFHMposition                 3
#define typeFHMpositionFromCurrent      4
#define typeFHMorientation              5
#define typeFHMorientationFromCurrent   6
#define typeFHMclearMemory              7
#define typeMCMdistance                 8
#define typeMCMangle                    9
#define typeFHMcertaintyToCurrent       10

/*
*   error messages
*
*/

#define SORRY_RETRY_LATER               101
#define OK                              102

/*
            A VERY SIMPLE SPEC OF THE INTERFACE TO WHEREAMI module
```

Ingoing and Outgoing messages

```
Each following item in Ingoin(Outgoing) entry tells relevant field
names in a WAIrequestMsg(WAIreplyMsg) type ingoing(outgoing)
message. Relevant ingoing message fields should be assigned values
properly before a client send a request message.

FHMsavePosition
    Ingoing : type
    Outgoing: type, position, errorMsg

    in case the robot is moving: SORRY_RETRY_LATER for errorMsg

FHMsavePositionDelta
    Ingoing : type, x,  y, angle
    Outgoing: type, position, errorMsg

    in case the robot is moving: SORRY_RETRY_LATER for errorMsg

FHMposition
    Ingoing : type, position1, position2
    Outgoing: type, x, stdX, y, stdY, errorMsg

FHMpositionFromCurrent
    Ingoing : type, position1
    Outgoing: type, x, stdX, y, stdY, errorMsg

    in case the robot is moving: SORRY_RETRY_LATER for errorMsg

FHMorientation
    Ingoing : type, position1, position2
    Outgoing: type, angle, stdAngle, errorMsg

FHMorientationFromCurrent
    Ingoing : type, position1
    Outgoing: type, angle, stdAngle, errorMsg

    in case the robot is moving: SORRY_RETRY_LATER for errorMsg

FHMcertaintyToCurrent
    Ingoing: type, position1
    Outgogin: type, certainty, errorMsg

    It returns the certainty of the current position w.r.t. position1

FHMclearMemory
    Ingoing : type
    Outgoing: type, errorMsg

MCMdistance
    Ingoing : type, position1, position2
    Outgoing: type, distance, stdDistance, errorMsg

MCMangle
    Ingoing : type, position1, position2, position3
    Outgoing: type, angle, stdAngle, errorMsg

*/
```

```
#include <stdio.h>
#include "/pro/ai/robot/software/ipc/src/ipc.h"
#include "/u/mlm/projects/robot/lowLevelControl/src/llc.h"
#include "WAIinterface.h"
#include "whereami.h"
#include <math.h>


/* WAI.c
*
*  main program of WHEREAMI module
*
*       author : Seungseok Hyun
*       date   : Aug. 21, 1989
*/

/* main
*
*       main gets requests by the ipc message passing mechanism,
*       calls procedures in whereami.c according to the requests, and
*       replies.
*
*/

main()
{

    clientId senderId, myOwnId, LLCId;
    WAIrequestMsg *request;
    WAIreplyMsg   reply;
    PositionName pos1;
    TwoPositions pos2;
    ThreePositions pos3;
    Coordinate   coor;
    Orientation  ori;
    Distance dis;
    Angle ang;
    Delta delta;
    int requestLength;
    LLCmessage *receiveMsgLLC, sendMsgLLC;
    int movingOn;
    long startTranslate, startRotate;
    long temp;
    char *buff;
    void FHMclearMemory();

    PositionName FHMsavePosition(), FHMsavePositionDelta();
    Coordinate   FHMposition();
    Coordinate   FHMpositionFromCurrent();
    Orientation  FHMorientation();
    Orientation  FHMorientationFromCurrent();
    Distance MCMdistance();
    Angle MCMangle();
    void DRMdeadReckoning(), DRMinitialize();
    float FHMcertaintyToCurrent();

    myOwnId = NSregisterSelf("WAI", 10);

    /* send requests for notifying Translate(Rotate)Start(End) */
    LLCId = NSgetClient(CONTROLLER_NAME);
    sendMsgLLC.type = LLCmessageNotify;
    sendMsgLLC.data = LLCtranslateStart;
    IPCsendMessage(&sendMsgLLC, sizeof(LLCmessage), LLCId);
    sendMsgLLC.data = LLCrotateStart;
```

```
        IPCsendMessage(&sendMsgLLC, sizeof(LLCmessage), LLCId);
        sendMsgLLC.data = LLCtranslateEnd;
        IPCsendMessage(&sendMsgLLC, sizeof(LLCmessage), LLCId);
        sendMsgLLC.data = LLCrotateEnd;
        IPCsendMessage(&sendMsgLLC, sizeof(LLCmessage), LLCId);

        /* Dead Reckoning Module initialization */
        DRMinitialize();
        movingOn = FALSE;
        buff = (char *) malloc(100);

        /* This program never halts. */

        while ( 1 ) {

            requestLength = sizeof(WAIrequestMsg);

            while((senderId = IPCrecvMessage(buff, &requestLength, 5))
                  == nullClient)
              ;

    if (senderId != nullClient){
            fprintf(stderr,"WAI got a message\n");
            fprintf(stderr,"The length of msg is %d\n", requestLength);
    }

            if (requestLength != sizeof(WAIrequestMsg)) {
                receiveMsgLLC =  (LLCmessage *) buff;

                switch(receiveMsgLLC->type) {
                    case LLCtranslateStart:
                        movingOn = TRUE;
                        startTranslate = receiveMsgLLC->data;
                        break;
                    case LLCtranslateEnd:
                        movingOn = FALSE;
                        temp = receiveMsgLLC->data - startTranslate;
                        DRMdeadReckoning((int)  temp, 0);
                        break;
                    case LLCrotateStart:
                        movingOn = TRUE;
                        startRotate = receiveMsgLLC->data;
                        break;
                    case LLCrotateEnd:
                        movingOn = FALSE;
                        temp = receiveMsgLLC->data - startRotate;
                        DRMdeadReckoning(0, (int)  temp );
                        break;
                }
            }
            else {
                request =  (WAIrequestMsg *) buff;

                switch(request->type) {
                    case typeFHMsavePosition:
                        if (movingOn == TRUE) {
                            reply.errorMsg = SORRY_RETRY_LATER;
                            reply.type = typeFHMsavePosition;
                            IPCsendMessage(&reply, sizeof(WAIreplyMsg), senderId);
                        }
                        else {
                            reply.position = FHMsavePosition();
                            reply.type = typeFHMsavePosition;
                            reply.errorMsg = OK;
```

```
            IPCsendMessage(&reply, sizeof(WAIreplyMsg), senderId);
        }
        break;

    case typeFHMsavePositionDelta:
        if (movingOn == TRUE) {
            reply.errorMsg = SORRY_RETRY_LATER;
            reply.type = typeFHMsavePositionDelta;
            IPCsendMessage(&reply, sizeof(WAIreplyMsg), senderId);
        }
        else {
            delta.valueX = request->x;
            delta.valueY = request->y;
            delta.theta  = request->angle;
            reply.position = FHMsavePositionDelta(delta);
            reply.type = typeFHMsavePositionDelta;
            reply.errorMsg = OK;
            IPCsendMessage(&reply, sizeof(WAIreplyMsg), senderId);
        }
        break;

    case typeFHMposition:
        pos2.position1 = request->position1;
        pos2.position2 = request->position2;
        coor = FHMposition(pos2);
        reply.x = irint(coor.valueX);
        reply.y = irint(coor.valueY);
        reply.stdX = sqrt(coor.covMatrix[0][0]);
        reply.stdY = sqrt(coor.covMatrix[1][1]);
        reply.type = typeFHMposition;
        reply.errorMsg = OK;
        IPCsendMessage(&reply, sizeof(WAIreplyMsg), senderId);
        break;

    case typeFHMpositionFromCurrent:
        if (movingOn == TRUE) {
            reply.errorMsg = SORRY_RETRY_LATER;
            reply.type = typeFHMpositionFromCurrent;
            IPCsendMessage(&reply, sizeof(WAIreplyMsg), senderId);
        }
        else {
            pos1 = request->position1;
            coor = FHMpositionFromCurrent(pos1);
            reply.errorMsg = OK;
            reply.x = irint(coor.valueX);
            reply.y = irint(coor.valueY);
            reply.stdX = sqrt(coor.covMatrix[0][0]);
            reply.stdY = sqrt(coor.covMatrix[1][1]);
            reply.type = typeFHMpositionFromCurrent;
            IPCsendMessage(&reply, sizeof(WAIreplyMsg), senderId);
        }
        break;

    case typeFHMorientation:
        pos2.position1 = request->position1;
        pos2.position2 = request->position2;
        ori = FHMorientation(pos2);
        reply.angle = ori.theta;
        reply.stdAngle = ToDegree(sqrt(ori.covMatrix[2][2]));
        reply.type = typeFHMorientation;
        reply.errorMsg = OK;
        IPCsendMessage(&reply, sizeof(WAIreplyMsg), senderId);
    break;
```

```
            case typeFHMorientationFromCurrent:
                if (movingOn == TRUE) {
                    reply.errorMsg = SORRY_RETRY_LATER;
                    reply.type = typeFHMorientationFromCurrent;
                    IPCsendMessage(&reply, sizeof(WAIreplyMsg), senderId);
                }
                else {
                    pos1 = request->position1;
                    ori = FHMorientationFromCurrent(pos1);
                    reply.angle = ori.theta;
                    reply.stdAngle = ToDegree(sqrt(ori.covMatrix[2][2]));
                    reply.type = typeFHMorientationFromCurrent;
                    reply.errorMsg = OK;
                    IPCsendMessage(&reply, sizeof(WAIreplyMsg), senderId);
                }
                break;

            case typeFHMclearMemory:
                FHMclearMemory();
                reply.type = typeFHMclearMemory;
                reply.errorMsg = OK;
                IPCsendMessage(&reply, sizeof(WAIreplyMsg), senderId);
                break;

            case typeFHMcertaintyToCurrent:
                if (movingOn == TRUE) {
                    reply.errorMsg = SORRY_RETRY_LATER;
                    reply.type = typeFHMcertaintyToCurrent;
                    IPCsendMessage(&reply, sizeof(WAIreplyMsg), senderId);
                }
                else {
                    pos1 = request->position1;
                    reply.type = typeFHMcertaintyToCurrent;
                    reply.certainty = FHMcertaintyToCurrent(pos1);
                    reply.errorMsg = OK;
                    IPCsendMessage(&reply, sizeof(WAIreplyMsg), senderId);
                }
                break;

            case typeMCMdistance:
                pos2.position1 = request->position1;
                pos2.position2 = request->position2;
                dis = MCMdistance(pos2);
                reply.distance = dis.distance;
                reply.stdDistance = dis.stdDistance;
                reply.type = typeMCMdistance;
                reply.errorMsg = OK;
                IPCsendMessage(&reply, sizeof(WAIreplyMsg), senderId);
                break;

            case typeMCMangle:
                pos3.position1 = request->position1;
                pos3.position2 = request->position2;
                pos3.position3 = request->position3;
                ang = MCMangle(pos3);
                reply.angle = ang.angle;
                reply.stdAngle = ang.stdAngle;
                reply.type = typeMCMangle;
                reply.errorMsg = OK;
                IPCsendMessage(&reply, sizeof(WAIreplyMsg), senderId);
            break;
        }
    }
}
```

}

```
/*
*
* type definitions for message data
*
*/

typedef struct _distance{
    double   distance;
    double   stdDistance;
} Distance;

typedef struct _angle {
    int   angle;
    int stdAngle;
} Angle;

typedef int PositionName;

typedef struct _twoPositions{
    PositionName position1;
    PositionName position2;
} TwoPositions;

typedef struct _threePositions {
    PositionName position1;
    PositionName position2;
    PositionName position3;
} ThreePositions;

typedef struct _coordinate {
    double valueX;
    double valueY;
    double covMatrix[3][3];
} Coordinate;

typedef struct _orientation {
    int theta;
    double covMatrix[3][3];
} Orientation;

typedef struct _delta {
    double valueX;
    double valueY;
    int   theta;
} Delta;


/*
*
*       type definitions for internal data structures
*
*
*/

typedef struct _frame {
    PositionName frameName;
    double forwardMeanX;
    double forwardMeanY;
    int forwardMeanTheta;
    double forwardCovMatrix[3][3];
    struct _frame *forwardChain;
} *Frame;

typedef struct _polar {
```

```
    double r;
    double pi;
    int    theta;
    double covMatrix[3][3];
} Polar;

/*
*
* constants
*
*/

#define PI 3.1415927

#define FORWARD  0
#define BACKWARD 1
#define UNKNOWN  2
#define TRUE     1
#define FALSE    0

/* given the error of rotation is 1% */
#define ROT_ERROR 0.01

/* given the error of translation is 3% */
#define TRANS_ERROR 0.03

/* the size of ellipse for determining the positional certainty is  */
/* defined by X-AXIS and Y-AXIS.                                    */
#define X_AXIS    30.0      /* 30 cm */
#define Y_AXIS    20.0      /* 20 cm */
#define CERTAINTY_RADIUS  1600.0   /* r = 40 cm */
#define DELTA     2

/* Variance simulator repeatition number */
#define N_REPEAT 50
```

```
#include "whereami.h"
#include <stdio.h>
#include <math.h>

#define DEBUG 1

/*
*       WHEREAMI module
*
*       The procedures in this file are called by the main program in WAI.c
*
*       Smith and Cheeseman's model is used to track the position of
*       the robot.
*
*       Author: Seungseok Hyun
*
*       Date: Aug. 10, 1989
*
*/

/*
*       External global variables
*/

Frame  startFrameList, endFrameList, currentFrame;
int    numberFrames;
char *malloc();

/*DRMinitialize()
*
*       This subroutine initializes the internal data structures
*
*/

void DRMinitialize()
{
        int i,j;

        startFrameList = endFrameList = NULL;

        currentFrame = (Frame) malloc((unsigned) sizeof(struct _frame));
        currentFrame->forwardMeanX = 0.0;
        currentFrame->forwardMeanY = 0.0;
        currentFrame->forwardMeanTheta = 0;

        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                currentFrame->forwardCovMatrix[i][j] = 0.0;

        numberFrames = 0;
}

/*DRMdeadReckoning()
*
*       This subroutine is a temporal version for mjr's modules.
*       The final version will get messages from mlm's modules.
*
*
*/

void DRMdeadReckoning(distance, angle)
int distance, angle;
{
        int i,j;
```

```
double x1, y1, x2, y2, x3, y3;
int theta1, theta2, theta3;
double matrixH[3][3], matrixK[3][3], matrixC1[3][3], matrixC2[3][3];
double matrixC3[3][3], matrixHt[3][3], matrixKt[3][3];
double matrixTemp1[3][3], matrixTemp2[3][3], matrixTemp3[3][3];
double matrixTemp4[3][3];
double Variance();

double ToRadian();
void MatrixTranspose(), MatrixAdd(), MatrixMultiply();

if (angle == 0){                    /* translation */

    x1 = currentFrame->forwardMeanX;
    y1 = currentFrame->forwardMeanY;
    theta1 = currentFrame->forwardMeanTheta;

    x2 = 0.0;
    y2 = distance;
    theta2 = 0;

/* from Smith and Cheeseman, equation (1) */

    x3 = x2 * cos(ToRadian(theta1)) -
             y2 * sin(ToRadian(theta1)) + x1;
    y3 = x2 * sin(ToRadian(theta1)) +
             y2 * cos(ToRadian(theta1)) + y1;
    theta3 = (theta1 + theta2) % 360;

    matrixH[0][0] = matrixH[1][1] = matrixH[2][2] = 1.0;
    matrixH[0][1] = matrixH[1][0] = matrixH[2][0] = matrixH[2][1]
             = 0.0;
    matrixH[0][2] = -1.0 * (y3 - y1);
    matrixH[1][2] = x3 - x1;

    matrixK[0][2] = matrixK[1][2] = matrixK[2][0] = matrixK[2][1]
            = 0.0;
    matrixK[2][2] = 1.0;
    matrixK[0][0] = matrixK[1][1] = cos(ToRadian(theta1));
    matrixK[0][1] = -1.0 * sin(ToRadian(theta1));
    matrixK[1][0] = sin(ToRadian(theta1));

    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++){
            matrixC2[i][j] = 0.0;
            matrixC1[i][j] = currentFrame->forwardCovMatrix[i][j];
        }

    matrixC2[1][1] = Variance((double) abs(distance), TRANS_ERROR);


/* from Smith and Cheeseman, equation (4)   */

    MatrixTranspose(matrixH, matrixHt);
    MatrixTranspose(matrixK, matrixKt);

    MatrixMultiply(matrixH, matrixC1, matrixTemp1);
    MatrixMultiply(matrixTemp1, matrixHt, matrixTemp2);
    MatrixMultiply(matrixK, matrixC2, matrixTemp3);
    MatrixMultiply(matrixTemp3, matrixKt, matrixTemp4);

    MatrixAdd(matrixTemp2, matrixTemp4, matrixC3);
```

```
    currentFrame->forwardMeanX = x3;
    currentFrame->forwardMeanY = y3;
    currentFrame->forwardMeanTheta = theta3;
    for (i = 0; i < 3; i ++)
        for (j = 0; j < 3; j ++)
            currentFrame->forwardCovMatrix[i][j] = matrixC3[i][j];
}
else if (distance == 0){        /* rotation    */
    x1 = currentFrame->forwardMeanX;
    y1 = currentFrame->forwardMeanY;
    theta1 = currentFrame->forwardMeanTheta;

    x2 = 0.0;
    y2 = 0.0;

/* angle increses counter-clockwise in Smith&Cheeseman model  */

    theta2 = -1 * angle;

/* from Smith and Cheeseman, equation (1) */

    x3 = x2 * cos(ToRadian(theta1)) -
             y2 * sin(ToRadian(theta1)) + x1;
    y3 = x2 * sin(ToRadian(theta1)) +
             y2 * cos(ToRadian(theta1)) + y1;
    theta3 = (theta1 + theta2) % 360;

    matrixH[0][0] = matrixH[1][1] = matrixH[2][2] = 1.0;
    matrixH[0][1] = matrixH[1][0] = matrixH[2][0] = matrixH[2][1]
             = 0.0;
    matrixH[0][2] = -1.0 * (y3 - y1);
    matrixH[1][2] = x3 - x1;

    matrixK[0][2] = matrixK[1][2] = matrixK[2][0] = matrixK[2][1]
            = 0.0;
    matrixK[2][2] = 1.0;
    matrixK[0][0] = matrixK[1][1] = cos(ToRadian(theta1));
    matrixK[0][1] = -1.0 * sin(ToRadian(theta1));
    matrixK[1][0] = sin(ToRadian(theta1));

    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++){
            matrixC2[i][j] = 0.0;
            matrixC1[i][j] = currentFrame->forwardCovMatrix[i][j];
        }
    matrixC2[2][2] = Variance(ToRadian(abs(angle)), ROT_ERROR);

/* from Smith and Cheeseman, equation (4)   */
    MatrixTranspose(matrixH, matrixHt);
    MatrixTranspose(matrixK, matrixKt);

    MatrixMultiply(matrixH, matrixC1, matrixTemp1);
    MatrixMultiply(matrixTemp1, matrixHt, matrixTemp2);
    MatrixMultiply(matrixK, matrixC2, matrixTemp3);
    MatrixMultiply(matrixTemp3, matrixKt, matrixTemp4);

    MatrixAdd(matrixTemp2, matrixTemp4, matrixC3);

    currentFrame->forwardMeanX = x3;
    currentFrame->forwardMeanY = y3;
    currentFrame->forwardMeanTheta = theta3;
    for (i = 0; i < 3; i ++)
        for (j = 0; j < 3; j ++)
            currentFrame->forwardCovMatrix[i][j] = matrixC3[i][j];
```

```
            }
        else{
          fprintf(stderr, "\tsorry, \n");
          fprintf(stderr,
            "\tsimultaneous translation and rotation not yet supported.\n");
        }
}

/*FHMsavePosition()
*
*       Input : void
*       Output : PositionName
*
*    This saves the current position and returns the name of the position.
*
*    insert currentFrame into the linked list and computer backward stuff.
*
*/

PositionName FHMsavePosition()
{


        PositionName NewFrameName();
        void MatrixAdd(), MatrixTranspose(), MatrixMultiply();
        double ToRadian();
        int i,j;


    /* insert at the end of the linked list    */

        if (startFrameList == NULL && endFrameList == NULL){
            startFrameList = endFrameList = currentFrame;
            endFrameList->forwardChain = NULL;
        }
        else{
            endFrameList->forwardChain = currentFrame;
            currentFrame->forwardChain = NULL;
            endFrameList = currentFrame;
        }

    /* get a new name for this position(frame) */

        endFrameList->frameName = NewFrameName();

    /* allocate new space to currentFrame */

        currentFrame = (Frame) malloc((unsigned) sizeof(struct _frame));
        currentFrame->forwardMeanX = 0.0;
        currentFrame->forwardMeanY = 0.0;
        currentFrame->forwardMeanTheta = 0;

        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                currentFrame->forwardCovMatrix[i][j] = 0.0;

    /* return name */

        return(endFrameList->frameName);
}

/*FHMsavePositionDelta()
*
*       Input :  Coordinate
```

```
*       Output : PositionName
*
*    This saves the current position and returns the name of the position.
*
*    insert currentFrame into the linked list and computer backward stuff.
*
*/

PositionName FHMsavePositionDelta(delta)
Delta delta;
{


        PositionName NewFrameName();
        void MatrixAdd(), MatrixTranspose(), MatrixMultiply();
        double ToRadian();
        int i,j;


    /* angle increses counter-clockwise in Smith&Cheeseman model  */

        delta.theta = - delta.theta;

    /* insert at the end of the linked list    */

        if (startFrameList == NULL && endFrameList == NULL){
            startFrameList = endFrameList = currentFrame;
            endFrameList->forwardChain = NULL;
        }
        else{
            endFrameList->forwardChain = currentFrame;
            currentFrame->forwardChain = NULL;
            endFrameList = currentFrame;
        }

        endFrameList->forwardMeanX += (delta.valueX *
           cos(ToRadian(endFrameList->forwardMeanTheta)) -
           delta.valueY * sin(ToRadian(endFrameList->forwardMeanTheta)));
        endFrameList->forwardMeanY += (delta.valueX *
           sin(ToRadian(endFrameList->forwardMeanTheta)) +
           delta.valueY * cos(ToRadian(endFrameList->forwardMeanTheta)));

        endFrameList->forwardMeanTheta += delta.theta;

    /* get a new name for this position(frame) */

        endFrameList->frameName = NewFrameName();

    /* allocate new space to currentFrame */

        currentFrame = (Frame) malloc((unsigned) sizeof(struct _frame));

        currentFrame->forwardMeanX =  (-delta.valueX) *
            cos(ToRadian(-delta.theta)) - (-delta.valueY) *
            sin(ToRadian(-delta.theta));
        currentFrame->forwardMeanY = (-delta.valueX) *
            sin(ToRadian(-delta.theta)) + (-delta.valueY) *
            cos(ToRadian(-delta.theta));
        currentFrame->forwardMeanTheta = - delta.theta;

        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                currentFrame->forwardCovMatrix[i][j] = 0.0;
```

```
    /* return name */

        return(endFrameList->frameName);
}


/*FHMposition()
*
*       Input : TwoPositionNames
*       Output : Coordinate
*
*    This returns the x,y coordinate of the first position in the
*       frame whose origin is the second position.
*
*/

Coordinate FHMposition(positionNames)
TwoPositions positionNames;
{
        Coordinate result;
        Frame tempFrame, FHMcompound();
        int i,j;

        tempFrame = FHMcompound(positionNames);

        result.valueX = tempFrame->forwardMeanX;
        result.valueY = tempFrame->forwardMeanY;

        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                result.covMatrix[i][j] = tempFrame->forwardCovMatrix[i][j];


        return(result);
}

/*FHMpositionFromCurrent
*
*       Input : PositionName
*       Output : Coordinate
*
*  This function returns the x,y coordinate of the position w.r.t.
*    the current position of the robot.
*
*/

Coordinate FHMpositionFromCurrent(positionName)
PositionName positionName;
{
        Coordinate result;
        Frame tempFrame, FHMcompoundFromCurrent();
        int i,j;

        tempFrame = FHMcompoundFromCurrent(positionName);

        result.valueX = tempFrame->forwardMeanX;
        result.valueY = tempFrame->forwardMeanY;

        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                result.covMatrix[i][j] = tempFrame->forwardCovMatrix[i][j];

        return(result);
}
```

```
/*FHMcertaintyToCurrent()
*
*
*       Input : PositionName
*       Output: double
*
*  The certainty that the robot lies within the ellipse defined by
*  X_AXIS and Y_AXIS w.r.t. the position is calculated.
*
*/

float FHMcertaintyToCurrent(positionName)
PositionName positionName;
{


        Frame tempFrame, FHMcompoundToCurrent();
        double sigmaX, sigmaY, rho, rhoSquare;
        double sigmaXsquare, sigmaYsquare;
        double tempA, tempB, tempC;
        double x,y;
        int i,j;
        float result;

        tempFrame = FHMcompoundToCurrent(positionName);

        sigmaX = sqrt(tempFrame->forwardCovMatrix[0][0]);
        sigmaY = sqrt(tempFrame->forwardCovMatrix[1][1]);
        sigmaXsquare = tempFrame->forwardCovMatrix[0][0];
        sigmaYsquare = tempFrame->forwardCovMatrix[1][1];
        rho = tempFrame->forwardCovMatrix[0][1] / (sigmaX * sigmaY);

for (i = 0; i < 3; i++){
    for (j = 0; j < 3; j++)
        fprintf(stderr,"%f ", tempFrame->forwardCovMatrix[i][j]);
    fprintf(stderr,"\n");
}



        rhoSquare = rho * rho;

        tempA = 1.0 / (2.0 * PI * sqrt(sigmaXsquare * sigmaYsquare
                                       * (1.0 - rhoSquare)));
        tempB = -1.0 / (2.0 * (1.0 - rhoSquare));

        result = 0.0;

        /* approximation of integral */

        if (sigmaXsquare != 0.0){
            for (x = 0.0 ; x < X_AXIS; x = x + DELTA)
                for (y = 0.0 ; y < Y_AXIS; y = y + DELTA){
                  tempC = tempA  * exp(tempB * (x*x/sigmaXsquare +
                          2.0 * rho * x * y / (sigmaX * sigmaY) +
                          y*y/sigmaYsquare));
                  result += tempC * DELTA * DELTA;
                }
        }
        else{
            if (sigmaYsquare == 0.0){
                fprintf(stderr, "result is 1.0 by zero covariance matrix\n");
                return(1.0);
```

```
            }
            tempA = 1.0 / (sigmaY * sqrt(2.0 * PI));
            for (y = 0.0 ; y < Y_AXIS; y = y + DELTA){
                tempC = tempA * exp(-1.0 * y * y / (2.0 * sigmaYsquare));
                result += tempC * DELTA;
            }
        }
        result = 2.0 * result;
        /* fix errors of the approximate calculation */
        if (result > 1.0)
          result = 1.0;

fprintf(stderr,"result is %f\n", result);

        return(result);
    }


/*FHMorientation()
*
*       Input : TwoPositionNames
*       Output : Orientation
*
*    This returns the orientation of the first position in the
*       frame whose origin is the second position.
*
*/

Orientation FHMorientation(positionNames)
TwoPositions positionNames;
{
        Orientation result;
        Frame tempFrame, FHMcompound();
        int i,j;

        tempFrame = FHMcompound(positionNames);

    /* angle increses counter-clockwise in Smith&Cheeseman model  */

        result.theta = - tempFrame->forwardMeanTheta;

        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                result.covMatrix[i][j] = tempFrame->forwardCovMatrix[i][j];

        return(result);
}


/*FHMorientationFromCurrent()
*
*       Input : PositionName
*       Output : Orientation
*
*    This returns the orientation of the position  w.r.t.
*       the frame whose origin is the current position of the robot.
*
*/

Orientation FHMorientationFromCurrent(positionName)
PositionName positionName;
{
        Orientation result;
        Frame tempFrame, FHMcompoundFromCurrent();
```

```
        int i,j;

        tempFrame = FHMcompoundFromCurrent(positionName);

    /* angle increses counter-clockwise in Smith&Cheeseman model  */

        result.theta = - tempFrame->forwardMeanTheta;

        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                result.covMatrix[i][j] = tempFrame->forwardCovMatrix[i][j];

        return(result);
}

/*FHMcompound()
*
*       Input : TwoPositions
*       Output : Frame
*
*    This subroutine does the "compound" operator in Smith and Cheeseman
*
*/

Frame FHMcompound(positionNames)
TwoPositions positionNames;
{
        Frame p, p1, p2, q;
        int   order;
        double x, y, covMatrix[3][3];
        int   theta;
        int i,j;
        double matrixH[3][3], matrixHt[3][3], matrixK[3][3], matrixKt[3][3];
        double x3, y3;
        int theta3;
        double matrixC3[3][3];
        double matrixTemp1[3][3], matrixTemp2[3][3], matrixTemp3[3][3];
        double matrixTemp4[3][3];
        double matrixR[3][3], matrixRt[3][3];
        double xp, yp;
        int thetap;
        double matrixCp[3][3];
        Frame result;
        double ToRadian();

        p = startFrameList;
        p1 = NULL;
        p2 = NULL;

    /* find the position names in the frame list */

        order = UNKNOWN;

        while (p != NULL) {
            if (p->frameName == positionNames.position1) {
                p1 = p;
                if (p2 == NULL) order = BACKWARD;
                else break;
            }
            else if (p->frameName == positionNames.position2) {
                p2 = p;
                if (p1 == NULL) order = FORWARD;
                else break;
            }
```

```
    p = p->forwardChain;
}

x = 0.0;
y = 0.0;
theta = 0;
for (i = 0; i < 3; i++)
    for ( j = 0; j < 3; j++)
        covMatrix[i][j] = 0.0;

if (p1 == NULL || p2 == NULL)
    fprintf(stderr,
      "\tposition name does not exist. Or same position names\n");
else {

    if (order == FORWARD) {
        p = p2;
        q = p1;
    }
    else {
        p = p1;
        q = p2;
    }

    do {
        p = p->forwardChain;

        x3 = p->forwardMeanX * cos (ToRadian(theta)) -
             p->forwardMeanY * sin (ToRadian(theta)) +
             x;
        y3 = p->forwardMeanX * sin (ToRadian(theta)) +
             p->forwardMeanY * cos (ToRadian(theta)) +
             y;
        theta3 = (theta + p->forwardMeanTheta) % 360;

        matrixH[0][0] = matrixH[1][1] = matrixH[2][2] = 1.0;
        matrixH[0][1] = matrixH[1][0] = matrixH[2][0] = matrixH[2][1]
             = 0.0;
        matrixH[0][2] = -1.0 * (y3 - y);
        matrixH[1][2] = x3 - x;

        matrixK[0][2] = matrixK[1][2] = matrixK[2][0] = matrixK[2][1]
            = 0.0;
        matrixK[0][0] = matrixK[1][1] = cos(ToRadian(theta));
        matrixK[2][2] = 1.0;
        matrixK[0][1] = -1.0 * sin(ToRadian(theta));
        matrixK[1][0] = sin(ToRadian(theta));

        MatrixTranspose(matrixH, matrixHt);
        MatrixTranspose(matrixK, matrixKt);

        MatrixMultiply(matrixH, covMatrix, matrixTemp1);
        MatrixMultiply(matrixTemp1, matrixHt, matrixTemp2);
        MatrixMultiply(matrixK, p->forwardCovMatrix, matrixTemp3);
        MatrixMultiply(matrixTemp3, matrixKt, matrixTemp4);
        MatrixAdd(matrixTemp2, matrixTemp4, matrixC3);

        x = x3;
        y = y3;
        theta = theta3;
        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                covMatrix[i][j] = matrixC3[i][j];
    } while (p != q);
```

```
            if (order == BACKWARD) {
                xp = -1.0 * x * cos(ToRadian(theta)) -
                     y * sin(ToRadian(theta));
                yp = x * sin(ToRadian(theta)) - y * cos(ToRadian(theta));
                thetap = -1 * theta;
                matrixR[2][0] = matrixR[2][1] = 0.0;
                matrixR[2][2] = -1.0;
                matrixR[0][0] = matrixR[1][1] = -1.0 * cos(ToRadian(theta));
                matrixR[0][1] = -1.0 * sin(ToRadian(theta));
                matrixR[1][0] = sin(ToRadian(theta));
                matrixR[0][2] = yp;
                matrixR[1][2] = -1.0 * xp;

                MatrixTranspose(matrixR, matrixRt);

                MatrixMultiply(matrixR, covMatrix, matrixTemp1);
                MatrixMultiply(matrixTemp1, matrixRt, matrixCp);

                x = xp;
                y = yp;
                theta = thetap;
                for (i = 0; i < 3; i++)
                    for (j = 0; j < 3; j++)
                        covMatrix[i][j] = matrixCp[i][j];
            }
        }

        result = (Frame) malloc((unsigned) sizeof(struct _frame));
        result->forwardMeanX = x;
        result->forwardMeanY = y;
        result->forwardMeanTheta = theta;
        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                result->forwardCovMatrix[i][j] = covMatrix[i][j];
        return(result);
}

/*FHMcompoundFromCurrent()
*
*       Input : PositionName
*       Output : Frame
*
*    This subroutine does the "compound" operator in Smith and Cheeseman
*
*/

Frame FHMcompoundFromCurrent(positionName)
PositionName positionName;
{
        Frame p, p1;
        double x, y, covMatrix[3][3];
        int   theta;
        int i,j;
        double matrixH[3][3], matrixHt[3][3], matrixK[3][3], matrixKt[3][3];
        double x3, y3;
        int theta3;
        double matrixC3[3][3];
        double matrixTemp1[3][3], matrixTemp2[3][3], matrixTemp3[3][3];
        double matrixTemp4[3][3];
        double matrixR[3][3], matrixRt[3][3];
        double xp, yp;
        int thetap;
        double matrixCp[3][3];
```

```
    Frame result;
    double ToRadian();

    p = startFrameList;
    p1 = NULL;

/* find the position names in the frame list */

    while (p != NULL) {
        if (p->frameName == positionName)
            p1 = p;
        p = p->forwardChain;
    }

    x = 0.0;
    y = 0.0;
    theta = 0;
    for (i = 0; i < 3; i++)
        for ( j = 0; j < 3; j++)
            covMatrix[i][j] = 0.0;

    if (p1 == NULL)
        fprintf(stderr,"\tposition name does not exist. \n");
    else{
        p = p1;
        if (p1 != endFrameList) {
            do {
                p = p->forwardChain;
                x3 = p->forwardMeanX * cos (ToRadian(theta)) -
                     p->forwardMeanY * sin (ToRadian(theta)) +
                     x;
                y3 = p->forwardMeanX * sin (ToRadian(theta)) +
                     p->forwardMeanY * cos (ToRadian(theta)) +
                     y;
                theta3 = (theta + p->forwardMeanTheta) % 360;

                matrixH[0][0] = matrixH[1][1] = matrixH[2][2] = 1.0;
                matrixH[0][1] = matrixH[1][0] = matrixH[2][0]
                              = matrixH[2][1] = 0.0;
                matrixH[0][2] = -1.0 * (y3 - y);
                matrixH[1][2] = x3 - x;

                matrixK[0][2] = matrixK[1][2] = matrixK[2][0]
                              = matrixK[2][1] = 0.0;
                matrixK[0][0] = matrixK[1][1] = cos(ToRadian(theta));
                matrixK[2][2] = 1.0;
                matrixK[0][1] = -1.0 * sin(ToRadian(theta));
                matrixK[1][0] = sin(ToRadian(theta));

                MatrixTranspose(matrixH, matrixHt);
                MatrixTranspose(matrixK, matrixKt);

                MatrixMultiply(matrixH, covMatrix, matrixTemp1);
                MatrixMultiply(matrixTemp1, matrixHt, matrixTemp2);
                MatrixMultiply(matrixK, p->forwardCovMatrix, matrixTemp3);
                MatrixMultiply(matrixTemp3, matrixKt, matrixTemp4);
                MatrixAdd(matrixTemp2, matrixTemp4, matrixC3);

                x = x3;
                y = y3;
                theta = theta3;
                for (i = 0; i < 3; i++)
                    for (j = 0; j < 3; j++)
                        covMatrix[i][j] = matrixC3[i][j];
```

```
    } while (p != endFrameList);
}

/* merge with CurrentFrame */

x3 = currentFrame->forwardMeanX * cos(ToRadian(theta)) -
     currentFrame->forwardMeanY * sin(ToRadian(theta)) +
     x;
y3 = currentFrame->forwardMeanX * sin(ToRadian(theta)) +
     currentFrame->forwardMeanY * cos(ToRadian(theta)) +
     y;
theta3 = (theta + currentFrame->forwardMeanTheta) % 360;

matrixH[0][0] = matrixH[1][1] = matrixH[2][2] = 1.0;
matrixH[0][1] = matrixH[1][0] = matrixH[2][0] = matrixH[2][1]
     = 0.0;
matrixH[0][2] = -1.0 * (y3 - y);
matrixH[1][2] = x3 - x;

matrixK[0][2] = matrixK[1][2] = matrixK[2][0] = matrixK[2][1]
    = 0.0;
matrixK[0][0] = matrixK[1][1] = cos(ToRadian(theta));
matrixK[2][2] = 1.0;
matrixK[0][1] = -1.0 * sin(ToRadian(theta));
matrixK[1][0] = sin(ToRadian(theta));

MatrixTranspose(matrixH, matrixHt);
MatrixTranspose(matrixK, matrixKt);

MatrixMultiply(matrixH, covMatrix, matrixTemp1);
MatrixMultiply(matrixTemp1, matrixHt, matrixTemp2);
MatrixMultiply(matrixK, currentFrame->forwardCovMatrix,
               matrixTemp3);
MatrixMultiply(matrixTemp3, matrixKt, matrixTemp4);
MatrixAdd(matrixTemp2, matrixTemp4, matrixC3);

x = x3;
y = y3;
theta = theta3;
for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
        covMatrix[i][j] = matrixC3[i][j];

/* reverse conversion */
xp = -1.0 * x * cos(ToRadian(theta)) -
     y * sin(ToRadian(theta));
yp = x * sin(ToRadian(theta)) - y * cos(ToRadian(theta));
thetap = -1 * theta;
matrixR[2][0] = matrixR[2][1] = 0.0;
matrixR[2][2] = -1.0;
matrixR[0][0] = matrixR[1][1] = -1.0 * cos(ToRadian(theta));
matrixR[0][1] = -1.0 * sin(ToRadian(theta));
matrixR[1][0] = sin(ToRadian(theta));
matrixR[0][2] = yp;
matrixR[1][2] = -1.0 * xp;

MatrixTranspose(matrixR, matrixRt);

MatrixMultiply(matrixR, covMatrix, matrixTemp1);
MatrixMultiply(matrixTemp1, matrixRt, matrixCp);

x = xp;
y = yp;
theta = thetap;
```

```
            for (i = 0; i < 3; i++)
                for (j = 0; j < 3; j++)
                    covMatrix[i][j] = matrixCp[i][j];
        }

        result = (Frame) malloc((unsigned) sizeof(struct _frame));
        result->forwardMeanX = x;
        result->forwardMeanY = y;
        result->forwardMeanTheta = theta;
        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                result->forwardCovMatrix[i][j] = covMatrix[i][j];
        return(result);
}

/*FHMcompoundToCurrent()
*
*       Input : PositionName
*       Output : Frame
*
*    This subroutine does the "compound" operator in Smith and Cheeseman
*
*/

Frame FHMcompoundToCurrent(positionName)
PositionName positionName;
{
        Frame p, p1;
        double x, y, covMatrix[3][3];
        int   theta;
        int i,j;
        double matrixH[3][3], matrixHt[3][3], matrixK[3][3], matrixKt[3][3];
        double x3, y3;
        int theta3;
        double matrixC3[3][3];
        double matrixTemp1[3][3], matrixTemp2[3][3], matrixTemp3[3][3];
        double matrixTemp4[3][3];
        Frame result;
        double ToRadian();

        fprintf(stderr," ToCurrent position is %d\n", positionName);
        p = startFrameList;
        p1 = NULL;

    /* find the position names in the frame list */

        while (p != NULL) {
            if (p->frameName == positionName)
                p1 = p;
            p = p->forwardChain;
        }

        x = 0.0;
        y = 0.0;
        theta = 0;
        for (i = 0; i < 3; i++)
            for ( j = 0; j < 3; j++)
                covMatrix[i][j] = 0.0;

        if (p1 == NULL)
            fprintf(stderr,"\tposition name does not exist. \n");
        else{
            p = p1;
            if (p1 != endFrameList) {
```

```
    do {
        p = p->forwardChain;
        x3 = p->forwardMeanX * cos (ToRadian(theta)) -
             p->forwardMeanY * sin (ToRadian(theta)) +
             x;
        y3 = p->forwardMeanX * sin (ToRadian(theta)) +
             p->forwardMeanY * cos (ToRadian(theta)) +
             y;
        theta3 = (theta + p->forwardMeanTheta) % 360;

        matrixH[0][0] = matrixH[1][1] = matrixH[2][2] = 1.0;
        matrixH[0][1] = matrixH[1][0] = matrixH[2][0]
                      = matrixH[2][1] = 0.0;
        matrixH[0][2] = -1.0 * (y3 - y);
        matrixH[1][2] = x3 - x;

        matrixK[0][2] = matrixK[1][2] = matrixK[2][0]
                      = matrixK[2][1] = 0.0;
        matrixK[0][0] = matrixK[1][1] = cos(ToRadian(theta));
        matrixK[2][2] = 1.0;
        matrixK[0][1] = -1.0 * sin(ToRadian(theta));
        matrixK[1][0] = sin(ToRadian(theta));

        MatrixTranspose(matrixH, matrixHt);
        MatrixTranspose(matrixK, matrixKt);

        MatrixMultiply(matrixH, covMatrix, matrixTemp1);
        MatrixMultiply(matrixTemp1, matrixHt, matrixTemp2);
        MatrixMultiply(matrixK, p->forwardCovMatrix, matrixTemp3);
        MatrixMultiply(matrixTemp3, matrixKt, matrixTemp4);
        MatrixAdd(matrixTemp2, matrixTemp4, matrixC3);

        x = x3;
        y = y3;
        theta = theta3;
        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                covMatrix[i][j] = matrixC3[i][j];
    } while (p != endFrameList);
}

/* merge with CurrentFrame */

x3 = currentFrame->forwardMeanX * cos(ToRadian(theta)) -
     currentFrame->forwardMeanY * sin(ToRadian(theta)) +
     x;
y3 = currentFrame->forwardMeanX * sin(ToRadian(theta)) +
     currentFrame->forwardMeanY * cos(ToRadian(theta)) +
     y;
theta3 = (theta + currentFrame->forwardMeanTheta) % 360;

matrixH[0][0] = matrixH[1][1] = matrixH[2][2] = 1.0;
matrixH[0][1] = matrixH[1][0] = matrixH[2][0] = matrixH[2][1]
     = 0.0;
matrixH[0][2] = -1.0 * (y3 - y);
matrixH[1][2] = x3 - x;

matrixK[0][2] = matrixK[1][2] = matrixK[2][0] = matrixK[2][1]
    = 0.0;
matrixK[0][0] = matrixK[1][1] = cos(ToRadian(theta));
matrixK[2][2] = 1.0;
matrixK[0][1] = -1.0 * sin(ToRadian(theta));
matrixK[1][0] = sin(ToRadian(theta));
```

```
            MatrixTranspose(matrixH, matrixHt);
            MatrixTranspose(matrixK, matrixKt);

            MatrixMultiply(matrixH, covMatrix, matrixTemp1);
            MatrixMultiply(matrixTemp1, matrixHt, matrixTemp2);
            MatrixMultiply(matrixK, currentFrame->forwardCovMatrix,
                           matrixTemp3);
            MatrixMultiply(matrixTemp3, matrixKt, matrixTemp4);
            MatrixAdd(matrixTemp2, matrixTemp4, matrixC3);

            x = x3;
            y = y3;
            theta = theta3;
            for (i = 0; i < 3; i++)
                for (j = 0; j < 3; j++)
                    covMatrix[i][j] = matrixC3[i][j];

        }

        result = (Frame) malloc((unsigned) sizeof(struct _frame));
        result->forwardMeanX = x;
        result->forwardMeanY = y;
        result->forwardMeanTheta = theta;
        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                result->forwardCovMatrix[i][j] = covMatrix[i][j];
        return(result);
}

/* FHMclearMemory()
*
*       Input : void
*       Output: void
*
*    This functions clears the memory allocated for positions so far.
*
*/

void
FHMclearMemory()
{
        Frame p,q;

        p = startFrameList;

        while ( p != NULL){
            q = p;
            p = p->forwardChain;
            free(q);
        }
        DRMinitialize();
}

/*MCMdistance()
*
*       Input : TwoPositions
*       Output : Distance
*
*    This returns the distance between two position.
*
*    search two positions in the linked list, and get relative coordinate
*    between them, and then compute distance.
*/
```

```
Distance MCMdistance(positionNames)
TwoPositions positionNames;
{
        Distance result;
        Polar    polarTemp;
        Frame tempFrame, FHMcompound();
        Polar  ToPolar();

        tempFrame = FHMcompound(positionNames);
        polarTemp = ToPolar(tempFrame);

        result.distance = polarTemp.r;
        result.stdDistance = sqrt(polarTemp.covMatrix[0][0]);

        return(result);
}


/*MCMangle()
*
*       Input : ThreePositions
*       Output : Angle
*
*    This returns the angle of three positions.
*
*
*
*/

Angle MCMangle(positionNames)
ThreePositions positionNames;
{
        TwoPositions tempPositions;
        Frame tempFrame1, tempFrame2, FHMcompound();
        Polar polarTemp1, polarTemp2;
        int r1, r2;
        Angle result;
        Polar ToPolar();

        tempPositions.position1 = positionNames.position1;
        tempPositions.position2 = positionNames.position2;
        tempFrame1 = FHMcompound(tempPositions);

        tempPositions.position1 = positionNames.position3;
        tempFrame2 = FHMcompound(tempPositions);

        polarTemp1 = ToPolar(tempFrame1);
        polarTemp2 = ToPolar(tempFrame2);

        result.stdAngle = ToDegree(sqrt(polarTemp1.covMatrix[1][1] +
                                polarTemp2.covMatrix[1][1]));

        r1 = ToDegree(polarTemp1.pi);
        r2 = ToDegree(polarTemp2.pi);

        if (r1 < 0)
                r1 = 360 + r1;
        if (r2 < 0)
                r2 = 360 + r2;
        result.angle = r1-r2;
        if (result.angle > 180)
            result.angle = - (360 - result.angle);
        else if (result.angle < -180)
            result.angle = 360 + result.angle;
```

```
        return(result);

}

/* ToPolar()
*
*       input : Frame
*       output : Polar
*
*  mapping from Cartesian coordinate system to polar coordinate system
*
*/

Polar ToPolar(frameArg)
Frame frameArg;
{
        Polar result;
        double matrixR[3][3], matrixRt[3][3], matrixTemp[3][3];

        result.r = sqrt(frameArg->forwardMeanX * frameArg->forwardMeanX +
                        frameArg->forwardMeanY * frameArg->forwardMeanY);
        result.pi = atan2(frameArg->forwardMeanY, frameArg->forwardMeanX);
        result.theta = frameArg->forwardMeanTheta;

        matrixR[0][2] = matrixR[1][2] = matrixR[2][0] = matrixR[2][1] = 0.0;
        matrixR[2][2] = 1.0;

        if (result.r != 0.0){
            matrixR[0][0] = frameArg->forwardMeanX / result.r;
            matrixR[0][1] = frameArg->forwardMeanY / result.r;
            matrixR[1][0] = frameArg->forwardMeanY / (result.r * result.r);
            matrixR[1][1] = frameArg->forwardMeanY / (result.r * result.r);
        }
        else{
            fprintf(stderr, "\t The result of ToPolar transformation");
            fprintf(stderr, "\t   may be wrong.");
            matrixR[0][0] = matrixR[0][1] = matrixR[1][0]
               = matrixR[1][1] = 1.0;
        }

        MatrixTranspose(matrixR, matrixRt);
        MatrixMultiply(matrixR, frameArg->forwardCovMatrix, matrixTemp);
        MatrixMultiply(matrixTemp, matrixRt, result.covMatrix);

        return(result);
}

/* ToRadian()
*
*       input : int degree
*       output : double
*
* returns a radian of a degree
*
*/

double ToRadian(degree)
int degree;
{
        return((double) PI * degree / 180.0);
}

/* ToDegree()
```

```
*
*       input : double radian
*       output : int
*
* returns a degree of a radian
*
*/

int ToDegree(radian)
double radian;
{
        return((int)(radian * 180.0 / PI));
}

/* MatrixTranspose(0
*
*
*  3x3 matrix transposition
*
*/

void MatrixTranspose(source, target)
double source[3][3], target[3][3];
{
        int i,j;

        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                target[j][i] = source[i][j];
}

/* MatrixAdd()
*
*  3x3 matrix addition
*
*/

void MatrixAdd(matrix1, matrix2, result)
double matrix1[3][3], matrix2[3][3], result[3][3];
{
        int i,j;

        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                result[i][j] = matrix1[i][j] + matrix2[i][j];
}

/* MatrixMultiply()
*
*
*  3x3 matrix mautiplication
*
*/

void MatrixMultiply(matrix1, matrix2, result)
double matrix1[3][3], matrix2[3][3], result[3][3];
{
        int i,j,k;

        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++){
                result[i][j] = 0.0;
                for (k = 0; k < 3; k++)
                    result[i][j] += matrix1[i][k] * matrix2[k][j];
```

```
            }
    }

    /* NewFrameName()
    *
    *
    *  generates new name, PositionName type is an alias of int.
    *
    */

    PositionName NewFrameName()
    {
            PositionName tempName;

            tempName = numberFrames;
            numberFrames++;

            return(tempName);
    }

    /* Variance()
    *
    *
    *  get the variance of a given motion and an error using a simple
    * simulation
    *
    */

    double Variance(motion_length, error)
    double motion_length, error;
    {

            double sum, sumXsquare, mean, result, x;
            double SimRandom();
            int i;

            sum = sumXsquare = 0.0;
            for (i = 0; i < N_REPEAT; i++){
                x = motion_length + motion_length * error * SimRandom();
                sum += x;
                sumXsquare += x * x;
            }

            mean = sum / N_REPEAT;
            result = sumXsquare / N_REPEAT - (mean * mean);

            return(result);
    }


    /*
    *
    *       SimRandom() returns random numbers between -1.0 and 1.0
    *
    */

    double SimRandom()
    {
            double result;

            result = (rand() - 1.07375e+09) / 1.07375e+09;

            return(result);
    }
```