DOE/MC/29115 -- 5658-Vol.I
(DE97002291)

An Intelligent Inspection and Survey Robot

Volume I

Topical Report

December 15, 1995

Work Performed Under Contract No.: DE-AC21-92MC29115

For

U.S. Department of Energy
Office of Environmental Management
Office of Technology Development
1000 Independence Avenue
Washington, DC 20585

U.S. Department of Energy
Office of Fossil Energy
Federal Energy Technology Center
Morgantown Site
P.O. Box 880
Morgantown, West Virginia 26507-0880

By
SCUREF
South Carolinas Universities
Research and Foundation
Strom Thurmond Institute
Clemson, South Carolina 29634-5701

ABSTRACT

ARIES #1 (*A*utonomous *R*obotic *I*nspection *E*xperimental *S*ystem), has been developed for the Department of Energy to survey and inspect drums containing low-level radioactive waste stored in warehouses at DOE facilities. The drums are typically stacked four high and arranged in rows with three-foot aisle widths. The robot will navigate through the aisles and perform an inspection operation, typically performed by a human operator, making decisions about the condition of the drums and maintaining a database of pertinent information about each drum.

A new version of the Cybermotion series of mobile robots is the base mobile vehicle for ARIES. The new Model K3A consists of an improved and enhanced mobile platform and a new turret that will permit turning around in a three-foot aisle. Advanced sonar and lidar systems were added to improve navigation in the narrow drum aisles. Onboard computer enhancements include a VMEbus computer system running the VxWorks real-time operating system. A graphical offboard supervisory UNIX workstation is used for high-level planning, control, monitoring, and reporting. A camera positioning system (CPS) includes primitive instructions for the robot to use in referencing and positioning the payload. The CPS retracts to a more compact position when traveling in the open warehouse. During inspection, the CPS extends up to deploy inspection packages at different heights on the four-drum stacks of 55-, 85-, and 110-gallon drums.

The vision inspection module performs a visual inspection of the waste drums. This system will locate and identify each drum, locate any unique visual features, characterize relevant surface features of interest (such as paint blisters, rusted areas, etc.), and update a database containing the inspection data. An adaptive algorithm and learning concept, requiring little effort by unskilled operators, will be featured to "train" the vision system prior to the actual inspection process. A supplemental prototype dexterity inspection package using a commercial robot arm has been developed to provide more sophisticated manipulation for more detailed inspection of the waste containers. Radiation hardening studies were performed to determine what design modifications would be required in order for ARIES #1 to meet potential future inspection and survey applications in higher-radiation areas and to provide a means of modifying the systems to a fully radiation-hardened version without major software revisions.

This report is Volume 1[1] of the two-volume Phase 2 Topical Report, *An Intelligent Inspection and Survey Robot.*

[1] Volume 2, Radiation Hardening Inspection and Survey Robots

DISCLAIMER

ACKNOWLEDGMENTS

This work is performed under Contract No. DE-AC21-92MC29115 for the U. S. Department of Energy's Morgantown Energy Technology Center.

We acknowledge and appreciate the interest, cooperation, and support of the DOE METC Project Manager, Mr. Cliff Carpenter, as well as other managers who have been associated with this project, Mr. Vijay Kothari and Dr. Kelly Pearce.

Special thanks and appreciation are extended to Dr. Judy Bostock, the previous Chief Operating Officer of SCUREF, for her support and many contributions to this project.

Thanks and appreciation is expressed to Mr. Brack Hazen of FERMCO. He provided video tapes containing images of drums stored at Fernald and storage warehouse drawings which were networked to our computer systems. He also provided additional documentation concerning the inspection of drums at Fernald and has arranged for testing and demonstrations of ARIES at Fernald warehouses. This provided us a "target" warehouse for our initial prototype.

We appreciate the interest expressed by Mr. Bruce Wilding of Lockheed-Martin Idaho Technology Co. who is currently gathering data and interest for future testing and demonstrations of robotic inspection systems at Idaho and other DOE storage sites.

We most especially acknowledge and appreciate the graduate students who have contributed many hours of good work to this project: Noel Eaton, Chris King, Blake Langland, Al Mew, Michael Sechrest, Otto Jansky, and Kinsler Vinson at University of South Carolina; and, Albrecht Carver, Nicolae Costescu, Khaled Shaaban, Atul Suri and Mike Mercier at Clemson University. As always with university projects, the graduate students make up the work force that gets the job done. In addition, through the funding of this project, METC has provided excellent "real world" experience for these students.

TABLE OF CONTENTS

1. EXECUTIVE SUMMARY

ARIES: Autonomous Robotic Inspection Experimental System

ARIES (Autonomous Robotic Inspection Experimental System), has been developed and demonstrated for the Department of Energy to survey and inspect drums containing low-level radioactive waste stored in warehouses at DOE facilities. The drums are typically stacked four-high and arranged in rows with three-foot aisle widths. The robot will navigate through the aisles and perform an inspection operation, typically performed by a human operator, making decisions about the condition of the drums and maintaining a database of pertinent information about each drum. This system was demonstrated at the University of South Carolina staging area on 30 November 1995. The staging area consists of approximately 200 55-, 85-, and 110-gallon drums arranged in three rows.

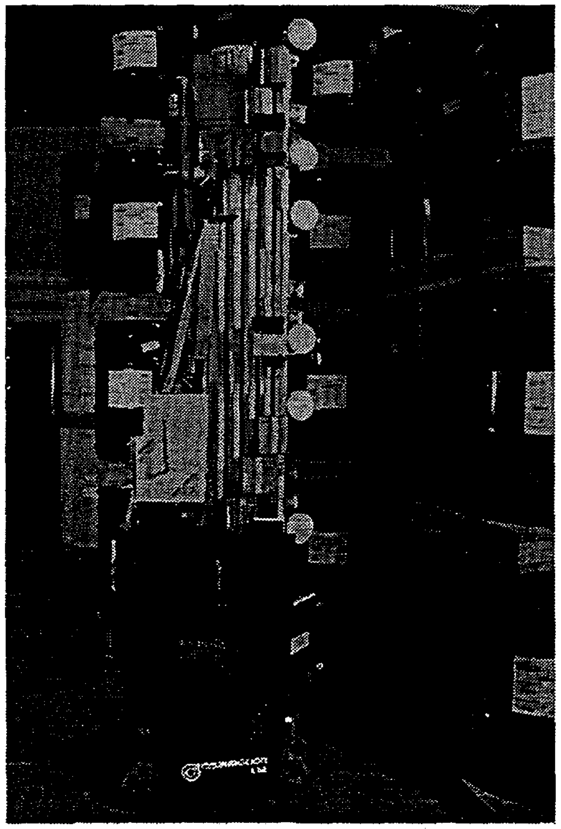


Figure 1-1 ARIES

The first phase of this three-phase project was a task-oriented, proof-of-principle phase in which demonstrations and reports were provided as the deliverables. The second phase was a technology integration effort to develop a single, commercializable prototype mobile robot capable of meeting many of the demands of the mission of environmental compliance and clean-up of DOE sites. Phase 3 will demonstrate and evaluate the prototype in a DOE warehouse storage facility, and then productize the system for commercial availability.

As a research and development industrial partner on the project team and in cooperation with researchers at the universities, Cybermotion, Inc. has developed a new model in their Navmaster series of mobile robots which is the base mobile vehicle for ARIES. This new version consists of an improved and enhanced mobile platform (Model K3A) and a new subturret that will permit turning around in a three-foot aisle of drums. Other Cybermotion enhancements to the robot sonar system and a new lidar system have improved the ability to navigate in the drum aisles and other close areas as well as the large open areas of the warehouses.

New designs and enhancements made at the University of South Carolina include an onboard VMEbus computer system running the VxWorks real-time operating system. An onboard mission handler communicates with other onboard Cybermotion computers via internal serial control links and with offboard workstations via a wireless Ethernet bridge. A graphical offboard supervisory UNIX workstation is used for high-level planning, control, monitoring, and reporting. Major software modules include: robot operations monitors, visualization tools, drum database, robot path planner, robot dispatcher, site manager, and visual displays.

A camera positioning system (CPS) was designed by the University of South Carolina and Cybermotion and is capable of performing survey and inspection of drums in the warehouse. The CPS was fabricated at Cybermotion. Drum-referencing and camera-positioning algorithms are included in the primitive instruction set for the robot. The CPS will position the vision camera and any other required instrumentation packages (bar-code reader, etc.) to perform the inspection process for each drum. This is a power-efficient system designed to complement the features of the enhanced mobile robot system.

The application payload of the camera positioning system includes computer vision and bar-code scanner modules developed at Clemson University. The vision module will perform a visual inspection of the waste drums. This system will locate and identify each drum, characterize relevant surface features (such as paint blisters, rusted areas, etc.), and update a database containing the inspection information. Color processing, using specialized algorithms written in C language, will incorporate supplemental multi-strobe lighting and differential strobe-based structured lighting. An adaptive algorithm and learning concept, designed for use by unskilled computer operators, will be featured to "train" the vision system prior to the actual inspection process.

An electrical power management module is being designed for ARIES. Distributing and controlling power to the various payload modules is recognized as a vital and necessary component for a complex mobile robot application. System power has been analyzed to determine additional batteries needed for the base system in order to realize an eight-hour inspection shift.

A supplemental prototype dexterity inspection package has been developed to provide more sophisticated manipulations for more detailed inspection of the waste containers. A camera mounted on the end of a six degree-of-freedom revolute manipulator can be used for close inspection of the surface. An additional sampling tool can be used to obtain surface samples of suspect areas and return for operator analysis. This module was demonstrated during the Phase 2 demonstration at USC, but there are no current plans for continued development or testing at a DOE warehouse site.

A radiation hardening study was performed for the ARIES #1 and is reported as Volume 2 of this Topical Report. The report includes plans and costs associated with the development of a radiation-hardened version of ARIES #1.

Phase 3 proposes enhancements based on testing and demonstration in the USC staging area. The ARIES #1 prototype will then be tested and demonstrated at a DOE storage warehouse at Fernald, Ohio. Successful testing there will lead to additional enhancements based on Fernald input and possible testing at other DOE facilities.

The Appendix contains the Phase 2 Task Descriptions and Work Breakdown Structure (WBS).

2. ARIES ROBOTIC VEHICLE

2.1 CYBERMOTION BACKGROUND

This work was performed under Task WBS 2.2 of the project Work Breakdown Structure.

Understanding Cybermotion's contribution to the **ARIES** program requires a brief explanation of the Company's philosophy and past. Although Cybermotion has been fortunate enough to receive several government contracts over the past decade, the Company views itself as a commercial manufacturer of mobile robots. Even within this view, Cybermotion further restricts its focus to relatively light vehicles of a single basic family. This basic vehicle is used predominately in indoor environments for security patrol, material handling, and hazardous monitoring applications. Customers for Cybermotion's vehicles range from J.C. Penny to the U.S. Army.

This focus has meant that all of the various improvements made to the system, whether on internal R&D, or as the result of a commercial or government contract, have served to enhance the vehicle's capabilities in all of its roles. Thus, **ARIES** has inherited many capabilities from this earlier work, and the **ARIES** developments will be available to address future applications of the technology. Had this not been the case, the cost of performing Cybermotion's part of the **ARIES** Phase 2 would have been prohibitive.

2.2 TASK 2.2.1 MOBILE VEHICLE:

2.2.1 K3A Mobile Platform

The narrow aisle requirement of **ARIES** dictated the use of a mobile platform with a narrower footprint and higher stability than standard Cybermotion K2A. This requirement was met by the K3A [Ref. 1]. Like the K2A, the K3A Mobile Platform is a Synchro-Drive vehicle using concentric drive shafts to transmit drive and steering forces to three "foot assemblies". The three foot assemblies are located on the ends of "leg tubes" oriented symmetrically about the base at 120 degree intervals. The name Synchro-Drive is derived from the fact that all wheels remain perpetually parallel because they are both steered and driven in mechanical synchronization.

Synchro-Drive vehicles have many characteristics that make them ideal for the **ARIES** application environment:

- Zero turning radius (turn in place).
- All wheel drive.
- Excellent odometry.
- Simplicity of control.
- Symmetric geometry (To allow right angle turns).
- High payload capacity.
- Supported by well proven and extensive navigation firmware.

The deficiency of the K2A for this application grew from the fact that each foot assembly of the base includes only one wheel assembly, and that wheel is offset from the foot. When the steering direction is such that a particular wheel on the K2A is oriented under the leg tube, the base of support is considerably reduced and thus the K2A has limited lateral (roll) stability. This is the case despite the fact that the leg tubes of the K2A extend from under the base. For normal applications this is not a problem, but the **ARIES** vehicle is required to operate in 36 inch aisles, and to carry a Camera Positioning System (CPS) that significantly increases (heightens) its center of gravity. A new mobile platform was required.

The K3A has many of the same internal structures and similar concentric shaft leg tubes as the K2A, but it sports two wheels on each foot assembly. These foot assemblies assure that the base of support remains reasonable through all orientations of steering, despite significantly shortened leg tubes. All wheels of the K3A are driven in absolute mechanical lock with each other without differentials. A proof of the K3A concept (one leg and foot assembly) already existed prior to the Phase 2 contract, but it had not been developed to the prototype vehicle stage.

A relative difference in rolling motion is required between the two wheels of a foot during steering. This difference is accomplished, independent of any drive action, by the use of a mechanical adder in each foot. An adder was chosen in place of a more conventional differential due to the problems expected from a differential drive on uneven surfaces.

The K3A is completely compatible with the electronics and software of the K2A, and thus inherited navigation and control firmware developed and refined over more than a decade. For expedience, the control panel from the K2A was fitted to the K3A on the Phase 2 model. This control panel represents a venerable and well tested design. A new control panel is being designed for both the K2A and the K3A. This control panel will utilize an MPP (Multi-purpose Processor) in place of the on-board computer. The MPP Computer was developed under DOD (MDARS) Contract DAAK70-94-C-0034. It has greatly enhanced performance compared to the DC-1, and is less than 1/3 the physical size. The MPP is also interchangeable with all control processors in the vehicle, reducing the number of spares required. Finally, the MPP uses FLASH program memory, and can thus be reprogrammed over the radio link without the need to physically change PROM memory chips. The first MPP board is being tested at this writing.

2.2.2 ST-1A Subturret

One characteristic of Synchro-Drive vehicles is that the mobile base does not turn (rotate around its vertical axis) as it executes steering maneuvers; only its wheels turn in the direction of steering. For this reason, a turret is provided that is coupled to the steering actuator in such a way that it always faces in the direction of motion. This turret is connected to the base electrically by a slip-ring, and consists of two parts; the Subturret and the Applications Payload. The Subturret acts as a foundation for a wide variety of applications, and includes many of the subsystems required to enable communications and navigation. In the case of **ARIES**, these include a Docking Beacon system, a Collision Avoidance System, a VME Card Cage (for vision related hardware), an autocharging receptacle, an expansion bus, and a Spread Spectrum Radio data communication link.

The Standard ST-1 Subturret is oblong in shape, thus requiring more clearance for turning than for simply driving straight. For the **ARIES** program, an ST-1 Turret was modified to shorten it and provide more deeply recessed side sonar ports. The shortening permits the vehicle to turn in a 36 inch aisle, and the deeper ports permits the accurate ranging of objects at very close proximity to the sides of the vehicle.

The top of the Subturret is flat with four mounting posts on 24 inch centers and a self-aligning connector to accommodate the payload. For **ARIES**, the mounting centers and connector were maintained at their standard (ST-1) locations despite the shortening of the overall length. This maintaining of the mounting standards allows the **ARIES** Subturret to be placed on a K2A platform, and the CPS to be placed on either an ST-1 or ST-1A Subturret. This flexibility proved most useful when the only vehicle available for testing the CPS was a K2A with an ST-1 Subturret.

The K3A and ST-1A Subturrets were fabricated with no significant difficulties. A second K3A vehicle was produced for use at Cybermotion (at no cost to the contract).

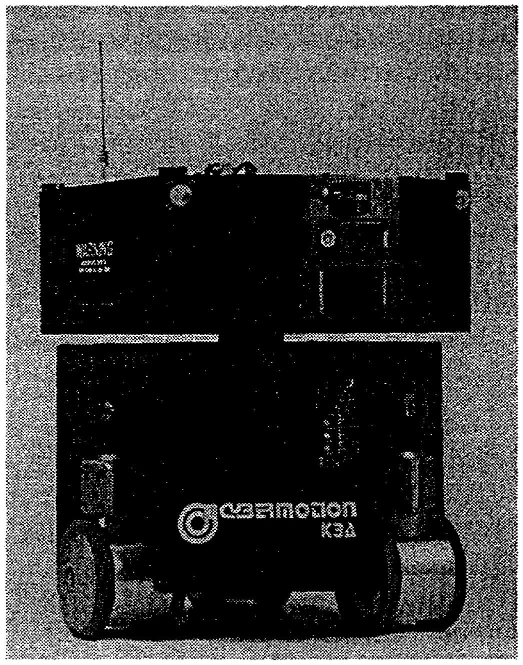


Figure 2.1 K3A Base With New Subturret

2.2.3 Lidar Navigation

The K3A vehicle inherited an extensive library of navigation techniques from the K2A. These techniques included WALL and HALL referencing, DOOR detection and navigation, reflective STRIPE navigation, GATE referencing, etc. Most of these techniques were developed for environments where sonar navigation is the preferred technique.

During Phase I, Cybermotion assisted USC in the development of a navigation technique that imaged drums using sonar and corrected the vehicle's position estimate. This technique used the law-of-cosines to locate the drum centers, and compared their location to the programmed locations. During the visit to Fernald at the beginning of Phase 2, it was learned that the position of drums is frequently changed, and that they cannot practically be used as precision navigation references. Without some precision reference, the vehicle would be forced to reference its movements only to the drum aisles when navigating from one area in the warehouse to another.

One of the navigation techniques inherited from the K2A utilizes a rotating lidar (light detection and ranging) laser ranger. The system uses passive reflectors at programmed positions in the environment. The lidar unit, manufactured by Transitions Research, had originally been integrated onto the K2A for Savannah River Laboratory to support the MACS (Mobile Autonomous Characterization System) floor survey vehicle, and thus did not include significant additional development.

The lidar requires that two or more reflectors called FIDs be in range (approximately 300 feet) in order to correct the vehicle's position and heading estimates. The advantage to the lidar is that it gives the vehicle the ability to navigate in wide open spaces, out of range of sonar. It was found that such spaces offered the best areas for navigation to and from inspection jobs in areas like the Fernald tension structures.

The lidar navigation algorithm was improved by fully integrating it into the vehicle's new "Uncertainty Model". This integration allows the vehicle to obtain a more stable lock on its position as it traverses the aisles. During tests, the heading in the aisles was typically held to within a third of a degree of true.

It was also essential that the vehicle become completely "locked" to the FIDs before attempting to enter an aisle. To accomplish this, FID navigation was expanded to operate both while the vehicle is moving, and while it is standing still. A five second period of waiting before beginning to move into an aisle was found to provide excellent entry accuracy as part of a robust navigation strategy.

2.2.4 Navigation Integration

The K2A / K3A incorporate a technique called "Navigation" which refers to the ability of multiple navigation algorithms to execute concurrently. An arbitrator integrates the results from various agents such as sonar and the lidar, using their results in context with calculations from the vehicle's Uncertainty Modeler. The technique extends earlier fuzzy logic techniques to include a historical dimension. The technique is used in **ARIES** to permit navigation to transition seamlessly from lidar to sonar techniques. During the Phase 2 Demonstration, the vehicle used its Docking Beacon, Sonar, and Lidar navigation intermixed.

2.2.5 Multi-Threaded Extensions to the Path Language

The K2A Virtual Path Language is a custom high level language, with rich embedded navigation capabilities and limited math functions. The language, developed by Cybermotion, supports subroutines and branching, but at the beginning of the **ARIES** project these were restricted to a single program thread.

At that time, for branching to occur, any test instruction would have to be located at a place in the program where the vehicle had stopped, or if moving, was between path segments. This limitation was dictated because the Path language considered "RUN" an instruction, and thus had to finish its execution before it could execute a test. An annoyance in less demanding applications, this limitation was unacceptable in the **ARIES** mission.

Under this structure, the programmer had no way of influencing the vehicle's behavior during a RUN operation. For the **ARIES** system it was decided that the vehicle would require the ability to "Discover" drums along an aisle, determine their precise location, position the camera payload, and trigger the vision system to execute the inspection. This requirement dictated the ability to execute an "Inspection Thread" of code every time a drum was detected. The division of these tasks between other computers on board (as was done in Phase I and for MACS), usually results in loose control. A new construct was needed.

2.2.6 *DO* Instruction

The first element of the multi-threading requirement was provided by creating a new Path Language construct similar to the "Do While" that is so central to the C and C++ Languages. This construct is not, however, merely a program looping device, but instead specifies a block of code that will be executed one or more times during the next RUN if certain conditions are met, and it will be executed *concurrently* with the RUN.

A DO instruction can include a conditional test and take several forms. A DOWHEN>, for example, will cause the thread that follows it (called its Do Group) to be executed once when the specified test is true and some variable is greater than a constant. Likewise, a DOWHILE> will cause its Do Group thread to execute 10 times per second as long as the test is true or until the RUN ends.

The DO construct was also extended to allow the trigger condition to be one of having reached a point on the path. This instruction group took two forms, DO@ and DOSTOP@. The DO@ instruction causes its Do Group thread to execute when the vehicle reached a specified position on the path, without necessarily affecting the speed of the vehicle. The DOSTOP@ causes the vehicle to stop at a specified location and then execute the Do Group thread. When the thread is finished, the vehicle resumes the RUN.

The DO concept has utility that extends well beyond **ARIES**. The basic DO instructions were added to all versions of the K2A and K3A, including the SR-2 security robot used by the DOD MDARS program. The result has been a plethora of new robot "behaviors". Future versions of the MACS survey robot could benefit from these constructs as well.

The DO construct, and in particular the DOSTOP@ instruction, would serve as the foundation for the **ARIES** drum discovery algorithm.

2.2.7 Drum Discovery

It was decided, after the Fernald site visit, that end user acceptance of the system would almost definitely require the vehicle to adapt to the changing location of drums without operator intervention. This requirement dictated that the vehicle would need to be capable of "discovering" stacks of drums at any place along a specified aisle and of identifying and inspecting them autonomously.

To accomplish discovery, the Law-of-Cosines (LOC) imaging which had been developed during Phase 1 as a navigation algorithm, was resurrected and coded into the K3A mobile platform itself. Instead of serving as a navigation instruction by modifying the vehicle's position estimate, however, the detection algorithm was used to determine the position of a drum stack and plan the equivalent of a "DOSTOP@" to occur. The Do Group thread would contain the inspection control program.

2.2.7.1 DODRUM

To control the drum-detection algorithm, and to couple it to the behavior of the robot, a "DODRUM" instruction was added to the Path Language. The DODRUM instruction specifies the parameters for the LOC (Law of Cosines) imager, and is followed by the thread to be executed upon finding a drum. The thread is included only once per aisle (RUN), and will execute every time a drum stack is detected. As part of its behavior, the DODRUM not only stops the vehicle, but then backs it up slightly to position the cameras of the main positioner (for inspecting the top three drums of the stack).

The LOC imaging algorithm, although completely recoded for **ARIES**, was based on the findings of both Phase 1 of this project [Ref. 2] and a much earlier DOE SBIR project. During that SBIR project, it was concluded that sonar imaging could most effectively be employed if some assumptions could be made about the target, and if the problem could be constrained to the sensor plane (or some limited set of planes). The LOC Imager does just that, by assuming it is "looking for" drums of one of several sizes.

Since the diameters of drums fall into a fairly narrow set of values, the imager simply assumes each of these and evaluates the results. As a result, the LOC Imager returns the position of the center of the drum, its probable size, and a confidence factor. As the vehicle approaches the drum, the imager continues the process of imaging. If a solution is found with a better quality factor than the previous one, the position and size estimates are revised.

2.2.7.2 DO4BAR and Thread Queuing

The four-bar positoner [see Task 2.2.2 Camera Positioning System], used for inspecting the bottom drum of a stack, is attached to the top of the turret, and deployed behind the vehicle. Thus, the vehicle must move considerably down the aisle before the four-bar system is in position to inspect the drum stack previously observed by the CPS mast system. To specify the actions to be taken for the four-bar positioner, the instruction DO4BAR was added to the Path Language. This instruction is followed by the Do Group thread that is to execute when the four-bar system is in position. DO4BAR must be preceded by a DODRUM, and its stopping point and thread pointer are queued with the main positioner stopping point and its pointer whenever a drum stack is detected.

This process is complicated by the fact that the vehicle may detect a second stack of drums before it has executed the four-bar system thread for the previous stack. To accommodate this condition, a self sorting Stop-n'-Do Que. was included in the vehicle's control program. Up to eight threaded stops can be sorted and queued at any given time, and the queuing is automatic and invisible to the programmer.

All of this capability is accomplished in a very concise (if Heretical) extension of the Path Language programming structure which is easily programmed and understood. Even this level of effort on the part of the programmer is reduced by the Automatic Path Planner, developed by USC, which automatically incorporates the appropriate DODRUM and DO4BAR Do Group threads into the program for any inspection aisle.

2.3 TASK 2.2.2 CAMERA POSITIONING SYSTEM (CPS)

Conceptual design and analysis of the CPS was performed by the mechanical systems task team at USC with the assistance and advice of Cybermotion. Fabrication was then performed at Cybermotion, where the electronic controls and software were also developed. Testing and calibration of the CPS were provided by Cybermotion after the system was shipped to USC for integration of the bar-code readers, the vision system, and associated cabling and support.

The CPS consists of a main telescoping lift and a four-bar mechanism. The main lift is used to position cameras for inspection of the top three drums in a stack, and the four-bar mechanism is used to position a camera system and bar-code reader for the bottom drum of a stack. The CPS will be described and discussed in detail in Section 3 of this report.

2.3.1 Main Lift Controls

The control system for the CPS is based on a K2A / K3A motor control panel. This assembly contains two 24 volt / 30 Amp PWM (Pulse Width Modulated) motor amplifiers and a DC-1 Computer. This combination makes use of the same electronics as the base of the vehicle, thus eliminating the need to design a new panel.

The software for the main lift drives two linear actuators with potentiometer feedback. The driver software is a swept-gain modified PID control executing at 40 Hz. The primitive motor driver is shelled to a the PID controller, which is in turn shelled to a master control. This structure allows control to be exerted at any level by writing to blackboard variables over the serial link.

The modified PID uses conventional Proportional and Derivative gain term combined with an Input (instead of Integral) gain term that feeds forward the velocity of the commanded position "rabbit". Acceleration and deceleration of the rabbit are controlled by a traditional trapezoidal velocity profile. As the servo controller reaches its end position, there tends to be a significant proportional droop which can cause a loss of closing accuracy. If the P gain is increased to reduce the droop, the system goes unstable during high-speed movements.

The closing error was greatly reduced by sweeping (continuously increasing) the gain of the amplifier during the final phase of a stroke. This technique improved the position repeatability from about 1.25 inches to less than .25 inches.

Many safety systems were incorporated in the CPS. The maximum torque output of the motors is limited electrically by two "Torque Limit" potentiometers. The amplifier and computer both contain an interlock to allow each to perform a safety stop if the other fails a confidence test. Each channel of the amplifier contains a software current (torque) limiter that will slow the lift if an over current condition is detected. If slowing does not eliminate the over current condition in approximately 1 second, a "Stall Kickout" shutdown will occur.

Finally, it is possible that in transition between legal states, the two positioners of the lift might reach a combination that is not safe (i.e.: threatens to pull a bearing off a runner, etc.). Thus a watchdog program monitors the position of the two actuators to assure that their combination is not nearing a condition that the lift can not physically accommodate. If the system approaches the edge of the safety envelope, the appropriate positioner is slowed or stopped until the other positioner has moved enough that the first positioner can safely resume its motion.

2.3.2 Four-Bar Positioner System

The four-bar positioner is much simpler that the main positioner. This was made possible because of the far smaller forces required. The drive motor for the four-bar is coupled to the mechanism through a gear reducer. The working voltage of the motor was chosen higher than 24 volts, so that the internal resistance of the motor would permit it to be safely stalled for several minutes at 24 volts. The controls consist of a simple 2 channel optical encoder and the controlling computer (the same computer that controls the other actuators) uses a "bang-bang" control method. Accuracy is improved by a predictor that anticipates if the target position will be crossed in the next time "tick", and thus issues a preemptive stop command.

All actuators, including the two main positioners and the four-bar positioner, contain safety limit switches at their extremes of motion and software limits at their normal limits of motion. Stall timers are implemented on all channels as well, and the control computer can remove all power from the actuators. The lift can be also be halted by any of four manual E-Stop (Emergency-Stop) buttons or by an internal E-Stop circuit. Any computer on the vehicle is capable of interrupting the E-Stop circuit if it senses a dangerous condition.

2.3.3 CPS Communication and Control

The CPS Computer continuously monitors the condition of the actuators and places their Status in globally-accessible blackboard memory. States of the individual actuators are combined to provide a Status for the CPS as a whole. For example, if any actuator is BUSY, then the CPS is BUSY, unless there is an over-riding fault Status. Only if all actuators are fully retracted (Stowed) does the CPS report its Status as STOWED. Status values for the CPS include; STOWED, BUSY, STALLED, etc.

The functioning of the CPS is controlled by writing to control variables in the blackboard memory. For a move operation, the controlling computer (or operator) simply writes the CPS Mode to "AUTO" and specifies the size and end of the drum to be viewed by the main and/or four-bar positioners. The CPS will then return a Status of BUSY until the operation has been completed. Both the vehicle control computer and the supervisory systems have access to all levels of control.

Since the K3A Mobile Platform can read and write the CPS Blackboard, it can operate the CPS in response to instructions in the Path Program it is executing. Such interaction may take the form of moving the positioner for an inspection, or simply checking that it is stowed before driving under a low building structure. Because the blackboard contains entries at almost all levels of shelled control, the range of possible control is highly flexible.

2.4 REFERENCES

1. D. Fisher, J.M. Holland, and K.F. Kennedy, "K3A Marks Third Generation SynchroDrive," *Proc. of ANS Robotics and Remote Systems,* American Nuclear Society, Chicago, IL (1994).

2. J. S. Byrd and K.H. Hill, "An Application of Mobile Robot Localization Using Sonar," *Proc. of ANS Robotics and Remote Systems,* American Nuclear Society, Chicago, IL (1993).

3. MECHANICAL SYSTEMS

3.1 INTRODUCTION

The Mechanical Deployment Systems work task WBS 2.4 is made up of three components: i) Vision System; ii) Prototype Inspection System; and iii) Camera Positioning System. The Camera Positioning System (CPS) is the subject of this section of the report. Other components are reported in appropriate sections. The main purpose of work task WBS 2.4 is to design a mechanical CPS capable of compact storage aboard the Cybermotion K3A mobile robot. The CPS must be deployed during vehicle movement and must be able to accurately position the vision payload during its survey mission.

The CPS is composed of four identical but separate camera packages for inspecting each drum in a column of four drums high. The CPS retracts to a more compact position when traveling in the open warehouse to effectively center and lower the mass of the system directly over the K3A mobile platform. Inside the aisles, a lift mechanism extends upward for inspecting the top three drums of a column and a parallelogram four-bar positioner folds out behind the K3A to inspect the bottom drum on the floor it has just passed. The lift mechanism consists of five-interlocking rail-elements connected by a series of cables and pulleys and uses two linear-servo actuators to achieve the desired motion. The parallelogram four-bar mechanism restricts the coupler (where the camera package is attached) to pure transitional motion and position is controlled with a worm-gear driven servomotor.

3.2 CONSTRAINTS

The constraints on the CPS are:

1) Allow the K3A mobile platform to traverse and turn in a 36-inch aisle.
2) Actively position cameras to inspect 55-, 85-, and 110-gallon drums stacked up to four high; required range of cameras up to 16 feet.
3) Perform inspection in a timely fashion to effect a high throughput of drums inspected.
4) Be capable of lowering the C.G. for negotiating spillway berms with minimum 9% grades, and to possibly traverse around the end of an aisle of drums.
5) Minimize movement to reduce power demands on batteries.

3.3 PAYLOAD HARDWARE COMPONENTS

To perform the inspection tasks the CPS carries and positions four camera packages. Each camera package consists of: one CCD camera, two photographic flash lamp units with separate battery units, two ambient-light sensors, one laser-stripe projector, and one bar-code scanner. Total weight of each camera package is approximately 30 pounds. Brackets and fixtures needed for mounting the payload components add additional weight to the overall system. Total weight of the CPS is approximately 300 pounds.

3.4 DESIGN PROCESS

The first step of the mechanical design process involved locating an existing commercially available system that required minimal retrofitting. When that search failed, off the shelf commercial components were sought. When that search failed, the last resort was pursued, which was to develop a new mechanical design. Some of the commercial products and ideas first evaluated were: industrial work platforms used for positioning people (retrofitted for this application); flexible metal bi-stem tubes used for TV camera positioning; square-nested telescoping pneumatic and hydraulic box-stem tubes; fixed mast with moving track or continuous lead screw; and a Puma 560 manipulator. None of these solutions were viable so a new design was pursued which most closely resembles the industrial work platforms available from a variety of vendors[1].

The concept behind an industrial work platform is to interlock several rail-elements, fix one rail-element to ground, and use a single hydraulic ram to move a chain-sprocket or cable-pulley system to position the outer most element where the worker stands. This concept would have been acceptable for the application at hand if just one camera package were needed to be positioned, that being the camera package on the outer-most element. Yet, what is required is to position three separate packages, simultaneously, to different heights. It was decided to take advantage of the fixed heights and spacing of the drums to position the camera packages simultaneously; also it was decided to take advantage of the positioning of not only the outside element but of two of the inside elements. This means that the lift needs two degrees-of-freedom. One degree-of-freedom is used to adjust the total height of the camera packages, and the other freedom is used to adjust the spread distance between packages. Through this arrangement, two actuators are used to position all three camera packages simultaneously at a chosen fractional height on each drum regardless of the column of drums being inspected. That is, if it is desired to be one-third the way up on all the top three 55-gallon drums simultaneously, it can be done; as well as the next move to be four-fifths (or any fraction) of the way up on all the top three 110-gallon drums simultaneously. Note the gross height and delta spread distances are different, yet can be achieved by the lift mechanism.

3.5 LIFT MECHANISM

Controlling a single output given two degrees of freedom yields an equation very similar to the equation of a plane; therefore, an analytical geometry analogy using planes and lines is made for developing the equations of motion.

The homogeneous coordinates of a plane are ($\underline{\mathbf{S}}$; s_0) which satisfies the equation

$$\underline{\mathbf{r}} \bullet \underline{\mathbf{S}} = s_0 \qquad (1)$$

where $\underline{\mathbf{r}}$ (x, y, z) is a position vector to the plane, $\underline{\mathbf{S}}$ (L, M, N) is the normal to the plane and the scalar s_0 is origin dependent. The line equation can be expressed in the form

$$\underline{\mathbf{r}} \times \underline{\mathbf{S}} = \underline{\mathbf{S}}_0 \qquad (2)$$

[1] Genie Industries, Redmond WA is one such vendor.

where the two vectors ($\underline{\mathbf{S}}$; $\underline{\mathbf{S}}_0$) are the Plücker coordinates of the line. Here $\underline{\mathbf{S}}$ (L, M, N) defines the direction of the line and $\underline{\mathbf{S}}_0$ (P, Q, R) is the moment of the line about the origin.

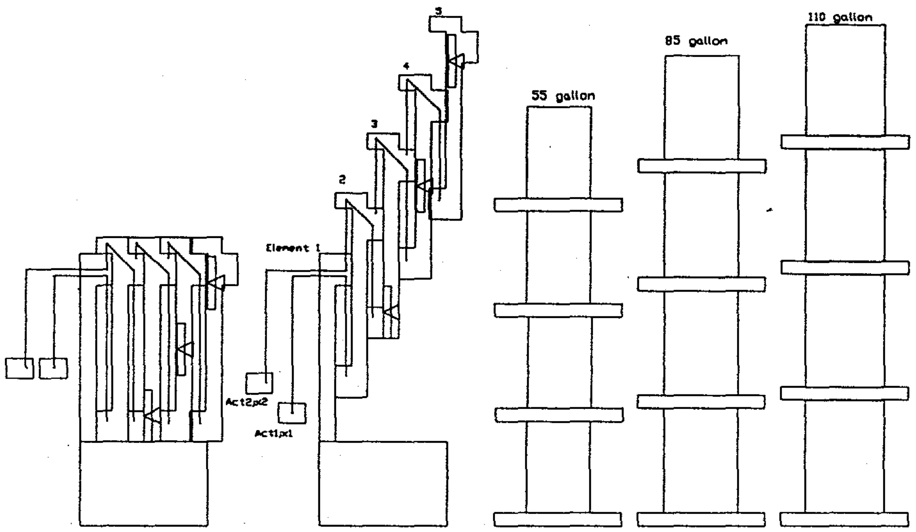


Figure 3-1 - Lift Mechanism Conceptual Design

The three camera packages (shown as vertical rectangles with triangles in Figure 3-1) for evaluating the top three drums are attached to elements i=3, 4, and 5 respectively. The goal is to move the elements so they move up with the same ratios to allow the camera packages to be placed at the same fraction of the way up on each drum simultaneously. The equations of motion for each element are a function of its initial height, C_i and the actuation's x_1 and x_2.

$$z_i = A_i x_1 + B_i x_2 + C_i \text{ for } i=3,4,5 \qquad (3)$$

Note that this is the equation of a plane. To achieve the same motion ratio upward the plane equations for each element have to intersect at the same line. The equation of the line and its associated Plücker coordinates where two planes intersect is found from (1) and (2) to be

$$\underline{\mathbf{r}} \times (\underline{\mathbf{S}}_1 \times \underline{\mathbf{S}}_2) = s_2 \underline{\mathbf{S}}_1 - s_1 \underline{\mathbf{S}}_2 \qquad (4a)$$

$$(\underline{\mathbf{S}}_1 \times \underline{\mathbf{S}}_2 \,;\, s_2 \underline{\mathbf{S}}_1 - s_1 \underline{\mathbf{S}}_2) \qquad (4b)$$

Using (3) with $\underline{\mathbf{r}} = (x_1, x_2, -z)$ the homogenous coordinates of the planes can be written

$$(A_i, B_i, 1 \,;\, -C_i) \qquad \text{for } i=3,4,5 \qquad (5)$$

From (4b) the Plücker line coordinates where 3 and 4 intersect must be the same line as where 4 and 5 intersect in order to maintain the desired height ratios on the drums. The intersections of 3-4 and 4-5 are

$$(B_3 - B_4, A_4 - A_3, A_3B_4 - A_4B_3 \,;\, A_4C_3 - A_3C_4, B_4C_3 - B_3C_4, C_3 - C_4) \qquad (6a)$$

$$(B_4 - B_5, A_5 - A_4, A_4B_5 - A_5B_4 ; A_5C_4 - A_4C_5, B_5C_4 - B_4C_5, C_4 - C_5) \qquad (6b)$$

If these lines are to be the same then

$$A_3 = 2A_4 - A_5 \qquad B_3 = 2B_4 - B_5 \qquad C_3 = 2C_4 - C_5 \qquad (7)$$

There is not a unique solution to the system of equations in (6), so a reasonable solution was found where a particular cable-pulley arrangement could be utilized to satisfy (7). Figure 3-1 shows one such arrangement and using (7) it is found that $A_3 = 2$, $A_4 = 3$, $A_5 = 4$, $B_3 = 1$, $B_4 = 2$, $B_5 = 3$; and, the initial at rest positions of the camera packages, C_3, C_4, and C_5, need to maintain similar ratios. Equation (3) can now be written for the cable-pulley arrangement shown in Figure 3-1.

$$z_3 = 2x_1 + x_2 + C_3 \qquad (8a)$$
$$z_4 = 3x_1 + 2x_2 + C_4 \qquad (8b)$$
$$z_5 = 4x_1 + 3x_2 + C_5 \qquad (8c)$$

The constants C_3, C_4, and C_5 are adjusted to meet the packaging constraints of the lift on top of the K3A robot and the maximum available range of the linear actuators x_1 and x_2. At rest, C_3 determines the height off the top of the K3A base, and C_5 determines the overall height of the lift. That is, for the overall height of the lift to be reduced, C_5 needs to be reduced, and the stroke length of the linear actuators x_1 and x_2 need to be increased. Due to the camera packages needing to be aligned vertically—with each camera package occupying approximately 30" in height—the values of C_i are chosen to be C_3 = 43", C_4 = 72.33", and C_5 = 101.67". Note that these values of C_i satisfy (7) and represent the floor to camera package distance when the lift is retracted. The height of the K3A is 32"; therefore, the overall height of the lift is (102" - 32") = 70", which is about six feet tall using these C_i values.

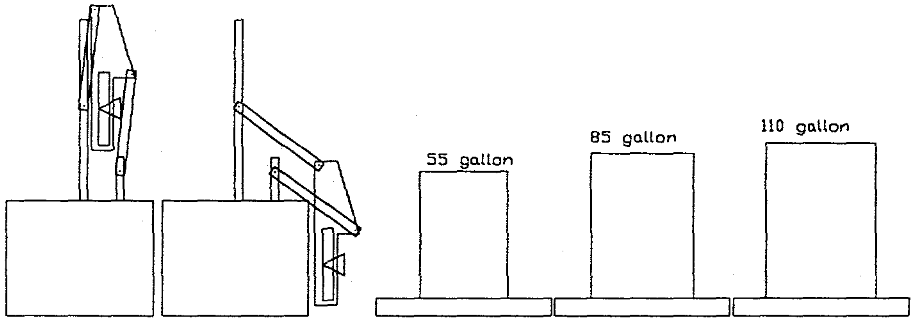


Figure 3-2 - Four-bar Conceptual Design

3.6 FOUR-BAR DESIGN

To inspect the drum on the floor a mechanism was needed to deploy a camera package off the back of the K3A. The first mechanism studied was a Watts six-bar. Due to the complexity of the six-bar, a special case Grashof parallelogram mechanism seemed to be more suitable. A unique aspect of the parallelogram four-bar is its ability to keep its camera package vertical at all times due to the pure translational motion of the coupler where the camera package is attached (Figure 3-2). The motion equations for the four-bar positioner are straight forward and are found in any elementary kinematics text. The constraints on the four-bar positioner are: 1) it fully retracts its camera package over the K3A; 2) it deploys out far enough for the camera field of view to not be obstructed by the sides of the K3A; and 3) the lateral displacement of the four-bar should be minimized between its lowest and highest camera heights. The reason for this last constraint is the desire for all camera positions on a particular drum to be along, or as close as possible to, a vertical line on that drum. The last constraint is accomplished by making sure the crank angle equals 90° halfway between the lowest and highest camera heights. Figure 3-2 shows the conceptual design of the four-bar.

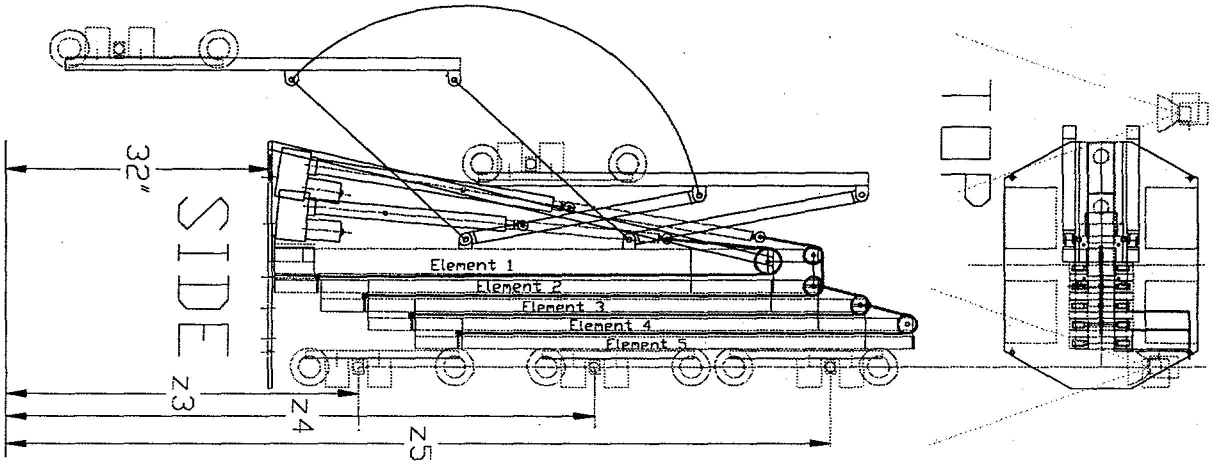


Figure 3-3 - CPS Working Drawings

3.7 LIFT FABRICATION

From the conceptual design of the CPS a final design was developed. The final design, and the associated parts from the supplier were delivered to Cybermotion, Inc. for fabrication. Figure 3-3 shows the top and side views of the working drawing for the fabricated mechanical lift. Due to packaging constraints encountered during the fabrication, the linear actuators were assembled differently than what was called for in the original design specifications. This can been seen by comparing the side view of Figure 3-3 with Figure 3-1, noting that in Figure 3-3 the cable-pulley arrangements are different than the conceptual design for the lift shown in Figure 3-1. The most obvious difference is the cable associated with actuator 2 no longer maintains the 90° relationship with the top of element 2. In fact, as element 2 travels up, the angle of the cable goes from 0° to nearly 90°. This assembly modification changes the relative motion of elements 1 and 2; therefore, new equations of motion are derived to reflect these changes.

Figure 3-4 (bottom) shows a single pulley arrangement with the cable on either side of the pulley at an α_1 and α_2 angle of less than 90°. This single pulley arrangement can be seen on the top of elements 1, 3, and 4 in Figure 3-3. As L_1 (the length of the cable on the left side of the pulley) increases when the coordinates (x_1, y_1) move down and to the left, L_2 (the length of the cable on the right side of the pulley) will be equally foreshortened as the coordinates (x_2, y_2) move straight up. The question is: when (x_1, y_1) change, what are the new coordinates of (x_2, y_2)? The answer is found by: 1) recognizing the total length, L of the cable stays constant; 2) θ_1, the angle of departure of the cable on the left side changes with the position of (x_1, y_1); 3) θ_2, the angle of departure of the cable on the right side changes with the position of (x_2, y_2); and 4) the x_2 coordinate is constant as y_2 travels straight up.

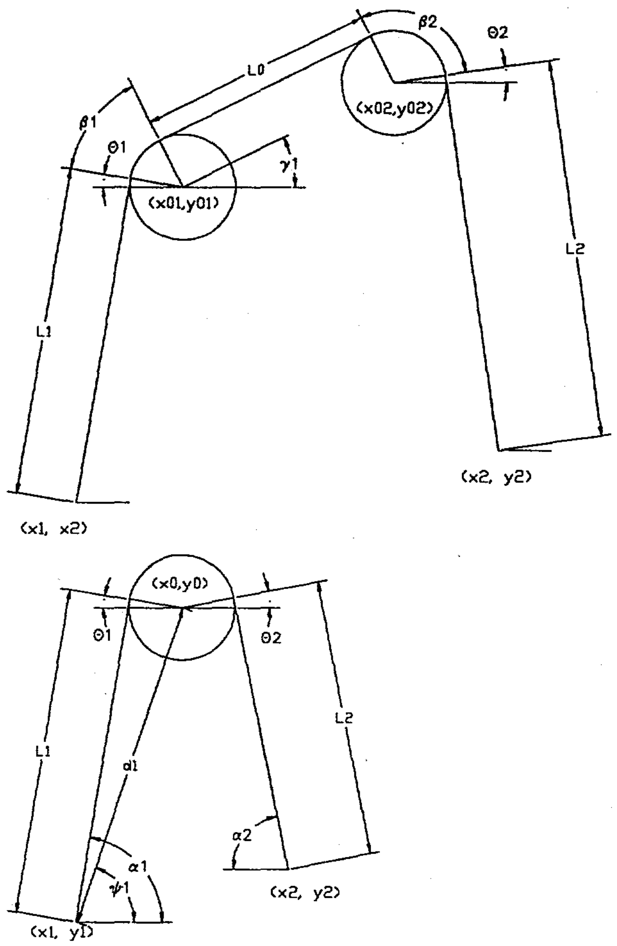


Figure 3-4 - Double Pulley (top) and Single Pulley (bottom) Arrangement

Using the above four insights, the givens are: Total cable length L, radius r of the pulley, the center of the pulley (x_0, y_0), the new (x_1, y_1) coordinates, and the constant x_2 coordinate.

Find: Three equations, 3 unknowns, solving for L_2, θ_2, and y_2 simultaneously.

$$\psi_1 = \tan^{-1} (y_0 - y_1)/(x_0 - x_1) \qquad (9a)$$

$$d_1 = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2} \qquad (9b)$$

$$\theta_1 = \cos^{-1} (r / d_1) - \psi_1 \qquad (9c)$$

$$L_1 = \sqrt{d_1^2 + r^2} \qquad (9d)$$

$$\alpha_1 = \pi/2 - \theta_1 \qquad (9e)$$

Solve for the following 3 unknowns L_2, θ_2, and y_2

$$r \cos\theta_2 + L_2 \sin\theta_2 + (x_0 - x_2) = 0 \qquad (10a)$$

$r \sin\theta_2 - L_2 \cos\theta_2 + (y_0 - y_2) = 0$ (10b)

$L - L_1 - L_2 - r(\pi - \theta_1 - \theta_2) = 0$ (10c)

An analogous approach is used for the two pulley arrangement. In Figure 3-4 (top) the double pulley arrangement (seen on the top of elements 1 and 2 in Figure 3-3) is shown with an additional variable, L_0 which is the distance between the centers of the two pulleys.

In this case the following is given: Total cable length L, radii $r = r_1 = r_2$ of the pulleys; centers (x_{01}, y_{01}) and (x_{02}, y_{02}) of the pulleys, the new (x_1, y_1) coordinates, and the constant x_2 coordinate.
Find: First solve for L_0, β_1, and β_2; then find three equations, 3 unknowns, solving L_2, θ_2, and y_2.

$L_0 = \sqrt{(x_{02} - x_{01})^2 + (y_{02} - y_{01})^2}$ (11a)

$\gamma_1 = \tan^{-1}(y_{02} - y_{01} / x_{02} - x_{01})$ (11b)

$\beta_1 = \pi/2 - \theta_1 - \gamma_1$ (11c)

$\beta_2 = \pi/2 - \theta_2 + \gamma_1$ (11d)

Solve for the following 3 unknowns L_2, θ_2, and y_2

$r \cos\theta_2 + L_2 \sin\theta_2 + (x_{02} - x_2) = 0$ (12a)

$r \sin\theta_2 - L_2 \cos\theta_2 + (y_{02} - y_2) = 0$ (12b)

$L - L_0 - L_1 - L_2 - r(\beta_1 - \beta_2) = 0$ (12c)

Once the above equations are found, the initial coordinate values of every pulley and cable connection are determined along with the corresponding cable lengths. Again note—the overall cable lengths remain constant. The solution of how every element moves proceeds from element 2 to element 5 is obtained in the following manner. Equations (9) and (10) are used to find the new position of element 2. Equations (11) and (12) are used to find the new position of element 3. Equations (9) and (10) are again used to find the positions of element 4 and then element 5. The result is non-linear but it can be approximated with linear equations similar to (8) based on the actuated height of the lift.

For small actuations used for the inspection of *55-gallon* drums the following set of equations is used as an approximation for the positions of the camera packages:

$z_3 = 1.829x_1 + 0.987x_2 + C_3$ (13a)

$z_4 = 2.646x_1 + 1.958x_2 + C_4$ (13b)

$z_5 = 3.45x_1 + 2.913x_2 + C_5$ (13c)

For *85-gallon* drums the set of equations is:

$z_3 = 1.864x_1 + 0.952x_2 + C_3$ (14a)

$z_4 = 2.720x_1 + 1.894x_2 + C_4$ (14b)

$z_5 = 3.565x_1 + 2.825x_2 + C_5$ (14c)

For *110-gallon* drums:

$$z_3 = 1.881x_1 + 0.947x_2 + C_3 \qquad (15a)$$

$$z_4 = 2.755x_1 + 1.886x_2 + C_4 \qquad (15b)$$

$$z_5 = 3.619x_1 + 2.815x_2 + C_5 \qquad (15c)$$

In (13)-(15) $C_3 = 46.366''$, $C_4 = 72.716''$, and $C_5 = 101.066''$. As can be seen with these new sets of equations, the relationship put forth in (7) does not hold exactly, but in tests of the mechanical lift the positioning accuracy of each camera package is ±0.5". This level of accuracy is more than adequate for the positioning of the camera packages.

3.8 DISCUSSION

Several design issues were addressed during the design process and those issues will now be briefly discussed.

3.8.1 Power Consumption

One of the main concerns of the CPS is for it to be energy efficient. To estimate power demands of the CPS a force analysis was performed on the actuators. The approximate weights of the moving elements 2, 3, 4, and 5 are: 52 lb., 33 lb., 33 lb., and 25 lb. respectively. From a static analysis, the tensions on each cable in Figure 3-1 (starting from the outside cable attached to the bottom of element 5 and working towards the cable attached to the bottom of element 3) are

T_5 =25 lb.

T_4 =82 lb.

T_3 =173 lb. (actuator 2 force)

T_2 =316 lb. (actuator 1 force)

With these measurements the power and positioning time needed to inspect columns of drums is as follows (the measured average velocities of actuator 1 and 2 are 1.3 in/sec and 1.7 in/sec respectively).

55	3kW-sec, 25 secs
85	3.8 kW-sec, 31 secs
110	4.3 kW-sec, 35 secs

The four-bar requires less than 0.3 kW-sec, 10 secs avg.

3.8.2 Stability

The obvious question when looking at the lift mechanism is its dynamic stability due to a relatively high center-of-gravity of the mast on a moving platform. The C.G. of the lift is 295 lb. located 6" forward of the center of the K3A base, and 75" from the ground. The 450 lb. center of gravity for the K3A is 12" off the ground. The maximum commanded acceleration of the K3A is 0.65 ft/sec^2 and the maximum commanded deceleration is -1.3 ft/sec^2. A dynamic analysis was performed using these accelerations and decelerations, and it was found that the stability of the entire system is very safe with these acceleration magnitudes. What is a concern is a sudden deceleration brought about by hitting a large crack in the floor or a fixed obstacle on the floor. If this occurs the system will be unstable and will tip over. To prevent this the operating environment's floor must be clear of obstacles of this type.

3.8.3 Bearing Load

For the bearing load analysis the supplier[2] of the aluminum extruded components, linear bearings, and shafts expressed concern over the maximum torque applied to the bearings due to the apparent cantilevered load. The design of the lift system purposely eliminates the lateral forces on the bearings and thus, the associated torques, by balancing the load over the top pulleys of each element. In fact, the maximum lateral load applied to a linear bearing is 11.5 lb. which occurs on the two lowest bearing units between the elements 1 and 2.

3.8.4 Testing

The linear actuators are ball-screw actuators, therefore they can be back-driven. Brakes are integrated into the actuators to prevent this. Tests show that upon sudden loss of power the brakes do indeed close and hold the lift in a fixed position. Another unanticipated test was the "drop test." During the test phase the lift was traveling up and before *safety* equations were coded into the controls software, the actuators overextended the lift causing bearings to collide and cable-pulley connectors to fail. The lift came down from its fully extended position with only minor damage occurring—in fact the lift was back repaired operating the same afternoon. To prevent this type of occurrence again, the actuators now have current limits on the control circuits such that when an obstacle is encountered and the force required by the actuators increases beyond the set current limits, the entire system shuts itself down.

3.9 CONCLUSION

The CPS on the vehicle, in its Phase 2 demonstration state, is shown in Figures 3-5 and 3-6. The stowed height of the CPS is 10 feet. The Maximum extended height is 16 feet. The entire CPS weighs approximately 300 lb.

The most immediate change proposed for Phase 3 will be the use of smaller aluminum profile components and smaller actuators. This will reduce the height and weight of the CPS to make it more stable, to allow it to operate faster, to reduce power consumption, and to reduce overall cost.

During Phase 2 the four-bar mechanism, although functionally sound, was not dynamically stable enough to carry all the weight of its vision payload. Therefore, during Phase 3 it is proposed to replace the four-bar positioner with an alternative mechanism that will be used to inspect the drums at the floor level. A conceptual idea for this mechanism is a mechanism that slides on the braces that support the back of the mast system. The mechanism would be a slotted or trolley-like device that is actuated with its own linear actuator and would slide on the braces.

[2] Item Industries, Houston TX

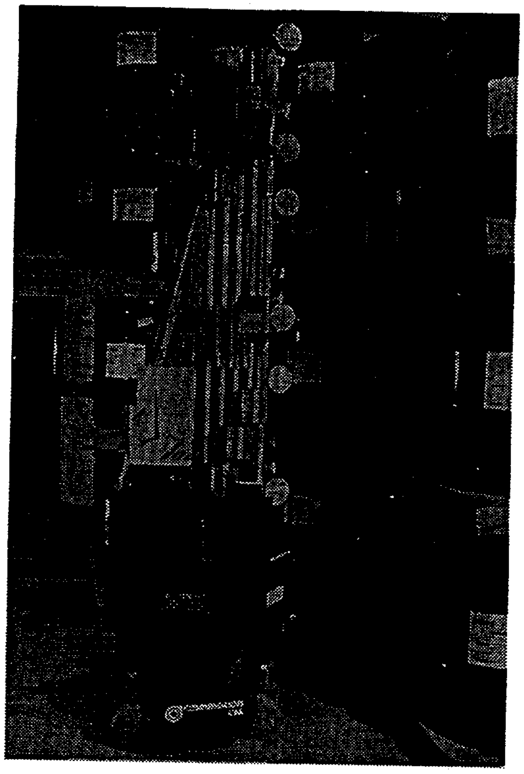


Figure 3-5 CPS in Stowed Position

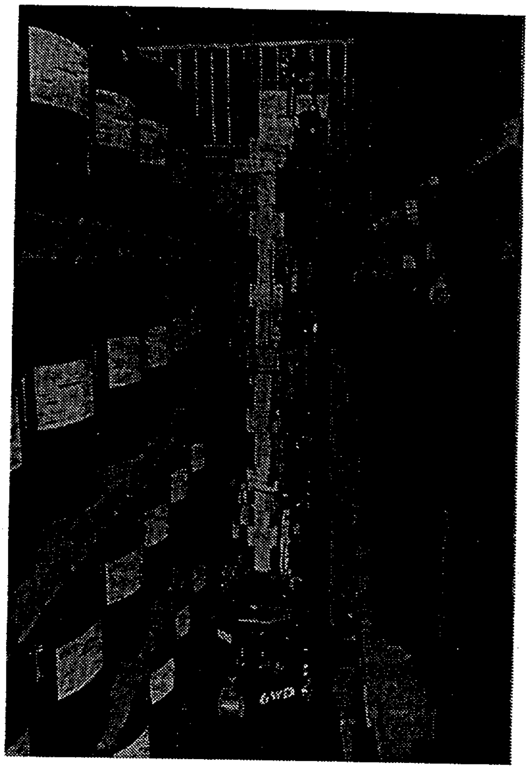


Figure 3-6 CPS in Extended Position

4. COMPUTERS AND CONTROL TASK

4.1 INTRODUCTION

The Phase 2 Computers & Controls requirement (WBS 2.3) was to

> "Deliver a *control environment system architecture* that includes the overall organization and major components of the system, their communications protocols, the *user environment* (interface, program development tools, etc.), and *the system mission structure*. The system's *applications payload* packages shall be integrated into this architecture."

This requirement was broken into six major tasks (WBS 2.3), which were:

1. Design and develop a *control environment architecture* capable of integrating all subunit protocols and programming.
2. Implement appropriate programs from Phase 1 on Phase 2 computers including the vehicle management system with onboard vehicle computer interface; operator interfaces for monitoring the vehicle, for training the vision algorithms, and for teleoperation of the dexterity capability; and the vision acquisition and processing programs.
3. Design and implement a control program that creates a database (containing drum information, etc.) and which is capable of supporting the applications payload(s).
4. Analyze all electrical and computer systems in the robot vehicle for power consumption.
5. Analyze power needs of the mobile vehicle and the applications payload(s).
6. Develop a system power management protocol and control system for most efficient power use in operations of the robotic system.

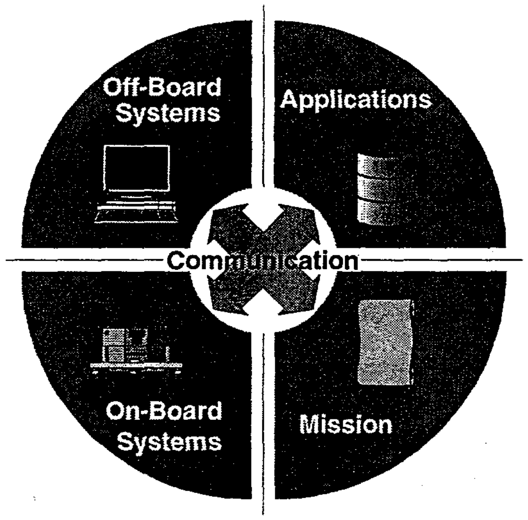


Figure 4-1. Computers and controls components

The primary computers and controls components, shown in figure 4-1, are *(i)* the mission, *(ii)* the off-board systems, *(iii)* the on-board systems, *(iv)* the applications, and *(v)* the communications system.

4.2 THE MISSION

The *mission* represents the system description of the tasks to be performed by the robot. The mission is created by the Site Manager (mission *planning*), down-loaded to the robot, (mission *assignment*), and executed by the on-board Mission Controller. The Site Manager may be used off-line to plan and create a mission or on-line to control and/or monitor the operation of the robot. It is the users primary access to the system at the task level. The mission controller carries out the mission in an autonomous fashion.

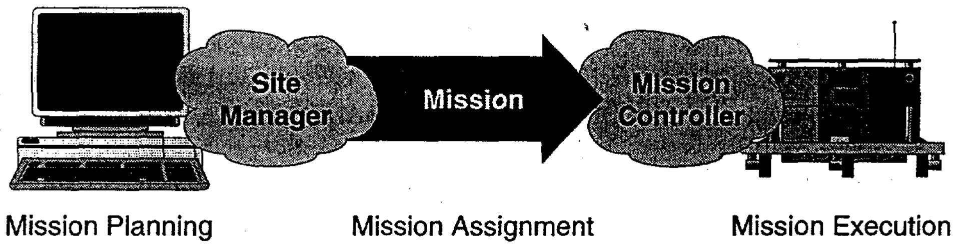


Figure 4-2. Roles of the Site Manager, Mission, and Mission Controller

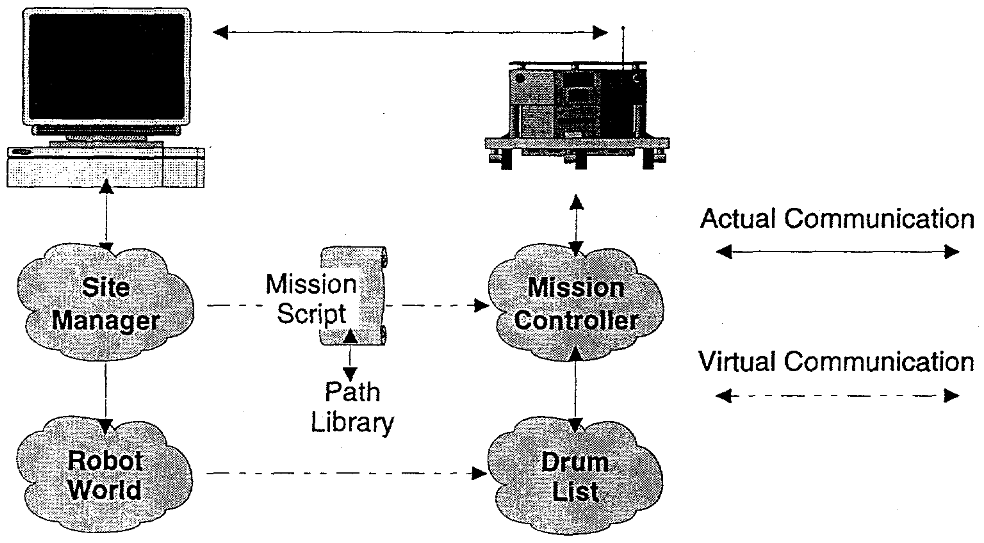


Figure 4-3. More detailed view of mission planning and assignment

Figure 4-3 gives a more detailed view of the mission planning and assignment process. The planning part, which is done off-line, takes place off-board, using the Site Manager. Mission planning depends on the use of information stored in both the Robot World and the Drum Database. Mission assignment is the process of down-loading the mission information to the on-board system. While all actual communication is done by the communications software, it is informative to view this as a peer level system in which the various off-board components communicate directly with their on-board counter parts.

4.2.1 Mission Components

Mission Script
Path Library
Drum List

Figure 4-4. Mission components

The mission consists of *(i)* the mission script, a high-level description of the mission used by the on-board Mission Controller, *(ii)* the Path Library, a set of path programs that may be used to travel between the nodes included in the survey area, and *(iii)* a local copy of the drum list (see Figure 4-4). It provides the Mission Controller with the following information:

- *Definition of the survey area.* The mission specifies the portions of the warehouse that are to be inspected by the robot.
- *Navigation information.*
- *Inspection procedures.* This information defines such items as the order in which the drums are inspected or
- *Exception procedures.* The exception procedures are used to handle unusual circumstances, such as a leaking drum, an obstruction in an aisle, or low battery.

The language used in the mission script is Mission Definition Language (MDL). MDL differs from conventional languages in that it contains both procedural and non-procedural parts. The mission and MDL will be defined in a later section.

4.2.2 Site Manager

The Site Manager uses the information contained in the Robot World, together with user input, to create the mission. It is invoked from the main control program. The primary functions of the Site Manager are to *(i)* provide a high-level programming interface to the system, *(ii)* support general queries concerning the state of the warehouse, including the visual inspection tour, and *(iii)* provide for control of the report generator. Programming a mission for a robot is largely visual with defaults supplied for many of the parameters. The Site Manager parallels the Cybermotion Dispatcher to some degree but is more application specific. It is invoked from the main control menu. Although Site Manager is not used for direct control of the robot, some of its functions are real-time in that they may interact directly with the robot during the inspection survey. The mission planning function, however, is primarily an off-line function.

4.2.3 Mission Controller

The Mission Controller is the on-board program which is charged with the task of carrying out the mission. It is a peer of the Site Manager (see Figure 4- 2) in that it receives the mission. However, it is primarily a real-time program since it directly controls the robot during the survey. The Mission Controller executes the procedural part of the MDL mission script, using the information contained in the non-procedural part.

4.2.4 The Robot World

The Robot World is an abstraction, or model, of the robot's environment. It is used to provide information to both the user and the robot. It consists of a 3-D AutoCAD description of the environment and a set of operations which may be performed on this description. It provides the information to construct both 2-D and 3-D views of the warehouse. The 2-D displays are typically used for planning purposes while the 3-D displays provide a more realistic representation. Both the Display Program and the Site Manager use the information from the Robot World to create operator displays. Figure 4-6 gives an example of the type of 3-D animation that can be constructed from the information contained in the Robot World.

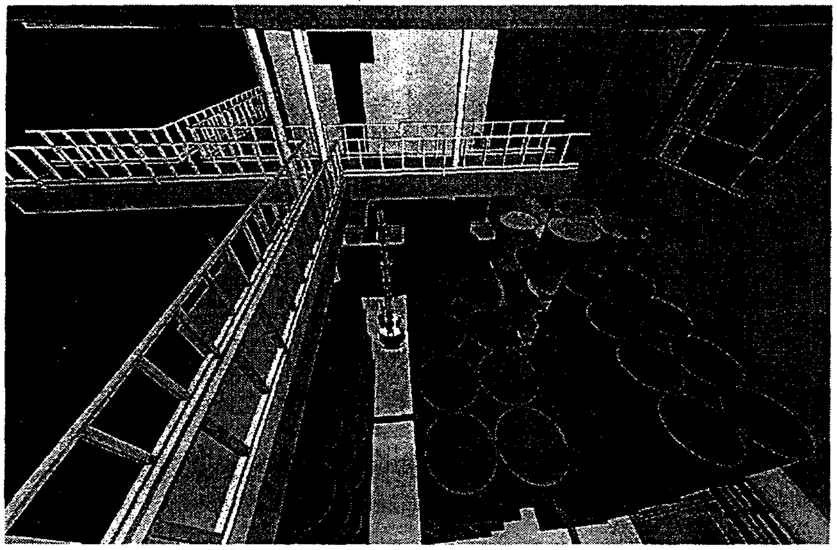


Figure 4-6. Operator view provided by Site Manager

The Robot World also provides the capability for applications to add application-specific information, which may be shared with other programs if desired. In this case, access to the information is provided by the application. The information may be restricted to the specific application or made public to other applications. The Site Manager is an example of an application that maintains information in the Robot World.

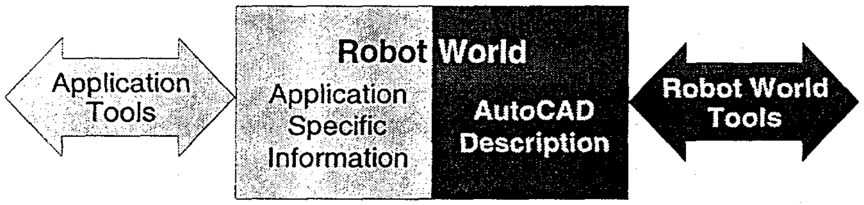


Figure 4-7. Robot World organization

4.2.5 The Drum List and Navigation Information

While AutoCAD is used to represent the warehouse, it is less suitable for the drums due to their dynamic nature. The information required to define the drums is kept in the drum list, a part of the Robot World that is managed by the Site Manager. The drum lists (see Figure 4-8) are linked lists which contain drum information needed for navigation (drum location, size) as well as specific information on each drum. The drum lists are organized by the bottom (or reference) drum since the floor projection (2-D location) is of primary interest. Each reference drum entry also contains entries for all other drums stacked on it. The specific drum information is obtained from the Drum Database.

Since this structure is to be used for navigation and to provide 2- and 3-D animations of the warehouse, it does not need to contain all of the information from the drum database. For example, the picture information contained in the drum database is used for image analysis and can not be compressed, while the picture information in the Robot World is only used for human viewing. Therefore, it is more efficient to create a new, smaller drum entry rather than reference the actual drum database entry. This also reduces the memory required, an important factor if the information is to be used on board the robot.

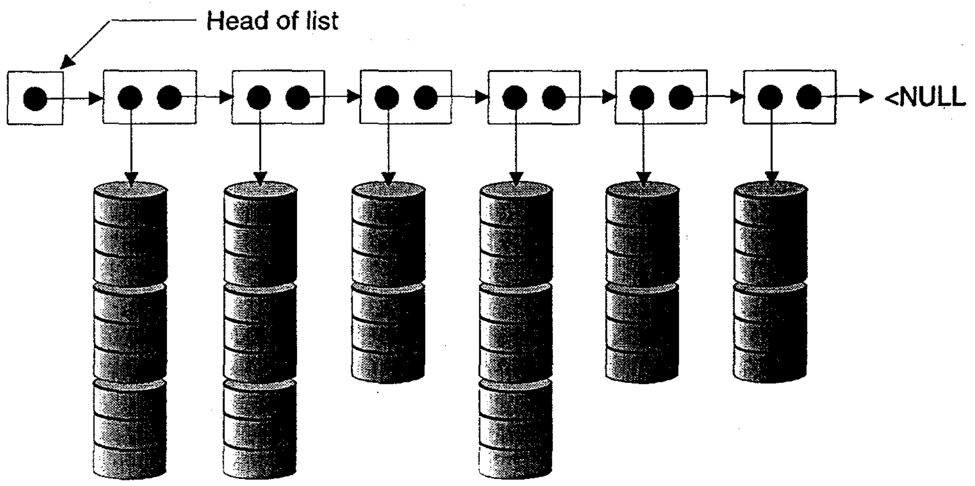


Figure 4-8. Example drum list

The Site Manager also maintains navigation information in the Robot World in the form of the LIDAR beacon locations. It may maintain the docking beacon information although this also can be done by the Dispatcher.

4.2.6 Mission Planning Using the Site Manager

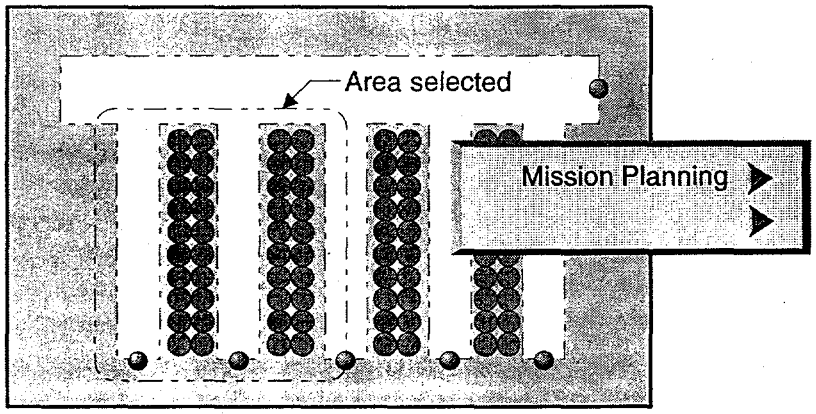


Figure 4-9. Selection of survey area for the mission

Mission planning begins with the selection of the area to be surveyed[1]. The Site Manager, using the 3-D AutoCAD description from the Robot World, provides a 2-D floor plan. The

[1] If more than one robot is available, then one robot will also have to be selected.

survey area is selected by enclosing the desired area on the 2-D layout of the warehouse, as shown in Figure 4-9. The Mission Planner then *(i)* creates a path abstraction of the selected (survey) area, *(ii)* uses the Path Planner to create a route through survey area, together with the path library programs required to implement the path, *(iii)* creates mission script used by the Mission Controller, and, finally, *(iv)* creates the on-board version of the Robot World.

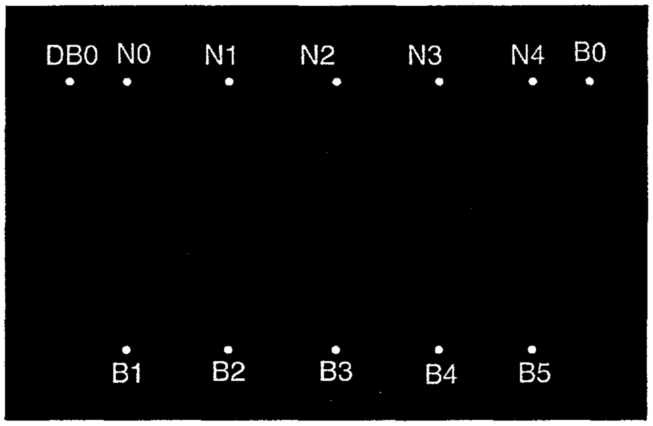


Figure 4-10. Path abstraction created by Path Planner from Robot World (DB = docking beacon, B = LIDAR beacon, N = node)

The path abstraction (or path description) of the warehouse contains the beacons, nodes, and paths, as shown in Figure 4-10. The drum lists for a given aisle are attached to the appropriate path defining that aisle. From the general path abstraction, a subset containing the information required for the survey area is created (see Figure 4-11). This structure provides the navigation information contained in the mission script.

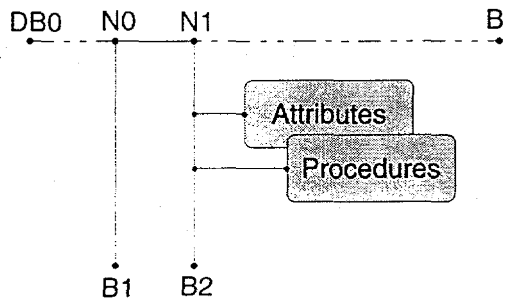


Figure 4-11. Path abstraction of the selected area

The path abstraction also provides a mechanism for determining the manner in which the robot will react to a given situation. It is possible to associate with each node or path both attributes and behaviors. This provides a mechanism by which the same instruction can cause a different response when executed in a different environment. This is done by attaching to each path component the appropriate attributes and procedures. For instance, the width would be an attribute which could be attached to a given path. Each attribute or procedure would have a scope equivalent to that of the path (or other component) to which it was attached. This would allow general conditions to be attached to the entire route and modified as appropriate in smaller sections.

4.2.7 The Mission Controller

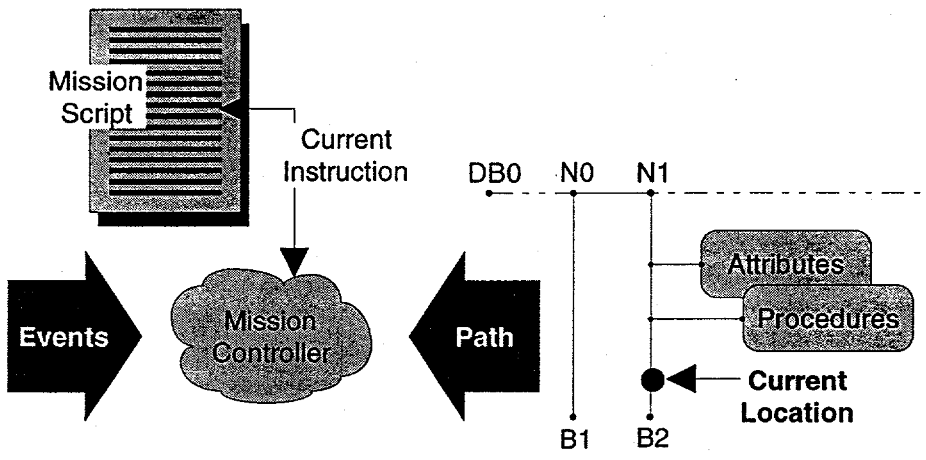


Figure 4-12. Operation of the Mission Controller

The mission controller is responsible for the actual execution of the mission. It performs the operations specified in the procedural part of the mission script, using the path information associated with the current path location. This information includes both attributes, such as the width of the path, and procedures, such as the path program. The mission controller also must respond to unscheduled events, which may be interrupts or be the consequence of a value reaching a limit. Figure 4-12 illustrates the general operation of the mission controller.

4.2.8 Mission Definition Language

A BNF description of the mission definition language (MDL) is given below.

```
<mission script> ::=            <mission object part> 'begin' <mission procedural
                                part> 'end' ';' ;
<mission object part>           { <object definition part> } ;
<object definition part> ::=    <survey definition> | <node definition> | <beacon
                                definition> | <path definition> | <exception part> |
                                <procedure definition> ;
<survey definition> ::=         'survey' '{' { <survey component> } '}' ';' ;
<survey component> ::=          <attribute part> | <procedure part> ;
<attribute part> ::=            'attribute' <attribute list> } ';' ;
<attribute list> :::            '{' { <attribute name> '=' <attribute value> ';' } '}' ;
<attribute name> ::=            identifier ;
<attribute value> ;;+           <scalar value> | <coordinate> ;
<scalar value> ::=              integer | real number ;
<coordinate>                    <2-D coordinate> | <3-D coordinate> ;
<2-D coordinate> ::=            <x part> ',' <y part> ;
<3-D coordinate> ::=            <x part> ',' <y part> ',' <z part> ;
<x part> ::=                    <scalar value> ;
```

```
<y part> ::=                      <scalar value> ;
<z part> ::=                      <scalar value> ;
<procedure part> ::=              { 'procedure' <procedure name> ';' } ;
<procedure name> ::=              integer ;
<node definition> ::=             'node' <node name> <2-D coordinate> ';' ;
<node name> ::=                   identifier ;
<beacon definition> ::=           <beacon type> 'beacon' <2-D coordinate> <beacon
                                  name> ';' ;
<beacon type> ::=                 'docking' | 'navigation' ;
<beacon name> ::=                 identifier | empty ;
<path definition> ::=             'path' <path name> '{' <path list> <path body> '}' ';' ;
<path name> ::=                   identifier;
<path list> ::=                   <node> <node> { <node> } ;
<path body> ::=                   { <attribute part> | <procedure part> };
<procedure definition> ::=        <script procedure> | <machine procedure> ;
<script procedure> ::=            'script' 'procedure' <script body> ;
<script body> ::=                 '{' <script statement> '}' ;
<script statement> ::=            <script command> { <parameter> } ;
<script command> ::=              'call' | 'inspect' | 'traverse' | 'find' | 'send' ;
<parameter> ::=                   <attribute value> | identifier ;
<machine procedure> ::=           'machine' 'procedure' <machine body> ;
<machine body> ::=                '{' <path statement> '}' ;
<path command> ::=                path assembler statement ;
<exception part> ::=              'exception' <exception name> '{' <exception body>
                                  '}' ';' ;
<exception name> ::=              identifier ;
<exception body> ::=              <script body> | <machine body> ;
<mission procedural part> ::=     { <script commands> | <procedure calls> } ;
<procedure call> ::=              <procedure name> ;
<procedure name> ::=              identifier;
```

4.3 THE PATH ASSEMBLER

4.3.1 Introduction

The Portable Path Assembler (PASM) is designed to support code development for the Cybermotion K2A Self-Guided Vehicle. While intended for use with the Unix environment developed by the Department of Electrical and Computer Engineering, University of South Carolina, it was designed to be compatible with existing Cybermotion code and may be hosted on a PC. The designation "portable" is given to indicate that PASM itself is designed to be hosted in any standard C environment and requires only a simple ASCII terminal.

By default, PASM provides for strict enforcement of the Cybermotion Path Language (CPL) as defined in the current Cybermotion manual. However, the Path Language has evolved over the years as new commands and functionality were added to the K2A. As a result, many commands have different parameter values or usage from when they were originally introduced. PASM provides an option to accept these early conventions, making it compatible with all existing Cybermotion code. This compatibility is made an option (as opposed to the default) to encourage programmers to meet current conventions with all new code. Since PASM works independent of a specific environment, it does not provide some of the features of the Cybermotion assembler when used in conjunction with the Dispatcher program.

The program is loosely implemented as a recursive-descent, top-down parser from a BNF (Backus-Naur Form) description of the Cybermotion Path Language. It departs from strict adherence to these principles for three reasons:

(i) Assemblers, while relatively simple compared to most compiled languages, have some complications, such as forward references and more than one token look ahead;

(ii) No precise expression of the CPL production rules or of the CPL semantics is available;

(iii) CPL has evolved over the years in conjunction with modifications to the K2A. In some instances these modifications result in inconsistencies with current CPL descriptions. Compatibility with all known existing K2A code is a high priority for PASM, therefore, the actual language implementation accepts all previous code to the greatest extent possible.

Note: While some test programs were obtained from Cybermotion, most test code was written since acquisition of the K2A at USC, therefore, it is likely that PASM may not accept all early versions of CPL.

All PASM limits, command values, etc., are contained in tables which may be maintained independent from the code. This allows these values, which frequently change with updates to the vehicle, to be modified without changes to the code. It also supports the use of a consistent technique of parameter checking. A separate program, the **PASM** Support Utility (PSU) was created to maintain these tables. Like PASM, PSU is written in C and will function in any standard C environment. It differs from PASM in that it is intended for use primarily by developers as opposed to end users and is considerably less "polished" code. PSU is described in a later section.

Several features have been added (at the suggestion of Cybermotion) to PASM. They include *(i)* support for units (inches, feet, meters, and centimeters), *(ii)* expressions (+, -, *, and ÷), *(iii)* decimal number support, and *(iv)* a disassembler.

PASM operates as a typical two-pass assembler in order to resolve the problem of forward references (see Figure 4-13). The primary purpose of the first pass is to create the symbol table although the parser performs error checking on both passes. The error messages are included in the listing program and may also be sent to the screen if desired. It generates an intermediate code (called *icode*) during the second pass. The main program uses this intermediate code to produce the binary output file, which uses the Cybermotion format.

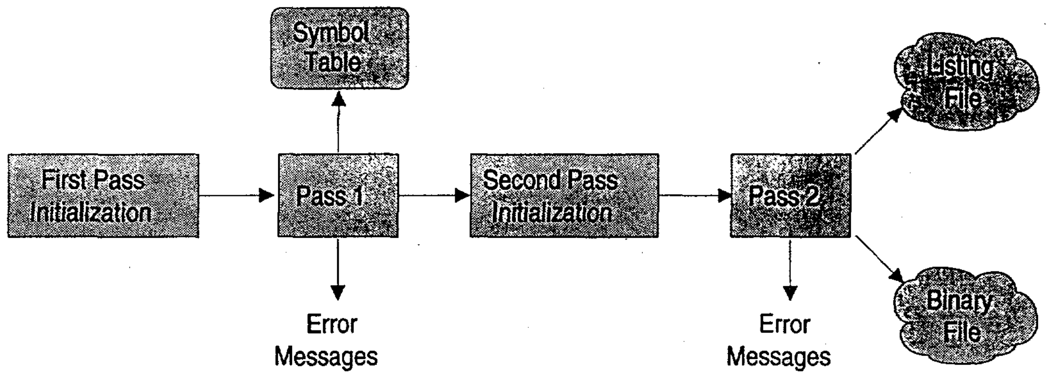


Figure 4-13. PASM operation

4.3.1.1 PASM Technology

This section details the technology used to implement PASM. It primarily applies to the scanner and parser since these components are most appropriately handled by formal techniques. PASM is loosely[2] implemented using a top-down approach known as recursive descent. Due to the lack of a formal grammar, it cannot be guaranteed that it meets the requirements for the subset of LL(1) grammars which are suitable for use with recursive descent. However, the relatively heuristic nature of this technique makes it more suitable for loosely defined grammars than a table-driven approach using an explicit stack. Recursive descent parsers may be designed by application of the following five design rules to the production rules:

1) A *sequence* of elements is translated into a *compound statement.*
2) A *choice* of elements is translated into a *switch statement* (or *if* statements if the number of choices is small).
3) A *loop* is translated into a *while or for statement.*
4) A *non-terminal* BNF production denoting *another production* is translated by a *function call.*
5) An element denoting a *terminal* symbol (*x*) is translated into a statement of the form:

```
IF symbol = x THEN
        GetNextSymbol
ELSE
        ReturnError;
```

In order to use these rules to construct a recursive descent parser from a particular grammar, the production of the grammar must meet the following restrictions.

Restriction 1 — Every branch emanating from a *choice of elements* must lead toward a distinct first symbol. An alternative way of stating this restriction is

[2] The term "loosely" is used here because the PASM grammar has not been formally defined due to the manner in which it has evolved.

that no symbol may be a *starter* of more than one of the alternatives of a given production (rule).

Restriction 2 — For the case in which there is an *empty alternative* as one of the choices , all *initial symbols* of the following statements must be distinct from the initial symbols of the choice in question. An alternative statement of restriction 2 is that no symbol may be a possible *starter* and a possible *follower* of an alternative which may possibly be empty.

The current plans for PASM v2.0 are to use a table-driven LL(1) parser with a formal stack. This should reduce the code size and improve the performance of the system. It will; however, require a formal definition of the PASM grammar.

4.3.1.2 PASM Usage

PASM is designed to be used either as a stand alone program or as an integral part of the control environment. When used as a stand alone program it is invoked by the command

```
pa { <options> } <filename>.
```

which follows standard Unix practice. The path assembler is designed so that a user who is reasonably experienced in assembler-level programming should be able to use it immediately. A minimal help function is included in the form of a help option. If the -h option is specified (`pa -h` or `pa /h`) the assembler will output a brief help message which gives the options available and the function of each. The response to `pa -h` is given in Figure 4-14 below. Note that most of the defaults are chosen for the "production case." For example, it is likely that most of the time a listing file will not be needed, therefore the default option is not to create it. Likewise, the default is to give only one error message per line although the "verbose" mode (-v) will give multiple error messages per line.

```
Virtual Path Language Assembler     v1.33
The Path Assembler is invoked by the command

      pa { <options> <filename> }

The following options are available:

-a                  Disassembler mode
-b                  Batch mode
-c                  Inhibits creation of .act file
-d                  Debug mode
-h                  Help
-i                  Print include files in the listing
-L                  Print listing in landscape format
-l                  Print listing file
-o <filename>       Allows specifying output file name
-p                  Parameter checking
-s                  Include symbol table in listing
-t                  Sets tab length for listing
-w                  Disables warning messages
-v                  Verbose mode.  Allows multiple error messages
-x                  Produces Hex listing
```

Figure 4-14. PASM help screen

4.3.1.3 PASM Examples

Several example programs (taken from the test suite used in developing the program) are given to illustrate some of the new features. The following program illustrates the use of units and decimal numbers. The symbol table listing is included to show the values assigned to each of the symbols.

```
; PASM Test Suite
; Program 1 - Assembler Command Test (defc and defp)
; 3 August 1994
;
; This program contains a number of versions of the defc and
; defp commands. It tests both the ability of the program to
; handle different formats and to handle different units.
;
asym    defc            1
        defc   bsym     1
csym    defp            1, 2
        defp   dsym     1, 2
esym    defc            0.1 m.
        defc   fsym     0.1 m.
gsym    defp            0.1 m.,       0.2 m.
        defp   hsym     0.1 m.,       0.2 m.
        defc   isym     -0.1 m.
        defc   jsym     1.5 in.
        defc   ksym     -1.5 in.
        defc   lsym     2.2 cm.
        defc   msym     -2.2 cm.
        defc   nsym     1.2 ft.
        defc   osym     -1.2 ft.
        defp   psym     5 cm., 1 m.
        defc   qsym     $a
        defc   rsym     $a cm.
        defp   usym     $a, $b
```

Figure 4-15. Example program 1.

```
SYMBOL TABLE CONTENTS:
Symbol          Type      Line    s|x      y
------          ----      ----   -----   -----
asym            Constant   10        1      -
bsym            Constant   11        1      -
csym            Position   12        1      2
dsym            Position   13        1      2
esym            Constant   14       32      -
fsym            Constant   15       32      -
gsym            Position   16       32     65
hsym            Position   17       32     65
isym            Constant   18      -32      -
jsym            Constant   19       12      -
ksym            Constant   20      -12      -
lsym            Constant   21        7      -
msym            Constant   22       -7      -
nsym            Constant   23      120      -
osym            Constant   24     -120      -
psym            Position   25       16    328
qsym            Constant   26       10      -
rsym            Constant   27       32      -
usym            Position   28       10     11
```

Figure 4-16. Symbol table for example program 1.

The second program, given below, illustrates the use of expressions. It also contains units to demonstrate that units may be included in expressions. A listing file is included.

```
;   PASM Test Suite
;   Program 2 - 3 August 1994
;   Expression test.  This program tests the ability of the
;   parser to handle expressions in command statements.
;
            defc    asym    10
            defc    bsym 2
            defp    csym 3, 4
            defp    dsym 5, 6
            defc    esym 10
main        run     esym    asym, bsym
            run     esym    asym+bsym, asym
            run     esym    asym+bsym, asym
            run     esym    asym+bsym, asym
            run     esym    csym+dsym
            run     esym    csym-dsym
            run     esym*esym-(esym+10), csym*dsym
            run     esym, asym+1.23 m., bsym
            run     ((10-8)*5/2)*2, 10, 2
            run     (asym+bsym), asym*(1.1 m. + 23 cm.)/2, -asym*(1.1 m.)/2
            run     (-(1.145 m. * asym)), 6.98 m./6.98 cm., 1 m./1 ft.
```

Figure 4-17. Example program 2.

Inclusion of the expression evaluator places some restrictions on the variable naming conventions unless spaces are required before and after operators. PASM v1.33 currently treats a-b as the symbol "b" subtracted from the symbol "a" as opposed to the single symbol a-b. This choice was based on standard convention and ease of implementation, however, the parser can be modified to treat a-b as a single symbol.

It might also be worthwhile including a symbol for the current instruction location. This would allow relative addressing and would enhance relocatability of the code.

```
Virtual Path Language Assembler v1.33 (t6.sgv)         page 1
Line Step  Cmd  s     x     y           Source Text
---- ----  --- --- ----- -----  ------------------------------
   1
   2  ;    PASM Test Suite
   3  ;    Program 2 - 3 August 1994
   4  ;    Expression test.  This program tests the ability of the
   5  ;    parser to handle expressions in command statements.
   6  ;
   7    0                              defc    asym    10
   8    0                              defc    bsym 2
   9    0                              defp    csym 3, 4
  10    0                              defp    dsym 5, 6
  11    0                              defc    esym 10
  12    0    1  10    10     2  main   run     esym    asym, bsym
  13    1    1  10    12    10   run   esym    asym+bsym, asym
  14    2    1  10    12    10   run   esym    asym+bsym, asym
  15    3    1  10    12    10   run   esym    asym+bsym, asym
  16    4    1  10     8    10   run   esym    csym+dsym
  17    5    1  10 65534 65534   run   esym    csym-dsym
  18    6    1  80     0     0   run   esym*esym-(esym+10), csym*dsym
  19    7    1  10   413     2   run   esym, asym+1.23 m., bsym
  20    8    1  10    10     2   run   ((10-8)*5/2)*2, 10, 2
  21    9    1  12  2175 63736   run   (asym+bsym),  asym*(1.1  m.  +  23  cm.)/2,  -asym*(1.1
m.)/2
  22   10    1-3750   104     3        run     (-(1.145  m.  *  asym)),  6.98  m./6.98  cm.,  1
m./1 ft.
ERROR REPORT:  warnings: [0], errors: [0]

   SYMBOL TABLE CONTENTS:
   Symbol          Type        Line     s|x      y
   ------          ----        ----    -----   -----
   asym            Constant      7       10      -
   bsym            Constant      8        2      -
   csym            Position      9        3       4
   dsym            Position     10        5       6
   esym            Constant     11       10      -
   main            Label        12        0
```

Figure 4-18. Listing file for example program 2.

The output obtained from the disassembler option (`pa -a t2`) is given in Figure 4-19. Use of the disassembler requires that the appropriate output file (`.act`) be available.

```
Virtual Path Language Assembler  v1.33
Disassembler mode ... disassembling [t6.act]
File contains [11] instructions, [4] additional bytes

        Command:       s:        x:      y:
        run (  1)     10        10       2
        run (  1)     10        12      10
        run (  1)     10        12      10
        run (  1)     10        12      10
        run (  1)     10         8      10
        run (  1)     10        -2      -2
        run (  1)     80         0       0
        run (  1)     10       413       2
        run (  1)     10        10       2
        run (  1)     12      2175   -1800
        run (  1)     90       104       3
Drive Acceleration: [  4]
Steer Acceleration: [ 10]
[0] milliseconds CPU time
```

Figure 4-19. Disassembler output for program 2.

4.3.2 PASM STRUCTURE

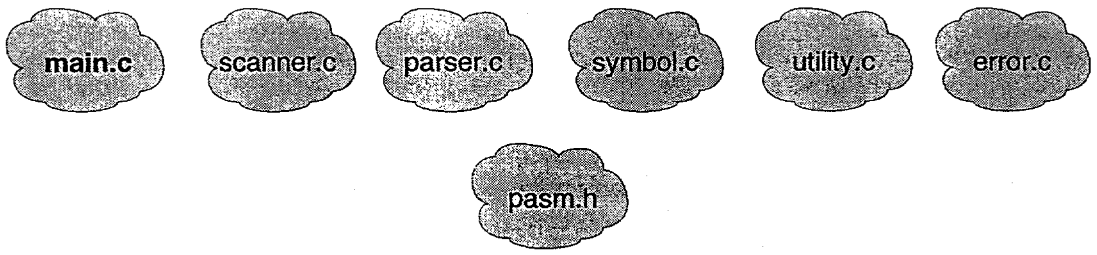


Figure 4-20. PASM file structure.

PASM is implemented in seven files: `main.c`, `scanner.c`, `parser.c`, `symbol.c`, `utility.c`, and, `error.c` , shown in Figure 4-20. This partitioning was chosen to minimize the number of external references, reducing the risk of harmful side effects as a result of program modifications. The header file, `pasm.h`, is used by each of the C files to obtain the necessary information to access functions or variables external to the file. Some of the values given in the header file are conditional in order to promote portability. For example, the file length values as shown below allow for differences between MSDOS and Unix.

```
#ifdef MSDOS
#define   FILENAME_LENGTH             8
#define   EXTENSION_LENGTH            3
#else
#define   FILENAME_LENGTH             32
#define   EXTENSION_LENGTH            32
#endif
```

A common header file is used to promote consistency and to enhance understandability since this means that all files see the same header. While most C compilers are tolerant of inconsistencies concerning uses of the *extern* command, conditional expressions are used to ensure that *not object* is declared to be both external and internal. For example, the externally visible objects in the file parser.c are given as follows:

```
#ifndef   PARSER
extern    int     ExternEntries;
extern    int     ParseCommand(int);
extern    void    ParsePseudop(int, SymType *);
#endif
```

The constant PARSER is defined at the beginning of parser.c so that the external declarations for that file are not seen in parser.c. The same procedure is used for each of the other files.

Considerable effort was made to choose a partitioning which would minimize the number of variables with external linkage[3]. All external variables which do not need to be accessed by any functions outside of the file have been declared to be static. A brief synopsis of the contents of each of the files is given in Table 1.

FILE	CONTENTS
main.c	main function, initialization routines, command line parser, disassembler, exception handler, and general shared variables for entire program,
scanner.c	Routines to get the next token from the source file, handle units and filenames, and the token tables.
parser.c	All parsing and most semantics routines. The two entry points are ParseCommand and ParsePseudop
symbol.c	Symbol table and access routines
utility.c	Stack routines, units conversion table
error.c	Error handling routines (RecordError, OutputErrors) plus various record and control variables for the error system.

Table 1. Contents of PASM files

4.3.3 PASM OPERATION

4.3.3.1 PASM Dataflow

A data flow model of PASM is given in Figure 4-21. The input text is fetched from the source file a line at the time by the routine NextLine (contained in main.c) and placed in the input line buffer (line[ ], in main.c). NextLine also handles the context change associated with the end of an include file by testing the include level when an end of file is found. It also updates the line number and handles some of the listing chores.

The routine NextToken, which is the primary module in scanner.c, returns the token values, together with appropriate semantic information, from the input line. NextToken returns one of the values from the enumeration terminals. It also updates the global vari-

[3] A C variable which is defined outside of a function is an external variable and, by default, has *external linkage*, which means that it may be accessed from any function in any file. If an external variable is defined to be *static*, then it no longer has external linkage and may be accessed only in the file where it is defined.

ables `id`, `IntVal`, `RealVal`, `ChPos`, and `IdPos` as appropriate. `id` contains the actual id string of a user identifier, `IntVal` contains the value of a number token (or `RealVal` if it is a decimal number), `ChPos` is a pointer to the current character in the input line, and `IdPos` is a pointer to the start of the string associated with the current token. `IdPos` is used by the error routines to insert the "marker" to identify the offending token. `NextToken` is passed an argument which specifies whether the token being fetched is a command or an object. This information is used to give more precise error and warning messages.

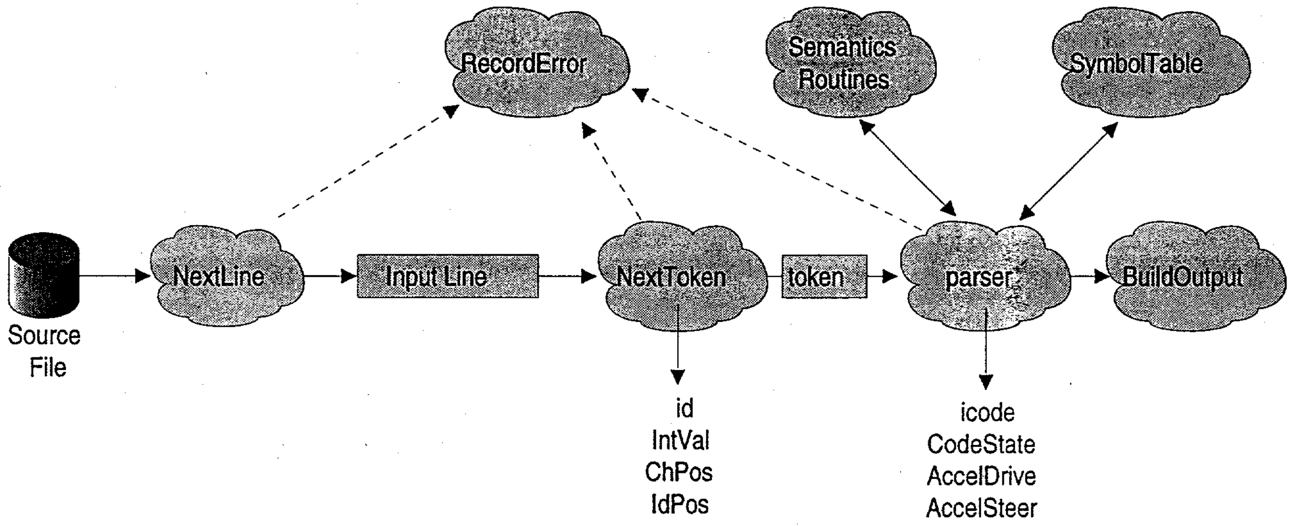


Figure 4-21. PASM data flow model

The bulk of the input processing is done by the parser. It checks for correctness, updates the symbol table, and creates an output where appropriate. The output created by the parser is a C struct (`icode`) which consists of four integers, representing the opcode and each of the three parameters. The function `BuildOutput` in `main.c` changes these values to Cybermotion binary format and writes it to the output file.

4.3.3.1.1 Pre-Fetching of Tokens and Characters

PASM adheres to a policy of one-token look ahead to the greatest extent possible. This implies that each PASM routine must either fetch a token initially, or expect to have one present and ensure that a new token is available at the exit of the routine. Since some routines can only determine that processing is complete by fetching a token which "does not fit", i.e., belongs to the next routine, then the best approach is to have all routines assume that a new token is available at entry. Likewise, each routine must ensure that a fresh token is available at exit. Since only one token is being processed at any given time, the information for the current token is stored in external variables and is available to all functions. The convenience and efficiency of this technique appear to justify any (possible) increase in possible side effects. The scanner follows the same policy in fetching characters and this information is handled in the same fashion.

4.3.3.2 The `main.c` File

The file `main.c` contains the functions and variables which generally apply to the entire program and are not suitable to encapsulation (as, for instance, the symbol table). The primary routines included in the `main.c` file are:

- the function `main`
- initialization routines, `init`, `GetInputFile`, `SetUpPass2`
- command line parser (for options), `ParseCommandLine`, `GiveHelp`
- pass 1 and pass 2 main functions, `pass1` and `pass2`
- `NextLine` function
- Output listing functions, `OutputListing` and `OutputHeader`
- Code output functions, `BuildOutput` and `OutputDriveSteer`
- `Disassembler`
- `HandleException`

The primary external variables include:

- file variables
- scanner and lexical variables
- code output variables
- program mode variable, `mode`

4.3.3.2.1 File Variables

The file variables are:

```
/*
!  PASM FILES AND RELATED VARIABLES:
!  The integer variables include and IncludeChange are used to implement
!  include files.  The file pointers are used as follows:
!
!  fp:          Pointer to current input file (source or include)
!  fplist:      Pointer to listing file
!  fpout:       Pointer to the (binary) output file
*/
char                    CurrentFile[FILENAME_LENGTH+EXTENSION_LENGTH+2];
FILE                    *fp, *fpextern, *fplist;
static    FILE          *fpout;
int                     IncludeChange = FALSE;
int                     IncludeLevel = 0;
static    char          FileName[FILENAME_LENGTH+EXTENSION_LENGTH+2];
static    char          ListFile[FILENAME_LENGTH+EXTENSION_LENGTH+2];
static    char          OutFile[FILENAME_LENGTH+EXTENSION_LENGTH+2];
static    char          SrcFile[FILENAME_LENGTH+EXTENSION_LENGTH+2];
static    char          ExternFile[FILENAME_LENGTH+EXTENSION_LENGTH+2];
static    int           InputFileSpecified = FALSE;
static    int           OutputFileSpecified = FALSE;
```

The filenames and the output file pointer are defined as static since they are not referenced except in `main.c`.

4.3.3.2.2 Scanner and Lexical Variables

The scanner and lexical variable of the main program are:

```
/*
!  SCANNER AND LEXICAL VARIABLES:
!  Current Token Definitions
!  These values are set by each call to the scanner function NextToken,
!  which returns the value for token.  id, InVal, and IdPos are set
!  directly.  These values are used by all PASM routines.
!
!  token          The token value, one of the values from the enumerated
!                 list of terminals in pasm.h
!  id             The identifer string for identifiers and keywords.
!  IntVal         Value of integer symbols.
!  tp             Pointer to the starting location of the current token.
!  cp             Pointer to the current character (start of next token)
!
!  The array line[ ] holds the current input line.
*/
char              id[ID_LENGTH+1];
char              line[LINE_LENGTH + 1];
char              *cp, *tp;
int               LineCount = 0, LineNumber = 0;
int               TabLength = TAB_LENGTH;
int               token, IntVal;
float             RealVal;
static    int     PageLength = PAGE_LENGTH_PORTRAIT;
static    int     page;
```

4.3.3.2.3 Code Output Variables

The code output variables are given below:

```
/*
!  CODE OUTPUT VARIABLES
!  The structure icode holds the intermediate code generated by the parser
!  during the second pass.  CodeState is a bit-vector variable which is set
!  to define the status of icode.  The constants used to set and test the
!  bits of CodeState are also used with the variable CmdType which is part
!  of the parsers command table.  These constants include:
!
!  S_PAR              1
!  X_PAR              2
!  Y_PAR              4
!  COMMAND_PAR        8
!  POS_PAR            16 (used by CmdType only)
!  S_ERR              32
!  X_ERR              64
!  Y_ERR              128
!
*/
IcodeType         icode;
int               CodeState = 0;
static    int     CodeGen;
```

The variable `icode`, which has the type `IcodeType` given below,

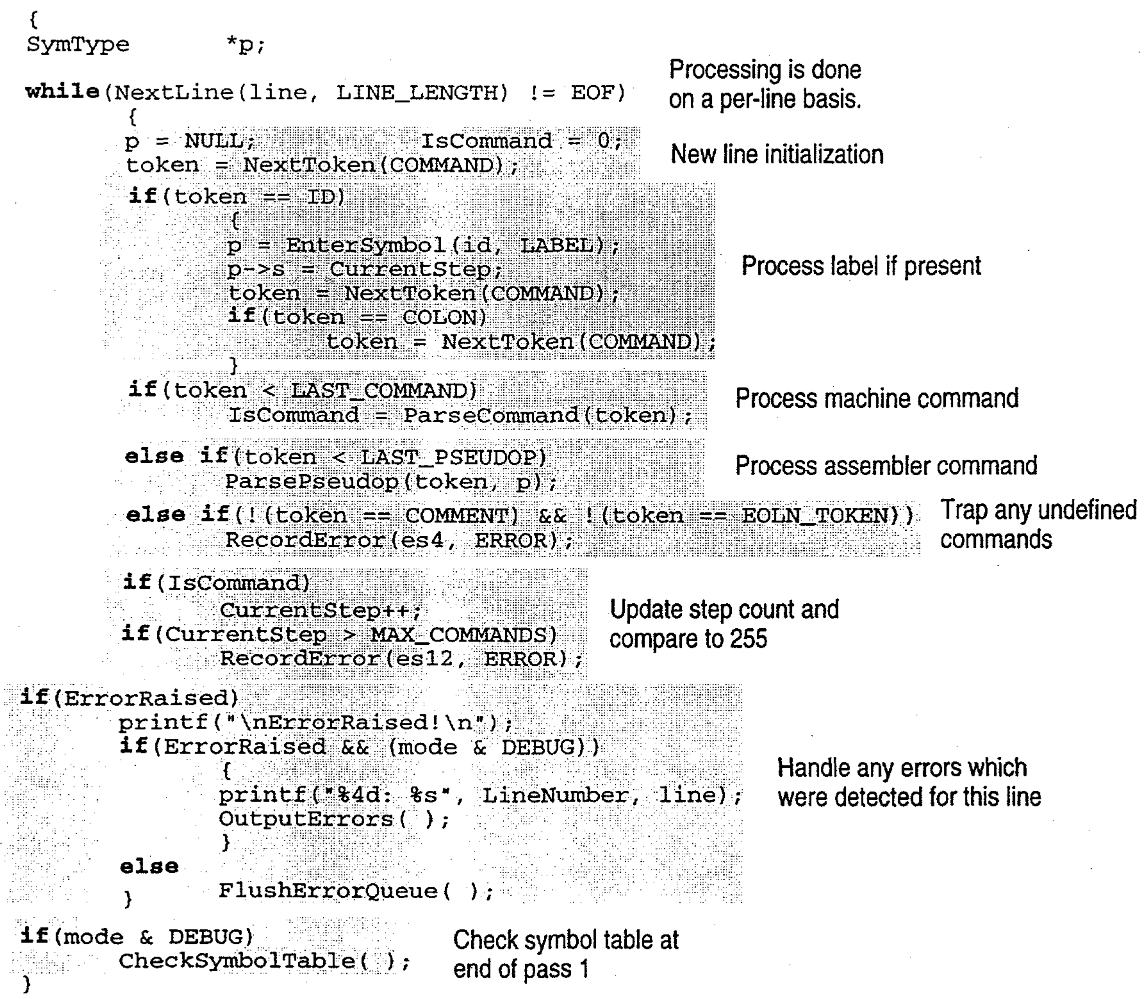
```
              typedef       struct
                    {
                    int             command;
                    long            pv[3];
                    } IcodeType;
```

is used to construct machine commands. `CodeState` is a bit-vector variable used to designate the form of the output command. Various bits of `CodeState` are set to indicate the number and type of operands present. This information is used by `BuildOutput` to properly format the output. `CodeGen` is used to control the generation of code. It is set false if an error occurs, inhibiting the generation of code.

4.3.3.2.4 Program Mode Variable

The program mode variable, mode, is used to specify the operating mode of the assembler. The various bits of mode are set by the command line option switches.

```
/*
!  PROGRAM MODE VARIABLE:
!  The variable mode is a bit-vector which holds the operating mode of the
!  program.  The various bits are set by the option switches when the
!  program is invoked.
!
!  Bit                        Function                             Default
!  DISASSEMBLER               Perform disassembly                  off
!  NO_MULTIPLE                One message per error                off
!  DEBUG                      Debugging option                     off
!  LISTING_ON                 Print output listing                 off
!  INCLUDE_SYMBOL_TABLE       No listing of symbol table           off
!  HEX_MODE                   Produce Hex listing                  off
!  BATCH_MODE                 Batch mode                           off
!  OUTPUT_INHIBITED           Inhibits creation of .act file       off
!  PARAMETER_CHECK            Enables parameter checking           off
*/
int       mode = 0;
```



```
 {
 SymType       *p;

 while(NextLine(line, LINE_LENGTH) != EOF)
        {
        p = NULL;           IsCommand = 0;
        token = NextToken(COMMAND);
        if(token == ID)
               {
               p = EnterSymbol(id, LABEL);
               p->s = CurrentStep;
               token = NextToken(COMMAND);
               if(token == COLON)
                      token = NextToken(COMMAND);
               }
        if(token < LAST_COMMAND)
               IsCommand = ParseCommand(token);

        else if(token < LAST_PSEUDOP)
               ParsePseudop(token, p);
        else if(!(token == COMMENT) && !(token == EOLN_TOKEN))
               RecordError(es4, ERROR);

        if(IsCommand)
               CurrentStep++;
        if(CurrentStep > MAX_COMMANDS)
               RecordError(es12, ERROR);
 if(ErrorRaised)
        printf("\nErrorRaised!\n");
        if(ErrorRaised && (mode & DEBUG))
               {
               printf("%4d: %s", LineNumber, line);
               OutputErrors( );
               }
        else
        }      FlushErrorQueue( );

 if(mode & DEBUG)
        CheckSymbolTable( );
 }
```

Processing is done on a per-line basis.
New line initialization
Process label if present
Process machine command
Process assembler command
Trap any undefined commands
Update step count and compare to 255
Handle any errors which were detected for this line
Check symbol table at end of pass 1

Figure 4-22. Annotated pass1 code.

4.3.3.2.5 Other main.c Variables

The file main.c contains other external variables, such as the disassembler command names table, used to supply the mnemonic names for the machine commands and the parameter sign table, which is used to format the disassembler output. The general program variables are given below:

```
/*
!  GENERAL PROGRAM VARIABLES
!
!  The variables given below store general program information.  Their uses
!  are consistent with the variable names.
*/
int                         AccelDrive = 4, AccelSteer = 10;
int                         pass = 1;
int                         MaxStep = 0;
static    int               CurrentStep = 0;
static    int               IsCommand;
static    int               CommentLine = FALSE;
static    long              CPU_time, start;
static    long              ListingInhibited = FALSE, lines = 0;
```

4.3.3.2.6 The Pass 1 and Pass 2 Functions

The functions pass1 and pass2 are the main programs for pass 1 and pass 2 respectively. An annotated version of the code from pass1 is given in Figure 4-22. The code for pass2 follows a similar organization but contains extra calls associated with creating the output code and the listing. pass2 also performs error checking on labels to catch errors such as duplicate labels. Both pass1 and pass2 perform the initial processing of each input line, handling labels and the command. Once the nature of the command is known, the appropriate parser routine (ParseCommand or ParsePseudop) is called and the parser handles the remaining processing for that source line.

4.3.3.2.7 The NextLine Function

A simplified flowchart of the NextLine function is given in Figure 4-23. This function is fairly complex due to both the amount of error checking required and the need to handle some of the processing of include files. Since NextLine is the function which detects an end of file, it is the logical place to put the processing associated with the end of an include file. This is done by testing the integer variable IncludeLevel whenever an end of file is detected. If an end of file is detected when IncludeLevel is non-zero, then the previous file context is popped from the include stack and restored. NextLine also checks the Boolean variable IncludeChange to determine if a new include file has been opened and, if so, causes a new file header to be output if appropriate. This is done here rather than while processing the include directive to avoid complications with the error handler.

4.3.3.2.8 The BuildOutput Function

BuildOutput creates the actual binary output values and writes them to the output file. The input values are contained in the variable icode discussed previously. It is a simple function except for the relatively esoteric computations required to create the binary output. The values for drive and steer acceleration are appended to the output by the function OutputDriveSteer which is called at the end of the source file.

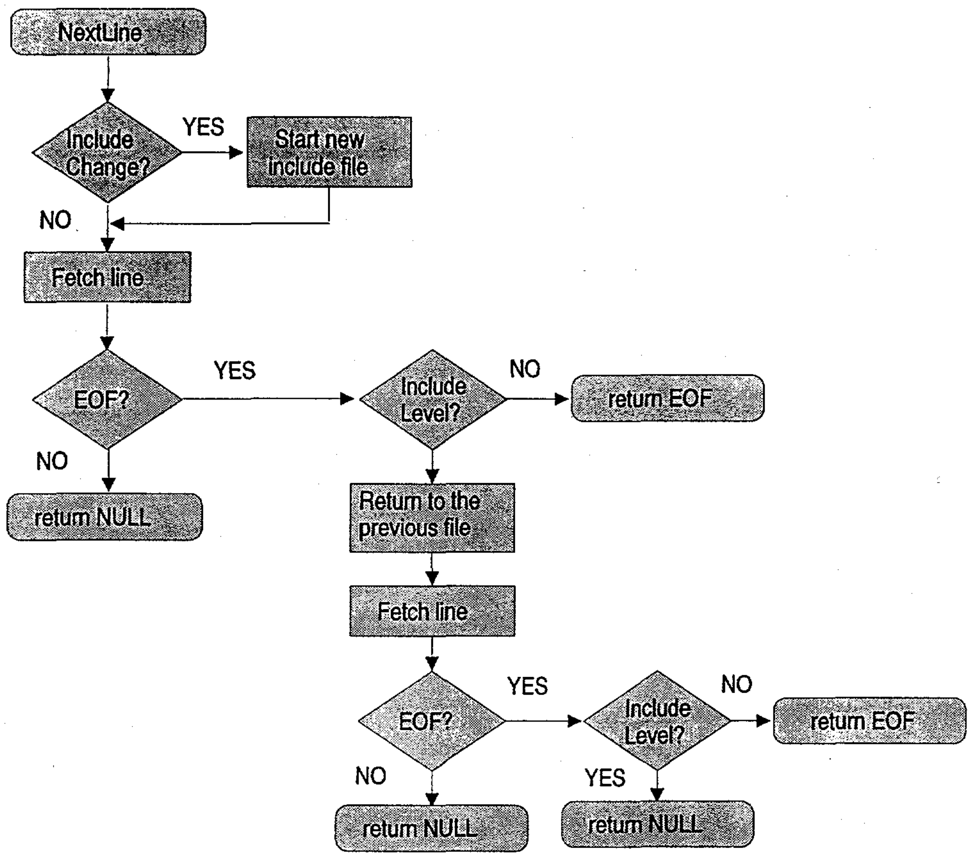


Figure 4-23. NextLine simplified flowchart

4.3.3.2.9 The Output Listing Functions

The output listing is created by the functions `OutputListing` and `OutputHeader`. `OutputListing` is called by `pass2` after the current line has been processed and prior to calling the function `OutputErrors`, which will insert any error messages accumulated for the line. `OutputListing` calls `OutputHeader` at the beginning of each page to create the page header and also updates the variables `PageCount` and `LineCount`. `OutputHeader` is implemented as a separate function since it is also called by the initialization routines and the `NextLine` function. `OutputListing` uses information from the variables `CodeState` and `mode` to determine the appropriate listing format (landscape or portrait, decimal or hex).

4.3.3.2.10 The Disassembler Function

If PASM is invoked with the `-a` option it functions as a disassembler. Selection of this option causes the program to default to the use of the ".act" rather than ".sgv" suffix and to call the function `Disassembler` rather than `pass1`. `Disassembler` first scans the input file to count the number of bytes and to determine if drive and or steer acceleration values have been included. It then rewinds the input file and fetches each line and prints the values for the opcode and parameters. The opcode mnemonics are supplied since they are known. The function uses the parameter sign table (`ParameterSign`) to determine if the parameter is signed (-32,768 to +32,767) or unsigned (0 to 65,535). As with

OutputListing, it also checks the mode variable to determine whether to display the output in decimal or hex format.

4.3.3.2.11 The Exception Handler

The exception handler (HandleException) is called if a PASM error (as opposed to a program error) is encountered. It provides a brief message as to the nature of the problem, closes the open files, and then calls the exit function. Whenever a PASM function has an appropriate program invariant, it uses it to test for exceptional conditions. For example, if a variable, such as the selector variable for a switch statement, has an "impossible" value, then an exception is declared and the program is halted. Occurrence of an exception strongly implies that there is an error in PASM itself. For this reason, exceptions are treated as terminating events and no attempt is made to resolve the problem. While this code could be removed after the program has been sufficiently tested, it requires little time or space and is a reasonable check for correct program operation.

4.3.4 The Scanner

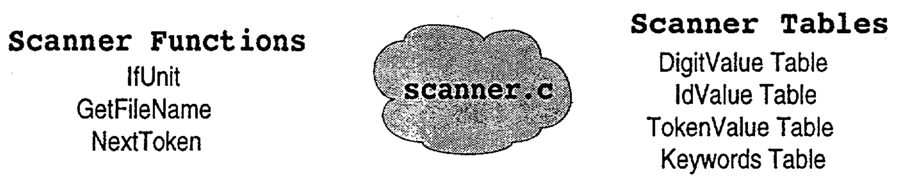


Figure 4-24. Scanner primary objects

The NextToken function returns a token for each input string which represents the value from vocabulary represented by that string. The PASM vocabulary is given in Figure 4-25. The entries LAST_COMMAND, LAST_PSEUDOP, and LAST_TERMINAL are included for differentiation between vocabulary classes and are not always used by the parser.

```
typedef   enum
   {
   NOP,  RUN,  TURN,  WAIT,  BACK,  WRITEB,  WRITEW,  READB,  READW,  JUMP,  JUMP_GT,
   JUMP_LT, JUMP_EQ, CALL, CALL_GT, CALL_LT, CALL_EQ, RETURN, ADD, SUB, HALT, DOCK,
   UNDOCK, SETXY, SETAZ, JOG, SETACC, COPYB, COPYW, DOCKOUT, DOCKIN, MEANAZ, AVOID,
   WARN,  MOVE,  PICK,  PUT,  MARK,  FOLLOW,  WALL,  RADIUS,  RUNON,  PORT,  UNPORT,
   APPROACH, SCAN, COMP, CURLIM, USE, CRUISE, STOP, VECTOR, ABS, MULT, DIV, CIRCUM,
   MTRSOFF, HALL, BREAK, CDEFLECT, WDEFLECT, PATROL, SURVEY, PAN, TILT, ZOOM, MAUX,
   DOOR,  SETSTDBY,  STANDBY,  GATE,  WBEGINS_AT,  WENDS_AT,  JUMP_NE,  CALL_NE,
   WALLOFF_AT, WALLON_AT, GATE_AT, APPRWALL, APPRJUNK, LAST_COMMAND, DEFC, DEFP,
   DEFD, DEFS, EXTERN, EXTERN_REF, INCLUDE, LAST_PSEUDOP,COLON, COMMA, COMMENT,
   DIVIDE, EOLN_TOKEN, EQUALS, ERR_NUM, ERR_TOKEN, ID, INTEGER, L_BRACKET, L_PAREN,
   MINUS,  MULTIPLY,  NUL,  PERIOD,  PLUS,  R_BRACKET,  R_PAREN,  REAL,  SEMICOLON,
   NULL_TOKEN, LAST_TERMINAL
   } Terminals;
```

Figure 4-25. PASM vocabulary

A flowchart of the operation of NextToken is given in Figure 4-26. NextToken first removes any leading spaces or tabs, then sets the token pointer (tp) to the same value as the character pointer (cp). The character pointer contains the address of the next character to be processed and the token pointer contains the location (in the input line) of the current token. Since the parser uses a one-token look ahead, the token pointer can be used by the error handler to mark the location of an error. The character is tested to see if it is the end of line ('\0' in the input line) and if so, the token value `EOLN_TOKEN` is returned. If the character is not the end of line character, then it represents the first character of an input word which must be processed by `NextToken`.

The first step is to assign a tentative token value to the input word by using the character as the index to the `TokenValue` table. This value is tentative because commands (such as turn, jog, etc) have the same form as used identifiers, thus all identifiers are initially given the token value `ID`. `NextToken` then checks for three special cases, identifiers, numbers, and error tokens, since each of these may contain multiple characters. If the token value is `ID`, then the routine `ProcessId` is called. `ProcessId` will continue to fetch input characters as long as they are legal identifier characters, placing them in the identifier buffer (`id`). The table `IdValue` is used to determine if the characters are legal identifier characters. Each identifier is then processed by the function `LookUpId` which attempts to match it with an entry in the `Keywords` table. If a match is found, then the token value from the table, representing the appropriate command, is returned. If not, then the token value `ID` is returned.

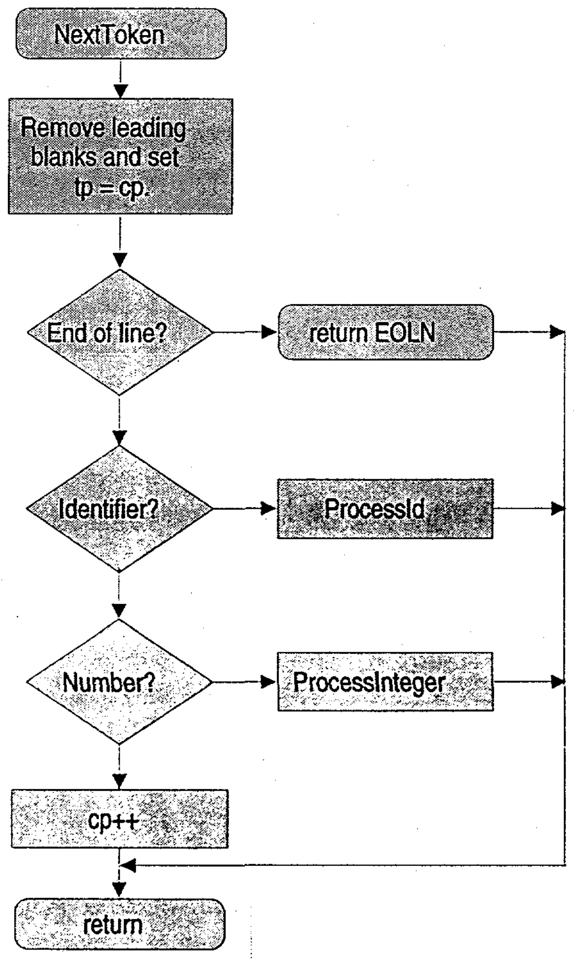


Figure 4-26. Flowchart of operation of `NextToken`

If the token value is not ID, then it is tested to see if it is INTEGER. If so, the routine ProcessInteger is called. ProcessInteger converts the character string to an integer value which is stored in the variable IntValue (or RealValue if it is a decimal number). It uses the table DigitValue to assign a value to the characters. The final test of the token value is to determine if it is an error character. If so, then all following error characters are collected and the value ERR_TOKEN is returned.

If the token is not one of the special cases ID, INTEGER, or ERR_TOKEN, then the value from the TokenValue table is returned and the character pointer is advanced to the next character. The code for NextToken is shown in Figure 4-27.

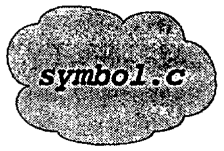
The scanner also contains the functions IfUnit and GetFileName. IfUnit is called in place of NextToken in any context where a unit is possible and handles all unit conversions. GetFileName is similar to NextToken but uses a different table (FileChar) to reflect the differences in the characters which are legal in a filename as opposed to a user identifier. It is primarily used when processing include file names.

```
/*--BEGIN FUNCTION--(NextToken)-------------------------------------------*/

int       NextToken(int context)
{
id[0] = '\0';
while( *cp == ' ' || *cp == '\t')
   cp++;
tp = cp;

if(*cp == '\0')
   {
   tx = (int) EOLN_TOKEN;
   return tx ;
   }
tx = TokenValue[(int) *cp ];
if(tx == ID)
   ProcessId(context);
else if(tx == INTEGER)
   ProcessInteger( );
else if(tx == ERR_TOKEN)
   while(TokenValue[(int) *cp++ ] == ERR_TOKEN)
         ;
else
   cp++;
return tx;
}
/*--END FUNCTION--(NextToken)---------------------------------------------*/
```

Figure 4-27. NextToken code

4.3.5 The Symbol Table

The symbol table stores the relevant information about user identifiers for later use. The two primary symbol table functions are QuerySymbol, used to get information concerning a given symbol, and EnterSymbol, which is used to enter symbols into the table. Both functions return a pointer to the appropriate entry in the table, allowing the calling routine to retrieve whatever information may be desired concerning the symbol. QuerySymbol returns the value NULL if the symbol is not found. The function CheckSymbolTable is called at the end of pass 1 to determine if there are any undefined symbols in the table. `ListSymbols` is used to create the symbol table listing.



Symbol Table Functions

QuerySymbol
EnterSymbol
CheckSymbolTable
ListSymbols

Figure 4-28. Symbol table access functions

4.3.5.1 *The Symbol*

The C struct used to hold the symbol information is shown below:

```
typedef       struct        SymTag
        {
        char          id[ID_LENGTH+1];
        int           LineNum;
        int           type;
        int           s, x, y;
        struct        SymTag         *link;
        } SymType;
```

It holds the identifier string (`id`), the line number where the symbol was defined, the symbol type, and the appropriate `s`, `x`, or `y` parameter values. The pointer `link` is used to build the table since the symbols are stored as linked lists.

4.3.5.2 *Table Implementation*

The table itself consists of an array of 26 pointers to a `SymType` struct, one for each letter of the alphabet (see Figure 4-29). Each symbol is entered into the appropriate string, in the order in which it is found. This scheme gives better performance than a single list since the average length of each list is less than if a combined list were used. A tree was considered (in fact, earlier versions of PASM used a tree) but the performance improvement of the tree did not seem justified by the increased code complexity. The `QuerySymbol` and `EnterSymbol` functions use the symbol pointers as their entry to the table, however, since the functions `CheckSymbolTable` and `ListTable` operate on the entire table, they are implemented as simple for loops on the actual table memory.

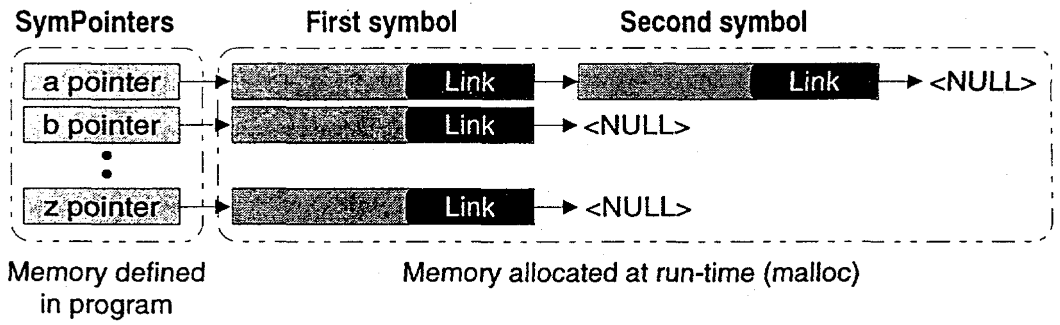


Figure 4-29. Symbol table structure

The memory for the symbol table, except for the 26 pointers, is dynamically allocated at initialization. The file `symbol.c` contains the following variable definitions for the table:

```
int                    NumEntries = 0;
static SymType         *SymbolMemory;
static SymType         *SymPntrs[26];
```

The initialization code is given below:

```
void   InitSymTable(void)
{
int          i;

SymbolMemory = (SymType *) malloc ((MAX_SYMBOLS) * sizeof(SymType));
if(SymbolMemory == NULL)
       HandleException(em1);
for(i = 0; i < 26; i++)
       SymPntrs[i] = NULL;
}
```

The number of symbols is given by the constant `MAX_SYMBOLS`, contained in the header file. It is currently set to 750, however; the `-n` option may be used to specify a different table size at run-time if desired. The reason for dynamically allocating the memory is to make it easier to adjust the size of the table. The `QuerySymbol` and `EnterSymbol` routines (described in following sections) could allocate space as required, which would give the smallest possible table. The disadvantage of this approach is that the malloc operation is relatively slow and the assembly time would increase. In addition, two of the symbol table functions, `CheckSymbolTable` and `ListSymbols`, treat the table as an array, requiring that the table be contiguous. This would not be the case if multiple calls to malloc were made. `CheckSymbolTable` and `ListSymbols` may be modified to search the table as a linked list, removing this restriction, although performance will decrease.

4.3.5.3 The QuerySymbol and EnterSymbol Function

FUNCTION	WHERE CALLED	PURPOSE
`QuerySymbol`	`pass2 (main.c)`	Checking labels to be sure they are in table.
	`primary (parser.c)`	Checking identifiers to determine if they are in the symbol table while parsing parameters.
`EnterSymbol`	`pass1 (main.c)`	Entering labels into the symbol table
	`GetDefineName (parser.c)`	Processing identifiers associated with DEFC and DEFP. EnterSymbol is used since the identifiers do not have to have been previously defined.
	`ProcessExtern (parser.c)`	Enter identifier associated with an EXTERN statement into the symbol table.
	`primary (parser.c)`	Called if an identifier found while parsing a parameter is not already in the symbol table.

Table 2. Use of `QuerySymbol` and `EnterSymbol`

The `QuerySymbol` function is used to search the symbol table for a given identifier to determine if it is in the table. It returns a pointer to the symbol if found, NULL otherwise. The code for `QuerySymbol` is given below:

```
SymType        *QuerySymbol(char *key)
{
int            found = FALSE;
SymType        *p = NULL;

p = SymPntrs[tolower(key[0]) - 'a'];
while(p != NULL && !found)
       if(strcmp(p->id, key) == 0)
              found = TRUE;
       else
              p = p->link;
if(found)
       return p;
else
       return NULL;
}
```

EnterSymbol first searches the table, using essentially the same code as shown for QuerySymbol. If the symbol is not in the table, then it is added using the code segment given below:

```
if(NumEntries < SymTableSize)
       p = &SymbolMemory[NumEntries++];
else
       HandleException("Symbol table is full");
strcpy(p->id, key);
p->type = type;
p->LineNum = LineNumber;
p->link = SymPntrs[i];
SymPntrs[i] = p;
```

If the symbol is found, then it is checked for consistency using the code shown below:

```
if((type == CONSTANT) || (type == POSITION) || (type == EXTERN))
       RecordError(es10, ERROR);
else if(type == LABEL)
       {
       if(p->type == UNDEFINED)
              p->type = type;
       else if(!(p->type == EXTERN))
              RecordError(es10, ERROR);
       }
else if(type == UNDEFINED)
       if(!((p->type == LABEL) || (p->type == CONSTANT)))
              RecordError(es13, ERROR);
```

The various checks on the symbol type are required to ensure that the symbol has been correctly defined. For instance, symbols which are defined by DEFC (CONSTANT) or DEFP (POSITION) should not already be in the table. They will be found by QuerySymbol when they are used in parameters.

It should be noted that these checks imply a coordination between the use of `EnterSymbol` and `QuerySymbol` which is not immediately evident. This should be cleaned up in future releases of PASM.

4.3.5.4 The CheckSymbolTable and ListSymbols Functions

Both CheckSymbolTable and ListSymbols are operations on the entire symbol table. Both functions contain a loop of the form:

```
for(i = 0, p = SymbolMemory; i < NumEntries; i++, p++)
        <perform appropriate operation on symbol>
```

The operation performed by CheckSymbol is to determine if the symbol is defined and to print a message for each undefined symbol. The code to perform this operation is:

```
if(p->type == UNDEFINED)
        {
        if(!flag)
                {
                printf("\tUndefined Symbols:\n");
                flag = TRUE;
                }
        printf("\t[%s] (line %d) is undefined at end of pass %d\n",
                p->id, p->LineNum, pass);
        }
```

The variable `flag` is set the first time an error is found so that the header "Undefined Symbols" will only be printed once. The code for ListSymbols is more complex due to the various formatting commands but it is relatively straight forward.

4.3.6 The Parser

The parser has two major entry points, `ParseCommand` and `ParsePseudop`, both called by the main program (`pass1` or `pass2`) for the appropriate pass. `ParseCommand` is called if the input line contains a machine command while `ParsePseudop` is called if the input line contains an assembler command (or pseudop). Once the appropriate parser function is called, the parser then performs all of the remaining activities to determine appropriate action for the current source line. `ParseCommand` is largely table driven since each K2A command has the same structure while `ParsePseudop` has separate routines for each assembler command. Both `ParseCommand` and `ParsePseudop` depend on the routine `expression` to process any arguments present in the current line.

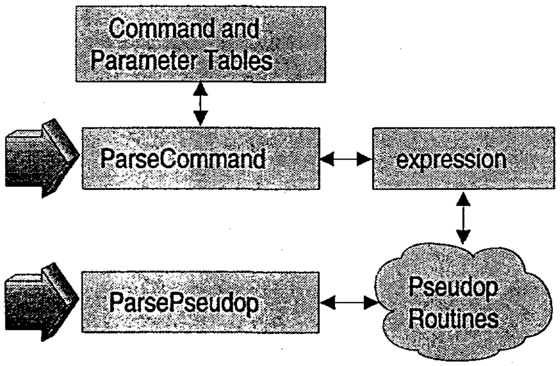


Figure 4-30. Parser structure.

4.3.6.1 The Semantic Record

Information concerning command arguments is passed in a semantic record, which is of the form:

```
typedef       struct
              {
              int    type;
              int    s, x, y;
              } SRType;
```

The type refers to the parameter type which is defined by the following typedef:

```
typedef       enum
              {
              S_TYPE, C_TYPE, CP_TYPE, P_TYPE, ERR_TYPE
              } ParTypes;
```

The various parameter types are given in table 3.

TYPE	DESCRIPTION
S_TYPE	A single byte constant, applicable only to the S parameter.
C_TYPE	A single word constant, applicable to either the X or Y parameter.
CP_TYPE	A single word constant (X or Y) or two word constants representing X and Y.
P_TYPE	A pair of word constants representing both X and Y (a position value).

Table 3. Description of the parameter types

When `expression` (and a number of the functions contained within `expression`) is called, the expected type is passed as a parameter, supplying the information required for proper processing. Since the use of a single position parameter (as opposed to an `X` and a `Y` parameter) is optional, the type `CP_TYPE` is always passed when processing an `X` value which could also be a position value. expression returns the actual type, which will be either a `C_TYPE` or `P_TYPE`, in this case. An error value (`ERR_TYPE`) is also defined and is returned if the parameter can not be properly evaluated. Two pre-defined semantic record values, `ErrSR` and `NullSR`, are defined as given below:

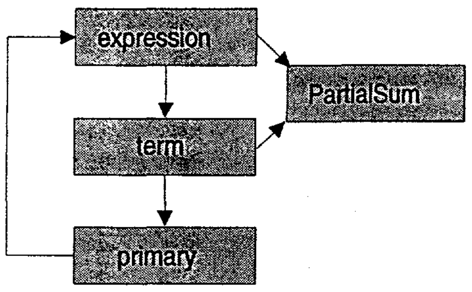
```
static  SRType      ErrSR = { ERR_TYPE, 0, 0, 0 }, NullSR = { 0, 0, 0, 0 };
```

`ErrSR` is returned by expression if an error occurs and `NullSR` is passed as a parameter when the applicable parameter is not required.

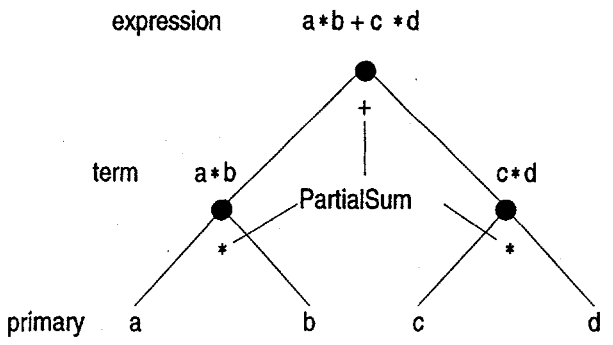
4.3.6.2 The Expression Evaluator

The expression evaluation routine is the most complex code in the entire program. It is based on the following portion of the PASM BNF:

<expression> ::=	<unary op> <term> {<add op> <term>} ;
<unary op> ::=	'+' | '-' | *empty* ;
<term> ::=	<primary> {<mult op> <primary>} ;
<primary> ::=	<number> | <identifier> | '(' <expression> ')' ;
<number> ::=	<number value> <unit> ;
<unit> ::=	*empty* | <unit specification> ;
<unit specification> ::=	'm.' | 'ft.' | 'cm.' | ,'in' ;
<mult op> ::=	'*' | '/' ;
<add op> ::=	'+' | '-' ;



(a) expression evaluator structure



(b) expression evaluator operation

Figure 4-31. Expression evaluator structure and operation

The general structure and operation of the expression evaluator is shown in Figure 4-31. It consists of four main functions, `expression`, `term`, `primary`, and `PartialSum`. The function `expression` is the entry routine and handles unary operators and "add ops" (addition or subtraction). It first calls the function `term` to handle "mult ops" (the terms formed by multiplication or division) for both the left and right operands since the mult ops have a higher precedence. After all of the mult ops have been completed for both operands, *expression* calls the function `PartialSum` to perform the appropriate (+ or -) operation. The function `term` operates in a similar manner, in that it first calls the function `primary` to evaluate the left and right operands, then calls `PartialSum` to perform the indicated (* or /) operation. The function `primary` handles parenthesized expressions (indicated by the indirectly recursive call to `expression`) as well as returning the value of numbers and variables. This allows the use of nested expressions to any desired depth. Information is propagated between these functions by use of semantic records (`SRType`).

```
if(token == MINUS)
        LeftOp = PartialSum(NullOp, MINUS, LeftOp);
else
        LeftOp = term( );
do
        {
        if(token == PLUS || token == MINUS)
                {
                op = token;
                token = NextToken( );
                }
        else
                return LeftOp;
        RightOp = term( );
        LeftOp = PartialSum(LeftOp, op, RightOp);
        }
while(token != NULL_TOKEN);
```

Figure 4- 32. Simplified `expression` code.

The general form of `expression` is given in Figure 4-32. Both the `LeftOp` and `RightOp` are semantic records. All of the code for error checking and handling optional commas has been removed. The function `PartialSum` performs the actual arithmetic operation and returns the appropriate value. The plus (+) and minus (-) operations have a similar form and are treated in the same fashion syntactically. The actual operand value is used by `PartialSum` to select the appropriate operation. A similar simplified form of `term` is given in Figure 4-33. As with the `expression` code, a number of details such as error checking, have been omitted.

```
LeftOp = primary( );
do
        {
        if(token == MULTIPLY || token == DIVIDE)
                {
                op = token;
                token = NextToken( );
                }
        else
                return LeftOp;
        RightOp = primary( );
        LeftOp = PartialSum(LeftOp, op, RightOp);
        }
while(token != NULL_TOKEN);
```

Figure 4-33. Simplified `term` code

Both of these modules are directly implemented from the BNF. Consider the first two lines of the BNF, which are:

<expression> ::=	<unary op> <term> {<add op> <term>} ;
<unary op> ::=	'+' | '-' | *empty* ;

The if statement at the beginning of the code for `expression` handles the optional unary op while the do - while loop implements the repetition indicated by the use of the braces ("{", "}"). The remainder of the expression evaluator (and the parser itself) are implemented in a similar fashion.

This technique is known as recursive descent parsing and is described in section 6, PASM Technology.

```
switch(token)
        {
        case ID:            <process identifier>;
                            break;
        case INTEGER:       <process integer>;
                            break;
        case REAL:          <process real number>;
                            break;
        case L_PAREN:       match(L_PAREN);
                            sr = expression( );
                            match(R_PAREN);
                            break;
        case EOLN_TOKEN:    sr = ErrSR;
                            RecordError(es9, ERROR);
                            break;
        default:            sr = ErrSR;
                            RecordError(es6, ERROR);
        }
token = NextToken(OBJECT);
return sr;
```

Figure 4-34. Simplified `primary` code.

A simplified version of the code for `primary` is shown in Figure 4-34. While this code is quite straight forward, `primary` is a very complex function because of the number of details which must be checked. For instance, `primary` has to search the symbol table for each identifier to retrieve the semantic information or, in pass 1, possibly enter the symbol into the table if it is not defined. Since variables may be addresses or byte values (`S_TYPE`), `X` or `Y` values (`C_TYPE`), or position variables (`P_TYPE`), `primary` must do type checking to determine if the indicated operation is appropriate. This is handled by table lookup using the array `TypeCheck`. `primary` must also check for the presence of units following either integers or real numbers. The semantic record variable (`sr`) is a local variable within `primary` and is set by the appropriate code for the first three cases (`ID`, `INTEGER`, and `REAL`).

A significant portion of the work performed by `primary` involves type checking, or resolving the match between the expected type and the actual type. This is handled by performing an action from the `TypeCheck` array, defined below:

```
typedef       enum
       {
       A1, A2, A3, A4, A5, A6, E1, E2
       } ParserActionTypes;

static char   TypeCheck[3][4] =
       {
       { A1, A2, E1, A3 },
       { E1, A4, E1, E2 },
       { E1, A5, A6, E2 },
       };
```

EXPECTED	ACTUAL TYPE			
TYPE	LABEL	CONSTANT	POSITION	UNDEFINED
S_TYPE	A1	A2	E1	A3
C_TYPE	E1	A4	E1	E2
CP_TYPE	E1	A5	A6	E2

Table 4. Contents of the TypeCheck array

The contents of the TypeCheck array are also shown in table 4 with the argument values given. The actual parser actions are given in table 5.

VALUE	ACTION TO BE TAKEN
A1:	sr.s = p->s
A2:	sr.s = p->x
A3:	if(pass == 2) E2
A4:	sr.x = p->x
A5:	sr.x = p->x, sr.type = C
A6:	sr.x = p->x, sr.y = p->y, sr.type = P
E1:	Type error (es13)
E2:	Undefined symbol (es6)

Table 5. Parse actions

The values contained in this array represent a parser action (A1 to A6) or an error condition (E1, E2). When the `primary` routine is called, it is passed a value which gives the expected type. If the actual parameter is a literal, then the type is taken to be `CONSTANT`.

If the parameter is an identifier, then it is looked up on the symbol table and the type stored in the table is used. The value from the `TypeCheck` array, using the expected type and the actual type as indices, is then used as the selector of a switch statement to select the appropriate action. This code is shown in Figure 4-35.

```
switch(TypeCheck[type][p->type])
        {
        case A1:        sr.s = p->s;
                        break;
        case A2:        sr.s = (int) p->x;
                        break;
        case A3:        if(pass == 2)
                                {
                                RecordError(es6, ERROR);
                                sr = ErrSR;
                                }
                        break;
        case A4:        sr.x = p->x;
                        break;
        case A5:        sr.x = p->x;
                        sr.type = C_TYPE;
                        break;
        case A6:        sr.x = p->x; sr.y = p->y;
                        sr.type = P_TYPE;
                        break;
        case E1:        RecordError(es13, ERROR);
                        sr = ErrSR;
                        break;
        case E2:        RecordError(es6, ERROR);
                        sr = ErrSR;
                        break;
        default:        HandleException("Type check error");
        }
```

Figure 4-35. Switch statement used with the TypeCheck array

The function used for parameter checking, `ParameterCheck`, is also considered part of the expression evaluation code. This routine, whose code is given in Figure 4-36, is passed the parameter value (`ParValue`), the parameter identity (`S`, `X`, or `Y`, `par`), and the parameter type (`type`). It uses the information in the parameter limit table (`ParLimits`) to determine if the parameter is within the legal range and issues a warning if not.

```
/*--BEGIN FUNCTION--(ParameterCheck)---------------------------------------*/

static void   ParameterCheck(long ParValue, int par, int type)
{
if(ParValue < ParLimits[type].NegMin)
        {
        sprintf(ErrBuf, "%s parameter [%ld] < negative minimum value [%ld]",
                parameters[par].name, ParValue, ParLimits[type].NegMin);
                RecordError(ErrBuf, WARNING);
        }
else if(ParValue > ParLimits[type].NegMax && ParValue < ParLimits[type].PosMin)
        {
        sprintf(ErrBuf, "%s parameter = [%ld] not [%ld - %ld] && [%ld - %ld]",
                parameters[par].name, ParValue, ParLimits[type].NegMin,
                ParLimits[type].NegMax, ParLimits[type].PosMin,
                ParLimits[type].PosMax);
        RecordError(ErrBuf, WARNING);
        }
else if(ParValue > ParLimits[type].PosMax)
        {
        sprintf(ErrBuf, "%s parameter [%ld] > maximum value [%ld]",
                parameters[par].name, ParValue, ParLimits[type].PosMax);
                RecordError(ErrBuf, WARNING);
        }
}
/*--END FUNCTION--(ParameterCheck)-----------------------------------------*/
```

Figure 4-36. Parameter checking code

4.3.6.3 Parsing of Machine Commands

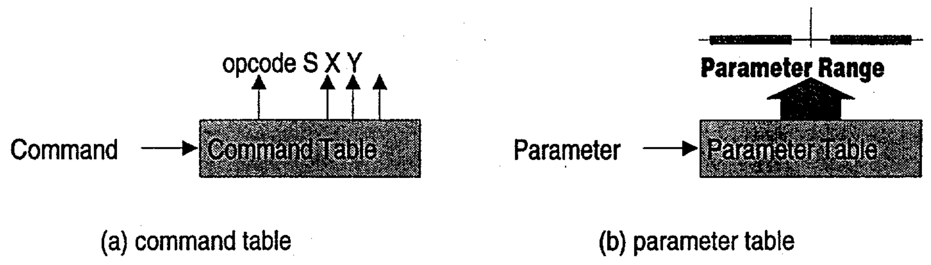


Figure 4-37. Primary parser tables

`ParseCommand` uses two tables (see Figure 4-37), the command table and the parameter table. The command table has entries of the form

```
typedef         struct
                {
                char    opcode;
                char    s;
                char    x;
                char    y;
                } CommandType;
```

for each command token value. The opcode is generally the same as the token value but it is placed in the table for generality. The s, x, and y values give the parameter type or have a value of 0 if the parameter is not used with the command. The *current*[4] parameter values are:

```
typedef         enum
                {
                NONE, ACCEL_DRV, ACCEL_STR, AZIMUTH, BEGREES, S_WORD, U_WORD, U_BYTE,
                S_WORD_NZ, DISTANCE, DIST_WALL, DIST_TURN, DIST_APPR, DIST_DOCK,
                DIST_UNDOCK, DOCK_NUM, R_SPEED, STEERV, DEFLECT, DEFLECT_RATE, SPI_MODE,
                S_512, S_999, BINARY, DIST_JUNK, DOOR_TYPE, S_700, DRIVE_CUR,
                STEER_CUR
                } ParameterTypes;
```

The parameter types are based on the allowable range of values for that parameter. This range of values is stored in the parameter table and is used to check to determine if the parameter is within the legal range. Each entry in the parameter table is of the form

```
typedef         struct
                {
                long    NegMin;
                long    NegMax;
                long    PosMin;
                long    PosMax;
                } ParameterLimitType;
```

which allows for separate positive and negative ranges where appropriate. This is the same type used by the `ParameterCheck` routine which was described previously.

[4] The modifier current is emphasized because the parameter values are subject to change and have not been totally defined as of PASM v1.33. This is one of the reasons they are stored in a table.

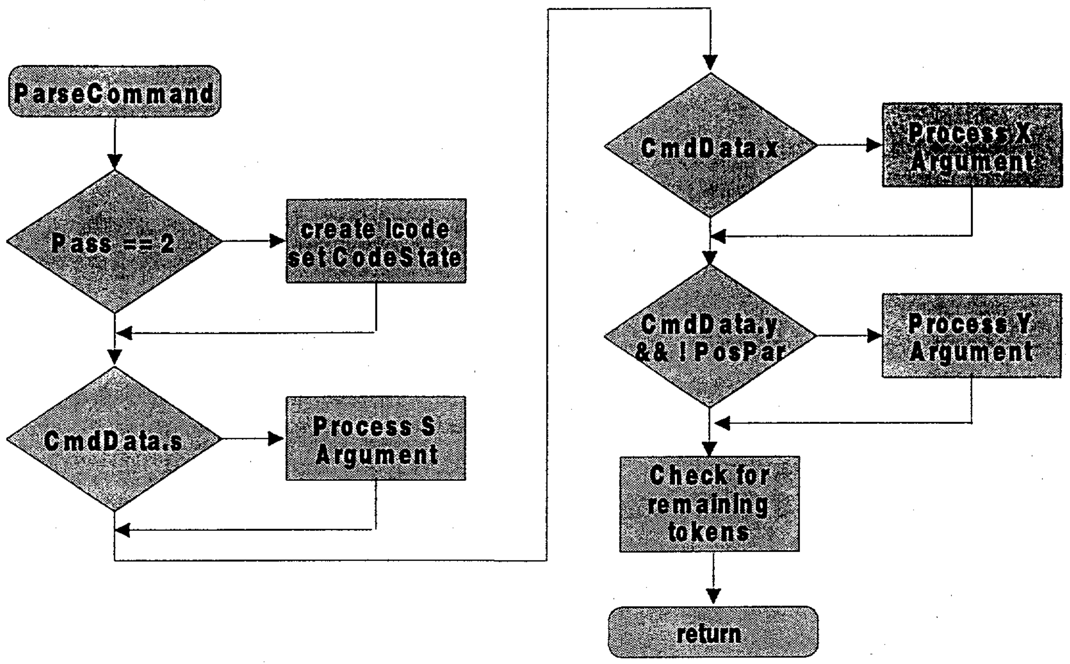


Figure 4-38. Flowchart of ParseCommand

A flowchart describing the operation of `ParseCommand` is given in Figure 4-38. The routine first gets the appropriate information on the command from the command table and places it in the C struct `CmdData`, which has the same type (`CommandType`) as the command table. If it is the second pass then the opcode will be placed in the variable `icode` and the `CodeState` set to `COMMAND_PAR`. The parameters are then parsed by blocks of code having the form

```
if(CmdData.<parameter>)
        {
        sr = expression(<parameter type>)
        if(pass == 2)
                <generate code>
        }
```

`CodeState` is a bit-vector variable used by the `BuildOutput` routine to determine if a value is present or not. `ParseCommand` sets the appropriate bit to one for each parameter present in the output. The corresponding value of the parameter is placed in the variable `icode`. The local variable `PosPar` is set true if a position parameter (`P_TYPE`) is found when processing the `X` parameter. In this case it is not necessary to process the `Y` value.

4.3.6.4 Parsing of Pseudo-Operations

The second entry point into the parser is the function `ParsePseudop`, which is called to handle the occurrence of one of the pseudo-operations `DEFC`, `DEFP`, `DEFD`, `DEFS`, `EXTERN`, `EXTERN_REF`, or `INCLUDE`. `ParsePseudop` is implemented as a switch statement with an entry for each operation. The code `ParsePseudop` is given in Figure 4-39.

`ParsePseudop` was implemented as a switch statement to make it more extensible. New assembler commands may be added by inserting new case statements with little or no impact on the remainder of the code. There are several aspects of the `ParsePseudop` code which are worth noting. First, note that most, but not all, of the case statements begin with a token fetch (`token = NextToken( )`), the two exceptions being `ProcessExternRef` and `ProcessInclude`. Since at least one of the cases did not require a pre-fetch, it was necessary to place individual statements in each case as opposed to a single statement prior to the switch. Second, note that a pointer to the statement label (if any) is passed to `ParsePseudop`. This is to handle the operations, such as `DEFC`, which may be expressed in either of the two formats give below:

```
DEFC          MaxSpeed      100
MaxSpeed      DEFC          100
```

The label pointer (`label`) is passed to `ParsePseudop` (and on to the actual code to implement the operation) so that it will be available if needed. If both an operand and a label are supplied, then the operand is used by default.

This code also provides a good example of the use of the exception handler. If `ParsePseudop` is called with an illegal assembler command, it notes the module in which the error was detected and calls `HandleException` to terminate the program. There are no "proper" conditions under which HandleException should be called. Some calls, such as this one, indicate an error in the PASM code. In other cases, such as overflow in the symbol table, the call indicates that a capacity has been exceeded but does not necessarily indicate an error in the program. It would be appropriate to better differentiate between these two cases in later releases of the program.

As with the other entry point into the parser (`ParseCommand`), the end of line token is checked to determine if it is a proper terminator (either a newline or a comment) and a warning is issued if this is not the case.

```
/*--BEGIN FUNCTION--(ParsePseudop)----------------------------------------*/
void        ParsePseudop(int command, SymType *label)
{
switch(command)
    {
    case DEFC:              token = NextToken(OBJECT);
                            DefineConstant(label);
                            break;
    case DEFP:              token = NextToken(OBJECT);
                            DefinePosition(label);
                            break;
    case DEFD:              token = NextToken(OBJECT);
                            SetDriveAccel( );
                            break;
    case DEFS:              token = NextToken(OBJECT);
                            SetSteerAccel( );
                            break;
    case EXTERN:            token = NextToken(OBJECT);
                            ProcessExtern( );
                            token = NextToken(OBJECT);
                            break;
    case EXTERN_REF:        ProcessExternRef(label->id);
                            token = NextToken(OBJECT);
                            break;
    case INCLUDE:           ProcessInclude( );
                            token = NextToken(OBJECT);
                            break;
    default:                HandleException("ParsePseudop logic error");
    }
if(!TerminatorToken(token))
    RecordError(es8, WARNING);
}
/*--END FUNCTION--(ParsePseudop)------------------------------------------*/
```

Figure 4-39. ParsePseudop

4.3.6.5 The DEFC and DEFP Commands

Both `DefineConstant` and `DefinePosition` first call the function `GetDefineName` (see Figure 4-40) to resolve which format applies and to return a pointer to the operand (in the symbol table). `GetDefineName` first determines if the next token is an ID. If so, then during pass 1 the symbol table is checked to determine what, and the symbol is entered if it was not already in the table. The symbol is also checked for type by comparing it to the type passed to the module when it was called (`CONSTANT` for `DefineConstant`, `POSITION` for `DefinePosition`). If the token is not an `ID`, then the label is returned as the operand if it exists, else an error message is generated and `NULL` is returned.

The code for `DefineConstant` is given in Figure 4-41. The first action is to `GetDefineName`, which returns a pointer to the operand. The type input parameter for `GetDefineName` is set to `CONSTANT` since this is the expected appropriate type here. `expression` is then called to evaluate the value and returns the results in a semantic record `sr`, which is defined as a local variable in `DefineConstant`. The code in the if statement which follows causes the values returned by the semantic record to be stored in the symbol table entry for the operand. This code is called on pass 1 since define values must be available at the beginning of pass 2. The test `(p != NULL)` inhibits action if no operand was found by `GetDefineName`. No error message is generated here since this was done by `GetDefineName`.

The code for `DefinePosition` follows the same form but is slightly more complex since two values are involved. No parameter checking is done since the intended use for variables associated with these commands cannot be inferred from the definition.

```
/*--BEGIN FUNCTION--(GetDefineName)-----------------------------------*/

static     SymType      *GetDefineName(SymType *label, int type)
{
SymType    *p = NULL;

if(token == ID)
    {
    if(pass == 1)
           {
           p = EnterSymbol(id, type);
           if(p->type != type)
                  p = NULL;
           }
    token = NextToken(OBJECT);
    }
else if(label == NULL)
    RecordError(es9, ERROR);
else
    p = label;

return p;
}
/*--END FUNCTION--(GetDefineName)-------------------------------------*/
```

Figure 4-40. `GetDefineName`

```
/*--BEGIN FUNCTION--(DefineConstant)-----------------------------------*/

static void   DefineConstant(SymType *label)
{
SRType sr = { 0, 0, 0, 0 };
SymType       *p;

p = GetDefineName(label, CONSTANT);
sr = expression(C_TYPE);
if((pass == 1) && (p != NULL))
       {
       p->type = CONSTANT; p->x = sr.x;        p->y = 0;
       }
}
/*--END FUNCTION--(DefineConstant)-------------------------------------*/
```

Figure 4-41. Code for `DefineConstant`

4.3.6.6 The DEFD and DEFS Commands

The DEFD and DEFS commands are used to set the drive (DEFD) and steer (DEFS) acceleration values. They are similar to DEFC and DEFP but are simplified by the fact that they control the value of program variables (`AccelDrive`, `AccelSteer`) rather than requiring an operand. These commands are implemented by the `SetDriveAccel` and `SetSteerAccel` functions. The code for SetDriveAccel is given in Figure 4-42. `SetSteerAccel` is not shown since it has the same format. Parameter checking is done on these values since the desired ranges are known.

```
/*--BEGIN FUNCTION--(SetDriveAccel)-----------------------------------*/

static void   SetDriveAccel(void)
{
SRType sr = { 0, 0, 0, 0 };

sr = expression(C_TYPE);
if(sr.type == ERR_TYPE)
       RecordError(es6, ERROR);
else
       {
       AccelDrive = (int) sr.x;
       if(mode & PARAMETER_CHECK)
              ParameterCheck(sr.x, X, ACCEL_DRV);
       }
}
/*--END FUNCTION--(SetDriveAccel)-------------------------------------*/
```

Figure 4-42. `SetDriveAccel` code

4.3.6.7 The Include Directive

```
/*--BEGIN FUNCTION--(ProcessInclude)----------------------------------*/

static void   ProcessInclude(void)
{
FILE                *fpinc;
StackElement  x;
char                IncludeFile[FILENAME_LENGTH+EXTENSION_LENGTH+2];

if(GetFileName(IncludeFile))
       {
       if((fpinc = fopen(IncludeFile, "r")) == NULL)
              {
                sprintf(ErrBuf, "[%s] %s", IncludeFile, es3);
                RecordError(ErrBuf, ERROR);
                }
        else
                {
                IncludeLevel++;
                x.LineNumber = LineNumber;
                strcpy(x.filename, CurrentFile);
                x.fp = fp;
                LineNumber = 0;
                fp = fpinc;
                strcpy(CurrentFile, IncludeFile);
                PushIncludeStack(x);
                IncludeChange = TRUE;
                }
        }
else
        RecordError(es16, ERROR);
}
/*--END FUNCTION--(ProcessInclude)-----------------------------------*/
```

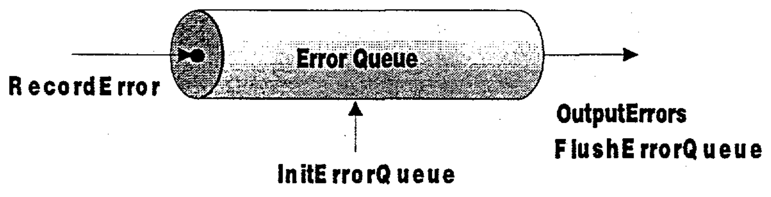
Figure 4-43. `ProcessInclude` code.

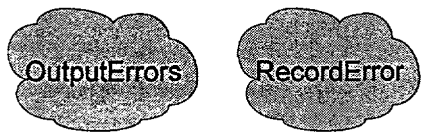
The code to process the include directive, `ProcessInclude`, is given in Figure 4-43. This code uses the scanner function `GetFileName` to read the filename into the program variable `IncludeFile`. `GetFileName` is used rather than `NextToken` because the legal characters for filenames are different from those of program variables. It also includes the actions required to open the file. `ProcessInclude` increments the variable `IncludeLevel` and pushes the current input file state (the line number, file name, and file pointer) onto the include stack. These values are then initialized so that the current file is now the include file and the variable `IncludeChange` is set to flag this change for NextLine so that it can output a new program header.

4.3.6.8 The EXTERN Command

The processing of the `EXTERN` command (`ProcessExtern`) and references to external objects (`ProcessExternalRef`) are included in the code for v1.33 but are not yet well defined. Their description is omitted for this reason.

4.3.7 Error Handler





(a) error queue (b) queue operations

Figure 4-44. Error queue

The primary components of the error handler (see Figure 4-44) are the error queue and several operations that queue, which are provided by the following functions:

```
void    InitErrorPackage(void)
void    RecordError(char *msg, int ErrType)
void    OutputErrors(void)
void    FlushErrorQueue(void)
```

The two primary functions of the error handler are the functions `RecordError` and `OutputErrors`. `RecordError` provides a mechanism to associate an error message with a given token and to place that message in the error queue for later use. `OutputErrors` provides a mechanism to have the error messages either displayed on the screen or placed in the output listing, generally after an entire line is processed. The error messages may contain run-time information if desired.

4.3.7.1 The Error Queue

The error queue is implemented as a singly-linked list. The error queue entries (`ErrorEntry`) are defined by the code given in Figure 4-45. The entry for each message consists of the following items:

- A pointer to the actual error message (`char *msg`)
- The index position of the related token in the input line buffer.
- The error class (`WARNING`, `ERROR`).
- `dynmsg` flag, set true if the message is stored in the error buffer (`ErrBuf`). If the message has been created at run time (stored in `ErrBuf`), then space will have to be allocated for it in the queue.
- `addmsg` flag, set true for other than the first message in the queue. Used when additional messages are suppressed.
- The pointer (`link`) used to construct the queue.

```
/*
!       ERROR QUEUE ENTRY STRUCTURE
!
!       dynmsg is set if the message pointer in RecordError points to
!       ErrBuf.  In this case space is dynamically assigned.  addmsg
!       is set for the second (and each succeeding) error for a given
!       symbol.  This allows the disabling of multiple error messages
!       for a given symbol.
*/
typedef         struct ErrTag
        {
        char            *msg;
        int             ErrPos;
        int             class;
        int             dynmsg;
        int             addmsg;
        struct ErrTag   *link;
        } ErrorEntry;
static ErrorEntry      *head = NULL;
static ErrorEntry      *tail = NULL;
```

Figure 4-45. Error entry and error queue code.

4.3.7.2 Error System Variables

The variables used by the error system are given in Figure 4-46. The variable `ErrorRaised` is a flag which may be tested to determine if there are any messages in the error queue.

The array `ErrBuf[60]` is used to store messages which are created dynamically at run-time. The file pointer `fperr` is static since it is set internally by testing the value of the `mode` and `pass` variables.

```
/*
!       Error system variables which are available externally
!       i.e., have external linkage.
*/
int             ErrorCount = 0;
int             ErrorRaised;
int             WarningCount = 0;
char            ErrBuf[60];
/*
!       Error system variables available in error.c only:
*/
static FILE     *fperr;
static int      FirstError = TRUE;
static int      FirstWarning = TRUE;
static int      LeftMargin;
static int      ErrorSystemInhibited = FALSE;
static int      ErrorSystemDisabled = FALSE;
static int      TabSize;
static int      unclassified = 0;
```

Figure 4-46. Error system variables

4.3.7.3 The Error Messages

Error messages which are used multiple times are stored in error.c so that they can be reused to conserve memory. The error messages as of v1.33 are shown in Figure 4-47.

```
/*
!       ERROR STRING STORAGE:
!       Those error strings which are used multiple times are
!       stored here to conserve memory.
*/
char    *es1  = "\tUSAGE pa { options } < source file >";
char    *es2  = "\t      pa -h for help";
char    *es3  = " could not be opened";
char    *es4  = "Illegal command";
char    *es5  = "Exceeds maximum number of extern entries";
char    *es6  = "Undefined symbol" ;
char    *es7  = "Input line truncated" ;
char    *es8  = "Token after end of statement" ;
char    *es9  = "Expecting parameter" ;
char    *es10 = "Symbol previously defined" ;
char    *es11 = "Possible missing comma" ;
char    *es12 = "Number of commands exceeds 255" ;
char    *es13 = "Type error" ;
char    *es14 = "Improper use of keyword" ;
char    *es15 = "/0 error" ;
char    *es16 = "Invalid filename" ;
```

Figure 4-47. Error message strings

4.3.7.4 The InitErrorPackage Function

The function InitErrorPackage is called at the beginning of each pass to initialize the various internal variables of the error system. This code follows the general form:

```
<initialize simple variables>
if(pass == 1)
        if(DEBUG mode)
                <set operating parameters for stdout>
        else
                ErrorSystemInhibited = TRUE;
else
        if(LISTING mode)
                <set operating parameters for listing>
                else if(DEBUG mode)
                        <set operating parameters for stdout>
```

when the simple variables are the counts and the flags. The operating parameters, which include the error file pointer (`fperr`), the tab size (`TabSize`), and the value of the left margin (`LeftMargin`), are a function of the pass and the mode variable. The error system is inhibited during pass 1 unless the `DEBUG` mode is set, in which case the errors are sent to the `stdout`. If a listing file is being created, all errors detected in pass 2 are sent to the listing file. If there is no listing file and `DEBUG` mode is set, all errors detected in pass 2 are reported to `stdout`.

4.3.7.5 The RecordError Function

The `RecordError` function is used to insert messages in the error queue. `RecordError` is passed an error message and a classification for the message (`WARNING` or `ERROR`). PASM uses one-token lookahead, which means that when an error is detected, it is always applicable to the current token. Since the position of the current token is known (the token pointer variable, `tp`, from the scanner), it is not necessary to pass this information to `RecordError`.

The `RecordError` function contains a large number of details since it must test a number of logical conditions, however the general flow of the code, given below, is straight forward.

```
<determine if message should be accepted>
<record type and increment appropriate count>
<test for maximum number of errors (ERROR_MAX)>
if(RecordInhibited || ErrorSystemInhibited)
        return;
<set addmsg>
if(msg == ErrBuf)
        {
        dynmsg = TRUE;
        <allocate space>
        }
ErrPos = (tp - line);
<link message into queue>
ErrorRaised = TRUE;
```

4.3.7.6 The OutputErrors Function

The `OutputErrors` function removes messages from the queue, frees the space used by the variable, and outputs it as determined by the operating variables. If the message is sent to the listing, `OutputErrors` also increments the line number variable (`LineNumber`) so that the correct number of lines will appear on the page.

When `OutputErrors` sends messages to the listing, it includes a marker (the "^" character) at the beginning of the applicable token. The message is written on the line below the marker, so the message actually occupies two lines. There are two major formatting problems. First, the starting point (the left margin) must be known. This is computed by the initialization routine and stored in the variable `LeftMargin`. The second problem is computing the tab stops. Since tabs are stored as a single ASCII code, the index position of a variable in the input line is not the correct actual location when tabs are used. The tab size (`TabSize`) is computed by the initialization routines. The function `PrintMarker` uses this information to compute the number of spaces required to correctly position the marker.

4.4 OFF-BOARD SYSTEMS

4.4.1 Introduction

The off-board systems consist of *(i)* control software, used to perform the actual inspection mission, *(ii)* programming tools, used to create the code used to perform an inspection mission, *(iii)* database software, used to maintain records derived from an inspection mission, and *(iv)* communications software, used to send information to and from the robot. The off-board may be divided in user functions, programs which are directly accessible by the user, and system functions, which support the user functions (Fig. 4-48). The user functions will be described in the section on the user interface.

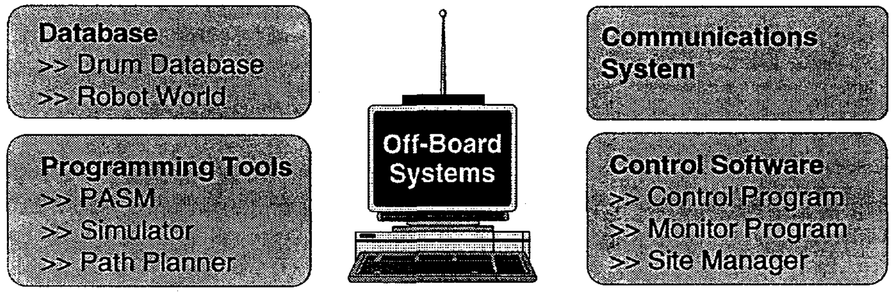


Figure 4-48. Off-Board Software

The control software is functionally and culturally compatible with the equivalent code supplied by Cybermotion with enhancements appropriate to the Unix operating system. The Cybermotion-equivalent functions are primarily used for debugging. The primary function used to control the mission is the Site Manager, which is a specialized version of the Cybermotion Dispatcher. The primary difference is that the Site Manager was created primarily for the warehouse inspection mission whereas the Cybermotion Dispatcher is more general. The Site Manager is a higher level tool and abstracts the user from most low-level details.

The off-board software is implemented in portable C and runs under the Unix operating system. The base user interface code uses standard X-windows while the advanced graphics features are implemented in GL. The reference platform for the base off-board code was a Digital Equipment DECStation 3100. However, the code is designed for portability and has been ported to Hewlett-Packard and Sun systems. The advanced features require a Silicon Graphics system, although they may be ported to any system supporting Open GL.

4.4.2 User Interface

The user interface software is not implemented as a separate sub-system. Each user-software function has its own appropriate interface code. Most interface code is implemented using menus and all functions have a consistent "look and feel." The main menu, shown in figure 4-50, illustrates the basic style used by **ARIES**. The menus associated with the various user functions will be described with that function.

4.4.3 ARIES Main Program

The **ARIES** *main* program is used to control the off-board applications and provides overall control of the robotic vehicle. The UNIX/X-Windows environment allows for distributed interfaces, that is, an application can be processed on one machine and displayed on another. *main* provides controls for selecting which machine processes and which machine displays the interface.

4.4.3.1 Invoking Main

To get familiar with the **ARIES** *main* program, type 'main.' It should already be installed in the **ARIES** installation directory, and the *main* executable should be in the **ARIES** path. Invoking *main* with no options gives the following standard UNIX usage output:

```
Usage: main <Server>
```

As outlined in the usage statement, *main* has only one required argument, the name of the vehicle computer to connect. Each of the **ARIES** vehicles has an on-board computer with a valid ethernet address. The name given this computer for addressing purposes should be passed to *main*.

4.4.3.2 System Configuration

4.4.3.2.1 'system.dat' File

In order for *main* to have information on the configuration of the local computer system, an input file called "system.dat" is used. It contains information on the different architectures and machine names available on the local network, as well as information on the **ARIES** applications executable locations and usage information. Figure 4-49 contains an example "system.dat" file. Each of the bold words are keywords to *main*.

4.4.3.2.2 Applications

Each of the **ARIES** applications are defined in this file. Information is stored about the title of the application, its executable name, and its parameters. Currently supported parameter types are: SERVER, FILE, and MULT_CHOICE.

- SERVER: This option currently is always followed by the keyword NO_CONFIG. *main* will always pass the name of the vehicle specified at start up to each of its child applications.
- FILE: This option is always followed by the description of the file needed, the extension type normally associated with it, and a default filename.
- MULT_CHOICE: Always followed by a description of the parameter and how the actual parameter appears on the command line for the application. Following this is a series of possible values for the parameter.

4.4.3.2.3 Systems

Each system is defined by entering the architecture name (SGI, HP, DEC), followed by the installation directory where the executables reside for the specified architecture.

4.4.3.2.4 Machines

Machines are grouped according to like architecture. They are specified by entering the architecture name as declared in the SYSTEM section, followed by a list of the names of the machines to be used with **ARIES**.

```
APPLICATIONS
{
Application = 'GLX Display'
        {
        exec = "glxdisplay"
        parameters =
                {
                SERVER, NO_CONFIG
                MULT_CHOICE, 'Robot Units', "-RobotUnits",
                        {
                        'inches', "inches",
                        'feet', "feet",
                        'mm', "mm",
                        'meters', "meters"
                        }
                MULT_CHOICE, 'World Units', "-WorldUnits",
                        {
                        'inches', "inches",
                        'feet', "feet",
                        'mm', "mm",
                        'meters', "meters"
                        }
                FILE, 'World iv File', "iv", "../ivs/3d38.iv"
                FILE, 'Robot File', "robot", "../robots/k3a4bar.high.cf"
                }
        }
Application = 'Site Manager'
        {
        exec = "site"
        parameters =
                {
                }
        }
Application = 'X Display'
        {
        exec = "display"
        parameters =
                {
                SERVER, NO_CONFIG
                FILE, 'World DXF File', "dxf", "../dxfs/3d38.dxf"
                FILE, 'Robot DXF File', "dxf", "../dxfs/k2a.dxf"
                }
        }
Application = 'Monitor'
        {
        exec = "monitor"
        parameters =
                {
                SERVER, NO_CONFIG
                }
        }
Application = 'Dispatcher'
        {
        exec = "dispatch"
        parameters =
                {
                SERVER, NO_CONFIG
                FILE, 'Database File', "db", "../PathDBs/squares/squares.db"
                }
        }
}
SYSTEMS
        {
        DEC, InstallDir = "/usr/people/robotics/tphase2/ULTRIX/";
        SGI, InstallDir = "/usr/people/robotics/tphase2/IRIX/";
        SUN, InstallDir = "/home/acct/e501/robot/sun/";
        }
MACHINES
        {
        SGI {enterprise decius excalibur excelsior endeavor}
        DEC {intrepid lexington bones picard}
        SUN {pepper lazarus}
        }
```

Figure 4-49. Example 'systems.dat' file

The systems file consists of three sections: Applications, Systems, and Machines.

4.4.3.3 MAIN Interface

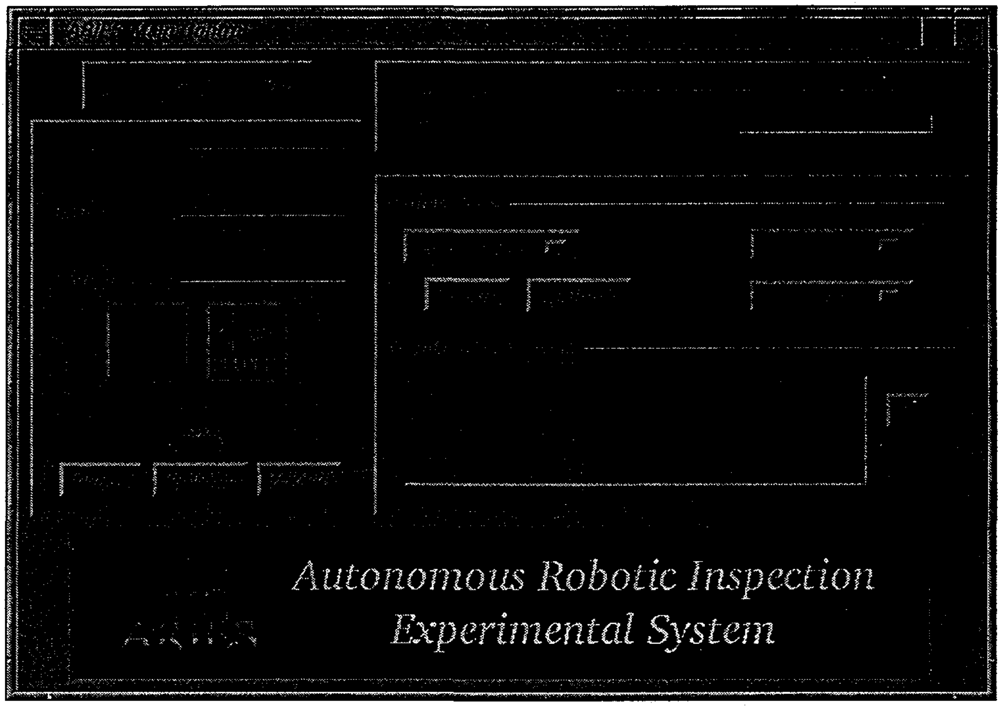


Figure 4-50. **ARIES** main interface

The **ARIES** *main* interface is shown in Figure 4-50. It is divided into several sections: status area, control area, communications status area, and applications area.

4.4.3.3.1 Status Area

The status area provides routine monitoring of the vehicle's status and current mode. These are provided from the communication software running on-board the vehicle at the internet address specified at the command before running *main*.

4.4.3.3.2 Control Area

The most prominent buttons in the control area are "HALT" and "RESUME." "HALT" immediately stops execution of the program being run and stops the vehicle. It will also bring it out of any of the manual modes. "RESUME" will continue the program currently in the vehicle's memory that may have been previously interrupted.

Other modes available under these buttons include MANUAL, REFLEXIVE, and VELOCITY. Refer to the Cybermotion K3A manual for information on these operating modes.

4.4.3.3.3 Communications Area

The communications area informs the user of the status of the comm link between the off-board host and the on-board computer. The load bar graph indicates the amount of available bandwidth being used.

4.4.3.3.4 Applications Area

4.4.3.3.4.1 Application Launcher

An option menu containing all of the available applications pertaining to **ARIES** is used to select the active application. The user is able to configure the parameters passed to this application by pressing on "CONFIGURE," which will pop up a dialog with the parameters specific to the active application.

4.4.3.3.4.2 Host Machine

This option menu contains a list of all of the machines available on the network as described in the "systems.dat" file. They are grouped according to architecture. This option menu controls which machine will do the processing for the application.

4.4.3.3.4.3 Display Machine

This option menu contains a list of all of the machines available on the network as described in the "systems.dat" file. They are grouped according to architecture. This option menu controls to which machine the display of the application will be sent.

4.4.3.3.4.4 Application List

The application list is a widget list containing all of the currently running applications complete with the host and display machine names. The user can kill an application not sent to his console by selecting an application and pressing "KILL."

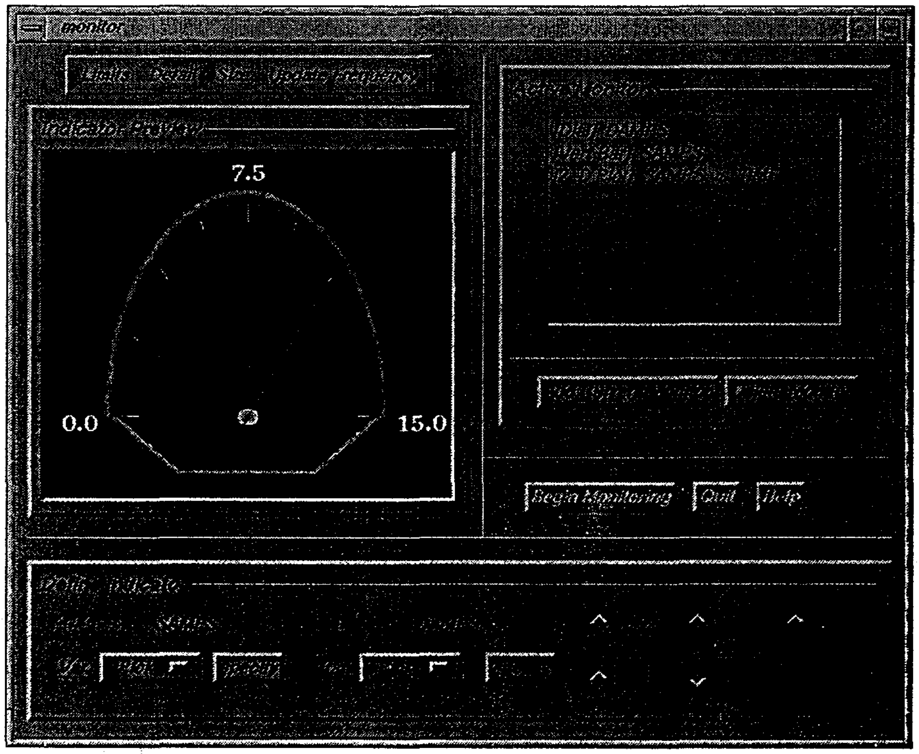


Figure 4-51. Monitor program main interface.

4.4.4 Monitor Program

The Monitor Program was written in X-Windows for use with the Unix control software for the Cybermotion K3A Self-Guided Vehicle. It allows the user to monitor specified variables on board the K3A in real-time. Controls are provided for specifying which variables are to be monitored, including a database with over 450 preset variables, and an interface for defining new variables.

4.4.4.1 Invoking the Monitor Program

To start the monitor program from a Unix prompt, the following steps should be taken:

- make sure the current directory is the installation directory for the appropriate operating system.
- usage of the program is "monitor <server machine> {-display machine:0}".
- <server machine> refers to the machine that is running the robot server process that communicates directly with the robot.
- {-display machine:0} refers to the machine name where the display is to be sent.
- If it is desired for the monitor to function without connecting to the server process, substitute "none" for the server machine.

4.4.4.2 Interface Layout

Figure 4-51 shows the overall interface for the monitor program. It is defined by several self contained regions.

4.4.4.2.1 Define Indicator

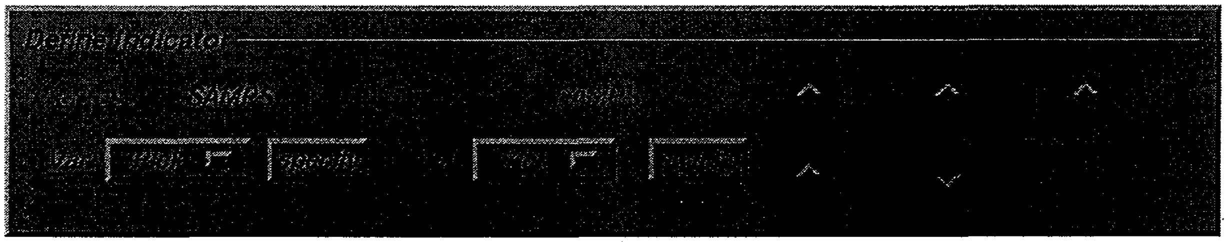


Figure 4-52. Define indicator interface.

The label *Address* in figure 4-52 refers to the current variable selected for monitoring. Underneath this label is an option menu for selecting the variable type. This menu contains the entries *Pick*, *Define*, and *Time*. When specify is clicked, a different popup menu will appear, depending on the option selected in the option menu.

The other option menu grouping labeled "Y-Axis" is used when the *plot* indicator is selected. Because a plot by nature needs a variable per axis, these controls are provided for selecting the second variable for the plot. These controls are only active when the *plot* indicator is active.

The five icons on the far right are used to select the indicator type to be used: *digital, horizontal bar graph, vertical bar graph, dial,* and *plot*. A preview of the indicator graphics is shown in the *Indicator Preview* section.

4.4.4.2.2 Indicator Preview

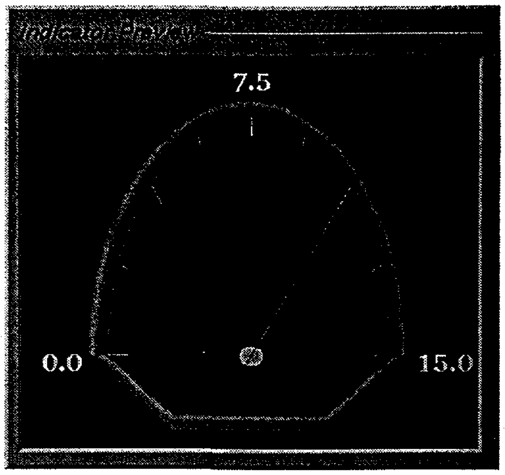


Figure 4-53. Indicator preview interface.

The indicator preview section of the interface shows how the indicator will look when actively monitoring the currently selected variable. The preview displays the current indicator type, using the upper and lower bounds of the currently selected variable.

4.4.4.2.3 Active Monitors List

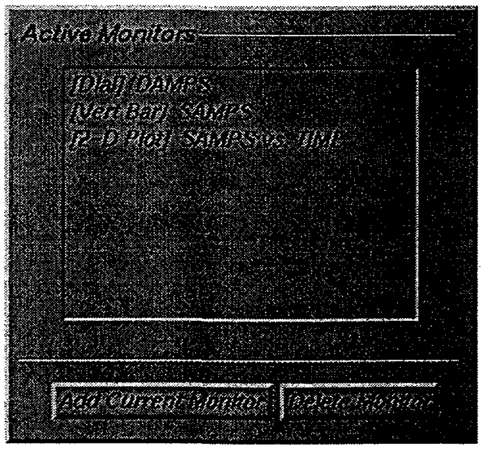


Figure 4-54. Active monitors interface.

Figure 4-54 shows the interface for managing the active monitors. This is a list of the variables, including the indicator type, that will be polled when monitoring is active. The button *Add Current Monitor* will take the current monitor configuration and add it to the list. *Delete Monitor* will remove the selected item from the list. Redundant entries will not be added.

4.4.4.2.4 Pulldown Menu

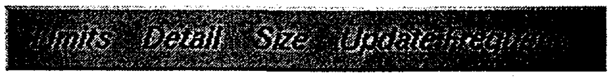


Figure 4-55. Indicator control pulldown menu.

Figure 4-55 shows the pulldown menubar for finer indicator control.

4.4.4.2.5 Control Buttons

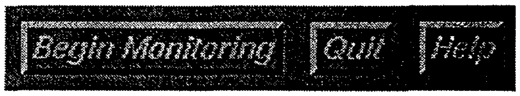


Figure 4-56. Control buttons.

The main controls for the monitor are shown in figure 4-56. They allow the user to *Begin Monitoring* (actively poll the variables from the robot), to *Quit*, or to request *Help* (currently not implemented).

4.4.4.3 Predefined Variables

4.4.4.3.1 Getting Around the Popup Menu

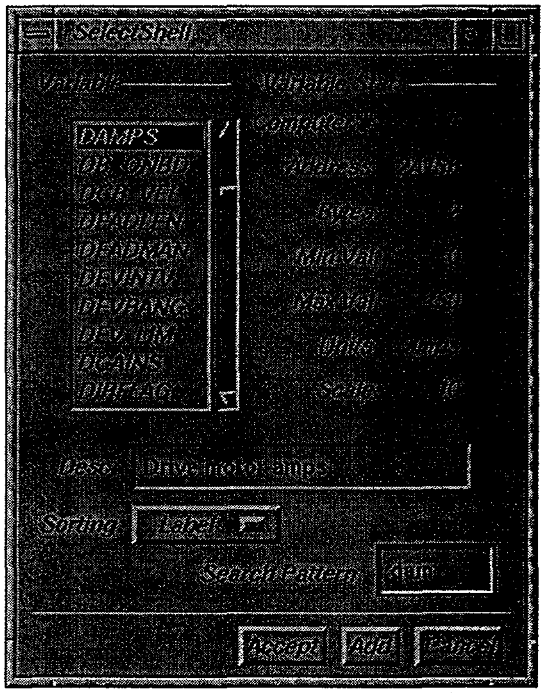


Figure 4-57. Variable select popup menu.

A database containing a set of most variables of the DC-01 is built into the monitor program and accessed through the popup shown in figure 4-57. The variable names are stored in the scrollable list in the top left of the popup. The list is sorted by either alphabetical order of the variable names, or ascending numerical order by the physical addresses of the variables. There is an option menu underneath the list to allow the user to change the sorting method used.

4.4.4.3.2 Navigating the Variable Database

To aid in searching through the database, the text window in the bottom right corner of the popup is used as a search pattern. The program will search through the database for the first occurrence of a variable name beginning with the specified pattern.

4.4.4.3.3 Variable Attributes and Descriptions

Once the user clicks on a variable, different statistics about that variable are given, including *Computer* number, *Address,* number of *Bytes, Minimum* and *Maximum* allowed values, *Units, Scale,* and a short *Description* of what the variable represents.

4.4.4.3.4 Control Buttons

At the bottom of the interface are three control buttons that tie back to the main interface. *Accept* will make the current selected variable the active variable for the main interface, and close the popup menu. *Add* will add the current variable along with the current configuration of the main interface to the *Active Monitors* list, without closing the popup menu. *Cancel* simply closes the popup menu.

4.4.4.4 Defining Variables

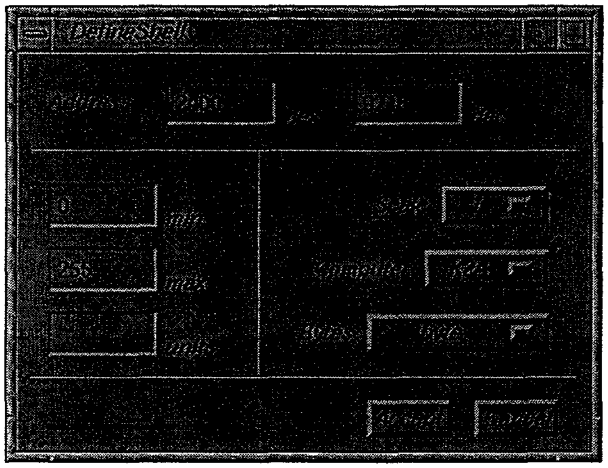


Figure 4-58. Variable definition popup menu.

4.4.4.4.1 Getting Around the Popup Menu

Controls are provided in the popup menu for outlining new variables to be monitored that are not in the existing database. After a variable is defined, *Accept* will accept the current variable as the variable defined in the main interface. *Cancel* will close the popup.

4.4.4.4.2 Specifying a Variable

When entering an address, the user may use either the *Hexadecimal* or *Decimal* text windows located at the top of the popup. Once [RETURN] is pressed in either text window, the other window will reflect the correct value in the respective base.

The option menu for the *Bytes* size is located in the bottom right of the popup. It allows the following choices: *Byte, Signed Byte, Word,* and *Signed Word.* Once a size is selected, the min and max fields on the popup are replaced with the appropriate values for the size and signedness of the variable.

The *Scale* option menu, located just under the *Decimal* text window, is used to specify how the raw data should be interpreted. For example, the drive motor current is read in tenths of an ampere. If the user wishes the readout to be in amperes, then a scale of 10 would be used.

The *Computer* number can be specified by the option menu in the middle right of the popup. It contains verbal descriptions of the different computers on board the robot.

4.4.4.5 Selecting the Indicator Type

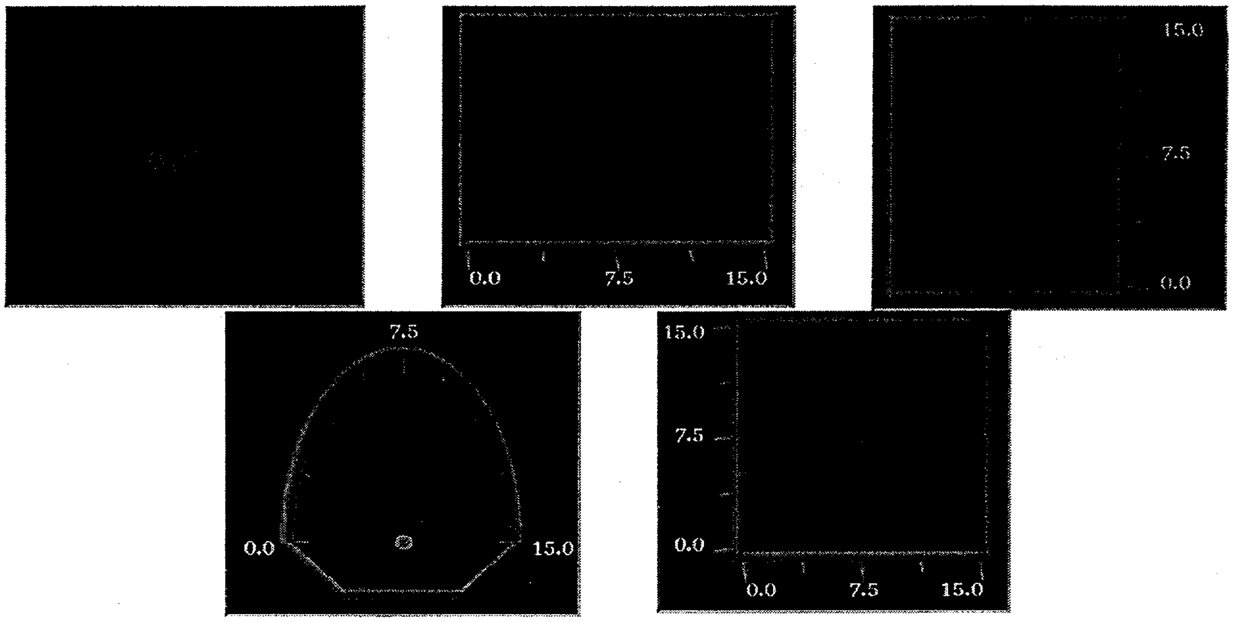


Figure 4-59. Available indicators.

Several different indicator types shown in Figure 4-59 are available through the *Define Indicator* section of the main interface.

4.4.4.6 Active Monitor List

An interface for managing the active monitors is provided. This displays a list of variables, including the indicator type, that will be polled when monitoring is active. The button *Add Current Monitor* will take the current monitor configuration and add it to the list. *Delete Monitor* will remove the selected item from the list.

4.4.4.7 Fine Controls

Controls are provided via the pulldown menu over the *Indicator Preview* section of the interface that allow the user to have more control over each individual indicator.

4.4.4.7.1 Limits

The limits of the indicator can either be fixed or floating. If the limits are fixed, then no value outside of the limits will be drawn, i.e., if the indicator is a bar graph, the bar will not move outside of the predefined minimum and maximum values.

If the limits are set to floating, then monitor will set the limits to be dynamic. That is, the limits will adjust themselves higher or lower depending on the data read from the robot.

4.4.4.7.2 Detail Level

There are two different detail levels: *High* and *Low*. With the *Low* level detail setting, indicators are drawn with thin lines, keeping the graphics processing at a minimal level.

The *High* detail graphics involves some light-shading on thicker lines defining the indicator, as shown in figure 4-60. This requires much more graphics processing and should be used only when CPU time is not a limiting factor.

4.4.4.7.2.1 Customizing Details

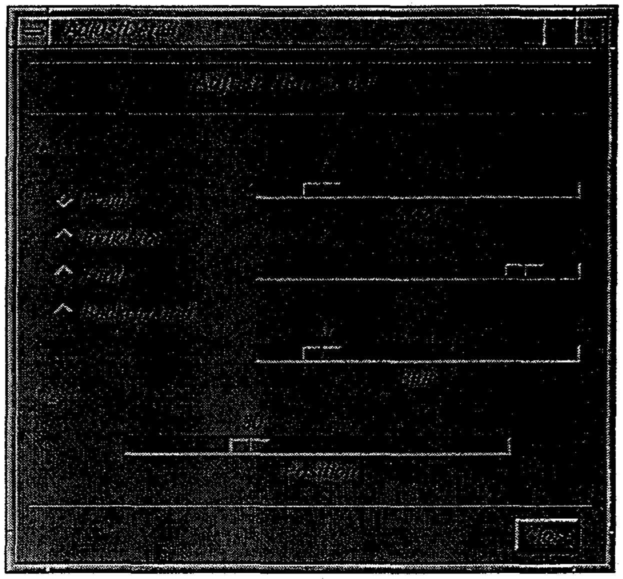


Figure 4-60. Detail customization interface.

Figure 4-60 shows the popup menu associated with customizing the colors and appearance of the indicators. Controls are provided for changing the color of four parts of the indicator: the *Frame*, *Indicator*, *Fonts*, and *Background* colors. Three sliders are provided corresponding to the *Red*, *Green*, and *Blue* components of the color.

The *Light* slider controls where the light highlight falls on the frame on the indicator. As this is used only with thick shaded lines, its effect will go unnoticed on low detail indicators.

4.4.4.7.2.2 Initial Size

The initial size can be specified as a *Small*, *Medium*, or *Large* window. The window can be resized once the monitoring state is active.

4.4.4.7.2.3 Update Frequency

This will control the frequency of updates of a given indicator (currently not implemented).

4.4.4.8 Monitoring Variables

4.4.4.8.1 Entering Monitoring Mode (Begin Monitoring)

When *Begin Monitoring* is pressed, the main interface will close down and a separate smaller window will appear per entry in the active monitor list. Each window will contain its own indicator with the attributes as specified by the *Active Monitors* list.

4.4.4.8.2 Redraw Scheduler

Monitor will redraw an indicator if a window is exposed to the foreground after being obscured by another window, or when a new value is read from the robot.

4.4.4.8.3 Popup Menu

If the third mouse button is pressed over any of the indicators, a popup menu will appear with several options.

4.4.4.8.3.1 Indicator Type

The indicator type can be changed on the fly, with a few exceptions. If the indicator was a plot type, then it cannot become any other indicator, as two variables were specified for the plot, and it is unclear which single variable should be used for the new indicator. Conversely, if the indicator is any other than the plot indicator, it cannot be changed to the plot indicator, as there would only be one variable specified.

4.4.4.8.3.2 Detail Level

The user can toggle the detail level on the fly.

4.4.4.8.3.3 Write to Address

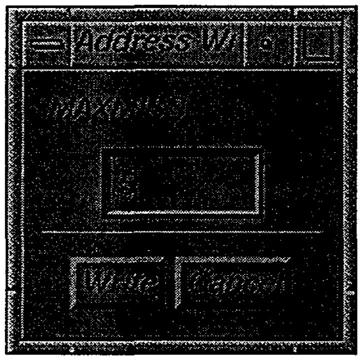


Figure 4-61. Address writing interface.

Figure 4-61 shows the popup window used to write to an address. The user should enter an integer number as it would appear without scaling inside the robot's computer. Negative numbers should be entered as true negative numbers, as monitor will do the conversion automatically. By pressing [RETURN] or pressing the *Write* button, monitor will send the new value to the robot.

4.4.4.8.3.4 Kill Monitor

This allows the user to dismiss an individual monitor while staying in monitoring mode.

4.4.4.8.3.5 Main Menu

This option will close the indicator windows and reopen the main interface, exiting monitoring mode.

4.4.4.8.3.6 Exit Program

This option will exit monitor completely.

4.4.4.9 Example Session

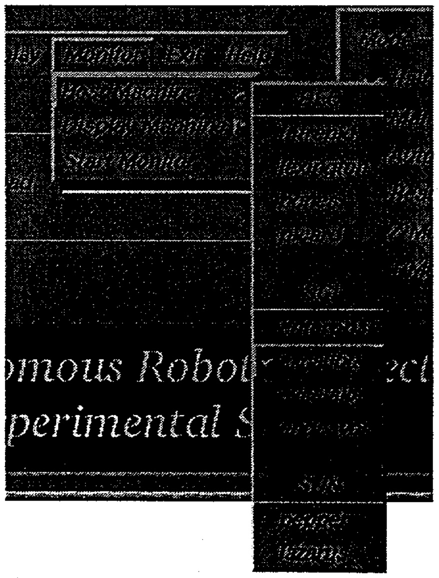
Example Task: Following is an outline containing a series of tasks that allow a user to become familiar with the monitor program:

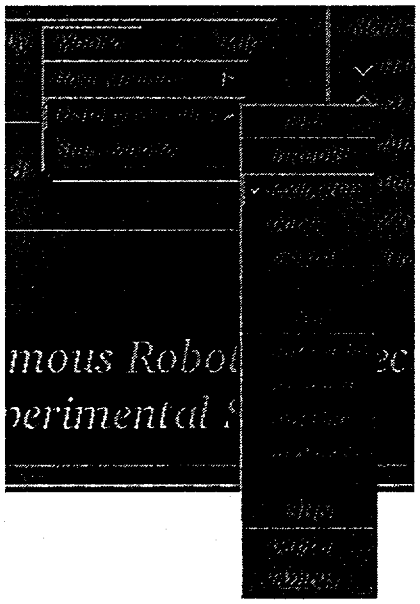
1. Invoke the monitor program from the control program.
2. Define an indicator to monitor the variable DAMPS and add it to the *Active Monitors* list. Define it as a horizontal bar graph with fixed limits, medium size, and high detail.
3. Define an indicator to monitor AZIMUTH vs. Time and add it to the *Active Monitors* list. Define it as a plot indicator with floating limits, large size, and low detail.
4. Define a digital indicator to monitor a variable that is not contained in the database and add it to the *Active Monitors* list. Use the define variables popup to define a variable that has the following attributes: address 2400 hex, minimum value of 0, maximum value 6, using the K2A computer, size is unsigned byte, and the units should read "mode."
5. Start monitoring by pressing *Begin Monitoring.*
6. Once the monitors are running, change the horizontal bar graph indicator to a vertical bar graph indicator via the popup menu.
7. Exit the program via the popup menu.

4.4.4.9.1 Invoke the monitor program from the control program.

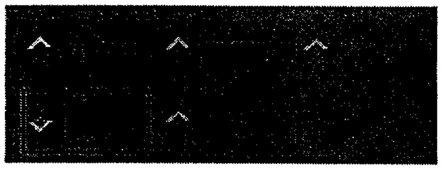
To start the monitor program, simply select *Start Monitor* from the pulldown menu entry *Monitor*. The *Host Machine* can be selected through the same pulldown menu entry. The menu will pull to the right and show a list of all of the workstations set up for running the control software, grouped by architecture type. Select the machine to run the software. The *Display Machine* is selected in the same manner under the *Display Machine* entry.

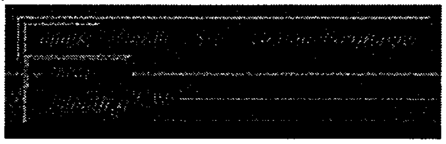
Define an indicator to monitor the variable DAMPS and add it to the *Active Monitors* list. Define it as a horizontal bar graph with fixed limits, medium size, and high detail.





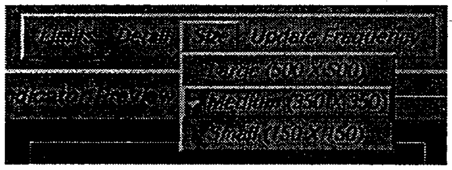
The indicator type selector under the *Define Indicator* section of the interface shows an iconic representation of the available types. Press the button corresponding to the horizontal bar graph indicator.

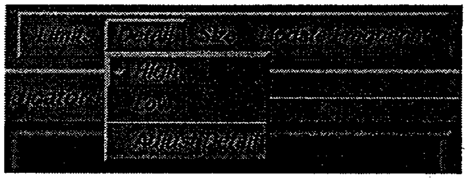




Under the pulldown menu above the *Indicator Preview* section of the interface is an entry labeled *Limits.* Choose the item labeled *Fixed.*

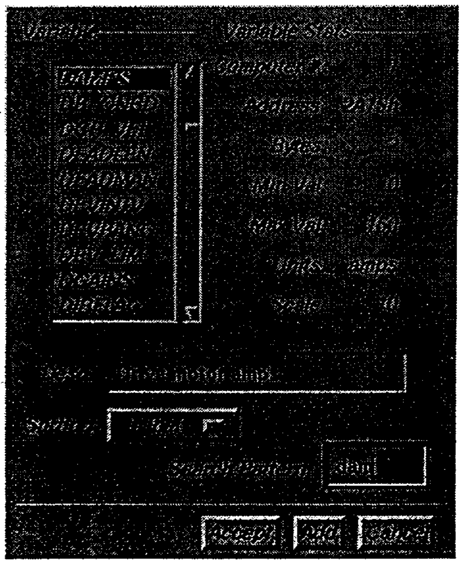
Under the same pulldown menu, there is an entry labeled *Size.* Under this entry, choose the item labeled *Medium (350 X 350).*





Under the same pulldown menu, there is another entry labeled *Detail.* Under this entry, choose the item *High.*

Under the *Define Indicator* section, there are controls for defining the *Address,* or variable, to be monitored. The option menu labeled *Var* contains three entries. Choose the entry *Pick.* This allows the user to pick from a preset database of variables. Click the button *Specify,* this will pop up the Select Variable popup.

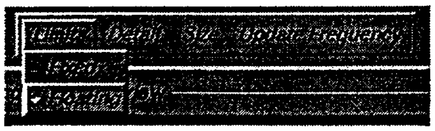


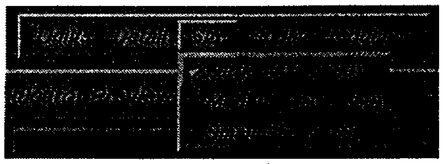
The scrollbar can be used to traverse the database to look for the entry DAMPS. Also, and more effectively, the user can begin typing the name of the variable in the *Search Pattern* text window until the variable is located. Press the control button *Add* to add the variable to the *Active Monitors* list. Press *Cancel* to remove the popup menu.

Define an indicator to monitor AZIMUTH vs. Time and add it to the *Active Monitors* list. Define it as a plot indicator with floating limits, large size, and low detail.

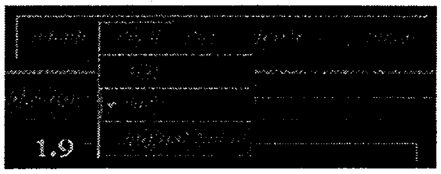
Press the button corresponding to the plot indicator in the *Define Indicator* section of the interface.

Under the pulldown menu above the *Indicator Preview* section of the interface is an entry labeled *Limits.* Choose the item labeled *Floating.*

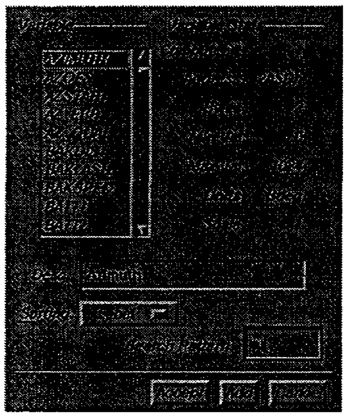




Under the same pulldown menu, choose the entry labeled *Size* and select the item *Large (500 X 500).*



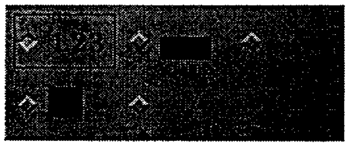
Under the entry labeled *Detail,* select the item *Low.*



Under the *Define Indicator* section, there are controls for defining the *Address* to be monitored. Choose the entry *Pick.* Click the button *Specify* to pop up the *Select Variable* popup menu. Choose the variable AZIMUTH in the same fashion as DAMPS was chosen in section 10.2. Press the control button *Add* to make AZIMUTH the currently active variable. It will be used as the X-Axis for the plot indicator.

Under the *Define Indicator* section, choose the Y-Axis variable as *Time.* This will set up the plot indicator as AZIMUTH vs. Time. Finally, press *Add Current Monitor* under the *Active Monitors* section of the interface.

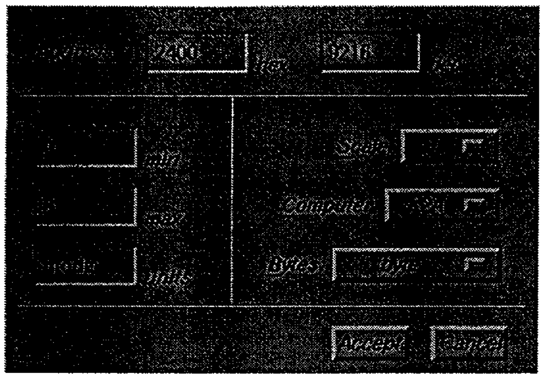
Define a digital indicator to monitor a variable that is not contained in the database and add it to the *Active Monitors* list. Use the define variables popup to define a variable that has the following attributes: address 2400 hex, minimum value of 0, maximum value 6, using the K2A computer, size is unsigned byte, and the units should read "mode."



Press the button corresponding to the digital indicator in the *Define Indicator* section of the interface.

Under the *Define Indicator* section, there are controls for defining the *Address,* or variable, to be monitored. Choose the entry *Define.* This allows the user to define a variable that is not defined in the preset database. Click the button *Specify,* this will pop up the *Define Indicator* popup.

The *hex* and *dec* text windows are used to enter the value of the address to be monitored. They both reflect the same address, using different numerical bases. Enter 2400 into the hexadecimal window and press [RETURN]. The decimal value will automatically update.



Select the size of *byte* from the *Bytes* option menu. The *min* and *max* text windows will automatically reflect the values of 0 and 255 respectively. Change these to 0 and 6 respectively. Choose a scale of 1 from the *Scale* option menu and the *Computer* as *K2A*. Press *Accept* after these selections are made to close the popup.

4.4.4.9.2 Start monitoring by press *Begin Monitoring.*

This will enter monitor into monitoring mode, closing the main interface and raising three separate smaller windows that represent the monitors themselves.

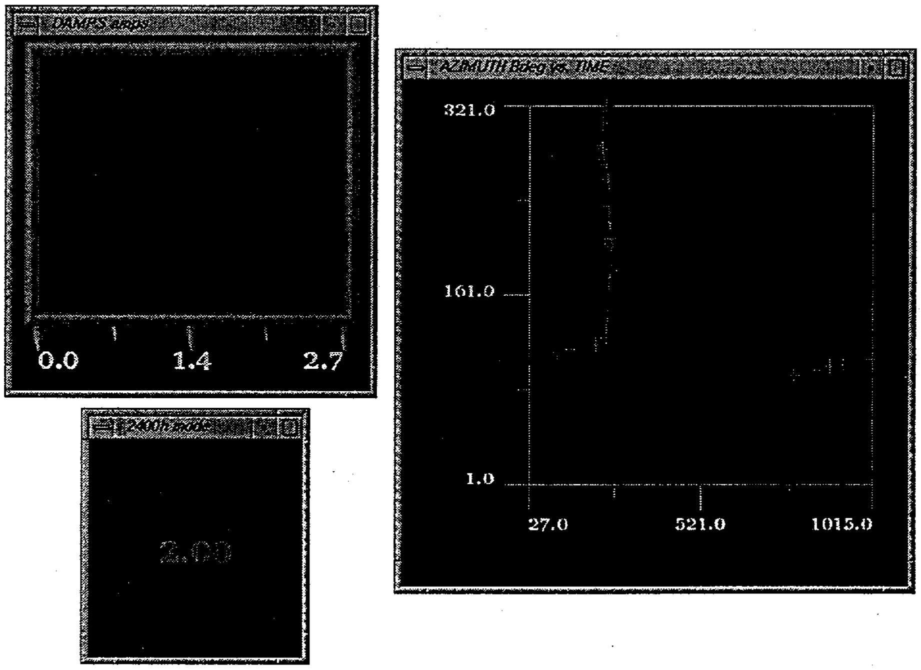
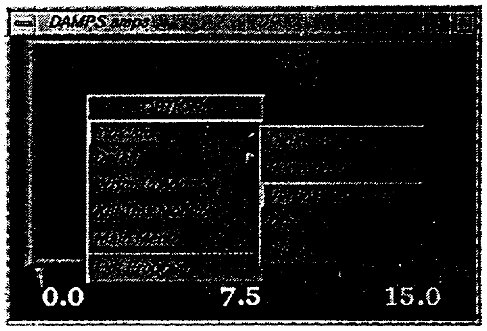


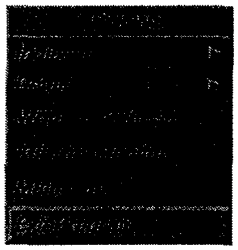
Figure 4-62. Active monitors.

Once the monitors are running, change the horizontal bar graph indicator to a vertical bar graph indicator via the popup menu.

Press the third mouse button over the horizontal bar graph indicator to raise the popup menu. Move to the entry labeled *Indicator*, and another menu pane will appear listing all of the available indicator types. Select the entry vertical bar graph.



4.4.4.9.3 Exit the program via the popup menu.



Press the third mouse button over any of the active monitors to raise the popup menu. Select the entry labeled *Exit Program*. This will exit monitor.

4.4.5 Display Program

The display program provides a graphical means of representing the Cybermotion Navmaster in its environment. AutoCAD files for both the Navmaster and the environment furnish the program with the data necessary for display. As the Navmaster moves along its path, the display program requests and receives position information (x, y, and azimuth) from the robot via the communications library. The position information is then used to render a new scene in the display window. Each scene can be rendered in a variety of methods ranging from a simplistic, two dimensional view to a more complex, three dimensional, solid faced view. In addition to the rendering styles, each scene can be viewed from several perspectives. These perspectives allow the user to view the scene as an outside observer, become a traveller in the Navmaster's environment, or view the scene as the robot itself. Three polygon sorting algorithms are available to enhance the quality of scenes rendered with solid faces. The frequency at which the scene is updated largely depends upon the selected rendering type (i.e., low detail rendering types update more frequently than high detail rendering types). The menu selections are described in detail in section III, Menu Selections.

4.4.5.1 Starting the display program

The following two sections describe how the display program is invoked. The first section, starting display from the control software, is the most common method of invoking the program.

4.4.5.2 Starting Display from the Command Prompt

The first step in starting display from the command prompt is to ensure that the Server from the communications library is currently running on the workstation connected to the R-F modem. If it is not, follow either of the two steps listed below to execute Server.

- Run the Control program using the "run" script. One function of this script is to execute the Server on the appropriate machine. The Control program must remain running if this method is used. If Control is closed, the Server is terminated with it.
- Execute the Server explicitly by entering "Server" from the command prompt in the appropriate installation directory (i.e., ULTRIX for DEC workstations). In order for the Server to function properly, it must be executed on the workstation connected to the R-F modem.

More information about the Server is located in the Communications manual. It is important to note the name of the machine on which the Server is running. The machine name will be used as a command line option for the display program.

Once the Server is running on the appropriate machine, the display program can be executed from the command prompt. Unlike the Server, display can run on any workstation regardless of the R-F modem connection; however, in order to receive communication from the robot there must be an ethernet connection between the workstation running display and the workstation running Server. The steps below describe how to execute display from the command prompt.

- Make sure that the current working directory is the appropriate installation directory. For example, the current working directory for DEC workstations should be the ULTRIX directory.
- A version of the display program compiled specifically for that workstation should reside in the directory (you can verify this by typing "ls" at the command prompt and looking for the "display" file). The syntax for invoking display is as follows:

```
display <server machine> <world dxf file> <robot dxf file>
```

Server machine should be replaced with the name of the workstation that is running the Server (note: both the Server and display can be executed on the same machine). World DXF file should be replaced with an AutoCAD file that describes the Navmaster's environment. Robot DXF file should be replaced with an AutoCAD file describing the robot itself. The command line invocation for the display program with the following conditions:

- The workstation named *bones* is running the Server.
- The world DXF file is located one directory above the current directory and is named *world.dxf.*
- The robot DXF file is located in the current directory and is named *robot.dxf.*

would be:

```
display bones ../world.dxf robot.dxf
```

One advantage to starting display from the command prompt is the "no server" option. Replacing server machine with "none" in the command line syntax will bring the display program up without connecting to the Server. Although no communication is possible in this mode, it is useful for viewing new AutoCAD files without the overhead of or dependency on the Server. More information on AutoCAD files is located in section IV, AutoCAD Requirements.

4.4.5.3 Menu selections

All display options are controlled from a central popup menu. This main menu can be popped up by placing the cursor in the display window and pressing the third (rightmost) mouse button. The main menu consists of four pullright menus and two actions (figure 4-63). The pullright menus can be selected in two ways.

- Press and hold the right mouse button. When the main popup menu appears, position the pointer over one of the pullright menus (denoted by the triangle). The chosen pullright menu will appear and can be traversed in the same manner.
- Press and Release the right mouse button. The main popup menu will appear. Position the pointer over the desired selection and tap the right mouse button. The chosen pullright menu will appear and can be traversed in the same manner. This method of menu traversal is generally the best choice for slower workstations.

The following sections describe each of the pullright menus and their respective selections.

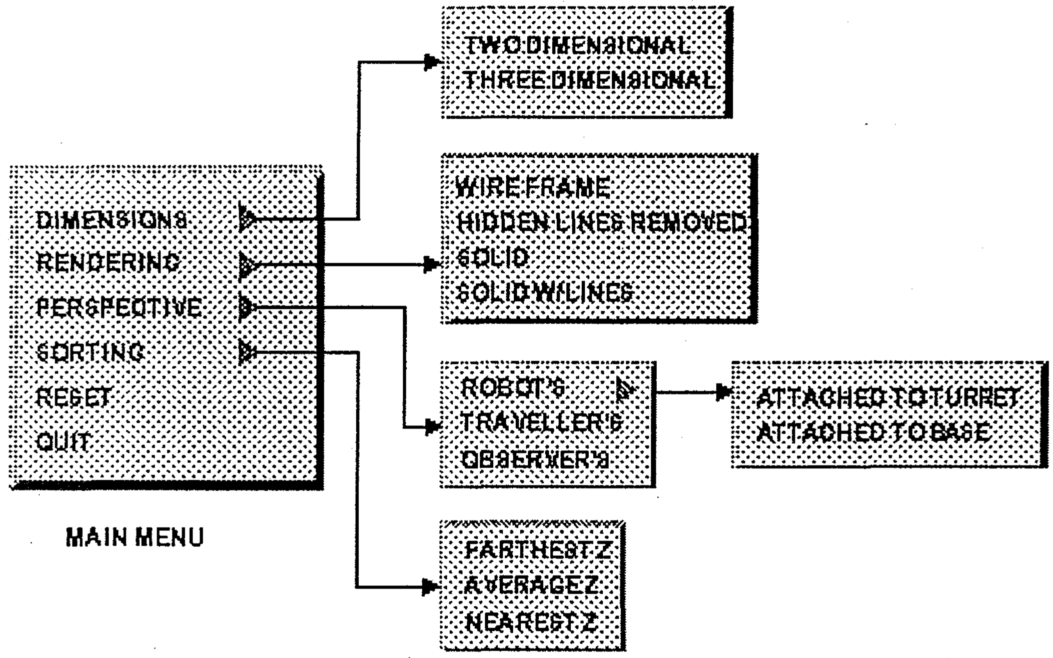


Figure 4-63. The menu tree

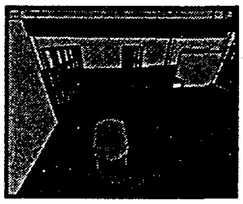
4.4.5.3.1 Dimensions

The *Dimensions* pullright menu controls controls whether each scene will be rendered in two or three dimensions. Notice when two dimensional rendering is selected, several other menu selections (i.e., *Rendering*) become disabled. These options are disabled because they do not apply in two dimensional rendering. The viewing context is saved when switching between two and three dimensional rendering. For example, if rendering were changed from two dimensional to three dimensional, all of the user's settings for two dimensional rendering would be saved. Upon returning to two dimensional rendering, the three dimensional settings would be saved and the two dimensional settings restored.

4.4.5.4 Two Dimensional Rendering

Two dimensional rendering is restricted to a wire frame only, bird's eye view of each scene (more on wire frame in the Rendering section). The viewer's location can be controlled through the Perspective pullright menu.

4.4.5.5 Three Dimensional Rendering

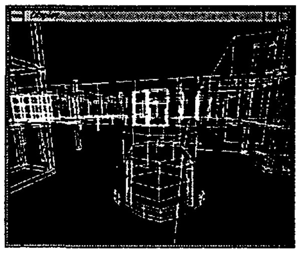


Three dimensional rendering enables each scene to be rendered in a more realistic fashion. Perspective transformations are used to achieve such three dimensional effects as parallax and vanishing points. All display options are available when three dimensional rendering is selected. Display uses three dimensional rendering as the default.

4.4.5.5.1 Rendering

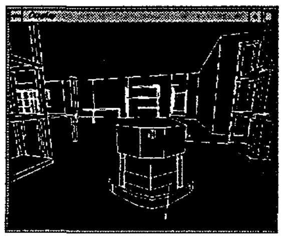
The *Rendering* pullright menu determines which type of rendering is used for three dimensional rendering. The choices are ordered from fastest rendering to slowest rendering.

4.4.5.5.1.1 *Wire Frame*



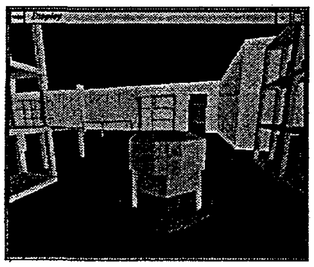
Wire Frame rendering outlines each polygon (solid face) in white. The polygons themselves are not drawn. Because the polygons are not being drawn (much less colored), no sorting or lighting calculations are necessary. Wire frame rendering is automatically selected for two dimensional rendering. Wire frame is the default rendering type.

4.4.5.5.1.2 *Hidden Lines Removed*



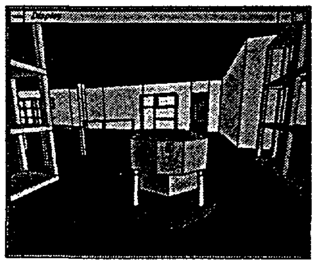
Hidden Lines Removed rendering draws each polygon in the background color as well as outlining them in white. The polygons must be sorted in order for the scene to appear to be composed of solid faces. The resulting scene has the appearance of wire frame rendering where all lines that should be hidden (obscured by solid faces) are not visible.

4.4.5.5.1.3 *Solid*



Solid rendering draws each polygon in a color based on its assigned AutoCAD color and its position relative to the light source. As in *Hidden Lines Removed,* the polygons must be sorted to give the appearance of solid faces. In addition to sorting calculations, lighting calculations must be performed to determine the appropriate shade for each polygon.

4.4.5.5.1.4 *Solid with Lines*

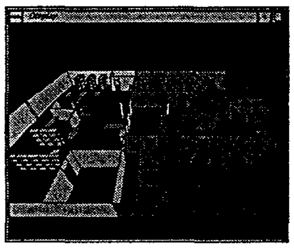


Solid with Lines rendering is similar to *Solid* rendering with the addition of a black outline around each polygon. Because this rendering type must sort, shade, and outline each polygon, it is the slowest of the rendering types.

4.4.5.5.2 Perspective

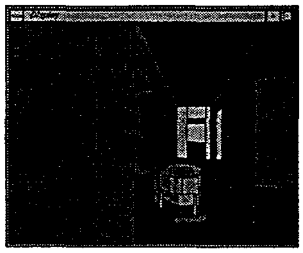
The *Perspective* pull right menu controls the point from which the scene is viewed. One selection, *Robot's*, has an additional pullright menu which allows the viewpoint to be attached to either the base or the turret. The viewing context is saved when switching between perspectives. Upon returning to a previously used perspective, that perspective will retain its last known settings. Section IV, Customizing the Viewpoint, describes how the mouse can be used to change the location of the viewpoint for each of the perspectives.

4.4.5.5.2.1 Observer's



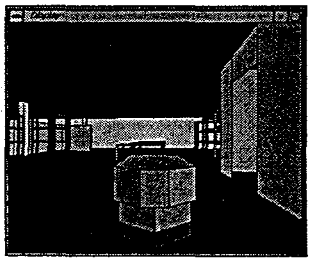
This perspective allows the viewpoint to be positioned anywhere in the scene. *Observer's* perspective is useful for obtaining views of large portions of the robot's environment. *Observer's* perspective is the default perspective.

4.4.5.5.2.2 Traveller's



Traveller's perspective allows the viewer to become a traveller in the robot's environment. Because the viewer is placed in the robot's environment, this perspective is useful for obtaining close up views of the robot's environment. *Traveller's* perspective is not valid for two dimensional rendering.

4.4.5.5.2.3 Robot's



Robot's perspective enables the viewpoint to be attached to the robot itself. Attaching the viewpoint to the robot causes the viewer to stay with the robot as it navigates its environment. When the robot turns, the turret rotates while the base maintains a fixed orientation. The two selections, *Attached to Turret* and *Attached to Base*, allow the viewer to either dynamically view the scene as the turret turns or maintain a fixed view of the robot as it traverses its paths.

4.4.5.5.3 Sorting

The *Sorting* pullright menu determines which sorting algorithm is used to render scenes which display solid faces. Each algorithm attempts to sort the entities (polygons and lines) so that they are drawn in the correct order. For example, entities in the background should be drawn before those in the foreground to ensure that foreground entities are visible. Comprehensive sorting algorithms which always sort the entities correctly are computationally expensive and are not suitable for rapid scene rendering. The algorithms provided by display are much quicker than comprehensive sorting algorithms but fail to sort the entities correctly every time. One of the three algorithms listed below should provide adequate sorting for any given scene.

4.4.5.5.3.1 Farthest Z

The *Farthest Z* algorithm sorts the entities based on the Z coordinate that is the greatest distance away from the viewpoint. Entities with distant Z coordinates are drawn before those with closer Z coordinates. Although long entities tend to cause sorting errors, *Farthest Z* is normally the best choice. *Farthest Z* is the default sorting algorithm.

4.4.5.5.3.2 Average Z

The *Average Z* sorting algorithm averages the Z value of each vertex of an entity and uses this value as the sorting criterion. This algorithm works well for scenes containing large numbers of overlapping entities.

4.4.5.5.3.3 Nearest Z

The *Nearest Z* algorithm uses the nearest Z coordinate of each entity as the sorting criterion. Entities with a distant nearest Z coordinate are drawn before those with a closer nearest Z coordinate. The *Nearest Z* algorithm sorts scenes drawn with no floor well.

4.4.5.6 Reset

The *Reset* option on the main menu resets the current perspective's viewpoint to its default position. Only the current viewpoint is affected.

4.4.5.7 Quit

The *Quit* option on the main menu disconnects the display program from the Server. Once the disconnection is confirmed, display is terminated.

4.4.5.8 Customizing the viewpoint

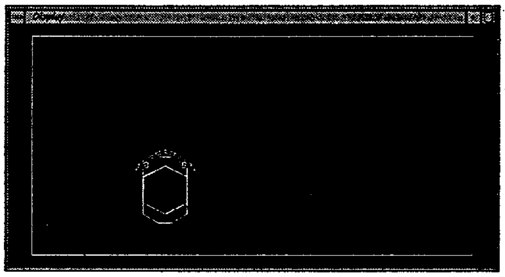
The viewpoint for any given scene can be changed using the mouse. Mouse controls vary for two and three dimensional rendering as well as for the different perspective types within these categories.

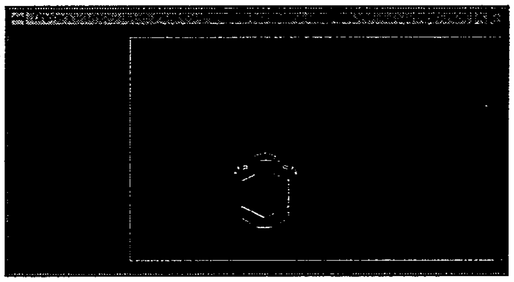
4.4.5.9 Two Dimensional Rendering

Translation, rotation, and zooming are the actions which can be accomplished with the mouse in two dimensional rendering. Translating the scene causes it to be shifted up or down, left or right in the display window. The scene can be rotated either clockwise or counterclockwise around the center of the world. Zooming allows the viewpoint to be placed as close to or as far away from the scene as is desired.

4.4.5.9.1 Observer's Perspective

- **Left Button Only** - Translation (Figure 4-64(b)). Press and hold only the left mouse button. As long as the button remains pressed, the scene will translate in any direction (in the X-Y plane) that the pointer moves.
- **Middle Button Only** - Rotation (Figure 4-64(c)). Press and hold only the middle mouse button. The scene will rotate clockwise when the pointer moves right, counterclockwise when the pointer moves left. Up and down motion of the pointer has no affect on two dimensional rotation.
- **Left and Middle Button** - Zoom (Figure 4-64(d)). Press and hold both the left and middle mouse buttons. To zoom in on the scene, move the pointer to the right. To zoom out, move the pointer to the left. Up and down motion of the pointer has no effect on zooming.


(a) initial view


(b) after translating right

(c) after counterclockwise rotation

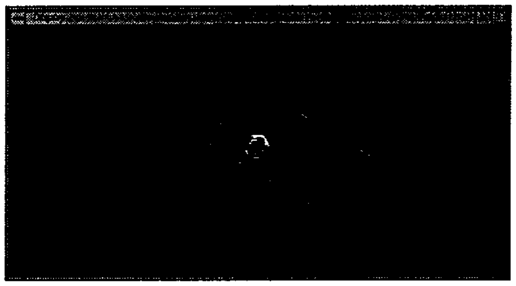

(d) after zooming out

Figure 4-64. Observers perspective

4.4.5.9.2 Traveller's Perspective

This perspective is not valid in two dimensional rendering.

4.4.5.9.3 Robot's Perspective

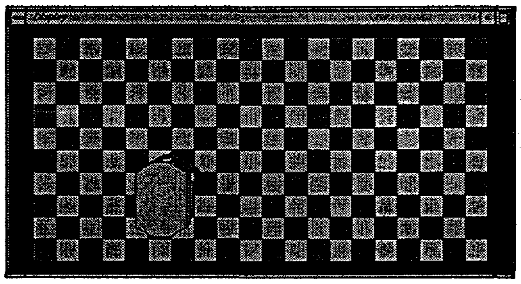
The mouse controls for *Robot's* perspective are identical to those for *Observer's* perspective with one exception; rotation is not permitted.

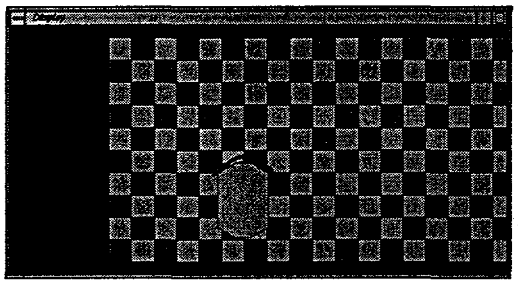
4.4.5.10 Three Dimensional Rendering

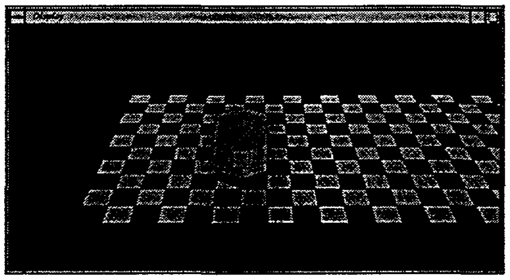
Mouse controls for three dimensional rendering are more diverse than those for two dimensional rendering. Translation, rotation (around two axes), and zooming are still available in Observer's perspective; however, the other perspectives require different actions. New actions are assigned to the mouse controls in *Traveller's* and *Robot's* perspectives.

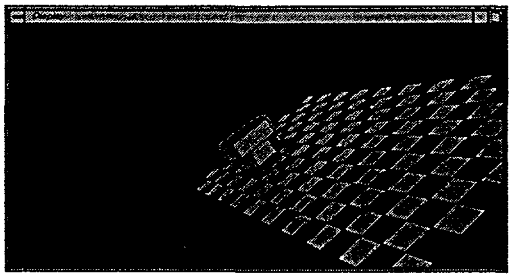
4.4.5.10.1 Observer's Perspective

- **Left Button Only** - Translation (Figure 4-65(b)). Press and hold only the left mouse button. As long as the button remains pressed, the scene will translate in any direction (in the X-Y plane) that the pointer moves.
- **Middle Button Only** - Rotation (Figure 4-65(c and d)). Press and hold only the middle mouse button. Up and down motion of the pointer rotates the scene around the X axis. Left and right motion of the pointer rotates the scene around the Y axis.
- **Left and Middle Button** - Zoom (Figure 4-65(e)). Press and hold both the left and middle mouse buttons. To zoom in on the scene, move the pointer to the right. To zoom out, move the pointer to the left. Vertical motion of the pointer does not affect zooming.


(a) intial view


(b) after translating right


(c) after rotating about the X axis


(d) after rotating about the Y axis

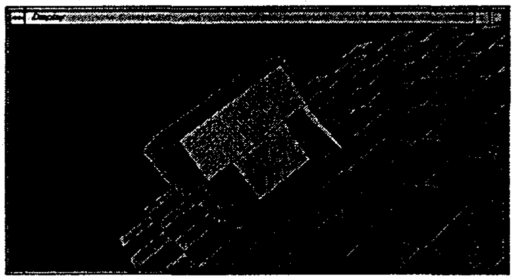
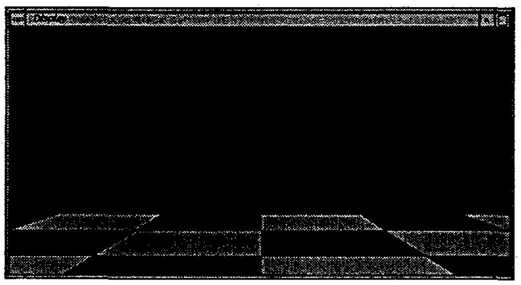

(e) after zooming in

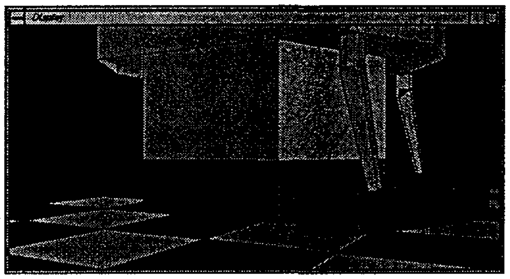
Figure 4-65. Observers perspective, 3D rendering

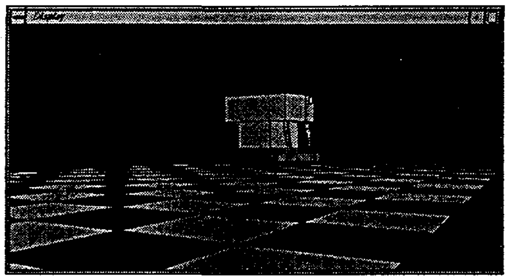
4.4.5.10.2 Traveller's Perspective

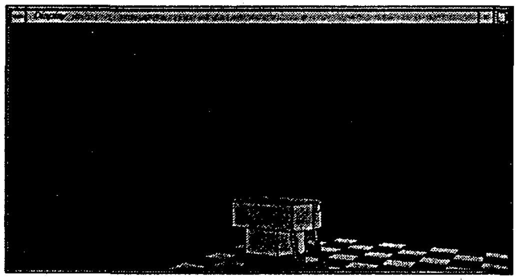
Traveller's perspective allows the viewer to become a traveller in the robot's environment. Mouse controls are provided to let the traveller walk forward and backward, rotate left and right, change the pitch of their head (i.e., look up and down), and change their height.

- **Left Button Only** - Navigation (Figure 4-66 (b and c)). Press and hold only the left mouse button. Up and down motion of the pointer causes the traveller to walk forward and backward respectively. Left and right motion of the pointer rotates the traveller.
- **Middle Button Only** - Pitch (Figure 4-66 (e)). Press and hold only the middle mouse button. Upward motion of the pointer causes the traveller to look up. Downward motion of the pointer causes the traveller to look down.
- **Left and Middle Button** - Height (Figure 4-66 (d)). Press and hold both the left and middle mouse buttons. Upward motion of the pointer makes the traveller grow taller. Downward motion of the pointer makes the traveller grow shorter. Left and right motion of the pointer has no effect on the height of the traveller.


(a) intial view


(b) after rotating left


(c) after walking backwards


(d) after increasing the height

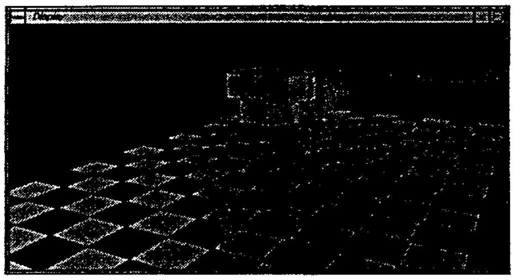
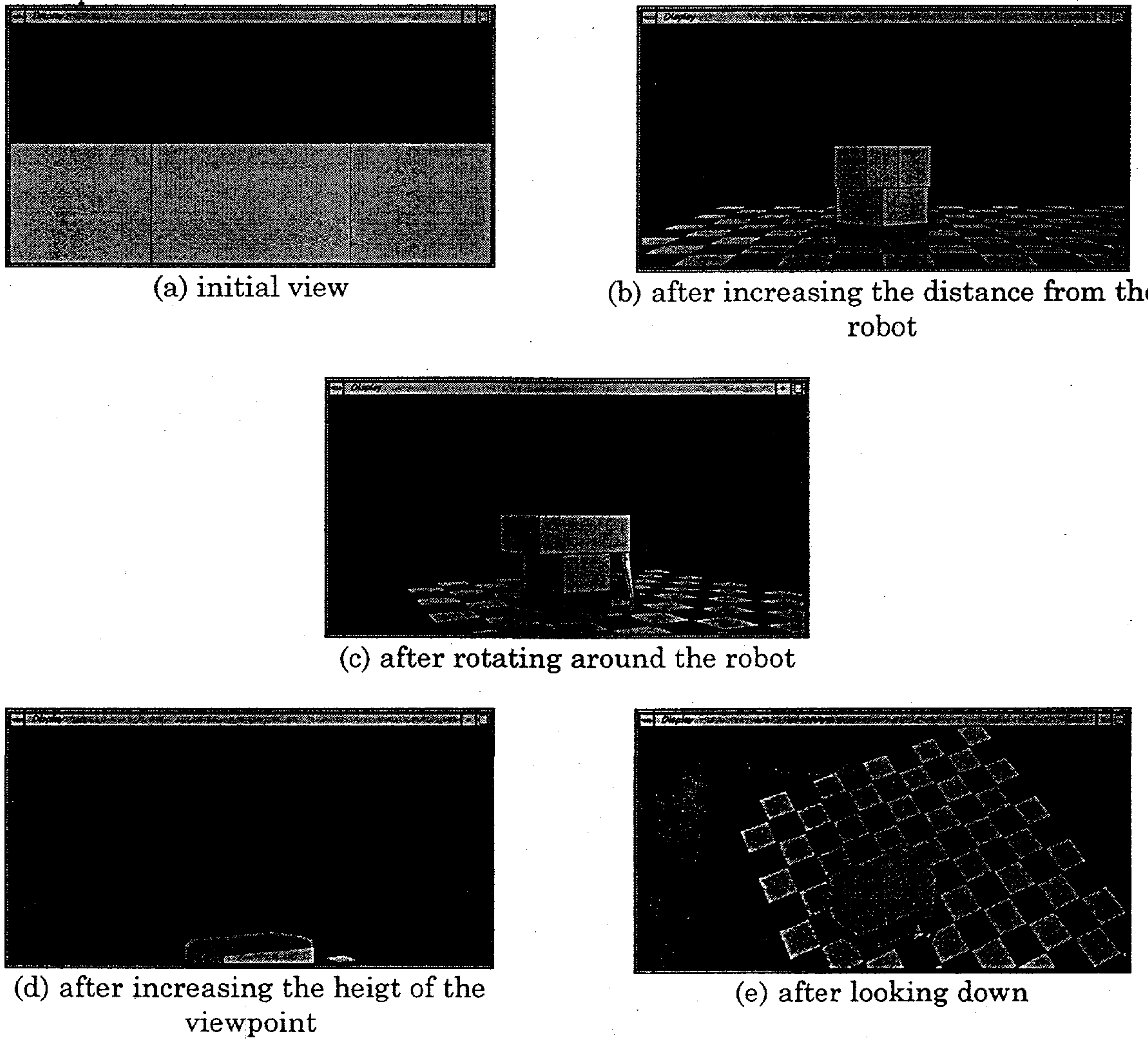

(e) after looking down

Figure 4-66. Traveler's perspective

4.4.5.10.3 Robot's Perspective

The controls are identical for both robot perspectives, *Attached to Turret* and *Attached to Base*. Controls are provided to modify the position of the viewpoint with respect to the robot, the pitch of the viewpoint, and the height of the viewpoint.

- **Left Button Only** - Position (Figure 4-67(b and c)). Press and hold only the left mouse button. Upward motion of the pointer brings the viewpoint closer to the robot. Downward motion pulls the viewpoint away from the robot. Left and right motion of the pointer adjusts the angle between the viewpoint and the robot.
- **Middle Button Only** - Pitch (Figure 4-67(e)). Press and hold only the middle mouse button. Upward motion of the pointer causes the viewer to look up from the position of the viewpoint. Downward motion causes the viewer to look down from the viewpoint. Left and right motion of the pointer has no effect on the pitch.
- **Left and Middle Button** - Height (Figure 4-67(d)). Press and hold both the left and middle mouse buttons. Up and down motion of the pointer moves the viewpoint up and down respectively. Left and right motion has no effect on the height of the viewpoint.


(a) initial view

(b) after increasing the distance from the robot

(c) after rotating around the robot

(d) after increasing the heigt of the viewpoint

(e) after looking down

Figure 4-67. Robot's perspective

4.4.5.11 AutoCAD requirements

The display program uses AutoCAD files to obtain data for drawing both the robot and its environment (the robot's environment will be referred to as the "world"). The display program supports only a small subset of AutoCAD's features. Therefore, certain restrictions apply when generating an AutoCAD file to be used with display. In addition to the restrictions, there are also guidelines which, when followed, ensure the accuracy of the display program. Finally, a list of hints is provided to generate AutoCAD files which will enhance the performance of the display program.

4.4.5.11.1 AutoCAD Restrictions

The following restrictions must be adhered to when generating AutoCAD files. Failure to do so will result in unpredictable behavior of the display program.

- Only the following entities are supported by display: 3DFACES, LINES, ARCS, and CIRCLES. Any other entities used will be ignored and will not appear in the display window. In addition, Thickness is ignored. Solid faces must be generated using 3DFACES. LINES with Thickness will not work.
- Layers are not supported. Any information contained in Layers will be ignored by display. For example, assigning colors " By Layer" will not be recognized.
- Different Linetypes are not supported. All LINES, ARCS, and CIRCLES will be drawn with solid lines.
- Changes in the User Coordinate System (UCS) are not permitted. All entities drawn in a UCS other than the World Coordinate System (WCS) will not appear correctly in display.
- All AutoCAD files used by display must be in the DXF file format. The DXF file format is an ASCII representation of the AutoCAD drawing. DXF files can be generated with the "dxfout" command or through the "FILE" pulldown menu under "export".

4.4.5.11.2 Guidelines for Accuracy

- Both the world and the robot AutoCAD files should be scaled so that 1 = 1 ft. Adhering to this scale ensures that the robot's position will be updated correctly upon receiving coordinates from the Navmaster. AutoCAD files may be generated in any desired units and then scaled to meet this requirement with the "scale" command. For example, an AutoCAD drawing generated in Architectual units (1 = 1 in.) should be "scaled" by 1/12 so that 1 becomes equal to 1 ft.
- The coordinates in the world file must correspond with those in the path programs used to navigate the Navmaster. If the Navmaster docks at position (0, 0), then the dock position in the world file should be (0, 0). Likewise, if the Navmaster encounters a door at position (4, 6), then there should be a door at position (4, 6) in the world file.
- Both the world and the robot files should be oriented so that the floor is parallel to the X-Y plane of the WCS and the positive Z axis points up. Any other orientation will not work with display.

- The Z coordinate of the floor and the smallest Z coordinate of the robot should be very close, if not equal. It is usually convenient to let the Z coordinate of the floor be 0.0. This guideline ensures that not only are the files of the same scale, but that the robot will appear in the room when the two files are displayed together.
- The robot file should be centered with respect to the X-Y plane about the point (0, 0). Display positions and rotates the robot based on this point. A robot file whose center, with respect to all three dimensions, was (0, 0, 2.5) would meet this requirement.

4.4.5.11.3 Tips for Performance Enhancement

- Avoid using numerous entities when generating AutoCAD files. The frequency of screen updates is reduced for every entity in the file simply because there is more to draw. In addition, rendering types which require sorting, lighting, or both must undergo additional calculations before a scene can be updated.
- Avoid using large polygons when possible. Large polygons are often incorrectly sorted by at least one of the three sorting algorithms. Breaking large polygons down into several smaller ones will often improve solid rendering.
- The floor is often an unnecessary polygon in world files. Try replacing the floor with colored lines representing robot paths.
- The "explode" command can be used to break down some solid objects into 3DFACES. For example, SPHERES can be "exploded" into their constituent 3DFACES. Once exploded, solid objects will be recognized by the display program.

4.4.6 ARIES UNIX Dispatcher

The **ARIES** UNIX *dispatcher* is used to dispatch the Cybermotion K2A/K3A vehicle using a series of preset path programs. It allows the user to create new paths, edit existing ones, view the paths against a scale drawing of the facility in AutoCAD, patrol the vehicle, and provide routine monitoring of the vehicle's status.

4.4.6.1 Invoking Dispatcher

To get familiar with the **ARIES** UNIX *dispatcher*, simply type '`dispatch.`' It should already be installed in the **ARIES** installation directory, and the *dispatcher* executable should be in the **ARIES** path. Invoking *dispatcher* with no options gives the following standard UNIX usage output:

```
Usage: dispatch <Server> [path database]
```

As outlined in the usage statement, *dispatcher* has only one required argument, the name of the vehicle to dispatch. Each of the **ARIES** vehicles has an on-board computer with a valid ethernet address. The name given this computer for addressing purposes should be passed to *dispatcher*.

An optional argument is a path database file. Including a database will cause *dispatcher* to pop up with the interface set up for that particular database. If no database is listed, the interface will simply pop up clean, with no path information contained in the dialogs. One example usage might be:

```
dispatch aries1 PathDatabases/warehouse.pdb
```

This would start *dispatcher*, have it connect to the `aries1` computer and load in the `ware-house.pdb` path database contained in the `PathDatabases` directory.

4.4.6.2 Dispatcher Interface

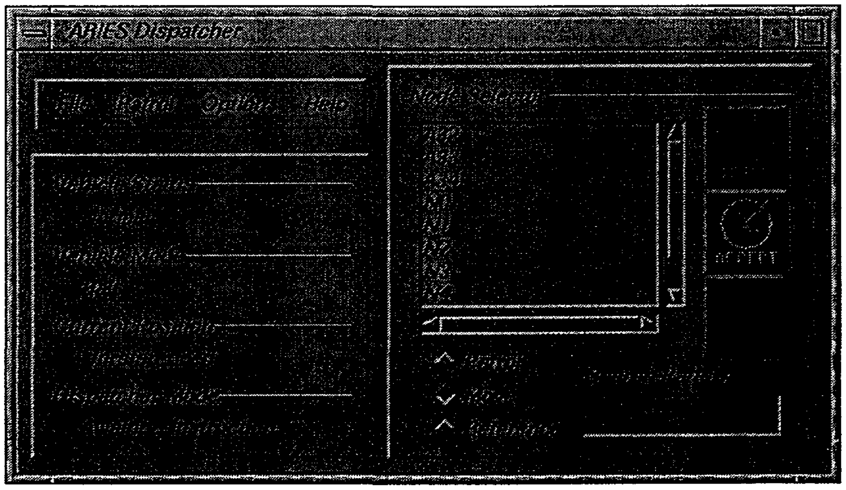


Figure 4-68. Dispatcher main interface

The **ARIES** *dispatcher* main interface is show in Figure 4-68. It is divided into several sections: the pulldown menu, the status area, and the dispatching area.

4.4.6.2.1 Pulldown Menu

4.4.6.2.2 File

dispatcher uses path database files to store the set of paths that describe how the robot is to navigate an installation. Per path, the start and end node names, the cost of traversal, and the action filename are stored. Listing 1 shows an example **ARIES** path database.

```
USCpathDatabasev1
Patrol { a1, a2, 1, "a1_a2.sgv" }
Patrol { a2, a1, 1, "a2_a1.sgv" }
Patrol { a2, a4, 1, "a2_a4" }
Move { a2, a8, 1, "a2_a8.sgv" }
Patrol { a4, a2, 1, "a4_a2.sgv" }
Patrol { a4, a5, 1, "a4_a5.sgv" }
Patrol { a5, a4, 1, "a5_a4.sgv" }
Reference { a6, a4, 1, "a6_ref.sgv" }
Reference { a7, a1, 1, "a7_ref.sgv" }
Move { a8, a2, 1, "a8_a2.sgv" }
Move { a8, a9, 1, "a8_a9.sgv" }
Drawing = "/usr/people/king/robot/dispatch/DBs/hallway.dxf";
```

Figure 4-69. Example dispatcher path database file

For each entry, one of the three keywords `Patrol`, `Move`, or `Reference` will appear. A patrol action is one that is used to normally dispatch the robot to a location, but may also be one of the randomly chosen paths for use during a patrol action. Move actions are identical to patrols, except that they cannot be used as patrol paths. Finally, reference actions are those that are used to reference the robot in its environment.

The file section of the pulldown menu gives the user access to file operations standard to most applications: **New**, **Load**, **Save**, **Save As**, and **Exit**. Each of these options, with the exception of **New** and **Exit** will pop up a file selector for the user to select the name of a path database. The **New** option will clean the interface of all path information so that a new database can be started.

4.4.6.2.3 Patrol

During patrol mode, *dispatcher* constantly sends the vehicle to random nodes from the path database. It will not send the vehicle to a node that has no return path. Only paths marked as *patrol* will be used in deciding the destination node.

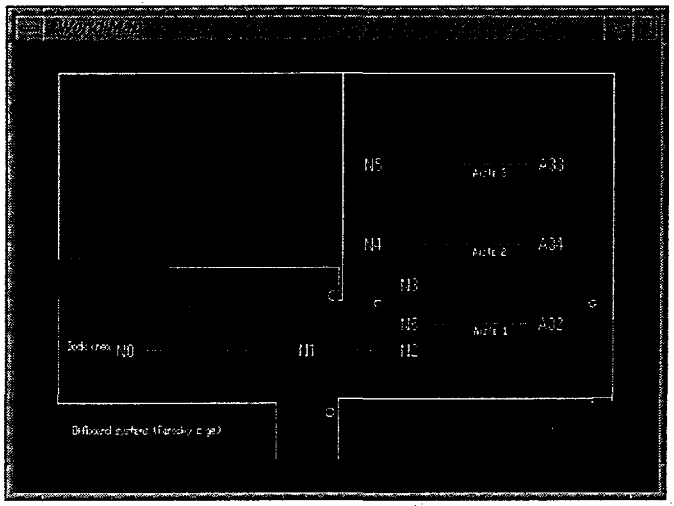
Controls are provided in this section of the pulldown interface. Once the patrol mode has started, the status area will display the current position of the robot as well as its destination. If "End Patrol" is selected, *dispatcher* will inform the user that patrol mode cannot be exited until the robot has reached its current destination node. However, pressing HALT in the dispatching area will still terminate the path program immediately.

4.4.6.2.4 Edit Database

The option will pop up the path database dialog of *dispatcher*. See Editing the Path Database for a detailed explanation of this dialog.

4.4.6.2.5 Enable Map Mode

By using the DXF file provided in the path database file, *dispatcher* provides a two-dimensional map of the facility as well as the paths, and path nodes defined in the database. The user can zoom to different areas of the map by selecting an area with the middle mouse button. Zoom out by pressing the third mouse button and selecting the "Zoom Out" option from the popup menu.



Also available on the popup menu is "Dispatch Mode." If this mode is entered, the user can dispatch the vehicle by simply clicking on the desired node. From the dispatching area in the main interface, the user may change the path type among Patrol, Move, and Reference. Changing the path type in this area also affects which paths and path nodes are displayed in the map area.

4.4.6.2.6 Queue Send Events

When dispatching the vehicle to several nodes, this option may be turned on so that several dispatch events are queuing, to be serviced sequentially after the vehicle has completed a path. For more on the dispatching process, see section 3.4. Dispatching Area.

4.4.6.2.7 Status Area

The status area provides routine monitoring of the state of the vehicle's status and mode. These are displayed in the first two lines in the status area. The current position of the vehicle is given by the name of the node, or the names of the nodes the vehicle is between. Finally, the dispatcher mode is last in the status list. It informs the user what mode *dispatcher* is in or what is causing it to wait.

4.4.6.3 Dispatching Area

The dispatching area is composed of four subdivisions: the node selector, action buttons, path type selector, and the search text widget.

4.4.6.3.1 Node Selector

Simply double-click on the name of the node to dispatch the vehicle. If no path exists between the vehicle's current position and the destination, *dispatcher* will alert the user. A node can be selected by a single click on the action buttons.

4.4.6.3.2 Action Buttons

The buttons allow the user to SEND, ACCEPT, or HALT. The SEND button simply dispatches the robot to the selected node, a duplication of the double-clicking functionality from the node selector. ACCEPT will change what *dispatcher* considers to be the vehicle's current position. HALT will immediately stop the execution of the path program running on-board the vehicle.

4.4.6.3.3 Path Type Selector

This group of toggle buttons determines which nodes are displayed in the node selector. A patrol action is one that is used to normally dispatch the robot to a location, but may also be one of the randomly chosen paths for use during a patrol action. Move actions are identical to patrols, except that they cannot be used as patrol paths. Finally, reference actions are those that are used to reference the robot in its environment.

4.4.6.3.4 Search Text Widget

Any pattern typed in the search text widget will be matched against the node selector list. The search compares the letters of the search pattern to the first letters of each entry in the list. If a match is found, the node is automatically selected.

4.4.6.4 Editing the Path Database

The *Database Manager* can be popped up by selecting "Edit Database" from the "Options" pulldown menu. This dialog is composed of four groups: the paths list, DXF file selector, options, and search text widget.

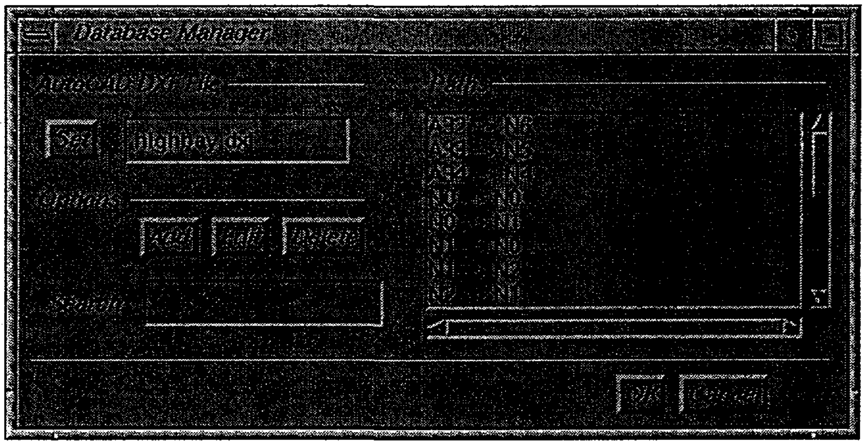


Figure 4-69. Database manager dialog

4.4.6.4.1 Paths List

The Paths List contains *all* of the paths in the path database, regardless of what type it is (Patrol, Move, or Reference). Double-click on a path entry to invoke the *Path Editor*, or click on an entry to make it the active path.

4.4.6.4.2 DXF File

These controls are used to specify the name of the DXF file used in map mode. Press the "Set" button to pop up a file selector to specify the file.

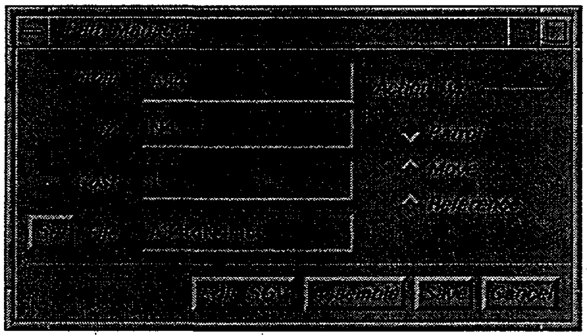
4.4.6.4.3 Options

These options directly affect the active path as selected in the Paths List. "Edit" will invoke the *Path Editor* set with the active path parameters. "Delete" will remove the active path from the Paths List, but will ask for confirmation. "Add" will invoke the *Path Editor* with its fields blank.

4.4.6.4.4 Search Text

Any pattern typed in the search text widget will be matched against the node selector list. The search compares the letters of the search pattern to the first letters of each entry in the list. If a match is found, the node is automatically selected.

4.4.6.5 Editing Path Entries



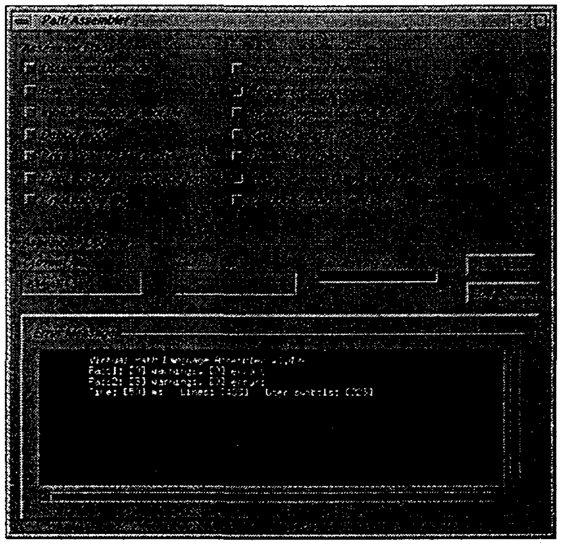
The *Path Editor* is used to modify and add new paths to the path database. It is composed of several different areas: the parameter editors, action type selector, and control buttons.

The parameters necessary to define a path are From, To , Cost, and Action File. These determine where the vehicle starts, ends, what the "cost" was, and which action file (path language assembled program) will accomplish the navigation. Each of these fields are text boxes, type into the boxes to enter the information. The Action File can also be entered by pressing "Set," to pop up a file selector.

The type selector defines what type of action the path should be considered as: Patrol, Move, or Reference.

The four control buttons at the bottom of the interface are: "Edit .SGV", "Assemble", "Save", "Cancel". "Edit .SGV" will pop up an *xterm* running *vi* on the SGV file that contains the navigation program. "Assemble" will pop up the *Path Assembler* interface with the current path program already entered.

4.4.6.6 Assembling Path Programs



```
[illegible] Path Language Assembler [illegible]
Pass1: [0] warnings, [0] errors
Pass2: [3] warnings, [0] errors
Time: [50] ms  Lines: [illegible]  User symbols: [illegible]
```

Figure 4-70. Path Assembler interface

The *Path Assembler* interface is composed of several sections: Assembler Flags, Assembler Options, and Error Messages.

4.4.6.6.1 Assembler Flags

The Assembler Flags are a series of toggle buttons that control different PASM parameters. See the *PASM* usage documentation for information on the types and functionality of the flag.

4.4.6.6.2 Assembler Options

These options allow the user to change the input and output files, set the tab length of the output listings, assemble the file, and edit the SGV source file.

4.4.6.6.3 Error Messages

This area displays the output of the path assembler. It will either be a report of a clean assembly, or a listing of errors that were found in the source code.

4.4.7 Database Program (Database Browser)

The **ARIES** database *browser* is used to view, add, and edit records in a drum database. It allows the user to browse through a list of drum records, click on one for further inspection, and edit its fields. Additional controls are provided enabling the user to search for drums based on one or many search criteria.

4.4.7.1 Invoking Browser

To invoke the **ARIES** database *browser*, simply type '`browser`.' It should be installed in the **ARIES** installation directory and the *browser* executable should be in the **ARIES** path. Typing '`browser -help`' gives the following standard UNIX usage output:

```
ARIES Database Browser/Editor v1.0

Usage: browser [.db file]
-sd <id>        read single drum matching <id>
```

As outlined in the usage statement, *browser* may be invoked by itself or given a drum database file as its first parameter. *browser* will automatically check to see if the file passed to it is a valid **ARIES** database file.

Also available is the `-sd` option, which allows the user to select a single drum from the database passed to *browser*. A database must be specified when using this option.

4.4.7.2 Browser Interface

The main interface for *browser* is shown in Figure 4-71. It consists of a pulldown menu, working data set area, and a series of control buttons.

4.4.7.3 Pulldown Menu

4.4.7.3.1 File

The file section of the pulldown menu gives the user access to file operations standard to most applications: **New**, **Load**, **Save**, **Save As**, and **Exit**. Each of these options, with the exception of **New** and **Exit** will pop up a file selector for the user to select the name of a database.

The **New** option will also pop up a file selector but requires the user to enter the name of an **ARIES** database *description* file. This is a file used to describe the format of the records stored in a database. Refer to section 4 for details about *description* files.

4.4.7.3.2 Working Set

The working set defines the records in the database that database operations will affect. By default, the working set will contain all entries in the database. The working set menu allows the user to read all of the entries into the set as well as search through the current working set for a subset of records.

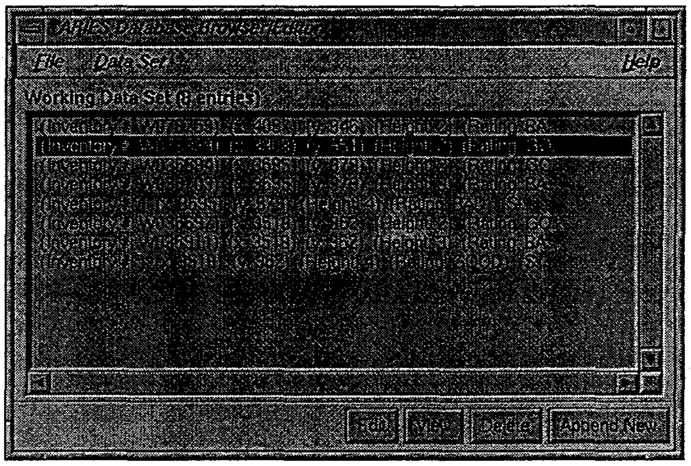
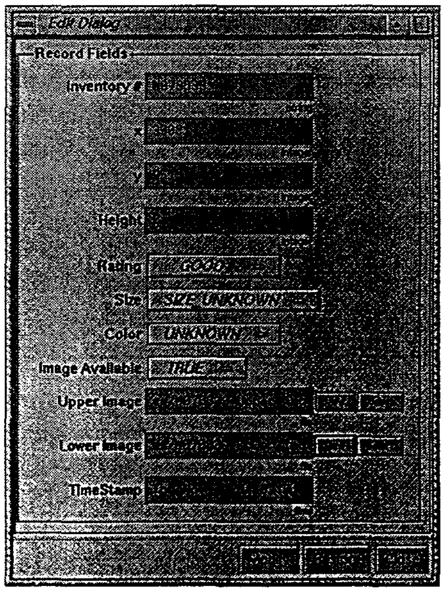


Figure 4-71. *browser* main interface

4.4.7.4 Editing Records

The control buttons at the bottom of the interface affect records in the working data set. These operations are:



Rating: GOOD
Size: SIZE UNKNOWN
Color: UNKNOWN
Image Available: TRUE

Edit: Pops up the record editing dialog with the "Modify" key enabled. This allows the user to save changes made to the records.

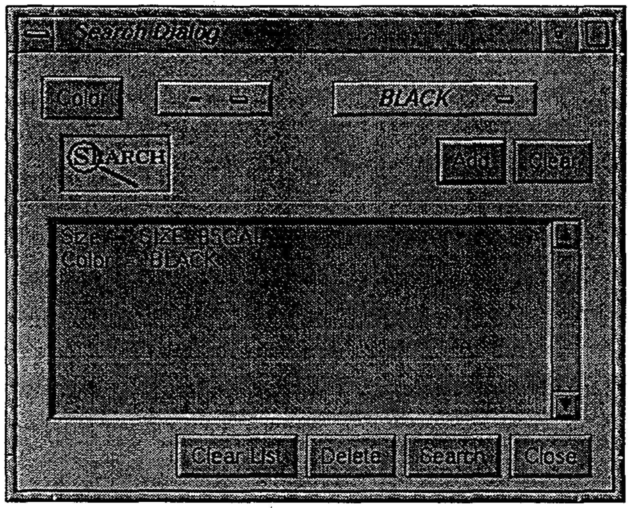
View: Pops up the record editing dialog with the "Modify" key disabled. This allows the user to view the record, without being able to modify the record.

Delete: Deletes the selected record from the working data set.

Append New: Pops up the record editing dialog with each field set to its default value. The "Modify" key changes to "Append" and when pressed, attaches the new record to the working data set.

The editing dialog is used to edit individual fields in the drum database records. It will change according to which type of fields compose the record. Some data types are edited via a text box widget (integers, floats, and character strings). The contents of the boxes are verified for the given data type. Enumerations are handled by providing the user with an option menu listing the only choices available to the user. Filenames are displayed in text box widgets where they may be editing by typing in the box. A “Select” button is provided that will pop up a file selector allowing the user to navigate the file tree. A “View” button is also provided that invokes the image viewing program *xv* on the file, allowing the user to view the image. If the file is recognized as an **ARIES** HIS image, it will invoke the **ARIES** HIS generic file format viewer.

4.4.7.5 Search Dialog



Size = SIZE_85GAL
Color = BLACK

The search dialog enables the user to narrow the list of drums in the working data set by searching for drums of a particular type. The dialog is brought up by selecting “Search...” under the “Working Set” pull-down menu.

As a tutorial on the usage of the search dialog, we will narrow the working data set by searching for drums that are “85 gallon in size and black in color.” The top left button of the dialog will be labeled “Search Field” when no field from the drum record has been chosen. Press this button to pop up the field selector dialog. This will contain a list of all of the fields that compose the record. Choose “Size” and press “OK.”

The “Search Field” button should now say “Size.” The option menu next to this button contains the comparator operations: =, <, >, ≤, ≥, and ≠. Choose “=.” Because “Size” is an enumerated type, the text box in the top right of the dialog will become an option menu once “Size” has been selected. The option menu will contain all of the available colors for drums. Choose “SIZE_85GAL.” We have now designated one search field and wish to add it to the search parameters. Do this by pressing “Add,” it will appear in the list widget in the lower section of the dialog. This list contains all of the active search parameters.

Using the method outlined in the previous paragraph, select “Color” as a search field, “=“ as the comparator, and “BLACK” as the sought value. Press “Add” to add it to the active search parameters.

Among the buttons at the bottom of the dialog is the “Search” button. Pressing this activates the search and replaces the working data set with the entries matching the search parameters.

4.4.7.6 Description Files

Description files are used to describe the record format of a given database. They are used by the browser when creating a new database. They provide for an additional level of flexibility in that databases can be created that store all types of data, not just **ARIES** drums. Furthermore, **ARIES** drum databases can be easily modified to remove fields or include new fields. Included below is the official **ARIES** drum database format at the time of the Phase II demonstration.

```
Record
    {
    char_array          "Inventory #";
    char_array          "Inventory #";
    integer             "X";
    integer             "Y";
    integer             "Height";
    integer             "Aisle";
    integer             "Row";
    integer             "Index";
    integer             "Drum #";
    float               "Net Weight";
    float               "Tare Weight";
    float               "Gross Weight";
    integer             "MEF #";
    { SIZE_UNKNOWN, SIZE_55GAL, SIZE_85GAL, SIZE_110GAL, SIZE_OTHER }
                        "Size";
    { UNKNOWN, BLACK, BLUE, YELLOW, GREEN, WHITE }
                        "Color";
    { RATING_UNKNOWN, RATING_GOOD, RATING_BAD, RATING_OTHER }
                        "Rating";
    { FALSE, TRUE }
                        "Image Available";
    file                "Lower Image";
    char_array          "Description";
    }
```

Listing 1. **ARIES** drum database record format

4.4.8 Programming Tools

4.4.8.1 Using the Path Assembler (PASM)

Source programs for the K3A are called *path programs* and are written in Virtual Path Language (VPL). VPL is a fairly low-level language. It is considered an assembler language since the actual K3A instructions are used. The process by which a path program is created and executed is similar to that of an assembler program for a small computer system. However, path programs differ from conventional computer programs in a number of ways, due to the nature of the K3A. The most important difference is that most K3A programs are "real-time" in nature. For a typical computer program, there is no connection between the speed of execution and the correctness of the program. A real-time program is incorrect if it does not perform the proper actions at the appropriate time. For instance, a K3A cruising down a corridor which is to turn into a doorway must turn at the proper time. Notice that the consequences of being too early are just as bad as being too late. It probably doesn't matter whether the hole in the wall is on the right- or left-hand side of the doorway!

The components of a path program (see figure 4-72) are *(i)* the K3A instructions, *(ii)* assembler directives, *(iii)* include(d) files, and *(iv)* external statements. K3A instructions consist of an opcode and from zero to three operands and is similar to a machine instruction of a typical computer program. Each K3A instruction, when assembled, produces a six-byte binary string which can be executed by the K3A. The K3A can store up to 255 instructions and executes them sequentially. The assembler directives (or "pseudops") are used to create symbolic constants, increasing the readability and maintainability of the path program. One of the assembler directives is the include command, which allows for inclusion of other files into the program when it is assembled. The include files are similar in nature to the header files of a C program. They allow commonly used constants or code segments to be placed in library files and included where needed.

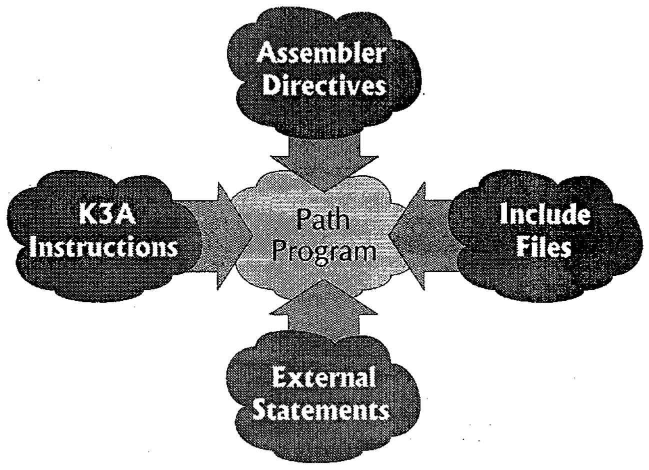


Figure 4-72. Components of a path program

The first three components of the path program are all analogous to similar components in a typical computer program. The fourth component, the external statements, reflect the real-time nature of a typical K3A (or K2A) application. An external statement causes a program to run concurrently on the host system (using the dispatcher program, described later) and allows cooperation between the host and the K3A while operating autonomously. For instance, a K3A could use an external statement to run a routine in the host which will call an elevator and open the door. The use of external statements is called *concurrent* programming because they execute concurrently with the internal program of the K3A.

The term *Virtual Path Language*, or VPL, is used for a K3A (or K2A) program to indicate that it directs the vehicle over some path (while possibly taking some action(s) along the path) and that most programs do not depend on an absolute coordinate system. The programs are also virtual in that a path program, when executed in conjunction with the dispatcher, may exceed the K3A internal limit of 255 instructions.

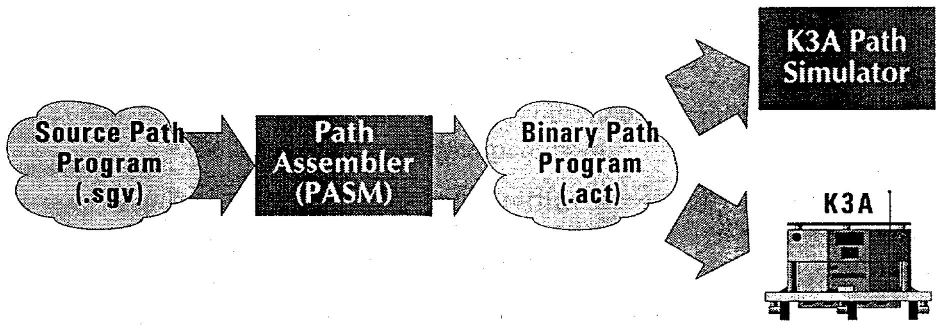


Figure 4-73. Use of PASM in the stand-alone mode

Path programs are converted to a binary form, suitable for execution, by the Path Assembler (PASM). PASM may be used in a stand-alone assembler or in conjunction with the

Dispatcher program running on the host system. Figure 4-73 shows the procedures used in the stand-alone mode. The source program is prepared off-line using a suitable ASCII editor and, by convention, is given the suffix ".sgv." PASM then creates a binary file having the suffix ".act" (for action file) which is suitable for downloading to the K3A for execution. A binary file created in this mode will generally contain a terminating instruction (HALT) at the end. This mode of operation is generally used only for small programs. The file may be downloaded to the K3A for execution, or alternatively, it may be executed on the K3A Path Simulator. The simulator (which is described in a later section) is useful during program development or while learning to program the K3A.

4.4.8.1.1 Program Development Using the Dispatcher

Most practical path programs are created by use of the dispatcher, which executes on the host system. The dispatcher, as its name implies, is used to control the vehicle while it performs a given task. The dispatcher provides *(i)* a user interface to the system, *(ii)* provisions for interaction with the K3A (or K2A) during program execution, and *(iii)* the ability to combine existing action files into complete path programs. The dispatcher acts as a linker, allowing two or more action segments to be combined into a complete path program. In addition, programs may be written such that new segments are downloaded during program execution, allowing programs to be longer than 255 instructions.

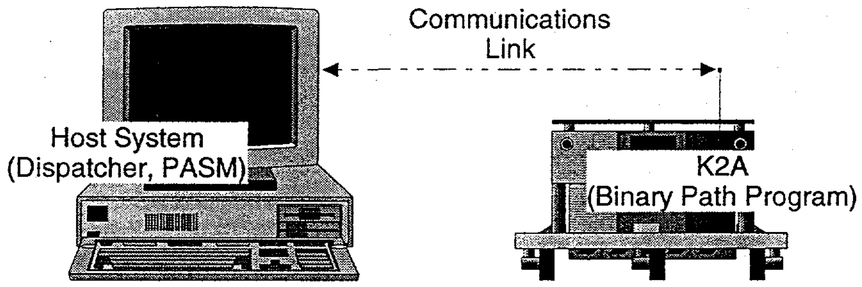


Figure 4-74. K3A used with host system and dispatcher

The stand-alone mode of programming mentioned above is good for small routines or while learning; however, the Dispatcher is the preferred programming environment for serious programs. The reasons for this include:

a) The ability to create programs longer than 255 instructions as mentioned above, as well as the ability to incorporate "library" routines.

b) The Dispatcher provides the "hooks" needed to interact with the vehicle while it is executing the on-board program.

c) It was mentioned earlier that programming the K3A differs from typical computer programming in that it is real-time. Another important difference is that it is more graphical in nature. The Dispatcher provides a graphical programming environment in which the path program may be composed in the context of the area to be traversed by the vehicle.

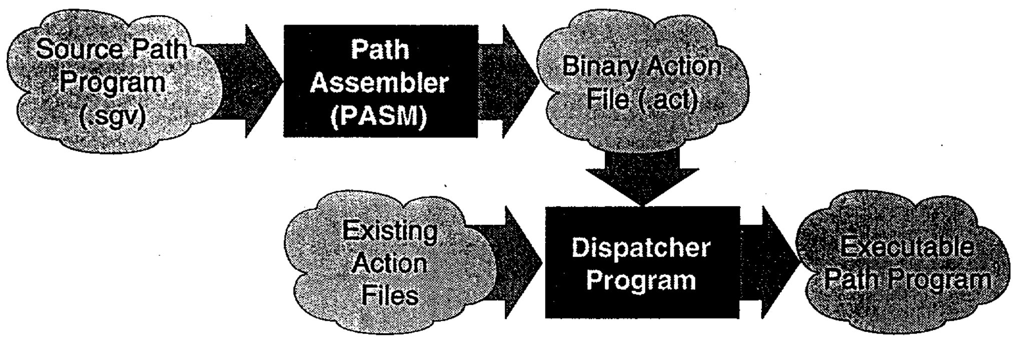


Figure 4-75. Program development using the dispatcher

4.4.8.1.2 The Environment

We will introduce the basic K3A programming techniques by way of a series of short example programs. Each of these programs can be done in the stand-alone mode. We will use the simulator, as well as data from an actual K3A, to illustrate the actual execution of the program. Path Programs are generally derived from a map or floor plan of the area to be traversed by the vehicle. For our example programs this room will be a 22' x 33' of the USC Robotics Lab. This area is shown in Figure 4-75. For convenience, the origin is displaced 4' from the top of the room since there are several tables in that area. One of these tables contains a docking station, which is located at position (3,2) with reference to the room origin. The room contains two rows of nine 55-gallon drums which are used to demonstrate navigation of the vehicle. Finally, several reference points have been marked which will be used in the example programs. All of our programs will assume that the robot is docked, which means that it will be a known distance (in this case, 5') in front of the docking station. This position is given as "`DockPos`."

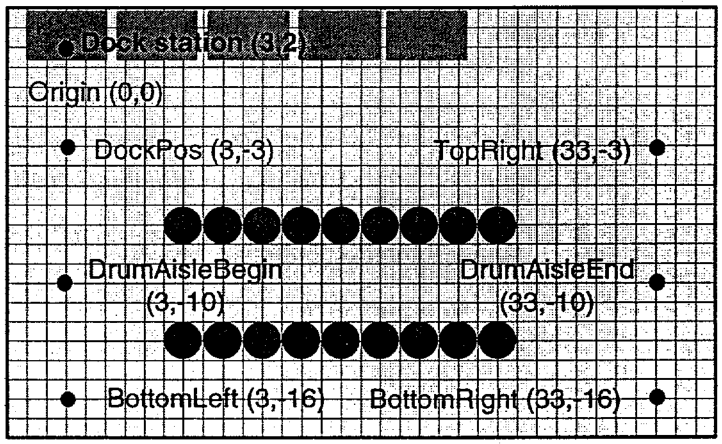


Figure 4-76. Layout of area to be used for the example programs

4.4.8.1.3 Invoking the Path Assembler

```
Virtual Path Language Assembler     v1.33
The Path Assembler is invoked by the command

       pa { <options> <filename> }

The following options are available:

-a                  Disassembler mode
```

```
-b                  Batch mode
-c                  Inhibits creation of .act file
-d                  Debug mode
-h                  Help
-i                  Print include files in the listing
-L                  Print listing in landscape format
-l                  Print listing file
-o <filename>       Allows specifying output file name
-p                  Parameter checking
-s                  Include symbol table in listing
-t                  Sets tab length for listing
-w                  Disables warning messages
-v                  Verbose mode.  Allows multiple error messages
-x                  Produces Hex listing
```

Figure 4-77. PASM help screen

PASM, when used as a stand alone program is invoked by the command

```
pa { <options> } <filename>.
```

which follows standard Unix practice. The path assembler is designed so that a user who is reasonably experienced in assembler-level programming should be able to use it immediately. A minimal help function is included in the form of a help option. If the -h option is specified (`pa -h` or `pa /h`) the assembler will output a brief help message which gives the options available and the function of each. The response to `pa -h` is given in figure 4-77. Note that most of the defaults are chosen for the "production case." For example, it is likely that most of the time a listing file will not be needed, therefore the default option is not to create it. Likewise, the default is to give only one error message per line although the "verbose" mode (`-v`) will give multiple error messages per line.

4.4.8.1.4 Example Program 1

The first program will cause the robot to traverse a closed rectangular path which is `DockPos` ⇒ `BottomLeft` ⇒ `BottomRight` ⇒ `TopRight` ⇒ `DockPos`. This path is illustrated in figure 4-78. This assumes that the vehicle is initially docked, which means that it is positioned a known distance directly in front of the docking beacon. This will be the default starting position for all example programs.

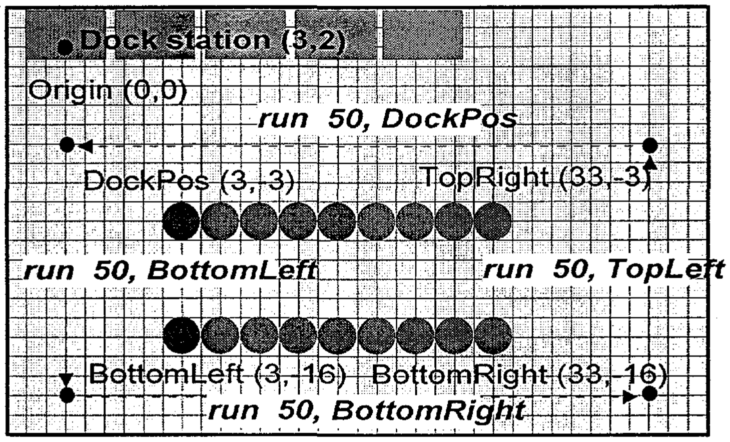


Figure 4-78. Path to be traversed for program 1

The source code for example program 1 is given in figure 4-79. The first instruction (PASM treats all text on a line following a ";" as a comment) is an include statement "include `r3d38.def`", which causes the contents of the file `r3d38.def` to be inserted at this point when the program is assembled.

```
;
;       Example Program 1
;       Traverse rectangular path in room 3D38
;
        include       r3d38.def           ; include constants for room 3D38
        setxy         DockPos             ; navigation instruction -
        warn          0, 50               ; personality instruction
        run           50, BottomLeft
        run           50, BottomRight
        run           50, TopRight
        run           50, DockPos
        halt
```

Figure 4-79. Source code for example program 1

The contents of the header file are given in figure 4-80. This file contains a series of position definitions using the PASM assembler directive `DEFP`, or DEFine Position. This command assigns an actual position value to a symbolic constant. For instance, the command

```
DEFP  DockPos     300, -300
```

assigns the values x = 3' and y = -3' (the PASM default units are one-hundredths of a foot) to the constant DockPos. Putting these values into an include file removes the necessity of entering them into each program and allows the programs to become more general. If changes are made to the room, then all programs which used the include file are automatically updated the next time they are assembled. The terms include file (since it is "included") and header file (since it generally comes at the first of the program) are used interchangeably.

```
;
;      Header File for Room 3D38 (Mobile Robotics Laboratory)
;
DEFP   DockPos             300,-300
DEFP   TopRight            3300,-300
DEFP   BottomRight         3300,-1600
DEFP   BottomLeft          300,-1600
DEFP   DrumAisleBegin      300,-1000
DEFP   DrumAisleEnd        3300,-1000
```

Figure 4-80. Contents of the include (header) file `r3d38.def`

A description of the activity associated with each instruction is given below:

`setxy DockPos` — Set the position, stored in the vehicle, to the docking location. This has the effect of referencing the internal position of the K3A to the reference system chosen for the room.

`warn 0, 50` — Controls the operation of the collision avoidance warning signal. The first parameter controls the number of steps for which the signal is to be disabled, 0 in this case. The second controls the maximum distance from an object at which the signal will be activated, in this case 0.5 feet or less.

`run 50, BottomLeft` — The run instruction causes the K3A to first turn in the direction specified, then move to that point. The first parameter is the speed, in units of 0.01 ft./sec. The second and third parameters give the x and y location to which the vehicle is to move. In this case a position parameter is used which specifies both (x = 3, y = -16). Since the vehicle always turns to the proper azimuth, the effects of run instruction are independent of the initial azimuth.

`run50, BottomRight; run 50, TopRight; run 50, DockPos;` — These run instructions cause the K3A to complete the next three segments of the path, returning to the original location at the dock.

`halt` — Causes the robot to go into the halt mode. Generally there is only one halt instruction in a program and it comes at the end.

The output file of the assembler, `p1.act`, is a binary file which consists of six bytes per instruction plus four additional bytes which specify the drive and steer acceleration values. If no values are specified, PASM assumes the default values of 40 begrees/sec^2 for the steer acceleration and 1 foot/sec^2 for the drive acceleration. (A begree is a binary degree and is equal to $2\pi/1024$ radians or $360°/1024$.) The default values can be changed by use of the assembler directives DEFS (steer) and DEFD (drive). Since the output file is in binary form, it cannot be readily read by a normal text editor; however, PASM has a disassembler mode which can be used to determine the contents of a .act file. Figure 4-81 gives the output of the disassembler for p1.act. The disassembler supplies the command mnemonics since they are unique; however, the constant labels are unique to the source code and cannot be inferred from the binary file.

```
Virtual Path Language Assembler  v1.33
Disassembler mode ... disassembling [p1.act]
File contains [7] instructions, [4] additional bytes

       Command:       s:       x:     y:
      setxy ( 23)     0       300   -300
       warn ( 33)     0        50      0
        run (  1)    50       300  -1600
        run (  1)    50      3300  -1600
        run (  1)    50      3300   -300
        run (  1)    50       300   -300
       halt ( 20)     0         0      0
Drive Acceleration: [  4]
Steer Acceleration: [ 10]
[0] milliseconds CPU time
```

Figure 4-81. Disassembler output for `p1.act`

4.4.8.1.5 Instruction Categories

The K3A currently has 77 instructions, which can be divided into eight categories, data transfer, event, mathematical, movement, navigation, personality, program control, and subsystem control. These categories, and the instructions in each, are shown in table 1.

4.4.8.1.6 Movement Instructions:

This group of instructions, including RUN, RUNON, BACK, JOG, TURN, SCAN, DOCK, UNDOCK, CRUISE, and STOP causes the vehicle to move, to turn, or to perform a series of movements and turns. These actions are not rigid, but are in fact goal oriented behaviors. Their actual execution is highly dependent on a wide range of navigation and personality variables. Most of the personality variables default to commonly used settings and are not normally of concern to the programmer.

CATEGORY	INSTRUCTIONS	APPLICATION
Data Transfer	COPYB, COPYW, DOCKIN, DOCKOUT, PORT, READB, READW, UNPORT, WRITEB, WRITEW	Used to move data between the various subsystems of the vehicle.
Event	GATE@, WALLOFF@, WALLON@, WBEGINS@, WENDS@	Allow scheduling of "events" during the path
Mathematical	ABS, ADD, DIV, MULT, SUB, VECTOR	Perform basic math operations
Movement	RUN, RUNON, BACK, JOG, TURN, SCAN, DOCK, UNDOCK, CRUISE, STOP	Control actual vehicle movement. Dependent on personality and navigation settings.
Navigation	APPROACH, DOOR, HALL, MARKER, MEANAZ, SETAZ, SETXY, WALL,	used to set variables that are used before, during, or after Movement Instructions to change the vehicle's vector position estimate
Personality	AVOID, CDEFLECT, CIRCUM, CURLIM, PATROL, SETACC, WARN, WDEFLECT	Determine the manner in which the vehicle executes the movement instructions.
Program Control	JUMP, JUMP=, JUMP>, JUMP<, JUMP!=, CALL, CALL=, CALL>, CALL<, CALL!=, and RETURN	Used to change program flow, test program state and branch accordingly
Subsystem	MAUX, MOVE, PAN, PICK, PUT, TILT, ZOOM	These instructions are used to control optional subsystems, such as the SPI package.

Table 4-7. K3A SRV instruction categories

4.4.8.1.7 Personality Instructions:

Personality Instructions including SETACC (set acceleration), AVOID (set avoidance distances), WARN, CDEFLECT, WDEFLECT, PATROL, CIRCUM, and CURLIM are used to modify personality variables that control the operation of the vehicle as it executes Movement Instructions. Less commonly used personality variables may be changed using Data Transfer Instructions.

4.4.8.1.8 Navigation Instructions:

Navigation Instructions are used to set variables that are used before, during, or after Movement Instructions to change the vehicle's position or azimuth estimates. The simplest Navigation Instructions set these estimates directly and unconditionally such as SETAZ (Set Azimuth) and SETXY (Set X/Y Position).

Other Navigation Instructions (WALL, HALL, APPROACH and DOOR) set variables that cause the vehicle to image building features during the next RUN, and then correct its position and azimuth estimates from the data it receives.

The MEANAZ Instruction is used to correct a vehicle's azimuth from data gathered during the preceding DOCK maneuver, and the MARKER instruction causes the vehicle to execute turns to image and correct its position from a sonar landmark.

4.4.8.1.9 Event Instructions:

Event Instructions are a special class (mostly navigation related) which schedule sequential "Events" to occur on the next path. The Events WALLOFF@ and WALLON@ modify the operation of Navigation Instructions WALL and HALL at certain points.

The GATE@, WBEGINS@, and WENDS@ Event Instructions cause special sonar imaging tasks to be executed at other points during a RUN. These tasks, if successful will also modify the vehicle's position estimates.

4.4.8.1.10 Subsystem Control Instructions

Subsystem Instructions such as MOVE, PICK, PUT, PAN, TILT, ZOOM, and MAUX control the operation of subsystems such as the T2C Paraflex Lift, and the SPI Security Patrol Instrumentation package. The K3A Platform performs this control by communicating with these systems over the Internal Control Link.

4.4.8.1.11 Data Transfer Instructions

Data transfers within the K3As memory or between the K3A and other subsystems on the vehicle may take place for a wide range of reasons. For example, a RUN instruction will cause the K3A to repeatedly read sonar collision ranges from the CA1 while it is moving.

For all cases not covered by specific instructions, a general set of data transfer commands is available, including READB (Read Byte), READW (Read Word), WRITEB, WRITEW, COPYB, COPYW, DOCKIN, and DOCKOUT. The PORT and UNPORT Instructions actually change the routing of communications between the base station and the vehicle.

4.4.8.1.12 Program Control Instructions

As with any language, it is sometimes necessary to divert program flow from a straight sequence execution. For this purpose a simple set of conditional and unconditional JUMPs and CALLs is provided (JUMP, JUMP=, JUMP>, JUMP<, JUMP!=, CALL, CALL=, CALL>, CALL<, CALL!=, and RETURN).

4.4.8.1.13 Mathematical Instructions

The language supports elementary mathematics (ADD, SUB, MULT, DIV, and ABS), and simple vector mathematics (VECTOR). Many applications will not require the use of these features and useful programs may be written without them. Direct addressing, indirecting, and even arrays and structures are possible (though awkward) with the USE Instruction.

4.4.8.2 Simulator

The control system on-board the mobile robot performs a number of complex tasks, for example, the robot often has to achieve multiple goals such as moving ahead while also avoiding local obstacles. The robot usually has multiple sensors providing sonar data which has to be analyzed to extract information used for collision avoidance. The algorithms used to implement these functions depend on the mobile robot being used and the requirements of the application for which the robot is being used. The algorithms determine the performance of the control system.

The need often arises to evaluate the efficiency of the control system of a robot for a particular application or to determine which algorithms when implemented in the control system would best satisfy the requirements and constraints of the application. Simulation is a useful technique to evaluate and predict the behavior of a mobile robot system for a particular application.

The general design goals for the simulator are:

1. The same control software used to control the operation of the mobile robot should be used to control the operation of the simulator.
2. The supervisory link interface of the host (and hence the control software) should be maintained for the interface of the host with the simulator.
3. The control mechanism should be the same as for the mobile robot, i.e., the control messages to either should be identical.
4. The design should offer the flexibility of running the control software and the simulator on separate machines.
5. The design should be modular, thus changes required in the implementation for new system requirements are localized.
6. As far as possible, an attempt is made to reduce the dependence of the simulator on any one type of mobile robot.
7. The design should allow an efficient implementation.

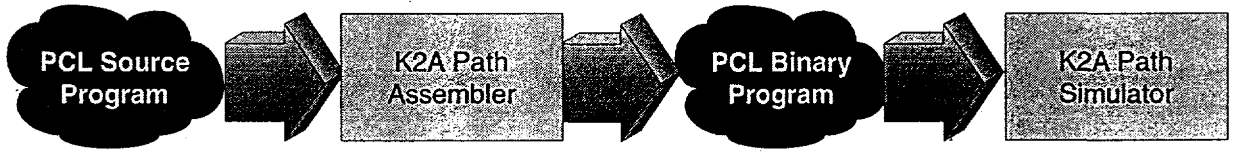


Figure 4-82. Use of the K2A simulator to execute a PCL program

The Cybermotion K2A Path Simulator (PSIM) simulates the path commands of the Cybermotion K2A Mobile Robot. PSIM accepts the standard binary output file of the Path Assembler.

The simulator may be executed as a stand-alone program with text output or may be interfaced to the control environment via the same socket mechanism as the actual robot. The latter is useful for training purposes and during development work on the environment.

4.4.8.2.1 The Simulator and the Robot World

The default world for the simulator is the two-dimensional x-y plane, as shown in figure 4-83. The absolute location of the K2A is given by a pair of x and y coordinate values. The simulator may also be initialized by use of a robot world file.

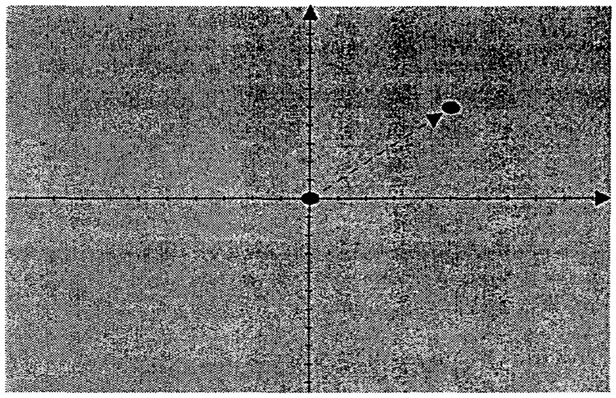


Figure 4-83. Robot world and coordinate system

4.4.8.2.2 Simulator Path Dynamics

A simulation represents an abstraction of the system being simulated. This abstraction suppresses details which are not relevant to the purpose of the simulation. For a simulation to be useful, it must be both *relevant* and *accurate.* By relevant, we mean that the simulation must address the appropriate attributes of the system being simulated. By correct, we mean that it must provide the same response for these items as does the real system. For the robot the key simulation item is the vector location of the robot, i.e., the robot's position, direction, and velocity. In order to compute these values we must know the *path dynamics,* i.e., the describing equations for movements by the robot.

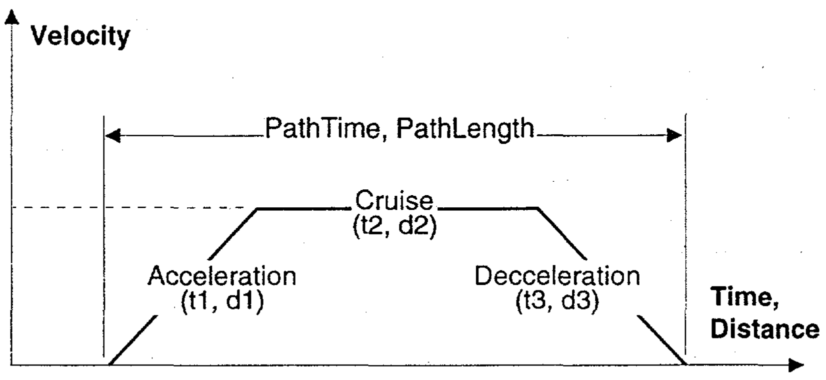


Figure 4-84. Typical path/steer velocity profile.

The K2A has two types of motion, turning and linear motion. We will refer to these as "turn" and "move" respectively. A typical velocity profile for a move or turn path is shown below. The path consists of three phases, *(i)* acceleration, *(ii)* steady-state (cruise), and *(iii)* deceleration.

Correctly simulating the motion of the robot requires that the general describing equations and the necessary coefficients be known for each phase. In addition, the simulator must be able to handle the special circumstances created by a short move or turn. If the path length is sufficiently small, then the robot will not have time to reach the cruise velocity and this phase will not occur. The profile for this type of motion is shown in figure 4-85.

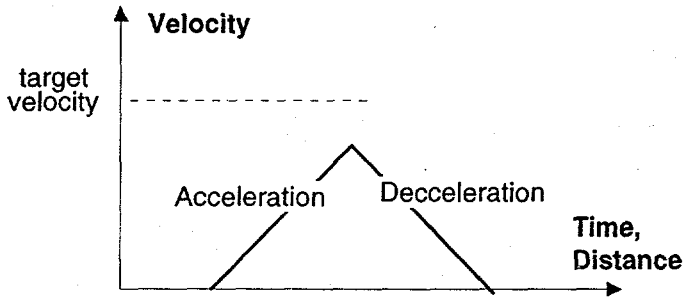


Figure 4-85. Exceptional (limited) case path/steer velocity profile

The simulator uses the equations derived for each of the three phases to handle all turns and moves. The coefficients were determined by *(i)* information from the Cybermotion manual, *(ii)* reading values from the robot while under operation, and *(iii)* consultation with Cybermotion. The simulated version of the robot appears to correctly model the behavior of the real robot.

4.5 ELECTRICAL SYSTEMS POWER ANALYSIS

Three of the six major tasks of WBS 2.3 involve power analysis and management of the inspection system. Two of the tasks overlap (see tasks 4. and 5. under section 4.1) in that they concern power consumption of the electrical and computer systems in the mobile base and the applications payload. These power consumption analysis tasks will be dealt with in an integrated way in the following sub-sections because of the interdependent construction of the inspection system.

The last task (see task 6. under section 4.1) is concerned with power management protocols and control. This task was mitigated during the evolution of the project to reflect the final design considerations of the mobile base and the payload. The final version of the vision system requires continuous operation of the image processing boards for maximum through-put. All computer and communications systems must be in continuous operation as well for correct and safe operation of the robot. The barcode lasers and rotating mirrors are powered continually because of the lag time in turning the mirrors on and off due to their inertia. Other components, such as laser diodes in the vision system, use very little power or are only powered up when needed (e.g. linear actuators for the CPS and 4-bar, and strobe lights for the camera heads).

4.5.1 Mobile Base, CPS, and Four-Bar

The energy use in the mobile robot base, the camera positioning system (CPS), and the four-bar mechanism is discussed. The drive motors, control electronics, and computer for the K3A platform are considered as one unit for analysis. The **ARIES** system is built upon a well-defined mobile platform provided by Cybermotion. This mobile platform is considered to be reliable and robust due to its long history of operation in the field. As such it was determined that the **ARIES** project would build upon this mobile platform with little modification of the internal electronic systems.

4.5.1.1 Mobile Base

A 16 hour test run of a K2A mobile system (predecessor of the K3A) was performed using a fixed path of travel. The path was as shown below in Figure 4-86. The vehicle was running a process that would stop at C and wait 10 seconds, run to B and wait 10 seconds, run to A and wait 10 seconds, run to D and wait 10 seconds, and then continue back to C. This process was designed to emulate a process of inspection and travel to drum columns and to new isles. The initial battery voltage was 26 V. After 16 hours of traversing the fixed path, the battery voltage was 24.1 V. Based on the recharge time, it was determined that the batteries had discharged to 20% of rated capacity and consumed 54.3 A•h.

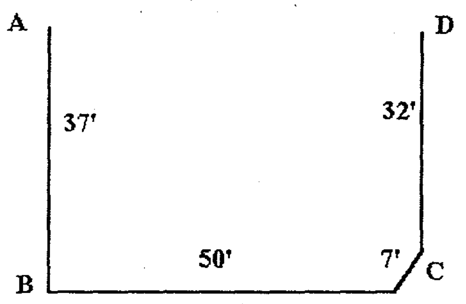


Figure 4-86. Mobile platform test path.

The extrapolated value of energy use by the mobile platform during its stationary inspection time (assumed to be 1 minute) of a column of four drums is calculated to be **0.42 A•h** (0.98 A x 26 V x 60 s). The extrapolated value of energy use by the mobile platform to move from one column to the next column (assumed to be 3 ft.) is **0.1 A•h**.

4.5.1.2 CPS and Four-Bar Actuators

	55 Gal.	**85 Gal.**	**110 Gal.**
CPS	0.82 W•h	1.05 W•h	1.19 W•h
CPS Time	25.2 s	31.1 s	34.6 s
Four-Bar	0.083 W•h	0.080 W•h	0.077 W•h
Four-Bar Time	10.1 s	9.7 s	9.4 s

Table 4-8. CPS and Four-Bar Energy Use and Movement Times

The CPS and four-bar actuators are dc servos. Two are required to operate the CPS and one for the four-bar mechanism. Tests were performed to determine their energy use and cycle time. The results are presented in Table 4-8. The values were obtained by measuring the energy use of the CPS starting from its rest position, moving to its appropriate height, corresponding to a column of 55, 85, or 110 gallon drums, and then returning to its fully stowed rest position. In like manner, the 4-bar energy use was measured with respect to its movement from the rest position to its correct inspection position on the lower drum, and then back to its fully stowed rest position.

4.5.2 Applications Payload and Computers

The continual power consumption of the applications payload is made up of various computer, control, communications, and processing boards that reside in the VME cage on-board. The boards are primarily for the control and vision systems. In addition, the power use of cooling fans and the barcode readers are included in Table 4-9.

Item	Qty	P @ 12 V	P @ 5 V	P @ -12 V
IMA Brd	2		40	
Camera	4	33.6		
AM-CLR	2	5.28	6	1.44
CM-HF	1		5	
CM-CLU	1		9	
Barcode	4	24		
HKMips	1	1.2	40	1.2
I/O Brd	1		7.5	
Serial Brd	1	2.28	23	2.28
1 GB Disk	1	5.04	2.5	
Ethernet	1	24		
Cooling Fan	3	5.76		
TOTAL		101.16 W	133 W	4.92 W

Table 4-9. Continual power consumption of vision and processing boards.

The strobe heads, strobe powerpacks, and laser diode range finders are energized intermittently. The energy use of these items has been measured and is given for a column of 4 drums as:

- Strobe head energy = 1,920 J/column = **0.53 W•h** (8 flashes/drum x 4 drums)
- Strobe powerpack energy = 720 J/column = **0.2 W•h.** (5 A x 12 V x 0.5 s x 6 charges/pack x 4 packs)
- Laser diode energy = 0.6 J/column = **0.0007 W•h.** (0.06 A x 5 V x 0.25 s x 2 per drum x 4 drums)

4.5.3 Total Energy Use And Available Energy

From the calculations and measurements of the preceeding two sub-sections, a worst-case energy consumption number for a column of four drums can be obtained. These values are for all mechancial and electrical systems except the on-board computer, control, and vision processing boards in the VME cage. The results are given in Table 4-10 where a stack of 55-gallon drums or 110-gallon drums is assumed for each item so as to give the highest energy consumption value.

Time for inspection of 1 column, image acquisition and processing, and movement to next column = 1 minute.

Drive System while stationary	0.42 W•h
Drive System while moving	0.10 W•h
CPS Actuators (2/3 of test value)	0.80 W•h
4-Bar Actuator	0.08 W•h
Strobe Heads	0.53 W•h
Strobe Powerpack	0.20 W•h
Laser Diodes	0.0007 W h
TOTAL	2.1307 W h

Table 4-10. Worst case energy consumption per column of 4 drums.

The total power consumed in one hour is then given as:

240 W + (2.1307 W•h /col. x 60 col./h) = **368** W.

The available energy on-board is **3,835 W•h**. This is from 235 A·h of batteries at 24 V. These are sealed lead-acid batteries in sizes of 4 each 100 A·h, 12 V and 4 each 17.5 A·h, 12 V connected to produce the 24 V dc bus. The available energy assumes discharge of the batteries to 20% of full charge and 85% efficiency in all the dc-dc converter modules. Other battery technologies were considered, but were discarded in favor of lead-acid because of cost and energy per weight. A full report on the technologies considered is given in the appendix.

By incorporating a safety factor of 30% on power usage (368 W x 1.3) in the system, an operating shift time of 8 hours (3,835 W•h /478 W) is still easily obtained with a corresponding inspection of over 1,900 drums. The time for recharging the system with the present charger is 12.5 h (at 15 A maximum). The charging scheme will be enhanced to reduce the total charge time to under 7 hours.

4.6 REFERENCES

1. J.L. Bostock, J.S. Byrd, and R.O. Pettus, "Topical Report: Mobile Robot Platform for an Intelligent Inspection and Survey Robot," Phase 1 METC DE-AC21-92MC29115, 31 March 1994.

2. K.H. Hill and J.S. Byrd, "An Intelligent Mobile Robot Control Architecture," *Proc. of ANS Fifth Topical on Robotics and Remote Systems,* Chicago, IL (1993).

5. VISION SYSTEM

5.1 ABSTRACT

This report documents the design of the ARIES #1 vision system (a component of Task WBS 2.3) used to acquire drum surface images under controlled conditions and subsequently perform autonomous visual inspection leading to a classification as 'acceptable' or 'suspect'. Specific topics considered herein include:

- Vision System Design Methodology.
- Algorithmic Structure.
- Hardware Processing Structure.
- Image Acquisition Hardware.

Most of these capabilities were demonstrated at the ARIES Phase 2 Demo which was held on November 30, 1995. Finally, Phase 3 efforts are briefly addressed.

5.2 INTRODUCTION

The ARIES #1 vision system, on the basis of visual information, makes autonomous decisions about the condition and size of stored drums. The system locates and identifies each drum, locates any unique visual features, characterizes relevant surface features of interest (such as paint blisters, rusted areas, etc.), and updates a database containing the inspection data. An adaptive algorithm and learning concept, requiring little effort by unskilled operators, will be featured to "train" the vision system prior to the actual inspection process.

5.3 VISION SYSTEM OVERALL OBJECTIVES

Visual assessment of drum condition is an autonomous assessment of visible and quantifiable surface characteristics based upon available image data. The problem is essentially a two-class risk minimization problem, where drums are classified as "acceptable" or "suspect." A drum would be considered "suspect" if it exhibits sufficient surface deterioration to warrant warning of possible failure. The system should err on the conservative side, i.e., the system should rarely miss a "suspect" drum, whereas misclassification of a good drum as "suspect,", while inconvenient, is not as significant.

The overlapping of class features makes this problem challenging. For example,

- Good and suspect drums exist in numerous (overlapping) colors.
- Good drums contain regular features, such as text and icons.
- Bad drums also contain regular features.
- Both classes contain bar codes (assumed).
- Both classes also display irregular texture due to other than shading; some is attributable to flaws and corrosion.

- Within drum color variation exists.

Based on expert opinions, the surface blemishes which indicate probable drum failure are rust patches on the order of 0.5" x 0.5" and paint blisters (indicating internal rust). The human inspector usually relies on the characteristic color of rust in classifying it, hence this was the obvious feature to utilize in segmenting rust patches. Likewise, the human uses patterns of the reflections from a blistered surface in classifying it. This suggested that the vision system could rely upon texture segmentation for classifying these regions. Detecting these features requires a color camera with sufficient resolution to resolve the texture elements of blisters and lighting with consistent color temperature and direction (with respect to the camera).

5.3.1 Summary of Key Vision System Development Parameters

The following are important features of the ARIES #1 vision system:

- Color image processing, based upon RGB to HSI conversion, is employed.
- All software is written in C; the top-level consists of mainly ITI hardware function calls. In addition, the vision hardware is independent from the ARIES robot platform and may be used with other image delivery systems.
- Supplemental multi-strobe lighting is used to reduce power consumption and compensate for non-uniform site illumination.
- Structured lighting is used for image segmentation and drum orientation detection.
- Other feature detection algorithms may be added, as needed.

5.3.2 Time and Power

5.3.2.1 Temporal Processing Constraints

The required drum inspection rate requires considerable computational power (which usually, given the temporal processing constraints, translates into electrical power). This is shown in Figure 5.1. *This constraint severely limits the amount of processing available per drum.* Other constraints include minimal size, weight, and power consumption.

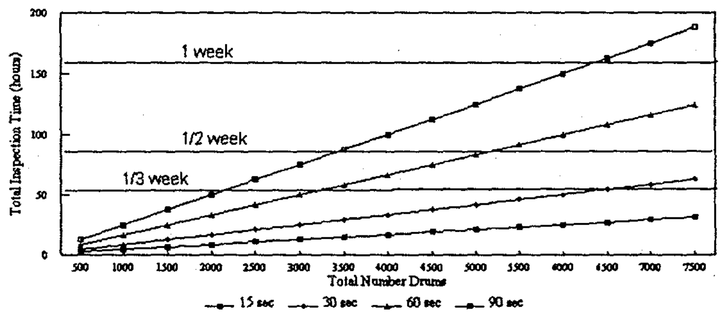


Figure 5-1: Inspection Capacity as Function of Single Drum Inspection Time.

5.3.2.2 *Current Temporal Processing Capability*

At the conclusion of Phase 2, the ARIES #1 vision system had the following temporal performance:

- All vision operations require approximately 3 seconds per half-drum (1 image).
- Total inspection time is therefore 6 seconds per drum. Further speedup may be possible, if faster strobe recharge times are enabled, and with further algorithm recoding.

5.4 ALGORITHMIC STRUCTURE

The overall structure of the visual inspection process which involves vision consists of the following steps:

1. Image delivery.
2. Image acquisition.
3. HSI region analysis.
4. Texture analysis.

In addition, an off-line learning algorithm is used to tune vision system parameters.

Processing of acquired image data occurs in two main steps:

1. The images(s) are segmented to ascertain that a drum, or drum portion, exists in the sensor field of view (FOV).
2. If a drum is found, the size and extent of the drum is computed (using active vision) and the images are subsequently classified by region.

Therefore, classifying a drum on the basis of passive visual information is carried out in four steps:

1. Segmenting the drum (section) from the total imaged scene, i.e., finding the image region corresponding to only the visible part of the drum in the image(s),
2. Segmenting rust regions,
3. Segmenting paint blisters; and,
4. Overall classification and recording of results.

5.5 SEGMENTATION ALGORITHMS

5.5.1 Definition

Segmentation is the process of finding a connected region with a specific property such as color or intensity, or a relationship (pattern) between pixels. Classification of a drum as "suspect" is done if the number of pixels in rusty regions or in paint blisters exceeds a specific threshold. The threshold is tunable, depending upon site-specific requirements. Since drum failure modes and human inspector assessments appear to be highly site-specific, it was deemed necessary that the algorithms should be adaptable to the site requirements. A

5.5.2 Addition of Features.

It is straightforward to add additional feature extraction approaches to the present system, once suitable decompositions into hardware-implementable computations are designed.

5.5.3 Features and Approaches Considered

Initially, all possible processing methodologies (model based, etc.) and potential features were investigated. Time constraints led to choice of a feature-extraction/classification (segmentation) strategy. Passive image features considered in Phases 1 and 2 include:

- Oriented Fourier features.
- Regional moments of various orders.
- Grey level difference statistics.
  - contrast
  - entropy
- Grey level run-length statistics.
  - grey-level distribution
  - run percentage (directionally oriented)

5.5.4 Training Data Available, Developed and Used

Training/Test Data Sources used in Phase 2 included:

- Photographs provided by WSRC/SRTC (several sets).
- Five videotapes of various sites/configurations (digitized large number of images from tapes) provided by FERMCO.
- Discussions during Hanford site visit, September 1993.
- Lab Drum(s): rusted and not rusted.
- USC mock-site drums: blistered, rusted and not rusted.

Since images were acquired under a variety of uncontrolled conditions (resolution, color accuracy, reproduction, lighting, etc.), we view the Phase 2 training data as necessary but not sufficient.

5.5.5 Image Compression

The use of compression algorithms could maximize on-robot 'suspect' image storage capacity. Stored images, if ultimately to be viewed by the operator, *could* be compressed with a lossy compression algorithm (jpeg), which is designed to 'fool' the human visual system. Our conclusion is that ARIES #1 should not use jpeg compression *prior to* computer processing for the following reasons:

- Resolution reduction occurs (jpeg characteristic).
- Texture-based algorithm performance is compromised (experimental results).
- Only modest space savings for reasonable (visible) image quality result.

5.5.6 Drum Segmentation

Due to warehouse imaging conditions, segmentation of the imaged drum from the scene background (which consists of, among other entities, other similar drums) is a challenging task. Traditional edge extraction and region segmentation techniques fail due to a variety of non-ideal working conditions (glossy drums, multiple reflections, gradual fading of intensity, etc.). This led to a knowledge-based technique which utilizes more information about the expected properties of the scene.

Another aspect of the drum segmentation is the exclusion of regions on the drum which are not to be analyzed for rust or blisters. These regions are paper labels such as barcodes and warning signs typically found on the drums. The diffuse reflections from paper are typically much greater than that from the paint. Since diffuse reflections are direction insensitive, the paper is segmented from the drum by finding regions which exhibit less change in intensity when the lighting direction is changed.

From the specification of the problem and the capabilities of the navigation/camera positioning system, the following is the list of assumptions which can be employed in segmenting the drum from the scene:

- The warehouse will contain only three drum sizes.
- Each stack of drums will contain a single drum size.
- Each drum has a homogeneous paint color. However there is no restriction on or *a priori* knowledge of the specific color.
- Projected laser dots are easily discernible on flat-painted surface.
- Specular reflections are easily discernible on glossy-painted surfaces.
- The height of the camera with respect to the floor is known.
- The distance to the drum surface from a camera will be within the range of 24" to 54".
- An image will contain the horizontal center and either the top or bottom edge of a drum.
- The dominant color of the visible region of a drum is its paint color (barcodes and other labels do not dominate the scene).

The technique that was developed which utilizes the above assumptions can be decomposed into a sequence of steps: finding the drum center and distance; finding a rough estimate of the vertical edges; finding the top or bottom edge; picking a drum size; and then refining the estimate of the vertical edges. Each of these steps is described in more detail below.

The process of finding the center of the drum starts by imaging the laser dots projected onto the surface of the drum. Using *a priori* knowledge of the laser/camera geometry, estimates of the three dimensional location of the dots are found. For each combination of 3 dots, the virtual vertical cylinder on which the dots lie is found. Any of these cylinders whose radius or location are not within the expected values corresponding to one of the three drum sizes and the known imaging geometry are excluded from further analysis. The set of projections of the virtual cylinder centers on the image plane will be referred to as the center-set.

In the case of glossy painted drums, the projected laser dots are much harder to detect since the majority of the laser energy is reflected specularly away from the camera. This reduces the reliability of the laser-center-finding technique. However, if the drum generates strong specular reflections, then the location of the specular spots from judiciously located lamps can be used to locate the center of the drum in the image. If the lights are vertically in line with the camera, specular reflections can only occur on surfaces for which the horizontal component of the surface normal vector is pointing at the camera. Since the drums are vertically oriented cylinders, only the vertical stripe closest to the camera, hence at the center of the drum image, will be oriented so that specular reflections can be seen. By isolating specular reflections matching the expected reflections from the lamps, candidate centers are determined and included in the center-set.

The drum center is then determined as the mean center of the largest cluster of estimated centers (a subset of center-set) whose span is less than a predetermined bound. Using this drum center and location of the laser dot closest to this center, the distance to the surface of the drum is estimated via geometry. Likewise, a rough estimate of the locations of the drum's vertical edges is obtained assuming a 55-gallon drum.

By restricting the analysis region to be within the rough estimates of the drum's vertical edges, it is safe to assume that the drum dominates the analysis region. Thus the ranges of hue, saturation, and intensity which characterize the drum can be determined via histograms. By selecting the pixels within these ranges, a binary image is constructed which represents the drum in the scene. This binary image is then compared to a set of previously generated templates. Each element in this set is a binary image which represents a drum whose top or bottom is at a known location in the image. Thus, the template which best matches the binary image provides the location of the actual top or bottom.

Knowing the vertical height of a drum top or bottom, the height of the camera, and the heights of the different drum sizes and pallets, the size of the drum is easily deduced. Once the drum size is known, the estimate of the locations of the drum's vertical edges can be improved which in turn defines the final analysis region.

5.5.7 Rust Segmentation

5.5.7.1 Color Space Fundamentals

One of the obvious properties of a rust region is its color; therefore, segmentation using pixel color as the characteristic element is used. Typically, the output of color video cameras is in a format which makes it difficult to detect colors, independent of variations in other parameters such as intensity. To facilitate the segmentation, the images are converted to a hue-saturation-intensity (HSI) color representation. In this color space, a color image is represented by the three gray-level images, referred to as planes.

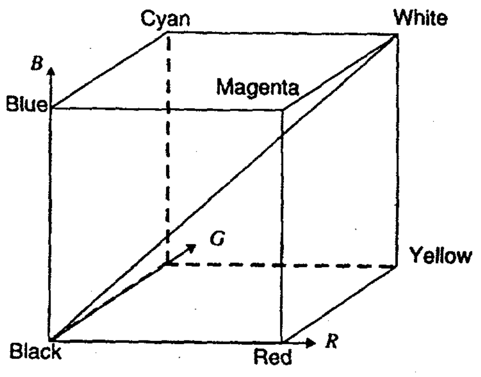


Figure 5-2: RGB Coordinate System

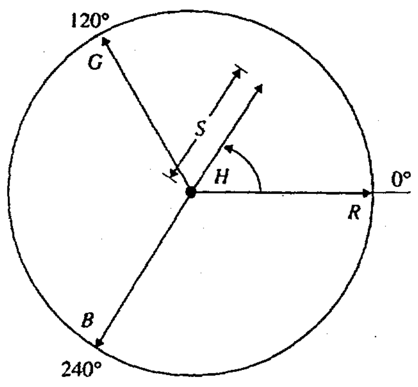


Figure 5-3: Hue Space Color Map (Red = 0 deg)

- Three corners (R,G,B) correspond to *primary colors.*
- The origin corresponds to no value of any primary color, and therefore is deemed 'black.'
- The maximum value of equal R,G,B values is 'white.'
- Locus of all points where R,G,B components are equal is the diagonal of the cube; referred to as *gray line.*
- The other 3 corners of the color cube correspond to *secondary colors*, yellow, cyan (blue-green), and magenta (purple).

In the hue plane, a pixel value is a numeric representation of the color. In the saturation plane, a pixel value is the purity of the color (high values indicate pure colors whereas low values indicate substantial mixing with white light). In the intensity plane, a pixel value denotes the brightness. In the implementation of HSI space, each pixel is represented by three values between 0 and 255. For the saturation and intensity, 0 represents the minimum values and 255 represents the maximum values. For the hue, the zero-reference is cyan; greens are around 45, reds are around 130, and blues are around 210. RGB to HSI conversion capability is provided in ITI hardware at a rate of 60 frames/second.

5.5.7.2 Application to Drum Images

In training data, rust was found to exhibit hue values typically between 90 and 160. It was also found that rust has a saturation value less than 60. Since this range is dependent on camera white balance, gamma corrections, and the lighting color temperature, this range must be established for any differences in these parameters. A pixel is considered a rust pixel if its hue and saturation falls within both of these ranges. Note, however, that if a pixel has either a high intensity or low saturation value, the hue information is unreliable. Considering this, the basic element for rust segmentation is a pixel with a hue value within the range of 90 to 160, a saturation value within the range of 12 to 60, and an intensity value less than 200.

Labeling all of the rust pixels is performed by thresholding the three planes over the ranges specified above and then performing a Boolean AND on the results. Connecting the rust pixels is the process of finding regions in which the density of rust pixels exceeds a predetermined threshold value. The density of rust associated with a given pixel is defined to be the number of rust pixels in a neighborhood of that pixel. A pixel with nine or more rust pixels in its 4x4 neighborhood is considered a member of a rusty region.

5.5.8 Paint Blister Segmentation

A paint blister is a conglomeration of several small, almost circular bubbles, protruding from the surface. Under controlled lighting conditions (intensity and geometry), the image of the light reflected off of one of these bubbles is a relatively consistent spatial intensity pattern. This suggested that the paint blister segmentation could rely on a spatial basic element. From frequency analysis over a set of training images, it was found that if a pixel, x(i,j), in the intensity plane satisfies the conjunction of the following constraints:

(c x(i,j) - x(i - DELTA i ,j) > 30 quad AND

x(i,j) - x(i + DELTA i,j) > 30 quad AND

x(i,j) - x(i,j-DELTA j) > 30 quad AND

x(i,j) - x(i,j+DELTA j) > 30 quad AND

x(i,j) > 50)

then, this pixel should be classified as part of a paint bubble. Depending upon the camera and optics specification (focal length, CCD chip size, resolution, etc.), DELTA i and DELTA j are chosen to optimize performance. Labeling the bubble pixels is performed by co-occurrence analysis with this pattern and connecting is performed as in the rust segmentation process.

5.6 LEARNING

Each of the vision algorithms used requires a set of operating parameters which directly affects system performance. To optimize this performance, an adaptation or optimization procedure to generate the optimum parameter set is required. Unfortunately, the problem cannot be directly formulated as a general nonlinear optimization problem because of the lack of a suitable quantitative performance measure or objective function. Instead of attempting to generate or estimate this performance measure, an interactive, operator-guided approach is used.

The operator interface used for system training has the form of a multi-window display. Each of the windows contains the resulting image from the segmentation algorithm using a specific set of parameters. The operators task is to only compare between windows to determine the relative quality of the results. The Nelder-Mead algorithm (Downhill Simplex Method) is the optimization procedure used. It requires a comparison of the objective function values at a limited number of search points (N+1 in N-dimension search space). For this algorithm, the operator need only decide the best and the worst images presented in the multi-window display. The algorithm uses the operator's decision to determine a new search "point" and hence a new set of images, derived from updated parameters, to present to the operator. This procedure continues until a satisfactory result is obtained.

5.7 RESOLUTION, WORKING DISTANCE AND FIELD OF VIEW

Design parameters for the image acquisition subsystem of the ARIES #1 vision system included the following concerns:

- Image Resolution vs. Available Camera Resolution.
- Number of Cameras vs. Number of Images .
- (Geometric) Distortion, which is especially significant when using structured light and requiring a large field of view (FOV).
- Design Goal: 1/2" x 1/2" minimum feature dimension, which requires 20 pixels/inch sensor resolution.

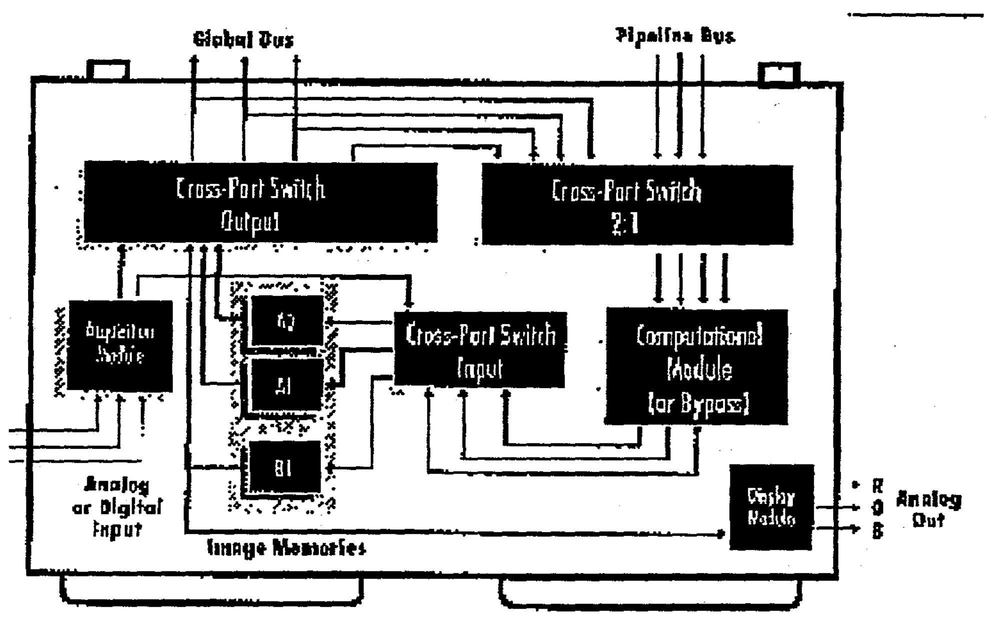
Ultimately, a 6 mm lens with 20-25 inch working distance, which yields 60% of the largest (110 gal) drum in the FOV was chosen. It should be noted that high resolution color technology, at the present time, requires multiple cameras.

5.8 ARIES #1 VISION HARDWARE AND ALGORITHM IMPLEMENTATION

5.8.1 Image Processing Hardware

Since system speed is an all-important constraint, specialized image processing/computer vision hardware is used to implement the aforementioned segmentation algorithms. The vision industry is following two design strategies, one in which hardware is tailored for specific vision processing tasks, and the other in which general DSP chip set is utilized to provide more generic capabilities. The former strategy provides for greater performance if one's algorithm can be decomposed into a series of subtasks implementable by the hardware. The latter strategy compromises speed for computational flexibility. The system selected falls within the first category.

ARIES #1 employs a modular vision system, manufactured by Imaging Technology Inc. (ITI), with full scale pipeline processing capabilities. The system uses two ITI IMA150/40 memory managers, each with four Mbytes of reconfigurable memory. These cards are designed to carry submodules that perform different image processing operations. One of its design features is a multi-input-multi-output cross-port switch that allows the reconfiguration of the pipeline for different operations. The current system uses two ITI IMA150/40 memory managers, each with four Mbytes of reconfigurable memory.



IMS-150/40 Standard Image Manager Block Diagram

Figure 5-4: ITI IMA

The following submodules are used:

1) *Acquisition Module:* The acquisition module is used for digitizing a true color image (24 bits) from an RGB PAL camera at 25 frames/second. The resolution used is (768 x 572), which leads to a digitization rate of (768 x 572 x 3 x 25 = 32.94 Mbytes/second). Also, this module performs the conversion of the camera's RGB output to the needed HSI color space in real time.

2) *Convolver/Arithmetic Logic Unit*: This is the main module used for most of the vision tasks required for image segmentation. This includes convolutions, thresholding, rank filtering and connecting. This module also carries a statistical processor which is used to count the number of pixels that belong to each class during the segmentation process.

3) *Histogram/ Feature Extraction Processor:* This processor is used to perform the necessary histogram operation required for identifying the drum color. The module is

also used to generate both vertical and horizontal projections of the image, which is necessary to locate the drum in the acquired image.

5.8.2 Image Acquisition System and Hardware

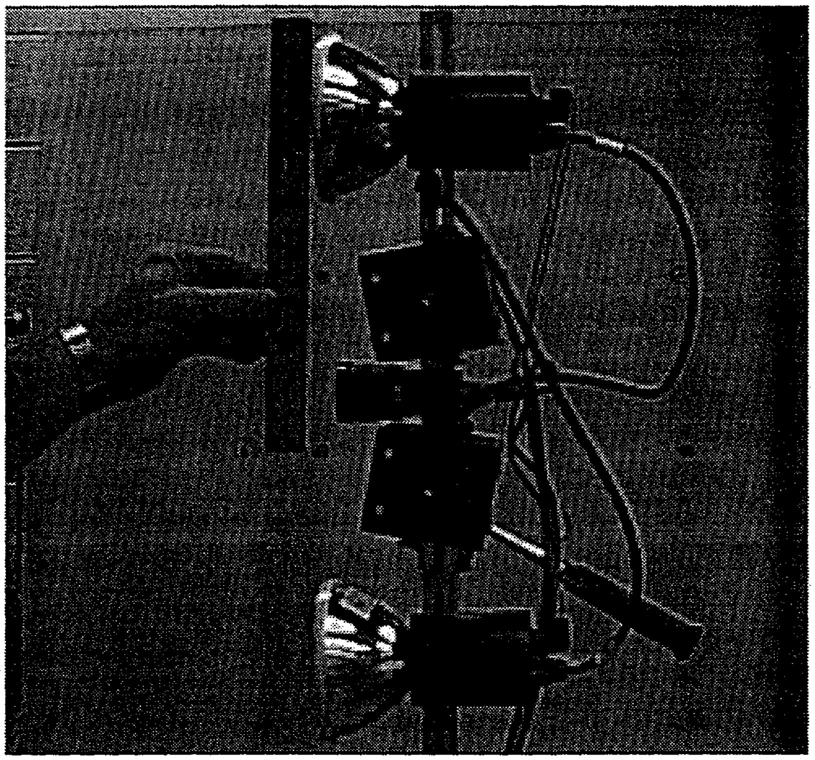


Figure 5-5: Basic Imaging Head (Dot Projector Not Shown)

5.8.3 Passive Components

The imaging system includes four camera subsystems, one for each drum in a stack of four. It also includes two digitizing modules which provide the ability to digitize images from all four cameras. Each camera subsystem consists of a camera with one strobe lamp above it and one below it. It also incorporates a five-dot laser structured-light source. The expected variability in site ambient lighting and power limitations of the mobile robot led to the development of a strobe-based image acquisition system. The camera used is a Panasonic GP-US502E, three-CCD high-resolution color camera.

To gain more vertical resolution, the version of the camera was chosen to follow the CCIR PAL standard. In this standard the camera provides 572 lines versus the 480 lines specified in the RS-170 NTSC standards commonly used in the USA. A single PAL image, referred to as a frame, consists of two interlaced fields separated in time by 0.02 seconds. To prevent "striping" in the image, one must provide consistent lighting to both of these fields. Since we are using strobe lighting, a single flash must be provided for each field and synchronized to that field (specifically at the start of the CCD integration time).

Utilizing the memory management card internal flags, which indicate the start of the next field, and a custom strobe control circuit, we are able to synchronize any strobe lamp to a field. However, since a single strobe has a recharge time on the order of two seconds, we are unable to provide the lighting for two successive fields (one frame) with a single strobe lamp. Even though the fields are lighted at the appropriate time, since the two strobes are spatially separated, the resultant lighting is not consistent in both fields yielding "striped" images. The software solution to this problem is addressed next.

5.8.3.1 Illumination

Each camera system consists of a camera with one strobe lamp above it and one below it. Since a single image acquired by the camera will have inconsistently illuminated fields due to the spatial separation of the lamps, two images are taken, one in which the first field is lit by the upper lamp while the second is lit by the lower and another in which the first field is lit by the lower lamp and the second by the upper. Combining the first field from the first image and the second field from the second image results in an image in which both fields are lit by the upper lamp. Similarly, an image lit by the lower lamp is generated. A technique using the ITI Convolver/Arithmetic-Unit module has been developed to accomplish this process at real-time rates.

5.9 ADDITIONAL ALGORITHMS DEVELOPED

During Phase 2, several ancillary vision algorithms were developed. Several of these may be considered for implementation in Phase 3. Specific algorithms were:

5.9.1 Real-Time Drum-Center Detection Using Active Vision

This capability will be fused with the sonar-based drum location algorithms and allows more precise positioning of the vision system. This capability has been demonstrated.

The objective of this algorithm is to find the center of the drum while the robot is moving. This operation is necessary for the robot to stop in the right position with the stack of drums to be analyzed in the center of the camera's field of view. Currently this operation is done using a sonar system.

The system utilizes one of the cameras and laser dot projectors. This projector is located about 10 inches above the center of the camera pointing down such that the dot is within the field of view for the anticipated working ranges. This camera-laser arrangement allows a rough estimate for the range between the camera point-of-projection and the reflection of the laser dot on the surface of the drum. The smaller the range is, the higher the location of the projection of laser dot on the image plane. Thus, as the robot moves, the laser dot moves across the surface of the drum changing the range from maximum at the edge of the drum to a minimum at the center to again a maximum at the other end. This change in range can be detected from the projection of the laser dot in the image plane. Based upon this idea, the following algorithm was developed in order to find the center of the drum:

1. Acquire an image with laser on.
2. Locate the laser dot in the image if it exists.

3. Depending upon the location of the dot in this image compared to the previous image, we may have one of the following three events:
    - UP: The location of the projection of the dot went up which implies a reduction in the range.
    - DOWN: The location of the projection of the dot went down which implies an expansion in the range.
    - FLAT: The location of the projection of the dot did not change which implies no change in the range.
4. If the laser dot is crossing the drum while acquiring a sequence of frames we should expect a sequence of events having the form:[UP, UP, ..., FLAT, FLAT ..., DOWN, DOWN, ...]. The occurrence of this pattern is a good indication that the laser dot has passed across a drum.

The implementation of this idea assumes that the speed of the robot is defined by range of maximum and minimum expected speeds. Accordingly, the algorithm was designed to tolerate variations of the robot speed. A more accurate version of this algorithm could be developed if accurate information about the instantaneous speed (or location) of the robot is provided.

5.9.2 Barcode Detection via Textural Analysis

This capability was carried over from Phase I, and implemented using the ITI hardware. In Phase 3, the possibility of reading barcodes via the vision system sensors, as opposed to freestanding barcode readers, will be investigated.

5.9.3 An Efficient and Accurate Algorithm for Direct Measurement of Cylindrical Surface Parameters

The basic idea is to project a horizontal line pattern onto the object, which is assumed to be a cylinder. A passive image is acquired by a camera and some characteristic coordinates are extracted from the image of the stripes. These points are then used for an initial estimation of the position, orientation and size of the imaged object, resulting in an initial set of surface parameters. Then, the position of the characteristic points for a projection of the same light pattern on the object given by the estimated surface parameters is computed. The position, location and size features of this "virtual" cylinder are estimated, using the same methods as for the actual image. The differences between the features computed from actual and estimated/virtual object are used for an Affine transform of the estimated cylinder parameters. This transform results in an updated, improved set of surface parameters describing the object. The resulting virtual passive image is again computed and the correction step repeated as necessary. The real image taken by the camera and the virtual image should match. Also, the surface parameters of the object should be very accurately estimated by the surface parameters obtained by the successive Affine transform steps. ICA is capable of delivering very accurate results with moderate computational cost. An advantage is that the result is successively improved. If only a coarse estimation is needed, the algorithm could be stopped after one or two steps.

5.10 REFERENCES

1. A. Busboom and R.J. Schalkoff, "Direct surface parameter estimation using structured light: A predictor-corrector based approach." *Image and Vision Computing,* 1996 (to appear).

2. R.J. Schalkoff et.al. "A modular and extendible image processing system: Update to version ip6.1 (including DOE inspection effort)." Technical Report TR-080693-0915-I, Dept. of Electrical and Computer Engineering, Clemson University, August 1993.

3. A.K. Jain. *Fundamentals of Digital Image Processing*. Prentice-Hall, 1989.

4. R.J. Schalkoff. *Digital Image Processing and Computer Vision*. John Wiley and Sons, 1989.

5. R.J. Schalkoff. *Pattern Recognition: Statistical, Structural and Neural Approaches*. John Wiley, 1992.

6. DEXTEROUS INSPECTION MODULE

6.1 INTRODUCTION

The overall objective of this work was to develop a pre-prototype of an enhanced dexterity inspection package to be deployed on a semi-autonomous mobile base being developed at the University of South Carolina. This work was a sub-task under Task WBS 2.4. Currently a vision module is deployed on the mobile base, however, in the case of a suspect drum which requires close inspection, the Enhanced Dexterity Package may be deployed. With its six-degree-of-freedom revolute manipulator equipped with an integrated six-axis force sensor and electric gripper, two-degree-of-freedom pan/tilt unit, and cameras mounted on the end effector and on the pan/tilt unit, the Enhanced Dexterity Package is capable of reading a drum's bar coded inventory number, performing a close range inspection of the drum's aisle facing surface, and, if necessary, interacting in a tactile manner with the drum surface for the purpose of obtaining surface samples and testing surface integrity.

An additional component of the Enhanced Dexterity Package is a Telepresence Module which may be deployed along with the Dexterous Inspection Module, or by itself. This system consists of a pan/tilt/vergence unit with two cameras mounted on it, and a stereo head-mounted display which the operator wears. The operator is immersed in the remote environment (where the Telepresence Module is physically located) and receives three dimensional stereo live video from the cameras. The operator's head motion controls the pan/tilt/vergence unit's motion via a head mounted magnetic tracker. If used in conjunction with the Dexterous Inspection Module, the Telepresence Module allows the operator to observe the manipulator's interaction with the workcell in real time using 3D video. If used alone, it can serve as a human operated vision package, where the pan/tilt/vergence unit is a proxy for a human inspector, being exposed to the hazardous environment, while the human operator interacts with the environment remotely from his workstation.

6.2 PHASE 1 SUMMARY

The following summarizes the assumptions, requirements, and work performed in Phase 1 for the Enhanced Dexterity Package.

6.2.1 Specifications

Environmental Assumptions - The assumptions listed below represent the current best knowledge of the environment. It is to be expected that dimensions and stacking of the waste drums in a given facility will be different. It is imperative that provisions for such variations be included in the final design.

- *Operating Heights* - The primary assumptions are in regard to the maximum and minimum heights at which the manipulator must operate. The problem is defined using the standard 208 liter (55 gallon) drum, which has a height of 0.876m (34.5") and a diameter of 0.584m (23"). Stacking this standard drum four units high, with a 0.1m (4") pallet between units and on the floor, will produce a stack which is approximately 4m (13') high. It is assumed that the manipulator must perform all functions up to the maximum height of 4m (13') and perform these same functions down to the minimum height of 0.1m (4"). Other drum stacking configurations based on different size drums are considered to fall within this set of operating heights.

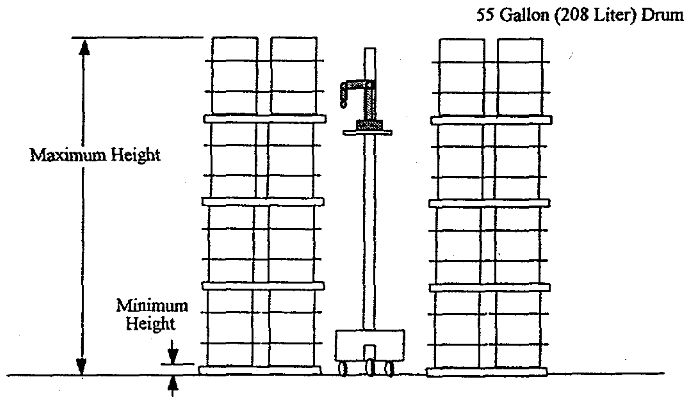


Figure 6-1 Operating Height Requirements

- *Accessible Drum Surface* - The size and stacking pattern of the drums will dictate the amount of exposed surface of a given drum. It is assumed that for the 208 liter (55 gallon) drum, the maximum inspection surface is approximately 1/3 of the total vertical drum surface. The different sized drums will have different amounts of exposed surface; however, the problem is again posed in terms of the 208 liter (55 gallon) drum with capacity planned for larger or smaller drums.

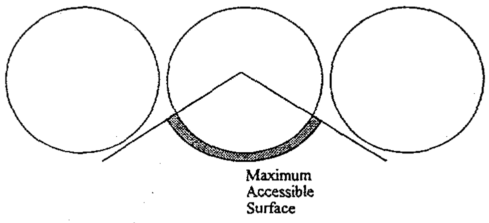


Figure 6-2 Accessible Drum Surface

- *Aisles* - The drums are assumed to be stacked such that a 0.92 m (36") aisle exists between facing drum stacks. The mobile robot will move along the center of the aisle, leaving 0.46 m (18") from the center of the mobile robot to the drums.

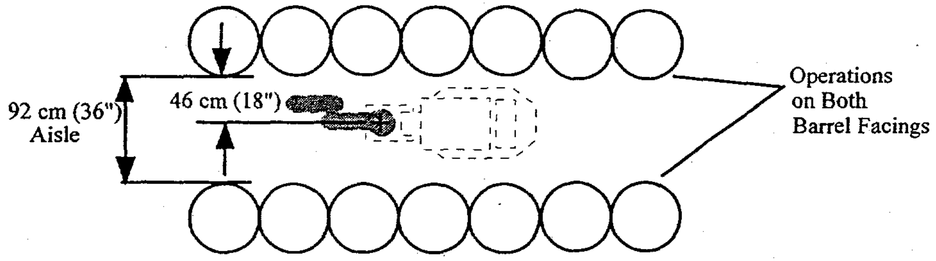


Figure 6-3 Aisle Width Assumptions

- *Communication* - A final assumption is that an antenna(s) can be mounted near the ceiling of the facility in order to allow wireless communication. This assumption is required to provide for teleoperation through a wireless connection.

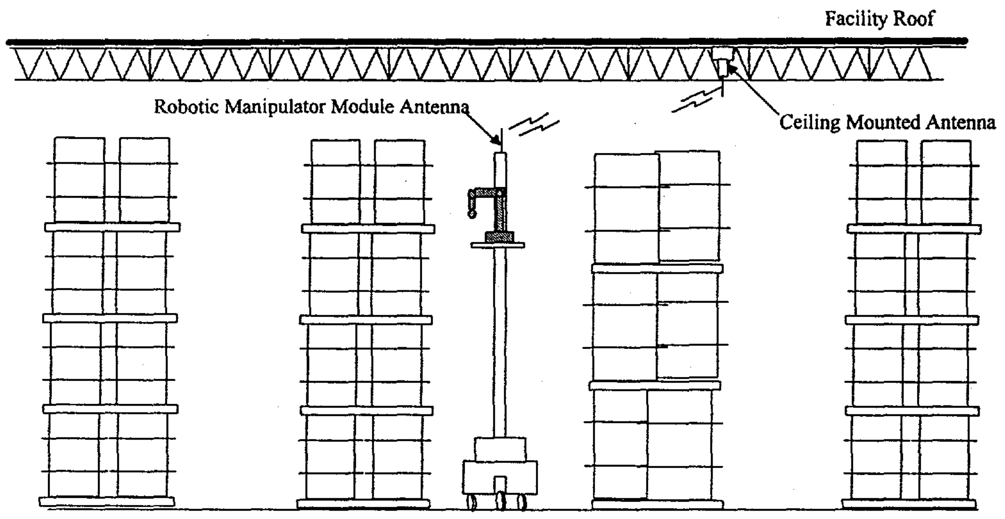


Figure 6-4 Communication

Tasks Identified - The tasks which were identified as being useful to an outside operator in the determination of the state of a given waste drum are listed below.

- *Bar Code Reading* - Each drum is assumed to be uniquely identified by a standard code 39 bar code. The Enhanced Dexterity Package should be capable of moving a portable reader to identify a particular drum from its bar code.
- *Wipe Sampling* - In order to identify a stain on the drum surface, a wipe sample should be taken for outside analysis. The sample must be correlated with the sample location.
- *Scraping/Probing* - To identify the extent of rust stain, scraping with a blunt blade will prove useful.
- *Movable Camera* - Video inspection by a remote operator.

Payload Requirements - Based on the identified tasks, the Enhanced Dexterity Package should be capable of moving a 0.91 kg (2 lb.) tool to an arbitrary orientation within its workspace.

Miscellaneous - Additional characteristics of the proposed system are listed below.

- *Teleoperation* - The operator can use any tools through teleoperation mode whereby the operator moves a small model of the dexterous arm to cause movement of the physical arm.
- *Video Interface* -The operator interface will include live and captured video in order for the operator to direct sampling and other inspection tasks.

Components - Specifications for a complete Enhanced Dexterity Package were developed. These specifications, in most cases, detail the commercial components and possible sources recommended in the final package.

6.2.2 Experimental Validation

Below are the tasks which were performed in order to validate the environmental assumptions, feasibility of performing certain tasks, and roadblocks to integration of a complete system. A mock-up of a proposed dexterity package was constructed and the following items were demonstrated.

Force Control - An implementation of a force controller which caused a robotic arm to move a low friction tool (a wheel) across a flat surface was performed.

Tool changing - A prototype tool changer system was built in-house. The changer was used to switch between task specific tools in a sample inspection.

Bar Code Reading - Using a portable bar code scanner, a commercial vision processing system, and a PUMA 560 robot the following experiment was successfully performed.

1. Video camera was pointed at a drum. A vision system used this image to locate the bar code by its characteristic size and shape.
2. Transformation of the output of the video system produced the physical location of the bar code on the drum.
3. The manipulator arm moved the portable bar code reader to the calculated location to read the bar code.

Wipe Sampling - An effective wipe sampling algorithm was demonstrated. The demonstration was performed using a PUMA 260 manipulator arm, Sun SPARC 10 with video board, and control hardware and software described below. The demonstration is described as follows.

1. Manipulator was positioned in front of a drum to be inspected.
2. A laser range finder was used to locate the center of the drum.
3. The manipulator arm was moved to capture an image of the drum surface. This image was displayed on the workstation.
4. The operator moved the workstation mouse to place a crosshair at a location on the video image where wipe sampling was to be performed (the drum was marked with chalk to simulate suspect spots).
5. Based on transformations of the crosshair on the video image into real world coordinates, the arm moved to wipe sample the desired locations (the chalk marks were wiped clean).

Lifting Platform - A prototype of a mechanical device for raising and lowering the PUMA 260 manipulator arm on a vertical stack of three drums was assembled. The platform was actively used in the wipe sampling demonstrations.

Task Tools - All tools used in the demonstration were integrated into the tool changer system. The tools constructed and used in the demonstrations are listed below.

- Bar Code Reader
- Laser Range Finder
- Wipe Sampler
- Moveable Inspection Camera

Control Hardware - A complete system of high and low-level control was implemented for the PUMA 260 and the PUMA 560 robots. The hardware involves a VME-bus scientific workstation, including three special purpose plug-in boards, which is connected to a factory supplied low-level controller to provide complete robot control and sensor integration. Similar hardware is envisioned for a mobile implementation.

Control Software - A package of C-language based robot control functions was installed on the hardware system described above. The software, called Robot Control C Library (RCCL)/ Real Time Control Interface (RCI) [Ref. 1], consists of a communication package installed on the low-level factory controller and a set of high-level programming commands installed on the operator workstation. The high-level commands are used to specify robot motions and commands for sensor reading. Based on these high-level commands, low-level commands are generated by RCI and sent to the low-level factory controller for execution.

Remote Operation - The wipe sampling demonstration was repeated from a desktop computer in a separate building from where the wipe sampling equipment was located. A connection was established between the two locations using a campus-wide Ethernet network. The wipe sampling experiment was successfully repeated. This is important as the digital link between the mobile system and remote operator will be a wireless system based on the Ethernet protocol.

6.2.3 Evolution from Phase 1

The main thrust of Phase 2 was the implementation and refinement of a prototype manipulation and telepresence payload. The project was divided into three systems: the actual Dexterous Inspection Module, the operator workstation, and the Telepresence Module. The major changes from Phase 1 to Phase 2 are listed below.

- A new manipulator was selected for Phase 2. The IMI Zebra Zero Force Manipulator was found to be better suited for the task than the Puma 260, and is also a more economical choice. One of the main concerns with the Puma was the large amount of power required by its controller (1200W). The Zebra runs off a 24VDC power supply, with typical operation drawing about 7A (depending on speed of operation). In addition, the Zebra's controller is integrated into the robot's base link, whereas the Puma 260's controller was a separate cabinet which weighs 80 pounds. The Zebra also has an integrated 6 axis force sensor, an electric gripper, and can be controlled from an embedded x86 based PC while the Puma 260 was controlled by a rack mounted VME based Sun workstation. The Zebra is also less expensive than the Puma 260, costing $30,500 new (including controller and PC), while the Puma costs about $28,000 for a used arm, not including all the necessary interface hardware. A new Puma 260 can cost up to $90,000.
- A new hardware platform for the operator console was chosen. The switch was made from a Sparc 10 with an IMS1000 video capture/display board to a Silicon Graphics Indy workstation with built in video capture/display. The reasons for switching include compatibility with the rest of the project investigators (USC uses Silicon Graphics workstations for their operator workstations), improved video capture/display capabilities, and the Indy workstation's exceptional graphics capabilities.

- To allow both sides of an aisle to be inspected, the fixed overhead camera mount of Phase 1 was replaced with a Directed Perception stepper-motor-based pan/tilt unit. This unit has low power operation modes, including a low-power hold mode in which there is no power applied to the motors, only a small current is applied to the electronic circuitry. Using this pan/tilt unit allows the operator to acquire images of all three drum sections, as well as of drums on both sides of the aisle.
- A video camera has been mounted to the manipulator's end effector. This camera is used to decode the inventory number bar code, as well as any other bar codes present on the drum, and to perform close range inspection of suspect spots on the drum. The end effector can be used as a local pan/tilt unit which can be positioned anywhere near the drum's visible surface.
- The Puma 260's tool changer has been replaced by the Zebra's electric gripper and the end effector mounted camera. Surface sampling tools are now acquired and held with the gripper. Since the bar code is decoded using the camera, there is no need for an additional bar-code reader tool.
- A Telepresence Module has been added. This consists of a brush DC motor TRC pan/tilt/vergence unit, a stereo head mounted display, and a head-mounted magnetic tracker. The TRC pan/tilt/vergence unit carries two video cameras, one for the left eye and one for the right eye. The horizontal displacement of the cameras provides a stereo image pair to the operator, who wears a stereo head mounted display. The head mounted display is equipped with a magnetic tracker which reports the operator's head position to the pan/tilt/vergence unit, allowing the operator to control the pan/tilt/vergence unit with his head motion. The fact that a stereo image pair is used gives the operator 3-dimensional live video, immersing him in the remote environment.

6.3 PHASE 2 SUMMARY

The following summarizes the assumptions, requirements, and work performed in Phase 2 for the Dexterous Inspection Module.

6.3.1 Specifications

The following specifications for the Enhanced Dexterity Package are from the Phase 2 proposal document [Ref. 2].

1. The contractor shall identify the control system specifications required of a robotics manipulator (i.e., arm) mounted on a mobile platform if the robot is to perform inspection and other tasks in an unstructured environment.
2. The contractor shall develop the control and sensor interfacing software which will enable appropriate manipulator movement and its effective integration with external sensing (i.e., vision and force).

Control System Specifications - The specifications for a manipulator that is to be deployed in a role such as mobile inspection of toxic waste containers have been identified below.

- Low power DC operation
- Six degrees-of-freedom for arbitrary position/orientation
- Electric gripper for tool manipulation
- Light weight
- Controller must be embeddable (i.e. light weight, small bulk, low power)
- Force sensor must be integrated into all tactile operations
- Payload capacity such that a camera and tool can be deployed on the end effector. Two pounds is sufficient.

Collision Avoidance - Collisions between the manipulator and the drum can occur during the inspection process if a linear-joint-space path planner is used. To avoid the added complexity of writing a custom path planner, paths which would cross certain points were segmented with via points. These added via points imposed constraints on the path planner such that the manipulator can not collide with other objects in its workcell (drum, pan/tilt unit, etc.).

Low Level Interface Software - Low level interface software was written for the Directed Perception pan/tilt unit, the TRC pan/tilt unit, and the Indy VINO (video in/no out) video capture/display hardware. Server programs were then written for all of these devices, and for the Zebra manipulator, which provided high level interfaces to clients. The clients can connect to the servers from anywhere on the Internet using TCP/IP connection-based sockets.

Interface/System Integration - All components, including the manipulator, controller, cameras, pan/tilt units, and operator workstation were integrated over a heterogeneous computing and networking environment. This involved developing high and low level control software, as well as network interface software, on DOS, QNX, and UNIX. All servers were developed so that they could be connected to by clients from anywhere on the Internet using TCP/IP sockets. Although for the purpose of this project the three components are integrated, they are bound together only with a standard Ethernet link, and by a well defined high level message passing interface. These three components could easily be integrated into other systems by simply connecting them on the same Ethernet network.

6.3.2 System Architecture

The Enhanced Dexterity Package consists of the two subsystems listed below.

- Dexterous Inspection Module
- Telepresence Module

Figure 6-5 below shows the software and hardware components which make up the Enhanced Dexterity Package, as well as their physical and logical interconnections. Hardware components are represented by boxes with single borders; software components are represented by boxes with double left and right borders and are enclosed within the hardware components on which they execute; and, physical connections are represented by fat arrows, while data flow is represented by thin connectors.

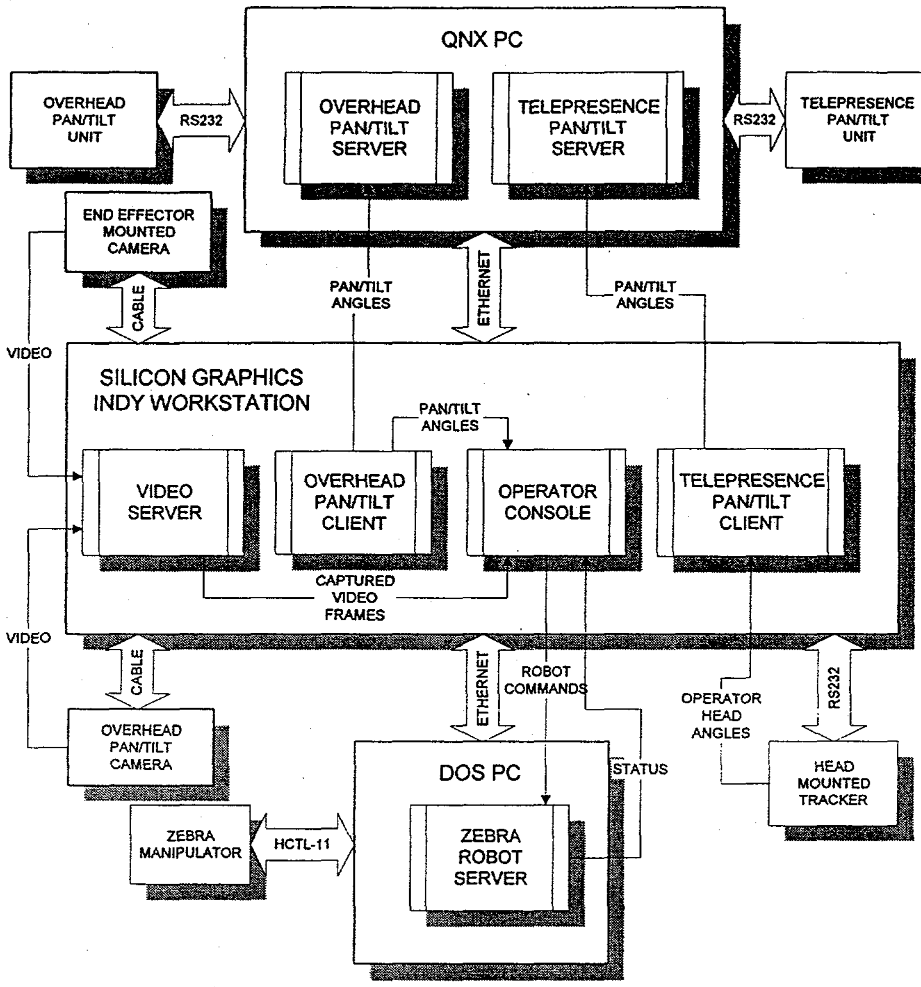


Figure 6-5 Block Diagram of Enhanced Dexterity Package

The dexterous inspection subsystem consists of the hardware listed below.

- IMI Zebra Zero Force manipulator
- PC based robot controller
- End effector mounted camera
- Directed Perception pan/tilt unit
- Pan/tilt unit mounted camera
- Silicon Graphics Indy workstation
- VINO (video in/no out) video capture/display hardware

The Dexterous Inspection Module performs the functions listed below.

- Bar code capture/decode
- Close range inspection of drum surface
- Obtain surface samples of suspect areas
- Inspect drum using overhead pan/tilt mounted camera

The Telepresence Module consists of the hardware listed below.

- Transition Research Corporation UniSight pan/tilt base with BiSight vergence head
- Two cameras
- PC controller
- Head mounted stereo video display
- Head mounted magnetic tracker
- Silicon Graphics Indy workstation (same as used with the Dexterous Inspection Module). Any computer with a high speed RS232 (38,400 BPS) port could be used, but since the Indy was used as the operator console for the Dexterous Inspection Module, and it has two serial ports, it was used with this module as well.

The Telepresence Module performs the following functions:

- Monitoring of the Dexterous Inspection Module while it is performing inspection functions
- Stand alone human proxy inspection

6.3.3 Experimental Validation

Below are the tasks used to validate the Enhanced Dexterity Package. These tasks would be performed during a routine inspection mission of a drum which had been identified as "suspect" by the semi-autonomous vision package. All of the above mentioned functions were incorporated into these tasks.

6.3.3.1 Dexterous Inspection Module

Bar Code Capture/Decoding - The end effector was used to position a video camera in front of the bar-code on the drum, and the bar-code was then decoded from the captured image from this camera. This bar-code identifies the inventory number of the drum, and possibly other information, such as drum weight, contents, etc. The location of the bar code on the drum was marked by the operator on an image captured from the overhead pan/tilt unit-mounted camera by clicking with a mouse pointer in a video display window.

Close Inspection of Suspect Spots - Suspect spots can be identified by the operator in the overhead pan/tilt camera view. The suspect spots are then marked on the image with a mouse click, and the manipulator positions the end effector mounted camera in front of the suspect spot. Local pan/tilt functions were implemented using the last three joints of the manipulator. The operator can also move the end effector camera over the surface of the drum with a few simple mouse clicks.

Surface Sample Acquisition - The manipulator acquired sponge sampling tools, and at the operator's direction, obtained surface samples of suspect spots on the drum surface. The wrist mounted force sensor was used to regulate contact forces during sampling (to avoid damage to the drum or the end effector) as well as during tool acquisition and discharge.

6.3.3.2 Telepresence Module

It was determined that although the operator could watch the inspection process in a live video window on the operator workstation, this did not really involve him in the inspection process in a natural manner. The decision was made to implement a Telepresence Module, which would allow the operator to immerse himself in the remote environment. This idea was validated by the reaction of volunteer test subjects who responded positively to the 3D stereo video effect, and to the natural method of control. An operator can don and move about the head mounted display to perform a remote inspection using the TRC pan/tilt/vergence unit with no prior training and with no interaction with a computer.

6.4 DEXTEROUS INSPECTION MODULE

The Dexterous Inspection Module represents one component of a modular solution to the waste drum inspection problem. The flowchart in Figure 6-6 outlines the dexterous inspection of one drum, suggesting the role of the Dexterous Inspection Module in the waste storage facility inspection process: to provide a more detailed investigation of an individual drum than can be accomplished by the semi-autonomous Video Inspection package. With this role in mind, a flexible system was developed which allows an operator to remotely inspect individual drums.

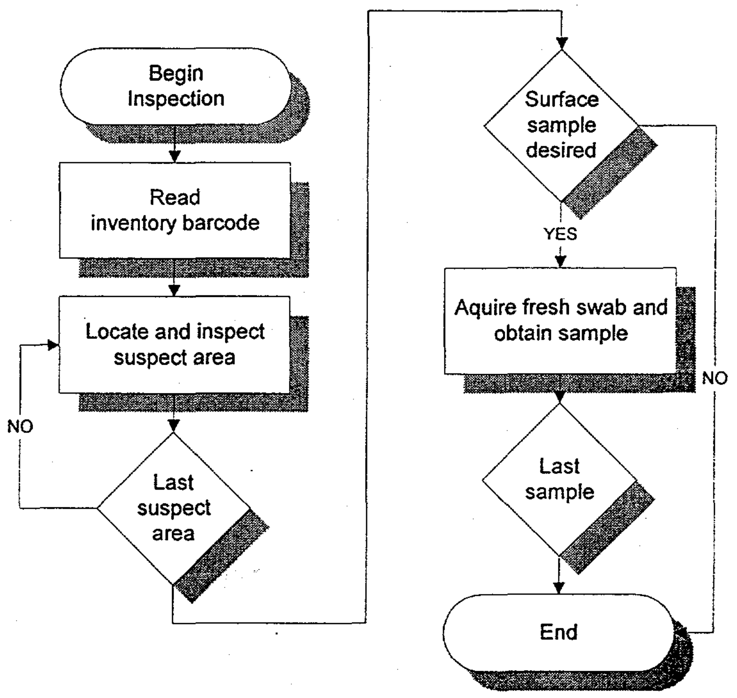


Figure 6-6 Inspection Process Flowchart

The Video Inspection package will identify waste drums that appear damaged or leaking. In order to determine the exact state of these suspect drums, the Dexterous Inspection Module will be equipped to provide various inspection functions for a detailed investigation. The functions currently performed by the Dexterous Inspection Module include:

- Inspection using an overhead pan/tilt unit mounted camera
- Close range inspection using an end effector mounted camera
- Capture and decoding of bar code labels on the drum
- Surface sampling

The use of a general purpose dexterous manipulator equipped with a gripper allows for addition of other functions through reprogramming. Figure 6-7 shows how the Dexterous Inspection Module would be combined with the mobile base for deployment.

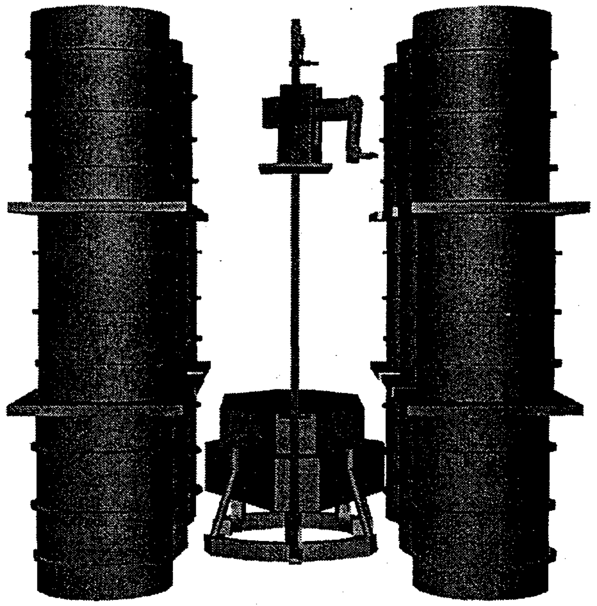


Figure 6-7 Proposed Deployment of Dexterous Manipulation Module

The following sections describe the individual components and interconnections that constitute the Dexterous Inspection Module. First, the problem is defined in terms of a typical drum inspection and the constraints of the storage facility. Then the electrical hardware, mechanical hardware, and computer software required to assemble the Dexterous Inspection Module are described in detail.

6.4.1 Problem Statement

The assumptions regarding operating heights, accessible drum surface, and aisle widths have not changed from Phase 1. These assumptions can be found in section 6.2.1, titled *Specifications,* on page 1 of this document.

6.4.2 Methodology

The operation and capabilities of the Dexterous Inspection Module are best summarized through the description of a typical drum inspection sequence. The flowchart in Figure 6-6 shows the sequence that will be followed to inspect a waste drum. The Video Inspection package will inspect a section of the facility and produce a list of suspect drums. The Dexterous Inspection Module will return to the suspect drums on the list and perform detailed investigations using cameras and surface sampling tools, which will probe individual drums for indications of leaks or loss of structural integrity. It is assumed that the mobile positioning mechanism is accurate enough to position the Dexterous Inspection Module along the centerline of a vertical stack of drums. Some small offset is allowed but the navigation system must communicate this offset to the Dexterous Inspection Module. The lift assembly should be capable of moving to at least one vertical position per drum.

The tasks which the Dexterous Inspection Module must perform are described in detail below:

- *Bar Code Capture/Decoding* - The first step is to make sure that the correct drum is being inspected. This is done by capturing an image of the bar coded inventory number on the drum with the end effector camera, using the following steps.

1. The manipulator is raised to the height of the drum to be inspected.
2. The overhead pan/tilt unit is oriented towards the side of the aisle on which the suspect drum is found.
3. The overhead pan/tilt unit is aimed at the middle drum section on the suspect drum. If the bar code label is not found on the middle drum section, the top and bottom drum sections are inspected with the overhead camera.
4. Once the bar code label has been located, the operator clicks on it in the captured image, and the manipulator positions the end effector mounted camera in front of the bar code.
5. The bar code is decoded from the captured image, and can be used to reference the drum database maintained by the K3A mobile base.

- *Close Inspection of Suspect Spots* - Once the drum's inventory number has been decoded and verified, the operator will inspect the suspect spots on the drum's surface using the following steps.

6. The operator will command the overhead pan/tilt unit to face the drum section containing the suspect spot.
7. The operator will mark the suspect spot on the overhead camera image by clicking on it with the mouse.
8. The operator commands the manipulator to enter inspect mode.
9. From the inspect mode dialog box, the operator can click various buttons to move the end effector along the contour of the drum, at a fixed distance from the surface of the drum, thereby inspecting at close range the entire drum surface if desired, starting at the suspect spot. There is also a local pan/tilt capability which can be used to pan and tilt the end effector mounted camera.
10. When the operator has finished performing the close range inspection of all suspect spots, he terminates inspect mode.
11. If the operator wishes to acquire surface samples of suspect spots, he can mark the suspect spots in the same manner as used for inspection, then issue the sample command.
12. The manipulator then acquires a clean sampling tool, and obtains a surface sample of the spot, afterwards placing the soiled sampling tool in a numbered container. This sample will be analyzed when the mobile robot returns to base. (In a similar manner other devices, such as eddy current or ultrasonic thickness transducers, could be placed at or near the drum surface.)

During the performance of the inspection tasks the operator will be viewing windows on the workstation monitor that contain live and/or still video of the remote environment.

A completely autonomous inspection could be implemented with high level vision software, although human monitoring is always important when dealing with a robotic manipulator in hazardous environments.

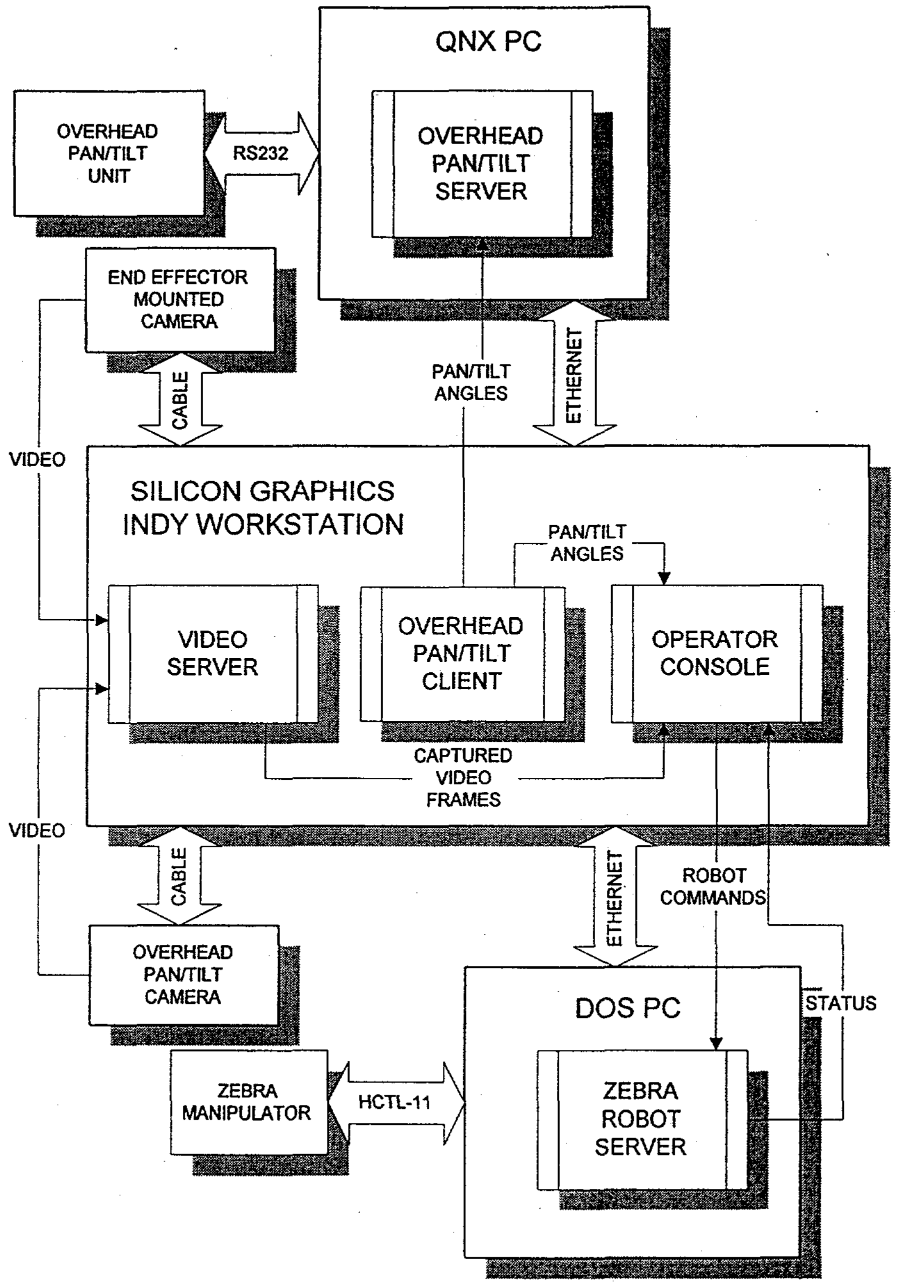


Figure 6-8 Dexterous Inspection Module Block Diagram

6.4.3 System Architecture

Figure 6-8 shows the hardware and software components, along with their interconnections, which comprise the Dexterous Inspection Module.

6.4.4 Hardware

The Dexterous Inspection Module is designed to provide the supplemental hardware and software to support a commercially available manipulator arm in order to perform waste drum inspection. The components of the Robot Dexterity module can be grouped into subsystems as shown in Table 6-1. The subsystems are divided by function as follows:

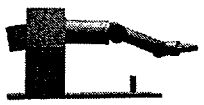
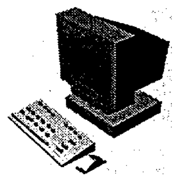
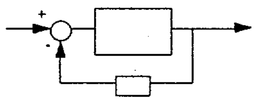
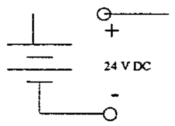
	*Robotic manipulator* - mechanical device which will move to perform inspection functions including close inspection and bar code decoding with camera and surface sampling.
	*Operator workstation* - Silicon Graphics based control panel used by the operator.
	*Control system* - hardware that controls movement of the manipulator and the pan/tilt units
	*Power supply* - battery system for supplying electrical power to the Robotic Dexterity module.
	Overhead pan/tilt unit - Used to position the overhead camera
	Overhead camera - used to locate bar code labels and suspect spots
	End effector mounted camera - used to decode bar codes and perform close range inspections of suspect spots

Table 6-1 Robotic Dexterity Module Subsystems

In Figure 6-9 the interaction between the subsystems of the dexterous manipulation module is shown in block form. The functional blocks and their interactions can be seen to implement the inspection operations described in the operational scenario. The flow of signals can be classified as:

- Directed to the operator
- Directed from the operator.

The signal flow to the operator will include information transmitted through the communication system and includes:

- Live/still video
- Robot status/position
- Overhead pan/tilt unit status/position

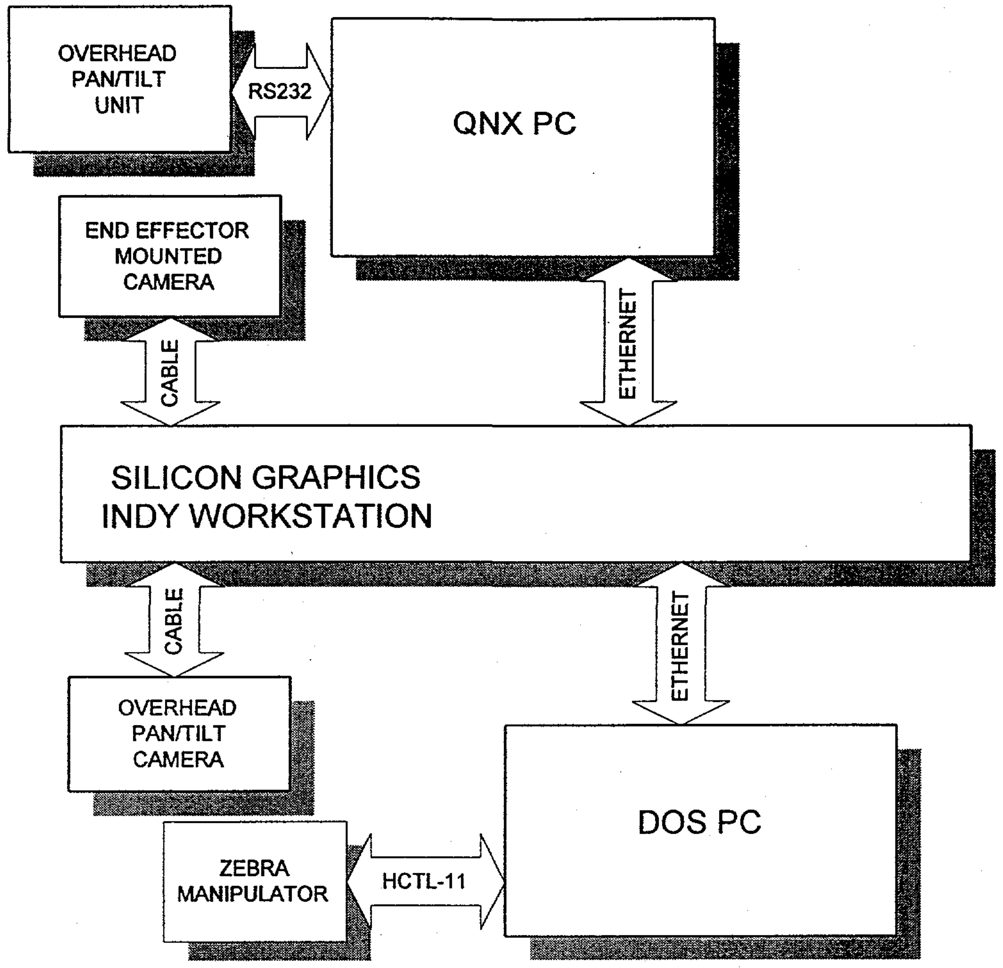


Figure 6-9 Block Diagram of the Dexterous Inspection Hardware

The incoming information is displayed to the operator and the operator will initiate actions based on this information. The operator will send commands through the communication channel to the control system. The control system then oversees the commanded use of tools or manipulator arm motion. In the following sections, the components that constitute the functional subparts are described.

6.4.4.1 IMI Zebra Zero Force Manipulator

Since the manipulator arm is the core of the Dexterous Inspection Module, the selection of manipulator arm is important. There are many manipulator arms currently available; however, few qualify for use here. The factors considered when selecting a manipulator arm are listed below:

- *Coverage of the drum surface by workspace of manipulator arm* - The intersection of the manipulator's workspace (within which it can position the end effector at an arbitrary orientation) and the surface of the drum, or a cylinder several inches larger in radius than the drum, determine how much of the drum surface can be sampled and inspected, respectively. Maximizing this intersection will decrease the number of times the lift platform will have to be moved per drum.

- *Power consumption of arm and low-level controller* - Obviously the manipulator must be able to operate from the mobile base's power supply, or a comparable DC power supply. This requirement excludes most, if not all, industrial robots.
- *Weight of arm/controller* - Since the manipulator is to be deployed on a mobile base, and lifted to various drum heights, the more it weighs, the more power the base and lift will draw while positioning it.
- *Available high-level control software* - In keeping with the underlying requirement to use off-the-shelf components, where possible, it is preferable to have a high level programming interface to the manipulator's controller. Although we are capable of developing both the low-level and high-level software, it is much more costly to maintain low-level device-dependent software.
- *Cost* - Dexterous manipulators can cost from $20,000 per degree of freedom and up, due to their complex nature and their vertical market.
- *Electrically driven actuators* - Since the mobile base uses an electric power source, a manipulator with pneumatic, hydraulic, etc. actuators would require an additional power source, adding mass and bulk to the entire mobile package.
- *Force sensor* - Tactile feedback is helpful in tasks such as picking up and replacing tools, obtaining surface samples, and detecting and limiting collisions.
- *Electric gripper* - Many electrically actuated manipulators are sold with pneumatic grippers. There is no pneumatic power source on the mobile base, so an electric gripper must be used.

Close examination of this reduced field of manipulator arms determined that the IMI Zebra ZERO Force Control manipulator arm is best suited for use with the Dexterous Inspection Module. The relevant specifications for the Zebra are given in Table 6-2 below.

Configuration	Articulated
Drive	DC servo motors
Reach	25 inches
Max. Payload	2.2 lb.
Force Sensing	6 axis wrist sensor
Standard Gripper	Electric, 0-3.5 inch grasp
Arm Weight	25 lb.
Power Requirements	24V, 7A typical moving slow, 25A peak at max. velocity/acc.
Programming Interface	High level C library

Table 6-2 Zebra Specifications

The IMI Zebra ZERO Force manipulator is manufactured by Integrated Motions, Inc. [Ref. 3]. The manipulator is composed of six revolute joints that allow the manipulator to position tools with arbitrary orientation within a characteristic workspace. The Zebra has a six-axis force sensor mounted in the wrist, an electrically actuated gripper as an end effector, and is controlled by an HCTL11-based 8 axis motor control card. The Zebra is shown in Figure 6-10.

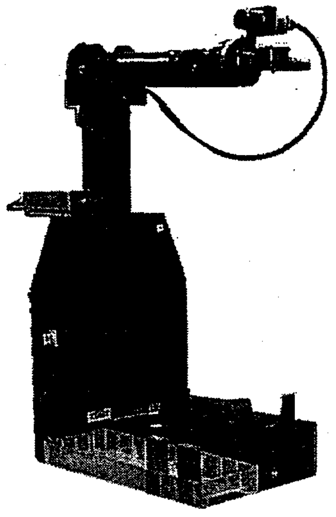


Figure 6-10 IMI Zebra Zero Force Manipulator

Figure 6-11 below shows the Zebra's joint axes and how the joint angles are measured. Table 6-3 lists the joint angle ranges.

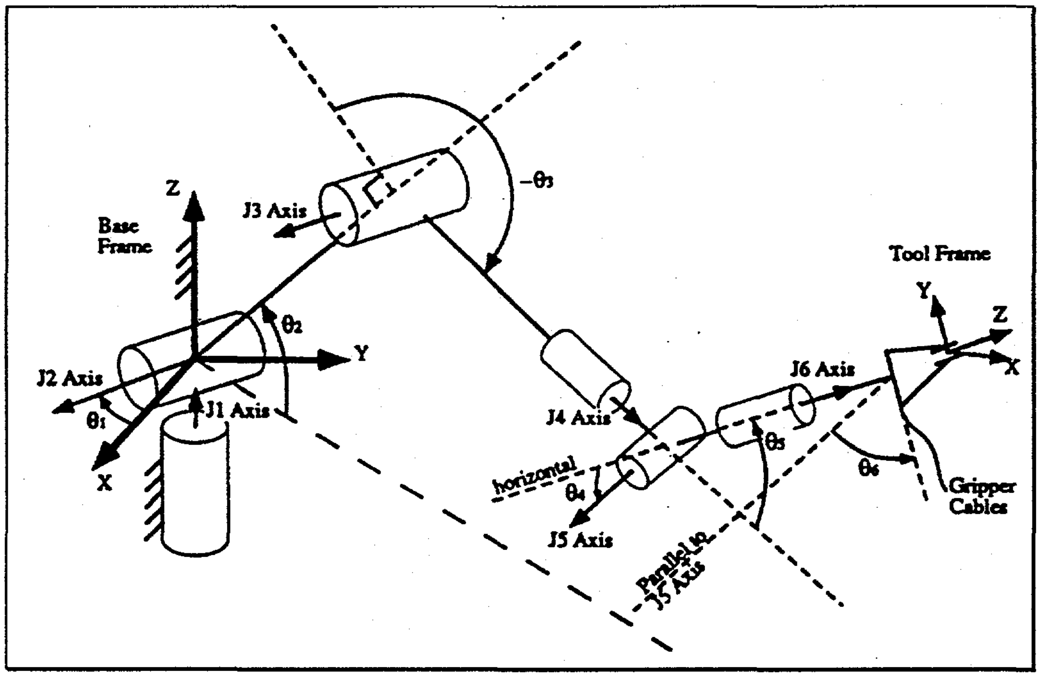


Figure 6-11 Zebra Joint Angles

Joint	Min	Joint Stop	Max	Joint Stop
1	-180	No	+180	No
2	-45	Yes	+135	No
3	-220	Yes	-90	Yes
4	0	Yes	360	Yes
5	-100	Yes	100	Yes
6	-180	No	180	No

Table 6-3 Zebra Joint Angle Limits

A full sized standard PC was used to control the Zebra in this pre-prototype phase, but several embedded single board computers have been identified which could run the robot. These all can run on a +5VDC power supply and draw about 1A of current.

6.4.4.2 End Effector Inspection Camera

The end effector mounted camera is used in the detailed visual inspection by the remote operator. The camera is mounted on a ring to the right of the end effector, with the axis of the camera coinciding with the Y axis of the robot's tool frame. This gives the operator the best vantage point for close up inspection of the drum, and allows the inspection of almost the entire visible drum surface without repositioning of the manipulator base.

The requirements for the camera are listed below.

- *Minimal weight* - Because of the limited payload capability of the end effector (2.2 lb.), the camera should be as light as possible, to allow as great a real payload as possible.
- *Small size* - A large bulky camera would interfere with the operation of the manipulator and increase the chance of collisions. Since it is being mounted on the manipulator's wrist, the camera must be small enough so that it does not interfere with rotations of the tool coordinate frame which are used during close inspection of the drum and sampling operations.
- *Low power* - Always a desirable characteristic for any component of a mobile, battery-powered system.
- *Color* - Color was deemed essential to the ability of the operator to identify suspect spots.
- *½ inch ccd and S-Video out* - It was determined experimentally that an imaging array smaller than ½ inch on a color camera would not provide the necessary resolution to decode the bar code at the distance required. Also, separate intensity and color signals (Y/C) were found to be necessary for the image to be of high enough quality to discern 15 mil narrow bars on the bar code labels at the distance specified, hence an S-Video output is required.
- *Rugged design* - The camera should be able to withstand the shocks and vibrations associated with powered motion, and the occasional collision.

Of the many commercially available camera systems surveyed, including manufacturers such as Sony, Panasonic, and Mitsubishi, only the Pulnix[i] TMC-7 with detachable remote head was found to meet all of the requirements [Ref. 4]. The hardest specification to fulfill was the small size and weight, because the weight and size of the lens must be added to that of the camera. The TMC-7 solves this problem by offering an option to separate the imaging head and lens from the rest of the camera body. A 4 foot cable connects the head to the body.

Only the remote imaging head and lens are mounted on the end effector to reduce the amount of mass carried by the arm and the possibility of interference with the manipulator's motion. The camera body is mounted to the manipulator's shoulder (see Figure 6-13) so that it exerts a force only along the negative J1 axis (down along the base joint's shaft) and therefore does not interfere with the robot's operation. The remote head mounting bracket is anchored behind the wrist force sensor, so that it does not interfere with operation of the force sensor during the inspection process.

The relevant specifications of the TMC-7 are listed in Table 6-4 below.

Weight	145g (weight of head + lens)
Size (W x H x L)	28mm x 28mm x 80mm (dimensions of head + lens)
Power Requirements	12VDC, 500mA
Resolution	460x400 TV lines
Video Output	Y/C (S-Video) and composite NTSC
Vibration/Shock	7G (200Hz to 2000Hz)/70G

Table 6-4 TMC-7 Specifications

The end effector camera's detachable head with an 8mm lens is shown in Figure 6-12.

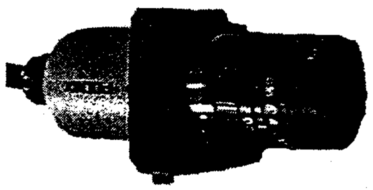


Figure 6-12 End Effector Camera Head

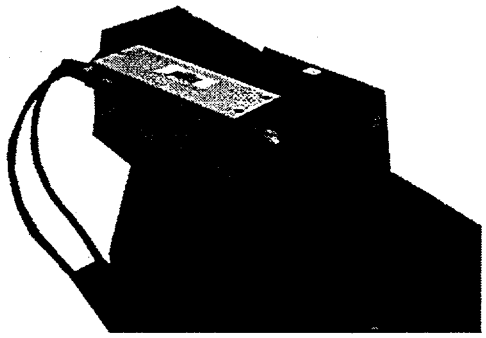


Figure 6-13 End Effector Camera Body

6.4.4.3 Operator Workstation

The operator station will be required to perform user interface, command, display, and communication functions. The display functions include display of live and still video of the inspection area, and display of status information, such as robot position. Required features for the operator workstation were identified and are listed below.

- *UNIX Operating System* - UNIX is an ideal software development and execution operating system. Most research facilities and many businesses rely on its robustness and the large body of scientific and technical software tools available for it. In addition, the ARIES team at USC was also using UNIX for their operator interface software, which provided an added incentive if compatibility with their system was desired.
- *X Windows and Motif development tools* - X Windows provides a network-capable graphical user interface (GUI) for UNIX and other platforms. One of the advantages of X over other non-network implemented GUIs (such as Microsoft Windows or OS/2 Presentation Manager) is the ability to remotely display program output and read program input. An X client can run on one machine and display its results and interact with a user anywhere on the Internet. Motif was chosen for the actual GUI development because it is an OSF standard and provides pre-built widget classes which facilitate the implementation of user friendly interfaces.
- *Live video display and capture* - Obviously, it is necessary for the operator to be able to see what is occurring in the remote environment being inspected. The ability to display live video in a window on the workstation screen was necessary to avoid having the operator constantly shifting his attention from the workstation display to a fixed function NTSC video monitor. The capture capability allows images to be stored and processed or cataloged later, and is used to store and decode an image of the bar coded labels identifying drum inventory number, contents, etc. Captured images from the overhead pan/tilt unit are also used to allow the operator to mark suspect spots for inspection and sampling.
- *C compiler* - UNIX is written in C and most if not all system libraries are written in C. C is popular, standard, and portable for the most part (barring system specific library function calls). Also, to maintain homogeneity across all computing platforms used in this project (DOS, QNX, UNIX), C was chosen, since it was the only language for which a compiler was provided on the DOS and QNX machines.
- *BSD style sockets API for communications* - Though available on most flavors of UNIX, BSD sockets are not standard. Since a sockets interface was the only means

found to communicate across an Ethernet layer with both the DOS-based robot server and the QNX-based pan/tilt servers, the operating system chosen for the operator workstation had to support it.

- *Fast RS-232 interface* - For interfacing to the magnetic tracker used in the Telepresence Module, a fast serial interface (RS232 or RS485) capable of 38,400 BPS continuous operation (while other tasks are executing) was needed.
- *Commercially available/well supported* - One of the underlying requirements for all of the equipment was that it be off-the-shelf whenever possible to reduce cost and enhance ease of servicing/replacement. The workstation selected should have a reliable manufacturer with a proven reputation for high quality products.

Two workstations from major vendors met the qualifications and were considered for the project; a Sun Sparc 10 with a third party video board, and a Silicon Graphics Indy workstation. Silicon Graphics [Ref. 5] is synonymous with high resolution, high color graphics and video displays, so it was decided that an SGI Indy would provide the best performance within budget. The Sparc met all of the other requirements but would have required support from a relatively unknown third party vendor for the video capture/display board. In addition the software libraries provided with that video capture card were completely proprietary and followed no standard, neither did they integrate with X windows. In fact at the time this report is being written, this third party developer (Image Manipulation Systems) has discontinued the video/capture board (IMS1000) for the Sparc and has also dropped all support for it.

The SGI Indy's VINO (video in no out) capture hardware is standard equipment on all Indy workstations, and SGI's digital media libraries are standard on all Silicon Graphics workstations. In addition all video display/capture functions are fully integrated with the X server through Silicon Graphics GL (Graphics Language). There was no doubt about support for the Indy's VINO video capture device, and its performance far exceeded that of the Sparc 10/third party video board (demos of both systems are available).

The SGI Indy, running the operator console software, is shown below in Figure 6-14.

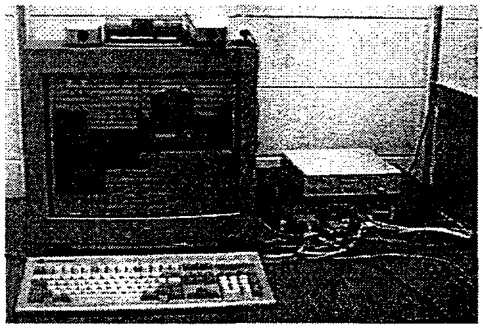


Figure 6-14 Operator Workstation

In addition to being used as a platform for development and execution of the operator console software, the Indy was also used as the NFS server for all of the source code for the DOS and QNX servers. In addition, software developed on the Indy is binary compatible with the SGI workstations used at USC on this project, as demonstrated in the Phase 2 demo. The Indy proved to be a very appropriate choice for the operator workstation.

6.4.4.4 Overhead Pan/Tilt Unit

In Phase 1, the camera used to mark suspect spots on the drum surface was fixed to the manipulator's shoulder. While this minimized error caused by inaccuracies in the model of the kinematic transformations from the camera to the end effector, it also severely limited what could be inspected. The manipulator platform had to be lifted to each drum section (3 levels per drum), which requires a large amount of power. In Phase 2 this camera was placed on an overhead downward facing pan/tilt unit. This not only eliminated the need to raise the manipulator platform to three drum sections (only one position per drum is required now), it also allowed for inspection of both sides of the aisle.

The requirements for the overhead pan/tilt unit are listed below.

- *Low power operation* - When the unit is not moving, it should be able to hold its position with little or no power consumption.
- *DC power supply* - Must operate on a DC voltage level available or obtainable from the mobile robot's power supply.
- *Mounting in arbitrary orientation* - To allow a maximum inspectable area, and to avoid intruding in the robot's workspace, the pan/tilt unit should be able to be mounted in any orientation (upside down, sideways, etc.)
- *PC interface* - Since the Zebra requires a PC based controller, the pan/tilt unit should also be able to interface and be controlled by a PC to avoid having multiple computers deployed with the Dexterous Inspection Module.
- *Large pan/tilt axes ranges* - To allow great flexibility in the inspection process, the operator should be able to pan/tilt as far as possible. Specifically, it should be possible to pan from one side of the aisle to the other, and from the top drum section to the bottom drum section.
- *Small size* - A bulky unit will increase the likelihood of collisions with the manipulator, and will limit the manipulator's workspace.
- *Light weight* - Since the unit must be lifted along with the manipulator to inspect drums at various heights, its weight should be limited.

The Directed Perception PTU-46-17.5 [Ref. 6] fulfilled all of the requirements. The relevant specifications are listed below in Table 6-5, and a picture of the DPPTU can be seen in Figure 6-15.

Power Requirements	11-40VDC unregulated, 16W peak (full power mode), 7.5W peak (low power mode), 1W peak (holding power off mode)
Possible Mounting Orientation	Any
Hardware Interface	RS232, 9600bps
Pan Axis Range	+/- 170°
Tilt Axis Range	+31° to -80°
Dimensions (W x H x L)	3.4" x 2.5" x 5.5"
Weight	12 oz

Table 6-5 Directed Perception Pan/Tilt Unit Specifications

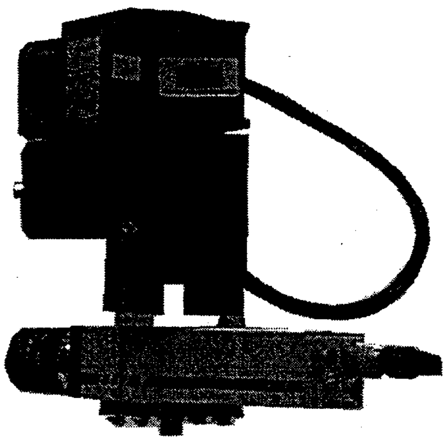


Figure 6-15 Directed Perception Pan/Tilt Unit

Though using a pan/tilt unit instead of fixing a camera to the manipulator has complicated the problem of determining position of objects in world space (because the kinematics of the pan/tilt unit must be added to the chain) it has increased the flexibility of the inspection package greatly, allowing inspection of both sides of the aisle, and of all drum sections without repositioning of the manipulator platform by the lift.

6.4.4.5 Overhead Camera

Requirements for the overhead camera are similar to those of the end effector mounted camera. To minimize the number of vendors used and the amount of training required, it was decided to purchase a camera identical to the end effector mounted unit.

6.4.5 Software Architecture

A client/server software architecture was adopted for the Dexterous Inspection Module. The client/server model fit very well with the proposed system; certain hardware devices (manipulator, pan/tilt units, video capture hardware, etc.) had to be encapsulated and made available on the network. These devices clearly were to be run by server programs. The operator would be using programs to command and interact with these devices over the network, so he would be using the client programs. This client/server architecture allows significant flexibility in how the hardware is used. Since the high level sockets interface is well defined, it is a simple matter to write a client program to interact with the servers. First, simple clients with no graphical user interface were developed. For example, a pan/tilt client which got pan/tilt angles from the command line was developed. Later the pan/tilt control panel with all of its Motif-based graphical sliders and menus was developed, but the same skeleton client code was used as in the simple command line pan/tilt client.

One of the major tasks for the Dexterous Inspection Module was system integration. Developing low level interface software for the pan/tilt units and the video capture hardware was challenging, but assembling all the software components across the heterogeneous computing and networking environment (UNIX workstation, DOS PC, QNX PC) using Ethernet and serial connections comprised the majority of the effort.

It was determined that to perform the inspection tasks specified, the five software components listed below had to be developed.

- *Robot server* - Since the Zebra manipulator was provided with a high-level C library of functions, there was no need to write low-level interface software for it. However, the Zebra must be controlled from the operator workstation, so a network interface must be provided. In addition, since network response is non-deterministic, and network failures may occur at any time, the operation of the robot must be shielded from dependence on the network. To do this, it was decided that a robot server program would be developed, which would allow clients on the net to connect to it and issue high-level commands (inspect this point, sample this point, etc.) to the robot. The server program itself would then make the appropriate C library calls and monitor the robot's operation. If the network fails, the robot server will not cause the robot to lose control, it will simply wait for the connection to be restored before doing anything else. The robot server is a DOS-based C program, which uses IMI's *robot.lib* for robot control functions, and the WATTCP library for socket-based network communication. The only reason DOS is used as the operating system is that the robot library is only provided for DOS. If desired, this library could be ported to QNX if some arrangement was made with Integrated Motions to acquire the source code. They would not provide the source code free of charge.
- *Video server*- If the interface software for the video capture hardware was made part of the operator console software, then it would not be able to execute over the Internet for a remote inspection. So a server for the video capture hardware was written. This server allows a client anywhere on the Internet to connect to it and request frames of video from the Indy's VINO capture device. Each frame is captured, then transmitted over the net using sockets.
- *Zebra control panel (operator console)* - The Zebra control panel software provides the user interface for the operator to control the manipulator. The Zebra control panel was designed for ease of use; it is completely GUI based, with pull down menus, buttons, and other Motif user interface widgets. No commands must be learned by the operator. In addition, the operator console connects to the video server to provide the operator with frames of video captured from the overhead pan/tilt mounted camera as well as the end effector mounted camera. Along with issuing high level robot commands to the robot server and obtaining video from the video server, the Zebra control panel program also translates two dimensional image-plane coordinates from the overhead pan/tilt camera view into three dimensional world-space coordinates which can be sent to the robot server as arguments to the inspect and sample functions.
- *Overhead pan/tilt server* - The Directed Perception pan/tilt unit was interfaced to an RS232 port on the QNX based PC. A socket-based server program was written to provide a high level network interface which hides the details of the unit's operation from client programs, and also allows the pan/tilt unit to operate over the Internet.
- *Overhead pan/tilt client* - This GUI based pan/tilt control panel provides two sliders, for pan/tilt angles respectively, as well as menu options for aiming the camera at various predefined locations (left/right drum, top/middle/bottom sections). This program executes along with the operator console on the operator workstation.

Figure 6-16 below shows the relationship between the software modules.

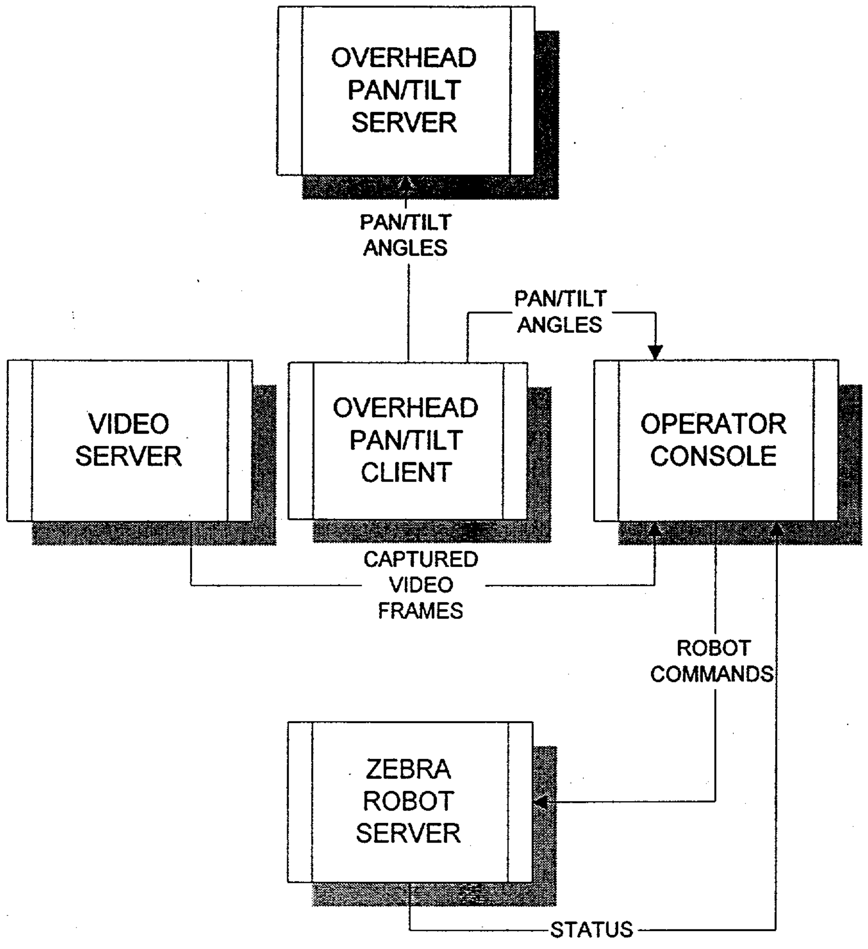


Figure 6-16 Block Diagram of the Dexterous Inspection Software

6.5 TELEPRESENCE MODULE

Telepresence allows a human operator to be immersed in a remote environment. During the course of developing the Dexterous Inspection Module, it was decided that watching video in a window on the operator workstation, and having to constantly reposition the camera with the overhead pan/tilt unit was not a very natural way to observe the remote environment. It was decided to attempt to adapt some tools normally associated with virtual reality research for use in a remote inspection. Virtual reality researchers use head mounted displays to immerse the user in a virtual environment. A magnetic tracker is attached to the head mounted display (also referred to as a helmet). The tracker reports the position and orientation of the user's head, and this information is used to position the viewer in the virtual environment. The same idea was implemented with the Telepresence Module, the only difference being that instead of controlling the position of a viewer in a virtual environment, the tracker now controls the orientation of the pan/tilt mounted cameras in the remote environment.

Since humans perceive the world in three dimensions through stereo vision, a stereo video display was deemed necessary for the operator to be immersed in the remote environment. This also requires two video cameras, with some fixed distance between them, to generate the left and right eye views.

The Telepresence Module's hardware and software components, along with their interconnections, are shown in Figure 6-17 below.

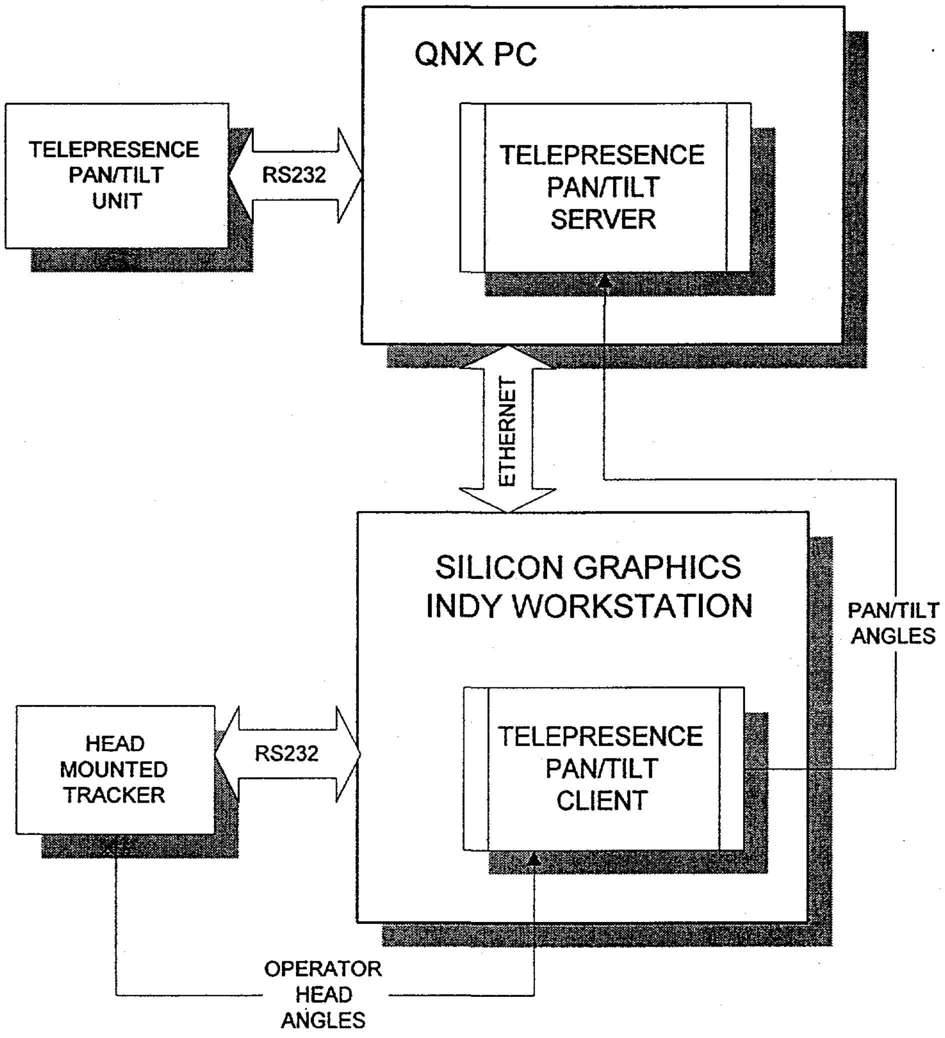

QNX PC
TELEPRESENCE PAN/TILT UNIT
RS232
TELEPRESENCE PAN/TILT SERVER
ETHERNET
SILICON GRAPHICS INDY WORKSTATION
PAN/TILT ANGLES
HEAD MOUNTED TRACKER
RS232
TELEPRESENCE PAN/TILT CLIENT
OPERATOR HEAD ANGLES

Figure 6-17 Telepresence Module Block Diagram

6.5.1 Telepresence Hardware

Table 6-6 below lists the hardware that was determined to be necessary in order to implement the Telepresence Module.

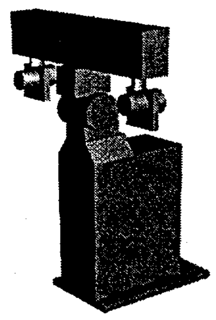
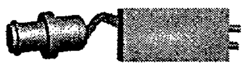
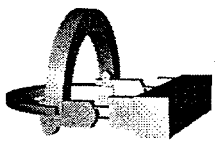
	*Pan/tilt/vergence unit* - The core of the Telepresence Module is a good pan/tilt/vergence unit, which must carry the cameras used to relay video back to the operator.
	*Color cameras* - Needed to generate stereo views of the remote environment.
	*Head mounted display* - The operator views the remote environment through a helmet display.
	Magnetic tracker - The tracker must provide position and orientation of the operator's head.

Table 6-6 Telepresence Hardware

The interconnection of the hardware components is shown in Figure 6-18 below.

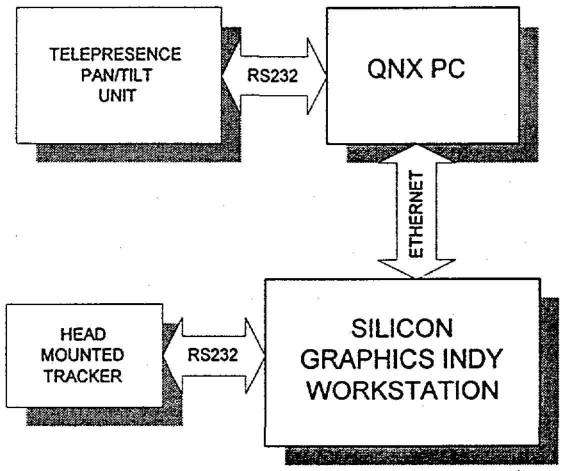


Figure 6-18 Telepresence Module Hardware

6.5.1.1 Pan/Tilt/Vergence Unit

For a pan/tilt unit to be used in the role specified here, it must meet some unusual requirements, which are listed below:

- *High torque/speed* - Because the pan/tilt unit must replicate the motion of a human head, it must be capable of high acceleration and velocity operation. In addition, it must be able to carry the dual camera payload at these high velocities.
- *Vergence axis* - To obtain stereo image pairs at varying distances, a vergence axis is required. This axis turns the cameras toward or away from each other, simulating the way a human's eyes converge as he focuses on objects that are close or far away.
- *PC based controller* - Since the Dexterous Inspection Module uses a PC to control its remotely deployed hardware, it is preferable that this pan/tilt unit also be controllable by a PC.

The TRC UniSight pan/tilt base [Ref. 7] with a BiSight vergence head meets all of these specifications. Unlike the Directed Perception pan/tilt unit used to orient the overhead camera in the Dexterous Inspection Module, which uses stepper motors for low power operation, the TRC unit uses brush DC motors for high torque/speed operation. Each camera mount has a separately controllable vergence axis so that the cameras can converge on targets at various distances. The controllers supplied with the pan/tilt/vergence unit have RS232 serial interfaces, and have been connected to the QNX-based PC which runs the overhead pan/tilt unit of the dexterous manipulation module.

The TRC UniSight pan/tilt unit and BiSight vergence head along with the dual color cameras are shown in Figure 6-19 below, and the specifications are shown in Table 6-7.

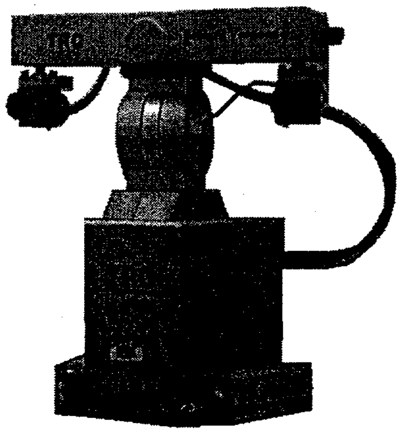


Figure 6-19 TRC Pan/Tilt/Vergence Unit with Cameras

Weight	14 lb.
Payload (per vergence axis)	1.5 lb.
Power requirements	14A@24VDC (max.)
Size (L x W x H) in inches	6.8 x 12.7 x15
Torque on pan/tilt axes	11 in.-lb. (1.24 N-m)
Torque on vergence axes	6.5 in.-lb. (.73 N-m)
Max. angular velocities (°/sec)	650 (pan) 500 (tilt) 1000 (vergence)
Max. angular accelerations (°/sec^2)	1350 (pan & tilt) 12000 (vergence)
Range of motion (degrees)	+/- 160 pan, +/- 90 tilt, +/- 90 vergence

Table 6-7 TRC Pan/Tilt/Vergence Specifications

6.5.1.2 Color Cameras

Requirements for the color cameras mounted on the vergence axes of the TRC pan/tilt/vergence unit were similar to those for the end effector mounted camera. Light weight and small size increase the performance and workspace of the pan/tilt unit. The camera used with the manipulator's end effector turned out to be of very high quality, small size, and light weight, so the same model was used for the TRC pan/tilt unit. The remote cylindrical heads of these cameras were essential because vergence angles would be severely limited if the camera body was mounted on the vergence joints (the camera bodies would collide with each other and the pan/tilt unit's mechanical structures).

6.5.1.3 Head Mounted Display

The requirements for a head mounted display to be used with this remote inspection system are as follows:

- *Stereo display capability* - To generate the 3D effect of depth perception, two images are needed, one from the left eye, and one from the right eye. A helmet with two separate video displays is needed.
- *Light weight/comfortable* - If the helmet is too heavy, or uncomfortable, the operator will not be able to wear it for any length of time, and it will not be used.
- *Allow viewing of monitor* - The helmet should not be of the completely enclosing type; the operator should be able to see both the helmet's displays, and the operator workstation monitor without removing the helmet.
- *Good color display* - Color is essential in determining the condition of drum surfaces.

The CyberEye CE100-S color stereo head mounted display by General Reality Company [Ref. 8] meets all of these requirements. This head mounted display with the magnetic tracker receiver mounted on it is pictured in Figure 6-20.

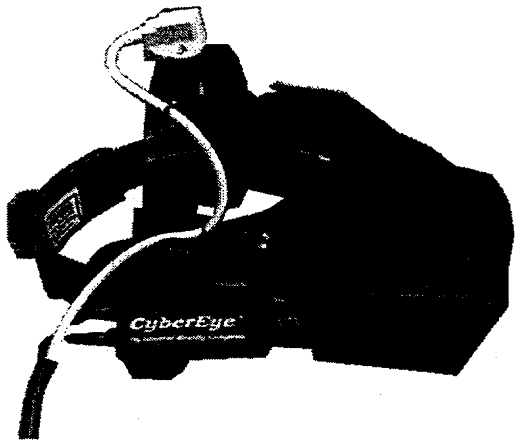


Figure 6-20 CyberEye Head Mounted Display

The specifications for the CyberEye helmet are shown in Table 6-8 below.

Display type	Dual active matrix LCD
Contrast ration	100:1
Resolution	420x230 (96,000) pixels per eye (triad)
Field of view	22.5°H x 16.8°V (27.5° diagonal)
Pixel size	3.2 arc minutes
Input signal	NTSC
Optical adjustments	Focus & interocular spacing, brightness
Weight	14oz

Table 6-8 CyberEye HMD Specifications

The CyberEye was the only helmet found which could be worn while still viewing the workstation monitor, as shown in Figure 6-21.

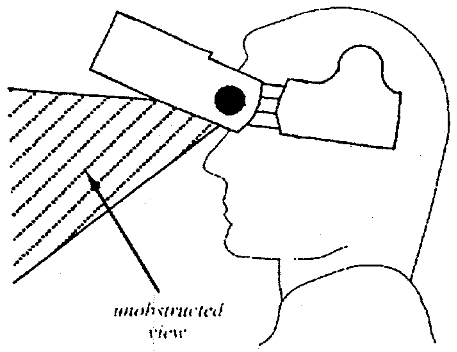


Figure 6-21 Unobstructed Monitor Viewing Area

6.5.1.4 Magnetic Tracker

The requirements for a head mounted tracker used to relay the operator's head position to the remote pan/tilt unit are as follows:

- *Light weight* - Must not encumber the operator's head, or else the operator will become fatigued quickly.
- *Fast sampling rate* - If the sampling rate is too low, the operator will experience excessive lag between his head motions and the motions of the pan/tilt unit. This can cause overcompensation by the operator, as well as motion sickness.
- *Interface to operator workstation* - The tracker must somehow be able to interface with the operator workstation, which communicates with the pan/tilt server.

The Ascension Technologies *Flock of Birds* six-degree-of-freedom (position and orientation) magnetic tracker [Ref. 9] meets all of these requirements. This type of tracker is used in virtual reality research. It has a reporting rate of 30 updates per second and uses an RS232 interface, which is available on the operator workstation. The tracker is pictured below in Figure 6-22, while the specifications appear in Table 6-9.

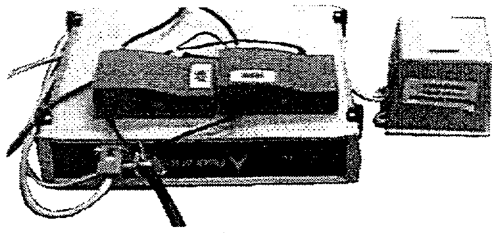


Figure 6-22 Ascension Technologies Magnetic Tracker

Positional range	+/- 36 in. along all axes
Angular range	+/- 180° azimuth & roll, +/- 90° elevation
Static positional accuracy	0.1 in. RMS
Static angular accuracy	0.5° RMS
Positional resolution	0.03 in. @ 12 in.
Angular resolution	0.1° @ 12 in.
Update rate	30 Hz
Interface	RS232 (2,400 - 115,200 BPS) RS485 (57,600 to 312,500 BPS)

Table 6-9 Ascension Technologies Magnetic Tracker Specifications

6.5.2 Software Architecture

As with the Dexterous Inspection Module, the major software related task for the Telepresence Module was system integration. Providing a high level interface for the various hardware devices, and implementing the necessary communications protocols comprised most of the effort. A fair amount of effort was expended in porting the DOS-based control software for the TRC pan/tilt unit to QNX. The TRC unit is far more versatile, and powerful, than the Directed Perception unit used with the overhead camera, but it is also far more complicated to control. It is a four-degree-of-freedom revolute robot with brush DC motors and PID controllers with user programmable control gains. Significant effort went into understanding its theory of operation and the software interface, as well as in the porting and restructuring of the software for use under QNX.

Since the Telepresence Module is not an integral part of the dexterous inspection package and was designed as an option which can either be deployed with the Dexterous Inspection Module (to be used by the operator to observe the inspection) or as a stand alone module which is used to perform a remote inspection of a drum with the most natural interface available (the operator simply moves his head around to look at different targets), the software was designed to be completely separate from the Dexterous Inspection Module software.

The two software components of the Telepresence Module are the pan/tilt server, and the pan/tilt client. Their interconnection is shown in Figure 6-23 below.

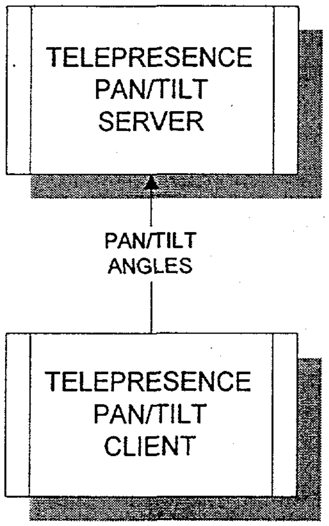


Figure 6-23 Telepresence Module Software

Pan/tilt server - This program executes on the QNX PC, and issues low-level commands to the TRC pan/tilt unit. It accepts connections from clients anywhere on the Internet. The client is expected to send the current position/orientation of the operator's head, along with a time stamp. The time stamp is used to interpolate desired velocities, although this is a crude method, especially with such a low sampling rate (30Hz). Ideally, the position signal would be filtered and then a velocity observer would be used to determine desired velocity, but the non-deterministic nature of the network, and the multiple serial connections would tend to make this effort not worthwhile.

Pan/tilt client - This program reads the current position/orientation of the magnetic tracker mounted on the operator's helmet, and sends this information along with a time stamp to the pan/tilt server program over the network using sockets.

6.6 RESULTS

Phase 2 of this project resulted in the specification of the hardware components, as well as the interconnections and software components required to construct a Dexterous Inspection Module which can be deployed as a payload on a mobile robot. This payload is capable of performing the tasks of decoding the inventory bar-code label, inspecting at close range suspect spots on a drum, and obtaining surface samples of suspect (i.e. rusty/leaky) areas for later analysis.

In addition, a Telepresence Module was introduced which allows the operator to observe the remote inspection process in a more natural fashion. This module can be deployed with the Dexterous Inspection Module, or as a stand-alone payload for use as a proxy for a human inspector, allowing the operator to remain in a safe location while inspecting drums in the hazardous environment.

Prototypes of both modules were built using commercially available components when available. The prototypes were demonstrated, and were capable of performing all of the tasks specified. A video of the demonstrations can be obtained by contacting the investigator [Ref. 10] for this subtask of the project.

Both modules were designed to be able to work as stand-alone systems, or to be integrated easily into a larger system. The only coupling between the various components of the system is Ethernet hardware and high-level socket messages using a well defined protocol. It would not be difficult to add intelligence to the Dexterous Inspection Module by interfacing it with a vision system that could detect and locate bar-codes and suspect defects. With this vision system capability, the human operator could be completely eliminated, allowing a totally autonomous inspection.

6.7 REFERENCES

1. John Lloyd, Department of Computer Science, University of British Columbia, Vancouver, B.C. V6T 1Z4, (604) 822-5109.
2. "An Intelligent Inspection and Survey Robot," Phase 2, Proposal No. 02-04-02-29634, Volume 2, December 22, 1993, Page 7, Section 2.2.4.1.
3. Integrated Motions, Inc., 758 Gilman Street, Berkeley, CA 94710. (510) 527-5810.
4. Subtechnique, Inc., 4819B Eisenhower Ave., Alexandria, VA 22304, (703) 212-0080.
5. Silicon Graphics, 1111 Alderman Drive, Suite 375, Building 300, Alpharetta, GA (404) 663-1238.
6. Directed Perception, Inc., 1451 Capuchino Ave., Burlingame, CA 94010-3308 (415) 342-9399.
7. Transitions Research Corp., Shelter Rock Lane, Danbury, CT 06810-8159 (203) 798-8988.
8. General Reality Co, 170 S. Morrison Ave., San Jose, CA 95126 (408) 289-8340.
9. Ascension Technology Co., PO 527, Burlington, VT 05402 (802) 860-6440.
10. Darren Dawson, Clemson University, 105 Riggs Hall, ECE Department, Clemson, SC 29634 (803) 656-5924, Email: ddawson@eng.clemson.edu.

7. Conclusions and Future Plans

A university/industrial technical team consisting of two universities (USC and CU) and a small business (Cybermotion, Inc.) has designed, developed, and fabricated a prototype mobile robotic system for the inspection of low-level radioactive waste stored in drums at DOE facilities. The autonomous demonstration held on 30 November 1995 at USC with all team members participating was evidence of the success of this project. Although the prototype was not perfect, it represented a technical effort that proved the feasibility of an autonomous inspection process.

Primary fabrication of the prototype ARIES #1 was conducted at Cybermotion, Inc., Salem, VA. Final detailed assembly and cabling was performed at the USC demonstration area with Cybermotion team members involved. Therefore, this system is well-known to Cybermotion, who will be expected to produce additional units if required by the DOE.

A dexterous manipulator inspector module was developed for consideration by DOE in future inspection applications. A complete study and cost analysis to radiation harden a mobile robot such as ARIES #1 was conducted (Volume 2 of this report).

Phase 3 (if funded by DOE) will focus on final productization of the prototype ARIES #1. Cybermotion will remain a vital partner in this effort, since they will provide two additional systems in addition to the Phase 2 prototype ARIES #1 as Cybermotion products during this phase. They will assemble, test, and install the new systems, ARIES #2 and ARIES #3, with cooperation from the University of South Carolina and Clemson University. During Phase 2 the DOE Fernald Site was identified as the customer for Phase 3 testing and demonstrations. The testing and demonstration at Fernald will be done in accordance with a test plan developed by Fernald. Also, a budget is included for planning visits and testing and demonstrations at another DOE site (such as Hanford or Idaho).

The proposed Phase 3 technical efforts focus on four general areas: productization of the Phase 2 prototype ARIES #1, testing and demonstration of the enhanced ARIES #1 at Fernald, enhancements as a result of customer input during the Fernald tests, and fabrication of two additional robots. The productization effort is the incorporation of improvements and enhancements to the prototype system produced in Phase 2 which was an initial thrust at integrating the technologies developed during Phase 1 of this project. Modifications and enhancements, identified during Phase 2 testing at the USC staging area, will be incorporated into the Phase 2 prototype prior to testing and demonstration at the Fernald Site. Customer feedback as a result of the Phase 2 demonstration will be used as the primary input to this activity. The Phase 2 prototype system with all ancillary equipment will be delivered to the DOE Fernald Site for testing and evaluation according to test plans developed by Fernald. After these tests, customer feedback and operational experience will be used to define additional improvements and enhancements to the design of additional systems to be fabricated.

Design uncertainties for future work are discussed below and summarized in Table 7-1.

Test Plan and Problem Definition The Fernald Site was identified late in Phase 2 as the customer site for the Phase 3 demonstrations. Only a brief site visit to Fernald has been made by two project investigators. Fernald has developed a test plan for the testing and demonstrations of ARIES and other inspection systems during FY96. We think it is important that we formally meet at Fernald to discuss the plan and the test expectations with the customer as early as possible in Phase 3 to eliminate any misunderstandings concerning test conditions and expectations.

The Environment To date the system has been tested and evaluated in simulated and laboratory environments. Until the robot is operated in the actual warehouse under normal warehouse operational conditions, there are many unknowns concerning environmental conditions (e.g. temperature, fork-lifts rearranging drums, etc.). Plans are underway to collect specific drum images at Fernald to be used for training the vision system.

Navigation The navigation goals for the project are to be able to *(i)* navigate a DOE standard 3-foot aisle, *(ii)* handle a mixture of 55-, 85-, and 100-gal drums, and *(iii)* deal with a dynamic environment in a large warehouse. The K3A has shown the ability to successfully navigate in and out of 3-foot aisle widths. We feel that the use of the lidar navigational system has resolved the problems associated with location identification in the large spacious areas of the warehouse. However, there are a number of unknowns concerning the aisles in the warehouse, the variable lengths of the aisles, the mixture of 55-, 85-, and 100-gallon drums and how that affects the aisle widths. Also, details to install lidar "targets" must be worked out with Fernald operators.

CONCERN	SOLUTION
Specific test plan and evaluation process is uncertain.	Meet as soon as possible with customers at Fernald to discuss specifics of test plan and evaluation process.
Actual warehouse test environment is unknown.	Collect drum images prior to delivery of ARIES at Fernald. Many of these concerns can only be addressed with ARIES at Fernald.
Navigation uncertainties.	Time must be allowed prior to formal test and demonstration at Fernald to resolve details concerning navigation in large open areas of warehouse.
Power consumption.	Analyze system during actual operation at Fernald.
Drum images.	More detail concerning drum images will be obtained early in Phase 3.

Table 3.2 Phase 3 Primary Concerns

Power Consumption The design goal for power consumption has been to enable the robot to operate for an entire 8-hour shift without re-charging. However, whenever a trade off has been made between power consumption and functionality during Phase 2, we have always favored functionality, especially for the processor and vision boards. Also, we tended to make "safe" choices concerning potential radiation hardening and various hardware and software standards. The result of these choices is that the system will require the installation of additional batteries in the current vehicle to meet the goal of an 8-hour shift. However, this may not be the approach taken for the additional vehicles. An important aspect of the Fernald test will be the collection of extended real-time data on power consumption. These data, together with better knowledge of the relevant standards, will be used to determine the optimum approach for the additional vehicles.

Drum Images Phase 2 design was done using photographs and videos of drums. There are many unknowns concerning the images until real data are obtained at Fernald. Plans are underway to collect those data early in 1996.

APPENDIX

Phase 2: ARIES

Project Objective and Task Descriptions

Large quantities of radioactive Low-Level Waste (LLW) and Transuranic Waste (TRU) are currently in storage at a number of sites throughout the United States. Environmental Protection agency (EPA) regulations call for regular inspection of such storage areas requiring significant increases in manpower. The overall objective of this work is to develop a semi-autonomous vehicle capable of inspecting a low radiation level contaminated storage area of stacked drums or other containers aligned in aisles. The vehicle will have the capability of autonomously entering, retracing the entry route, and avoiding all hazardous obstacles in the route traveled.

Major Tasks Descriptions (Work Breakdown Structure)

WBS 2.1	Provide coordination of tasks, budget administration, general management of project, and reporting according to the DOE's requirements.
WBS 2.2	Develop and fabricate an autonomous vehicle capable of locating and navigating 3-foot aisles of drum storage sites. The vehicle will have the capability of autonomously entering, retracing the entry route, and avoiding all hazardous obstacles in the route traveled. Assist Task 2.4 in the design of a vision payload system (Camera Positioning System) for the autonomous vehicle and then fabricate the module.
WBS 2.3	Deliver a control environment system architecture that includes overall organization and major components of the system, their communications protocols, the user environment, and the system mission structure. Integrate system application payloads into this architecture.
WBS 2.4	Design a mechanical Camera Positioning System (CPS) capable of compact storage aboard the robot during vehicle movement and capable of deploying the vision application payload with sufficient accuracy for the survey mission. Design and fabricate a prototype inspection support system.
WBS 2.5	Analyze all mechanical, electrical, and computer systems of the new robotic vehicle for radiation tolerance, and deliver a report on the radiation hardness of all electrical, electronic, computer, and other control systems with recommendations for future development work for operations in a high-level radiation environment.
WBS 2.6	Integrate and assemble all subsystems developed in Tasks 2.2, 2.3, and 2.4, and test the overall robotic system in a simulated drum storage area demonstrating all of its various features and capabilities. Deliver a topical report according to contract requirements.